优炫数据库参考手册 2.1

UXSINO
优 炫 软 件

优炫数据库参考手册 2.1

法律声明

优炫数据库管理系统(简称：UXDB) 是由北京优炫软件股份有限公司开发并发布的一款商业性数据库管理系统。

注意

由于产品版本安装或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

北京优炫软件股份有限公司（总部）

- 地址：北京市海淀区学院南路62号中关村资本大厦11层（邮编：100081）
- 网址：http://www.uxsino.com
- 邮箱：<uxdb_support@uxsino.com>
- 电话：010-82886998
- 传真：010-82886338
- 服务热线：400-650-7837

目录

插图清单

表格清单

范例清单

前言

1. 文档目的

本文档介绍优炫数据库标准模式与兼容模式下SQL命令、SQL语言、SQL关键词的相关内容，为相关技术人员和用户提供必要参考。

2. 文档对象

- 技术支持工程师
- 维护工程师
- 优炫数据库用户

3. 修改记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本的更新内容。

表 1. 文档更新记录

工具版本	发布日期	修改说明
2.1.1.5C	2023-01-09	第一次正式发布。

部分 I. 参考

这份参考中的条目意欲提供关于相应主题的权威、完整和正式的总结。关于使用UXDB的更多信息（以叙述、教程或例子的形式）可以在本书的其他部分找到。见每个参考页面上列出的交叉引用。

这些参考条目也在传统“man”页面中可用。

目录

SQL 命令

这部分包含UXDB支持的SQL命令的参考信息。每条命令的标准符合和兼容的信息可以在相关的参考页中找到。

目录

名称

ABORT — 中止当前事务

大纲

```
ABORT [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

描述

ABORT回滚当前事务并且导致由该事务所作的所有更新被丢弃。这个命令的行为与标准SQL命令ROLLBACK(7)的行为一样，并且只是为了历史原因存在。

参数

WORK
TRANSACTION

可选关键词。它们没有效果。

AND CHAIN

如果规定了AND CHAIN ，新事务立即启动，具有与刚刚完成的事务相同的事务特征(参见SET TRANSACTION(7))。否则，不会启动新事务。

注解

使用COMMIT(7)成功地终止一个事务。

在一个事务块之外发出**ABORT**会发出一个警告消息并且不会产生效果。

例子

中止所有更改：

```
ABORT;
```

兼容性

这个命令是一个因为历史原因而存在的UXDB扩展。**ROLLBACK**是等效的标准 SQL 命令。

参见

BEGIN(7), COMMIT(7), ROLLBACK(7)

名称

ALTER AGGREGATE — 更改一个聚集函数的定义

大纲

```
ALTER AGGREGATE name ( aggregate_signature ) RENAME TO new_name
ALTER AGGREGATE name ( aggregate_signature )
            OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER AGGREGATE name ( aggregate_signature ) SET SCHEMA new_schema
```

其中 *aggregate_signature* 是：

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

描述

ALTER AGGREGATE更改一个聚集函数的定义。

要使用**ALTER AGGREGATE**，必须拥有该聚集函数。 要更改一个聚集函数的模式，还必须具有新模式上的 CREATE特权。要修改拥有者，还必须是新拥有角色 的一个直接或者间接成员，并且那个角色必须在聚集函数的模式上拥有 CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该聚集函数来做任何不能做的事情。不过，一个超级用户可以更改任何聚集函数的所有权）。

参数

name

一个现有聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或VARIADIC。 如果省略，默认为IN。

argname

一个参数的名称。注意**ALTER AGGREGATE** 并不真正关心参数名称，因为决定聚集函数的身份时只需要参数的数据类型。

argtype

聚集函数要在其上操作的输入数据类型。要引用一个零参数聚集函数，在参数 说明列表的位置写上*。要引用一个有序集聚集函数，在直接参数 说明和聚集参数说明之间写上ORDER BY。

new_name

聚集函数的新名称。

new_owner

聚集函数的新拥有者。

new_schema

聚集函数的新模式。

注解

引用有序集聚集的推荐语法是在直接参数说明和聚集参数说明之间写上 ORDER BY，这和CREATE AGGREGATE(7) 中的风格相同。不过，省略ORDER BY并且只把直接和 聚集参数说明放到一个单一列表中也是可以的。在这种简写形式中，如果 在直接和聚集参数列表中都使用了VARIADIC "any"，只用 写一次VARIADIC "any"。

示例

要把用于类型integer的聚集函数 myavg重命名为my_average：

```
ALTER AGGREGATE myavg(integer) RENAME TO my_average;
```

要把用于类型integer的聚集函数 myavg的拥有者改为joe：

```
ALTER AGGREGATE myavg(integer) OWNER TO joe;
```

把带有float8类型直接参数和integer 类型聚集参数的有序集聚集mypercentile 移动到 模式myschema中：

```
ALTER AGGREGATE mypercentile(float8 ORDER BY integer) SET SCHEMA myschema;
```

这也能行：

```
ALTER AGGREGATE mypercentile(float8, integer) SET SCHEMA myschema;
```

兼容性

在 SQL 标准中没有**ALTER AGGREGATE**语句。

另见

CREATE AGGREGATE(7), DROP AGGREGATE(7)

名称

ALTER COLLATION — 更改一个排序规则的定义

大纲

```
ALTER COLLATION name REFRESH VERSION

ALTER COLLATION name RENAME TO new_name
ALTER COLLATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER COLLATION name SET SCHEMA new_schema
```

描述

ALTER COLLATION更改一个排序规则的定义。

必须拥有要对其使用**ALTER COLLATION**的排序规则。要更改拥有者，必须是新的拥有角色的直接或者间接成员，并且该角色必须在排序规则的模式上具有CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该排序规则来做任何不能做的事情。不过，一个超级用户可以更改任何排序规则的所有权）。

参数

name

一个现有排序规则的名称（可以是模式限定的）。

new_name

排序规则的新名称。

new_owner

排序规则的新拥有者。

new_schema

排序规则的新模式。

REFRESH VERSION

更新排序规则的版本。 参阅下面的“注意”一节。

注意

使用ICU库提供的排序规则时，创建排序规则对象时，系统目录中会记录排序规则的特定ICU版本。 使用排序规则时，将根据记录的版本检查当前版本，并在发生不匹配时发出警告，例如：

```
WARNING:  collation "xx-x-icu" has version mismatch
DETAIL:  The collation in the database was created using version 1.2.3.4, but the operating system
 provides version 2.3.4.5.
```

HINT: Rebuild all objects affected by this collation and run ALTER COLLATION ux_catalog."xx-x-icu" REFRESH VERSION, or build UXDB with the right library version.

排序规则定义的更改会导致索引损坏和其他问题，因为数据库系统依赖于具有特定排序顺序的存储对象。 通常，应该避免这种情况，但它可以在合法的情况下发生，例如使用ux_uuxrade 升级到与更新版本的ICU链接的服务器二进制文件。发生这种情况时， 应该重建所有依赖于该排序规则的对象，例如，使用REINDEX。 完成后，使用命令ALTER COLLATION ... REFRESH VERSION可以刷新排序规则版本。 这将更新系统目录以记录当前的排序规则版本，并会使警告消失。请注意， 这实际上并不检查是否所有受影响的对象都已正确重建。

以下查询可用于识别当前数据库中需要刷新的所有排序规则以及依赖它们的对象：

```
SELECT ux_describe_object(refclassid, refobjid, refobjsubid) AS "Collation",
    ux_describe_object(classid, objid, objsubid) AS "Object"
 FROM ux_depend d JOIN ux_collation c
    ON refclassid = 'ux_collation'::regclass AND refobjid = c.oid
 WHERE c.collversion <> ux_collation_actual_version(c.oid)
 ORDER BY 1, 2;
```

例子

要把排序规则de_DE重命名为german：

```
ALTER COLLATION "de_DE" RENAME TO german;
```

要把排序规则en_US的拥有者改成joe：

```
ALTER COLLATION "en_US" OWNER TO joe;
```

兼容性

在 SQL 标准中没有ALTER COLLATION语句。

参见

CREATE COLLATION(7), DROP COLLATION(7)

名称

ALTER CONVERSION — 改变一个转换的定义

大纲

```
ALTER CONVERSION name RENAME TO new_name
ALTER CONVERSION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER CONVERSION name SET SCHEMA new_schema
```

描述

ALTER CONVERSION改变一个转换的定义。

必须拥有要对其使用**ALTER CONVERSION**的转换。要更改拥有者，必须是新的拥有角色的直接或者间接成员，并且该角色必须在转换的模式上具有CREATE特权（这些限制强制要求拥有者不能通过丢弃并重建该转换来做任何不能做的事情。不过，一个超级用户可以更改任何转换的所有权）。

参数

name

一个现有转换的名称（可以是模式限定的）。

new_name

转换的新名称。

new_owner

转换的新拥有者。

new_schema

转换的新模式。

例子

要把转换iso_8859_1_to_utf8重命名为latin1_to_unicode：

```
ALTER CONVERSION iso_8859_1_to_utf8 RENAME TO latin1_to_unicode;
```

要把转换iso_8859_1_to_utf8的拥有者改成joe：

```
ALTER CONVERSION iso_8859_1_to_utf8 OWNER TO joe;
```

兼容性

在 SQL 标准中没有**ALTER CONVERSION**语句。

参见

CREATE CONVERSION(7), DROP CONVERSION(7)

名称

ALTER DATABASE — 更改一个数据库

大纲

ALTER DATABASE *name* [[WITH] *option* [...]]

这里 *option* 可以是：

ALLOW_CONNECTIONS *allowconn*
CONNECTION LIMIT *connlimit*
IS_TEMPLATE *istemplate*

ALTER DATABASE *name* RENAME TO *new_name*

ALTER DATABASE *name* OWNER TO { *new_owner* | CURRENT_USER | SESSION_USER }

ALTER DATABASE *name* SET TABLESPACE *new_tablespace*

ALTER DATABASE *name* SET *configuration_parameter* { TO | = } { *value* | DEFAULT }
ALTER DATABASE *name* SET *configuration_parameter* FROM CURRENT
ALTER DATABASE *name* RESET *configuration_parameter*
ALTER DATABASE *name* RESET ALL

描述

ALTER DATABASE更改一个数据库的属性。

第一种形式更改某些针对每个数据库的设置（详见下文）。只有数据库拥有者或者超级用户可以更改这些设置。

第二种形式更改数据库的名称。只有数据库拥有者或者超级用户可以重命名一个数据库，非超级用户拥有者还必须拥有CREATEDB特权。当前数据库不能被重命名（如果需要这样做请连接到一个不同的数据库）。

第三种形式更改数据库的拥有者。要修改拥有者，必须拥有该数据库并且也是新拥有角色的一个直接或间接成员，并且必须具有CREATEDB特权（注意超级用户自动拥有所有这些特权）。

第四种形式更改数据库的默认表空间。只有数据库拥有者或超级用户能够这样做，还必须对新表空间具有创建特权。这个命令会在物理上移动位于该数据库旧的默认表空间中的任何表或索引到新的表空间中。新的默认表空间对于这个数据库必须是空的，并且不能有人可以连接到该数据库。在非默认表空间中的表和索引不受影响。

剩下的形式更改用于一个UXDB数据库的运行时配置变量的会话默认值。接下来只要一个新的会话在该数据库中开始，指定的值就会成为该会话的默认值。数据库相关的默认值会覆盖出现在uxsinodb.conf中或者从uxdb命令行接收到的设置。只有数据库拥有者或超级用户可以更改一个数据库的会话默认值。一些变量不能用这种方式设置或者只能由超级用户更改。

参数

name

要被修改属性的数据库名称。

allowconn

如果为假，则没有人能连接到这个数据库。

connlimit

与这个数据库可以建立多少个并发连接。-1 表示没有限制。

istemplate

如果为真，则任何具有CREATEDB特权的用户都可以从这个数据库进行克隆。如果为假，则只有超级用户或者这个数据库的拥有者可以克隆它。

new_name

数据库的新名称。

new_owner

数据库的新拥有者。

new_tablespace

数据库的新默认表空间。

这种形式的命令不能在事务块内执行。

configuration_parameter
value

将这个数据库的指定配置参数的会话默认值设置为给定值。如果*value*是DEFAULT，或者等效地使用了RESET，数据库相关的设置会被移除，因此系统范围的默认设置将会在新会话中继承。使用RESET ALL可清除所有数据库相关的设置。SET FROM CURRENT会保存该会话的当前参数值作为数据库相关的值。

更多关于允许的参数名称和值的信息可参考SET(7)。

注解

也可以把一个会话的默认值绑定到一个特定角色而不是一个数据库，见ALTER ROLE(7)。如果有冲突，角色相关的设置会覆盖数据库相关的值。

例子

要在数据库test中默认禁用索引扫描：

```
ALTER DATABASE test SET enable_indexscan TO off;
```

兼容性

ALTER DATABASE语句是一个UXDB扩展。

参见

CREATE DATABASE(7), DROP DATABASE(7), SET(7), CREATE TABLESPACE(7)

名称

ALTER DEFAULT PRIVILEGES — 定义默认访问特权

大纲

```
ALTER DEFAULT PRIVILEGES
  [ FOR { ROLE | USER } target_role [, ...] ]
  [ IN SCHEMA schema_name [, ...] ]
  abbreviated_grant_or_revoke
```

其中*abbreviated_grant_or_revoke*是下列之一：

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON TABLES
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
  [, ...] | ALL [ PRIVILEGES ] }
  ON SEQUENCES
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
  ON { FUNCTIONS | ROUTINES }
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
  ON TYPES
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | CREATE | ALL [ PRIVILEGES ] }
  ON SCHEMAS
  TO { [ GROUP ] role_name | PUBLIC } [, ...] [ WITH GRANT OPTION ]

REVOKE [ GRANT OPTION FOR ]
  { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
  [, ...] | ALL [ PRIVILEGES ] }
  ON TABLES
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
  { { USAGE | SELECT | UPDATE }
  [, ...] | ALL [ PRIVILEGES ] }
  ON SEQUENCES
  FROM { [ GROUP ] role_name | PUBLIC } [, ...]
  [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
  { EXECUTE | ALL [ PRIVILEGES ] }
  ON { FUNCTIONS | ROUTINES }
```

```
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPES
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | CREATE | ALL [ PRIVILEGES ] }
    ON SCHEMAS
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

描述

ALTER DEFAULT PRIVILEGES允许设置将被应用于未来要创建的对象的特权（它不会影响分配给已经存在的对象的特权）。当前，只能修改用于模式、表（包括视图和外部表）、序列、函数和类型（包括域）的特权。其中，可设置权限的函数包括聚集函数和过程函数。当这个命令应用于函数时，单词FUNCTIONS和ROUTINES是等效的。（推荐使用ROUTINES，因为它是用来囊括函数和过程的一个标准术语。在较早的UXDB发行版中，只允许单词FUNCTIONS。无法为函数或过程单独设置默认特权。）

只能改变自己或者属于其中的角色所创建的对象的默认特权。这些特权可以对全局范围设置（即对当前数据库中创建的所有对象），或者只对在指定模式中创建的对象设置。

如第 2.7 节 “权限”中所述，用于任何对象类型的默认特权通常会把所有可授予的权限授予给对象拥有者，并且也可能授予一些特权给PUBLIC。不过，这种行为可以通过使用ALTER DEFAULT PRIVILEGES修改全局默认特权来改变。

每个模式规定的默认权限将添加到特定对象类型的全局默认特权的内容中。 这意味着不能撤销每个模式的特权，如果它们已经被全局授予（默认情况下，或者根据以前未规定模式的ALTER DEFAULT PRIVILEGES命令）。 每个模式的REVOKE仅用于反转以前每个模式GRANT的影响。

参数

target_role

一个现有角色的名称，当前角色是它的一个成员。如果FOR ROLE被忽略，将假定为当前角色。

schema_name

一个现有模式的名称。如果被指定，以后在那个模式中创建的对象的默认特权会被修改。如果IN SCHEMA被忽略，全局默认特权会被修改。 当设置特权给模式时不能使用IN SCHEMA，因为模式不能嵌套。

role_name

要为其授予或者收回特权的一个现有角色的名称。这个参数以及所有*abbreviated_grant_or_revoke*中的其他参数会按照GRANT(7)或者REVOKE(7)中描述的方式运作，不过这里是为一整类的对象而不是特别指定的对象设置权限。

注解

使用psql的\ddp命令可以获得关于默认特权的现有分配信息。 特权显示的含义和第 2.7 节 “权限”中的\dp描述的相同。

如果希望删除一个默认特权被修改的角色，有必要撤销其默认特权上的改变或者使用DROP OWNED BY来为该角色去除默认特权项。

例子

为后续在模式myschema中创建的所有表（和视图）授予 SELECT 特权，并且也允许角色webuser向它们之中 INSERT 数据：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT SELECT ON TABLES TO
 PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema GRANT INSERT ON TABLES TO
 webuser;
```

撤销上面的动作，因此后续创建的表不会有任何不寻常的权限：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE SELECT ON TABLES FROM
 PUBLIC;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema REVOKE INSERT ON TABLES FROM
 webuser;
```

为后续由角色admin创建的所有函数移除通常在函数上会授予的公共 EXECUTE 权限：

```
ALTER DEFAULT PRIVILEGES FOR ROLE admin REVOKE EXECUTE ON FUNCTIONS FROM
 PUBLIC;
```

但是注意不能使用限制为单个模式的命令来实现该效果。 此命令不起作用，除非它撤消匹配的GRANT：

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public REVOKE EXECUTE ON FUNCTIONS FROM
 PUBLIC;
```

这是因为每个模式的默认特权只能向全局设置添加特权，而不能移除它授予的特权。

兼容性

在 SQL 标准中没有ALTER DEFAULT PRIVILEGES语句。

参见

GRANT(7), REVOKE(7)

名称

ALTER DOMAIN — 更改一个域的定义

大纲

```
ALTER DOMAIN name
  { SET DEFAULT expression | DROP DEFAULT }
ALTER DOMAIN name
  { SET | DROP } NOT NULL
ALTER DOMAIN name
  ADD domain_constraint [ NOT VALID ]
ALTER DOMAIN name
  DROP CONSTRAINT [ IF EXISTS ] constraint_name [ RESTRICT | CASCADE ]
ALTER DOMAIN name
   RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER DOMAIN name
  VALIDATE CONSTRAINT constraint_name
ALTER DOMAIN name
  OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER DOMAIN name
  RENAME TO new_name
ALTER DOMAIN name
  SET SCHEMA new_schema
```

描述

ALTER DOMAIN更改一个现有域的定义。有几种形式：

SET/DROP DEFAULT

这些形式设置或者移除一个域的默认值。注意默认值只会应用到后续的 **INSERT**命令，它们不影响使用该域的已经 存在于表中的行。

SET/DROP NOT NULL

这些形式更改一个域是被标记为允许 NULL 值还是拒绝 NULL 值。只有当使用该 域的列不包含空值时才能SET NOT NULL。

ADD *domain_constraint* [NOT VALID]

这种形式使用和CREATE DOMAIN(7)相同的语法为一个 域增加一个新的约束。当一个新的约束被增加到一个域时，所有使用该域的列 都会被根据新加的约束进行检查。可以通过增加使用 NOT VALID选项的新约束来抑制这类检查，而该约束则可 以在以后使用 **ALTER DOMAIN ... VALIDATE CONSTRAINT** 变为可用。新插入和更新的行总是会被根据所有约束进行检查（包括被标记为 NOT VALID的约束）。只有CHECK约束接受 NOT VALID。

DROP CONSTRAINT [IF EXISTS]

这种形式删除一个域上的约束。如果指定了IF EXISTS并且 约束不存在，不会抛出错误。在这种情况下会转而发出一个提示。

RENAME CONSTRAINT

这种形式更改一个域上的一个约束的名称。

VALIDATE CONSTRAINT

这种形式验证一个之前作为NOT VALID增加的约束，也就是说 它验证该域类型的表列中所有值满足指定的约束。

OWNER

这种形式更改域的拥有者为指定用户。

RENAME

这种形式更改域的名称。

SET SCHEMA

这种形式更改域的模式。任何与该域关联的约束也被会移动到新的模式中。

要使用ALTER DOMAIN，必须拥有该域。要更改一个域的模式， 还必须具有新模式上的CREATE特权。要更改拥有者，还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该域的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建域做不到的事情。不过，一个超级用户怎么都能更改任何域的所有权。）。

参数

name

要修改的一个现有域的名称（可能是模式限定的）。

domain_constraint

用于该域的新域约束。

constraint_name

要删除或重命名的一个现有约束的名称。

NOT VALID

不为约束的合法性验证现有的存储数据。

CASCADE

自动删除依赖于该约束的对象，并且接着删除依赖于那些对象的 所有对象（见第 2.14 节“依赖跟踪”）。

RESTRICT

如果有任何依赖对象则拒绝删除该约束。这是默认行为。

new_name

域的新名称。

new_constraint_name

约束的新名称。

new_owner

域的新拥有者的用户名。

new_schema

域的新模式。

注解

尽管ALTER DOMAIN ADD CONSTRAINT尝试验证现有存储的数据是否满足新约束，但此检查不是万无一失的，因为命令无法“see”新插入或更新但尚未提交的表行。 如果存在并发操作可能插入坏数据的危险，则处理方法是使用NOT VALID选项添加约束，提交该命令，等到所有事务在提交完成之前启动，然后发出ALTER DOMAIN VALIDATE CONSTRAINT以搜索违反约束的数据。 此方法是可靠的，因为一旦提交约束，所有新事务都保证针对域类型的新值强制执行约束。

当前，如果域或者任何衍生域被数据库中的任意表的一个容器类型 列（组合、数组、范围类型的列）使用，ALTER DOMAIN ADD CONSTRAINT、ALTER DOMAIN VALIDATE CONSTRAINT和 ALTER DOMAIN SET NOT NULL将会失败。这些命令最终将 会被改进成能够对这类嵌套值进行约束验证。

示例

要把一个NOT NULL约束加到一个域：

```
ALTER DOMAIN zipcode SET NOT NULL;
```

要从一个域中移除一个NOT NULL约束：

```
ALTER DOMAIN zipcode DROP NOT NULL;
```

要把一个检查约束增加到一个域：

```
ALTER DOMAIN zipcode ADD CONSTRAINT zipchk CHECK (char_length(VALUE) = 5);
```

要从一个域移除一个检查约束：

```
ALTER DOMAIN zipcode DROP CONSTRAINT zipchk;
```

要重命名一个域上的一个检查约束：

```
ALTER DOMAIN zipcode RENAME CONSTRAINT zipchk TO zip_check;
```

要把域移动到一个不同的模式：

```
ALTER DOMAIN zipcode SET SCHEMA customers;
```

兼容性

ALTER DOMAIN conforms to the SQL standard, except for the 除OWNER、RENAME、SET SCHEMA 以及VALIDATE CONSTRAINT变体之外（它们是 UXDB的扩展）， **ALTER DOMAIN**符合SQL标准。 ADD CONSTRAINT变体的NOT VALID子句也是一个 UXDB扩展。

另见

CREATE DOMAIN(7), DROP DOMAIN(7)

名称

ALTER EVENT TRIGGER — 更改一个事件触发器的定义

大纲

```
ALTER EVENT TRIGGER name DISABLE
ALTER EVENT TRIGGER name ENABLE [ REPLICA | ALWAYS ]
ALTER EVENT TRIGGER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER EVENT TRIGGER name RENAME TO new_name
```

描述

ALTER EVENT TRIGGER更改一个现有事件触发器 的属性。

必须作为一个超级用户才能修改一个时间触发器。

参数

name

要修改的现有触发器的名称。

new_owner

该事件触发器的新拥有者的用户名。

new_name

该事件触发器的新名称。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置事件触发器的触发。一个被禁用的触发器对系统来说仍然是可知的， 但是当期触发事件发生时却不会执行它。

兼容性

在 SQL 标准中没有**ALTER EVENT TRIGGER**语句。

另见

CREATE EVENT TRIGGER(7), DROP EVENT TRIGGER(7)

名称

ALTER EXTENSION — 更改一个扩展的定义

大纲

ALTER EXTENSION *name* UPDATE [TO *new_version*]
ALTER EXTENSION *name* SET SCHEMA *new_schema*
ALTER EXTENSION *name* ADD *member_object*
ALTER EXTENSION *name* DROP *member_object*

其中 *member_object* 是：

ACCESS METHOD *object_name* |
AGGREGATE *aggregate_name* (*aggregate_signature*) |
CAST (*source_type* AS *target_type*) |
COLLATION *object_name* |
CONVERSION *object_name* |
DOMAIN *object_name* |
EVENT TRIGGER *object_name* |
FOREIGN DATA WRAPPER *object_name* |
FOREIGN TABLE *object_name* |
FUNCTION *function_name* [([[*argmode*] [*argname*] *argtype* [, ...]])] |
MATERIALIZED VIEW *object_name* |
OPERATOR *operator_name* (*left_type*, *right_type*) |
OPERATOR CLASS *object_name* USING *index_method* |
OPERATOR FAMILY *object_name* USING *index_method* |
[PROCEDURAL] LANGUAGE *object_name* |
PROCEDURE *procedure_name* [([[*argmode*] [*argname*] *argtype* [, ...]])] |
ROUTINE *routine_name* [([[*argmode*] [*argname*] *argtype* [, ...]])] |
SCHEMA *object_name* |
SEQUENCE *object_name* |
SERVER *object_name* |
TABLE *object_name* |
TEXT SEARCH CONFIGURATION *object_name* |
TEXT SEARCH DICTIONARY *object_name* |
TEXT SEARCH PARSER *object_name* |
TEXT SEARCH TEMPLATE *object_name* |
TRANSFORM FOR *type_name* LANGUAGE *lang_name* |
TYPE *object_name* |
VIEW *object_name*

并且 *aggregate_signature* 是：

* |
[*argmode*] [*argname*] *argtype* [, ...] |
[[*argmode*] [*argname*] *argtype* [, ...]] ORDER BY [*argmode*] [*argname*] *argtype* [, ...]

描述

ALTER EXTENSION更改一个已安装扩展的定义。 有几种子形式：

UPDATE

这种形式把该扩展更新到一个新版本。该扩展必须提供一个适当的更新 脚本（或者一系列脚本）来把当前已安装的版本修改成所要求的版本。

SET SCHEMA

这种形式把该扩展的对象移动到另一个模式中。要使这个命令成功，该扩展 必须是可重定位的。

ADD *member_object*

这种形式把一个现有的对象加入到该扩展中。这主要对扩展更新脚本有用。 该对象后续将被当作该扩展的一个成员。尤其是该对象只有通过删除扩展 才能删除。

DROP *member_object*

这种形式从该扩展移除一个成员对象。这主要对扩展更新脚本有用。只有 撤销该对象与其扩展之间的关联后才能删除该对象。

要使用ALTER EXTENSION，必须拥有该扩展。 ADD/DROP形式还要求被增加/删除对象的所有权。

参数

name

一个已安装扩展的名称。

new_version

想要得到的该扩展的新版本。这可以写成一个标识符或者一个字符串。如果没有指定，ALTER EXTENSION UPDATE会尝试更新到该扩展的控制文件中的 默认版本。

new_schema

该扩展的新模式。

object_name
aggregate_name
function_name
operator_name
procedure_name
routine_name

要从该扩展增加或者移除的对象的名称。表、聚集、域、外部表、函数、 操作符、操作符类、操作符族、过程、例程、序列、文本搜索对象、类型和视图的名称 可以被模式限定。

source_type

该转换的源数据类型的名称。

target_type

该转换的目标数据类型的名称。

argmode

一个函数、过程或者聚集参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略，默认值是 IN。注意，ALTER EXTENSION 并不真正关心OUT参数，因为决定该函数的身份时只需要输入 参数。因此列出IN、INOUT和 VARIADIC参数足矣。

argname

一个函数、过程或者聚集参数的名称。注意， ALTER EXTENSION并不真正关心参数名称，因为 决定该函数的身份时只需要参数的数据类型。

argtype

一个函数、过程或者或聚集参数的数据类型。

left_type
right_type

该操作符参数的数据类型（可以用模式限定）。对一个前缀或后缀操作符的缺失的 参数可以写NONE。

PROCEDURAL

这是一个噪声词。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

示例

把hstore扩展更新到版本 2.0：

```
ALTER EXTENSION hstore UPDATE TO '2.0';
```

把hstore扩展的模式更改到utils：

```
ALTER EXTENSION hstore SET SCHEMA utils;
```

要向hstore扩展增加一个现有函数：

```
ALTER EXTENSION hstore ADD FUNCTION populate_record(anyelement, hstore);
```

兼容性

ALTER EXTENSION是一个UXDB 扩展。

另见

CREATE EXTENSION(7), DROP EXTENSION(7)

名称

ALTER FOREIGN DATA WRAPPER — 更改一个外部数据包装器的定义

大纲

```
ALTER FOREIGN DATA WRAPPER name
  [ HANDLER handler_function | NO HANDLER ]
  [ VALIDATOR validator_function | NO VALIDATOR ]
  [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ]) ]
ALTER FOREIGN DATA WRAPPER name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER FOREIGN DATA WRAPPER name RENAME TO new_name
```

描述

ALTER FOREIGN DATA WRAPPER更改一个 外部数据包装器的定义。该命令的第一种形式用于更改外部数据包装器的 支持函数或者一般选项（至少要求一个子句）。第二种形式更改外部数据包 装器的拥有者。

只有超级用户能修改外部数据包装器。此外，只有超级用户能够拥有外部数 据包装器。

参数

name

一个已有的外部数据包装器的名称。

HANDLER *handler_function*

为外部数据包装器指定一个新的处理器函数。

NO HANDLER

用于指定该外部数据包装器不再具有一个处理器函数。

注意使用没有处理器的外部数据包装器的外部表不能被访问。

VALIDATOR *validator_function*

为外部数据包装器指定一个新的验证器函数。

注意，新的验证器可能会认为该外部数据包装器或者依赖于它的独立服务器 的已有选项、用户映射、外部表无效。UXDB 不会做这种检查。在使用修改过的外部数据包装器之前确认这些选项正确是 用户的责任。不过，在这个**ALTER FOREIGN DATA WRAPPER**命令中指定的选项将会被使用新的验证器检查。

NO VALIDATOR

用于指定该外部数据包装器不再拥有一个验证器函数。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

为该外部数据包装器更改选项。ADD、SET 以及DROP指定要被执行的动作。如果没有显式地指定操作， 将假定为ADD。选项名称必须唯一，选项名称和值（如果有） 也会使用该外部数据包装器的验证器函数来验证。

new_owner

该外部数据包装器的新拥有者的用户名。

new_name

该外部数据包装器的新名称。

示例

更改一个外部数据包装器dbi，增加选项 foo并删除bar：

```
ALTER FOREIGN DATA WRAPPER dbi OPTIONS (ADD foo '1', DROP 'bar');
```

把外部数据包装器dbi的验证器改为 bob.myvalidator：

```
ALTER FOREIGN DATA WRAPPER dbi VALIDATOR bob.myvalidator;
```

兼容性

ALTER FOREIGN DATA WRAPPER符合 ISO/IEC 9075-9 (SQL/MED)。不过HANDLER、VALIDATOR、OWNER TO 以及RENAME子句是扩展。

另见

CREATE FOREIGN DATA WRAPPER(7), DROP FOREIGN DATA WRAPPER(7)

名称

ALTER FOREIGN TABLE — 更改一个外部表的定义

大纲

```
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    action [, ... ]
ALTER FOREIGN TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    RENAME TO new_name
ALTER FOREIGN TABLE [ IF EXISTS ] name
    SET SCHEMA new_schema
```

其中 *action* 是以下之一：

```
    ADD [ COLUMN ] column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
    ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
    ALTER [ COLUMN ] column_name SET DEFAULT expression
    ALTER [ COLUMN ] column_name DROP DEFAULT
    ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
 MAIN }
    ALTER [ COLUMN ] column_name OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
    ADD table_constraint [ NOT VALID ]
    VALIDATE CONSTRAINT constraint_name
    DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    SET WITHOUT OIDS
    INHERIT parent_table
    NO INHERIT parent_table
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ])
```

描述

ALTER FOREIGN TABLE更改一个现有外部表的定义。 有几种子形式：

ADD COLUMN

这种形式使用与CREATE FOREIGN TABLE(7)相同的语法把 一个新的列增加到该外部表。和为常规表增加一列不同，这种形式并不影响底层 的存储：这个动作只是简单地声明通过该外部表可以访问某个新的列而已。

DROP COLUMN [IF EXISTS]

这种形式从一个外部表删掉一列。如果在该表外部有任何东西依赖于该列， 将需要写上CASCADE，典型的例子就是视图。如果指定了 IF EXISTS并且该列不存在，将不会抛出错误。在这种 情况下会转而发出一个提示。

SET DATA TYPE

这种形式更改一个外部表的一列的类型。同样，这种形式并不影响底层 的存储：这个动作只是简单地更改UXDB 相信该列所具有的类型。

SET/DROP DEFAULT

这些形式设置或者移除一列的默认值。默认值只会应用于后续的 INSERT或UPDATE命令，它们 不会导致已经在表中的行被更改。

SET/DROP NOT NULL

把一列标记为允许或者不允许空值。

SET STATISTICS

这种形式为后续的ANALYZE(7)操作设置针对每列 的统计收集目标。详见ALTER TABLE(7)的类似形式。

SET (*attribute_option* = *value* [, ...])
RESET (*attribute_option* [, ...])

这种形式设置或重置针对每个属性的选项。详见ALTER TABLE(7) 的类似形式。

SET STORAGE

这种形式设置一个列的存储模式。详见 ALTER TABLE(7)中类似的模式。注意存储模式 不会产生效果，除非该表的外部数据包装器选择处理它。

ADD *table_constraint* [NOT VALID]

这种形式为外部表增加一个新的约束，使用的语法和 CREATE FOREIGN TABLE(7)中相同。当前只 支持CHECK约束。

和向常规表增加约束的情况不同，为外部表增加约束时不会做任何事情来 验证该约束是否正确。这个动作只是简单地声明了该外部表中所有的行都 应该满足的某种新的条件（见CREATE FOREIGN TABLE(7)中的讨论）。如果该 约束被标记为NOT VALID，那么它不被假设为有效，而只是 被记录下来以备未来使用。

VALIDATE CONSTRAINT

这种形式把一个之前被标记为NOT VALID的约束标 记为有效。不会做任何动作来验证该约束，但是未来的查询将会假定该 约束是保持的。

DROP CONSTRAINT [IF EXISTS]

这种形式删掉在一个外部表上的指定约束。如果指定了 IF EXISTS但约束并不存在，则不会抛出错误。 在这种情况下会发出一个提示。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置属于该外部表的触发器的触发情况。详见 ALTER TABLE(7)的类似形式。

SET WITHOUT OIDS

向后兼容性语法，表示移除oid系统列。 由于不能再为外部表添加oid系统列，此语句不再产生效用。

INHERIT *parent_table*

这种形式把目标外部表作为指定的父表的新后代。详见 ALTER TABLE(7)的类似的形式。

NO INHERIT *parent_table*

这种形式把目标外部表从指定的父表的子女列表中移除。

OWNER

这种形式把该外部表的拥有者改成指定的用户。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

更改该外部表或者其中一个列的选项。ADD、SET 以及DROP指定要执行的动作。如果没有显式地指定操作，将假定 为ADD。不允许重复的名称（不过一个表选项和一个列选项可以重 名）。选项名称和值也会用外部数据包装器库来验证。

RENAME

RENAME形式更改外部表的名称或者外 部表中一个列的名称。

SET SCHEMA

这种形式把外部表移动到另一个模式中。

所有除了RENAME和SET SCHEMA的 动作都能被整合到一个多修改列表以便能被并行应用。例如，可以在一个 命令中增加几个列并且/或者修改几个列的类型。

如果该命令被写作ALTER FOREIGN TABLE IF EXISTS ...并且 该外部表不存在，则不会抛出错误。这种情况下会发出一个提示。

必须拥有该表以使用ALTER FOREIGN TABLE。要更改一个 外部表的模式，必须还拥有新模式上的CREATE特权。要 更改拥有者，还必须是新拥有角色的一个直接或者间接成员，并且该角色必须 具有在该表的模式上的CREATE特权（这些限制强制修改拥有 者不能做一些通过删除和重建该表做不到的事情。不过，一个超级用户怎么都能 更改任何表的所有权）。要增加一列或者修改一个列的类型，还必须具有该数 据类型上的USAGE特权。

参数

name

一个要修改的现有外部表的名称（可以被模式限定）。如果在表名前指定了 ONLY，则只有该表被修改。如果没有指定ONLY， 该表和它所有的后代表（如果有）都会被修改。可选地，在表名后面指定 *可以显式地表示将后代表包括在内。

column_name

一个新的或者现有列的名称。

new_column_name

一个现有列的新名称。

new_name

该表的新名称。

data_type

新列的数据类型或者一个现有列的新数据类型。

table_constraint

New table constraint for the foreign table.

constraint_name

Name of an existing constraint to drop.

CASCADE

自动删除依赖于被删除列或约束的对象（例如，引用该列的视图），　并且接着删除依赖于那些对象的 所有对象（见第　2.14　节　“依赖跟踪”。

RESTRICT

如果有任何依赖对象就拒绝删除该列或约束。这是默认行为。

trigger_name

要禁用或启用的一个触发器的名称。

ALL

禁用或者启用所有属于该外部表的触发器（如果任何触发器是内部生成　的触发器，这都要求超级用户特权。核心系统不会向外部表增加这类触发 器，但是附加代码会这样做。）。

USER

禁用或者启用属于该外部表的除了内部生成的触发器之外的所有触发器。

parent_table

要与这个外部表关联或者解除关联的父表。

new_owner

该表的新拥有者的用户名。

new_schema

该表要被移动到其中的模式的名称。

注解

关键词COLUMN是噪声词并且可以被忽略。

当使用ADD COLUMN或 DROP COLUMN增加或移除一列、增加一个NOT NULL 或者CHECK约束或者用SET DATA TYPE更改一个列类型时，不会检查与外部服务器的一 致性。确保该表定义匹配远端是用户的责任。

关于有效参数的进一步描述可参考CREATE FOREIGN TABLE(7)。

示例

要把一列标记为非空：

```
ALTER FOREIGN TABLE distributors ALTER COLUMN street SET NOT NULL;
```

要更改一个外部表的选项：

```
ALTER FOREIGN TABLE myschema.distributors OPTIONS (ADD opt1 'value', SET opt2 'value2',
 DROP opt3 'value3');
```

兼容性

形式ADD、DROP以及 SET DATA TYPE符合 SQL 标准。其他形式是 SQL 标准的 UXDB扩展。在一个 ALTER FOREIGN TABLE命令中指定多于一个操作也是一种扩展。

ALTER FOREIGN TABLE DROP COLUMN可以被用来删除 一个外部表的唯一一列，从而留下一个没有列的表。这是一种 SQL 的扩展，它 允许没有列的外部表。

另见

CREATE FOREIGN TABLE(7), DROP FOREIGN TABLE(7)

名称

ALTER FUNCTION — 更改一个函数的定义

大纲

```
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER FUNCTION name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中 *action* 是以下之一：

```
    CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SUPPORT support_function
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

描述

ALTER FUNCTION更改一个函数的定义。

必须拥有该函数以使用ALTER FUNCTION。要更改一个函数 的模式，还必须具有新模式上的CREATE特权。要 更改拥有者，还必须是新拥有角色的一个直接或者间接成员，并且该角色必须 具有在该函数的模式上的CREATE特权（这些限制强制修改拥有 者不能做一些通过删除和重建该函数做不到的事情。不过，一个超级用户怎么都能 更改任何函数的所有权）。

参数

name

一个现有函数的名称（可以被模式限定）。如果没有指定参数列表， 则该名称必须在它的模式中唯一。

argmode

一个参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略，默认为 IN。注意ALTER FUNCTION 并不真正关心OUT参数，因为在决定函数的身份时只需要输 入参数。因此列出IN、INOUT以及 VARIADIC参数即可。

argname

一个参数的名称。注意**ALTER FUNCTION** 并不真正参数名称，因为在确定函数的身份时只需要参数的数据类型即可。

argtype

该函数的参数（如果有）的数据类型（可以被模式限定）。

new_name

该函数的新名称。

new_owner

该函数的新拥有者。注意如果该函数被标记为 SECURITY DEFINER，它的后续执行将会使用新拥有者。

new_schema

该函数的新模式。

extension_name

该函数所以来的扩展名。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT将该函数改为在某些 或者全部参数为空值时可以被调用。RETURNS NULL ON NULL INPUT或者 STRICT将该函数改为只要任一参数为空值就不被调用而 是自动假定一个空值结果。详见CREATE FUNCTION(7)。

IMMUTABLE
STABLE
VOLATILE

把该函数的稳定性更改为指定的设置。详见 CREATE FUNCTION(7)。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

更改该函数是否为一个安全性定义者。关键词EXTERNAL 是为了符合 SQL，它会被忽略。关于这项能力的详情请见 CREATE FUNCTION(7)。

PARALLEL

决定该函数对于并行是否安全。详见 CREATE FUNCTION(7)。

LEAKPROOF

更改该函数是否被认为是防泄漏的。关于这项能力的详情请见 CREATE FUNCTION(7)。

COST *execution_cost*

更改该函数的估计执行代价。详见CREATE FUNCTION(7)。

ROWS *result_rows*

更改一个集合返回函数的估计行数。详见 CREATE FUNCTION(7)。

SUPPORT *support_function*

设置或更改计划器支持函数以使用这个函数。必须是超级用户才可以使用此选项。

此选项不能用于完全删除支持功能，因为它必须命名新的支持函数。 如果需要这样做，可以使用CREATE OR REPLACE FUNCTION。

configuration_parameter
value

当该函数被调用时，要对一个配置参数做出增加或者更改的赋值。如果 *value*是DEFAULT或者使用等价的RESET，该函数本地的设置将会被 移除，这样该函数会使用其环境中存在的值执行。使用RESET ALL可以清除所有函数本地的设置。 SET FROM CURRENT把ALTER FUNCTION 执行时该参数的当前值保存为进入 该函数时要应用的值。

有关允许的参数名称和值可详见SET(7)。

RESTRICT

为了符合 SQL 标准存在，被忽略。

示例

要把用于类型integer的函数sqrt 重命名为square_root：

```
ALTER FUNCTION sqrt(integer) RENAME TO square_root;
```

要把用于类型integer的函数sqrt 的拥有者改为joe：

```
ALTER FUNCTION sqrt(integer) OWNER TO joe;
```

要把用于类型integer的函数sqrt 的模式改为maths：

```
ALTER FUNCTION sqrt(integer) SET SCHEMA maths;
```

要把类型integer的函数sqrt 标记为依赖于扩展mathlib：

```
ALTER FUNCTION sqrt(integer) DEPENDS ON EXTENSION mathlib;
```

要调整一个函数的自动搜索路径：

```
ALTER FUNCTION check_password(text) SET search_path = admin, ux_temp;
```

要禁止一个函数的search_path的自动设置：

```
ALTER FUNCTION check_password(text) RESET search_path;
```

这个函数将用其调用者使用的搜索路径来执行。

兼容性

这个语句部分兼容 SQL 标准中的**ALTER FUNCTION**语句。该标准允许修改一个函数的更多属性，但是不提供 重命名一个函数、标记一个函数为安全性定义者、为一个函数附加配置参 数值或者更改一个函数的拥有者、模式或者稳定性等功能。该标准还要求 RESTRICT关键字，它在UXDB中是可选的。

另见

CREATE FUNCTION(7), DROP FUNCTION(7), ALTER PROCEDURE(7), ALTER ROUTINE(7)

名称

ALTER GROUP — 更改角色名称或者成员关系

大纲

ALTER GROUP *role_specification* ADD USER *user_name* [, ...]
ALTER GROUP *role_specification* DROP USER *user_name* [, ...]

其中 *role_specification* 可以是：

role_name
| CURRENT_USER
| SESSION_USER

ALTER GROUP *group_name* RENAME TO *new_name*

描述

ALTER GROUP更改一个用户组的属性。 这是一个被废弃的命令，不过为了向后兼容还是会被接受，因为组（以及用户） 已经被更一般的角色概念替代了。

前两个变体向一个组增加用户或者从一个组中移除用户（为了这个目的， 任何角色都可以扮演“用户”或者“组”）。这些变体 实际上等效于在被称为“组”的角色中授予或者回收成员关系， 因此最好的方法是使用GRANT(7)或者 REVOKE(7)。

第三种变体会更改该组的名称。这恰好等效于用ALTER ROLE(7) 重命名该角色。

参数

group_name

要修改的组（角色）的名称。

user_name

要被加入到该组或者从该组移除的用户（角色）。这些用户必须已经存在， **ALTER GROUP**不会创建或者删除用户。

new_name

该组的新名称。

示例

向一个组增加用户：

ALTER GROUP staff ADD USER karl, john;

从一个组移除一个用户：

```
ALTER GROUP workers DROP USER beth;
```

兼容性

在 SQL 标准中没有**ALTER GROUP**语句。

另见

GRANT(7)， REVOKE(7)， ALTER ROLE(7)

名称

ALTER INDEX — 更改一个索引的定义

大纲

```
ALTER INDEX [ IF EXISTS ] name RENAME TO new_name
ALTER INDEX [ IF EXISTS ] name SET TABLESPACE tablespace_name
ALTER INDEX name ATTACH PARTITION index_name
ALTER INDEX name DEPENDS ON EXTENSION extension_name
ALTER INDEX [ IF EXISTS ] name SET ( storage_parameter = value [, ... ] )
ALTER INDEX [ IF EXISTS ] name RESET ( storage_parameter [, ... ] )
ALTER INDEX [ IF EXISTS ] name ALTER [ COLUMN ] column_number
  SET STATISTICS integer
ALTER INDEX ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
  SET TABLESPACE new_tablespace [ NOWAIT ]
```

描述

ALTER INDEX更改一个现有索引的定义。下面描述了几种子窗体。 注意每个子窗体所需的锁级别可能不同。除非显式说明，ACCESS EXCLUSIVE锁被持有。 列出多个子命令时，锁的持有将是任何子命令所需的最严格的子命令。

RENAME

RENAME形式更改该索引的名称。如果索引与一个表约束（UNIQUE、PRIMARY KEY或者EXCLUDE）关联，该约束也会被重命名。这对已存储的数据没有影响。

重命名索引取得一个 SHARE UPDATE EXCLUSIVE锁。

SET TABLESPACE

这种形式更改该索引的表空间为指定的表空间，并且把与该索引相关联的数据文件 移动到新的表空间中。要更改一个索引的表空间，必须拥有该索引并且具有新表 空间上的CREATE特权。可以使用 ALL IN TABLESPACE形式把当前数据库中在一个表空间内的 所有索引全部移动到另一个表空间中，这将会锁定所有要被移动的索引然后挨个移 动它们。这种形式也支持OWNED BY，即只移动属于指定角色 的索引。如果指定了NOWAIT选项，那么当该命令无法立刻获 得所有锁时将会失败。注意这个命令不会移动系统目录，如果想要移动系统目录， 应使用**ALTER DATABASE**或者显式的 **ALTER INDEX**调用。另见 CREATE TABLESPACE(7)。

ATTACH PARTITION

导致提到的索引变成附着于被修改的索引。提及的索引必须在包含被修改索引的表的一个分区上，并且具有一种等效的定义。一个附着索引不能被单独删除，它会在其父索引被删除时自动连带删除。

DEPENDS ON EXTENSION

这种形式把该索引标记为依赖于扩展，这样如果该扩展被删除，该索引也将被 自动删除。

SET (*storage_parameter* = *value* [, ...])

这种形式为该索引更改一个或者多个索引方法相关的存储参数。可用的参数详见 CREATE INDEX(7)。注意这个命令不会立刻修改索引内容， 根据参数可能需要用REINDEX(7)重建索引来得到想要的 效果。

RESET (*storage_parameter* [, ...])

这种形式把一个或者多个索引方法相关的存储参数重置为其默认值。正如 SET一样，可能需要一次REINDEX来完全更新 该索引。

ALTER [COLUMN] *column_number* SET STATISTICS *integer*

这种形式为后续的ANALYZE(7)操作设置针对每个列的统计信息收集目标，不过只能用在被定义为表达式的索引列上。由于表达式缺少唯一的名称，我们通过该索引列的顺序号来引用它们。收集目标可以被设置为范围0到10000之间的值。另外，把它设置为-1会恢复到使用系统的默认统计信息目标。更多有关UXDB查询规划器使用统计信息的内容，请参考第 11.2 节“规划器使用的统计信息”。

参数

IF EXISTS

如果该索引不存在不要抛出错误。这种情况下将发出一个提示。

column_number

引用该索引列的顺序（从左往右）位置的顺序号。

name

要更改的一个现有索引的名称（可能被模式限定）。

new_name

该索引的新名称。

tablespace_name

该索引将被移动到的表空间。

extension_name

该索引所依赖的扩展的名称。

storage_parameter

一个索引方法相关的存储参数的名称。

value

一个索引方法相关的存储参数的新值。根据该参数，这可能是一个数字或者一个 词。

注解

也可以用ALTER TABLE(7)来做这些操作。实际上， ALTER INDEX只是ALTER TABLE应用在索引 上的形式的别名而已。

以前有一种**ALTER INDEX OWNER**变体，但现在已被忽略（会出现 一个警告）。一个索引的拥有者不能与其基表的拥有者不同。更改基表的拥有者 会自动地更改索引的拥有者。

不允许更改系统目录索引的任何部分。

示例

要重命名一个现有索引：

```
ALTER INDEX distributors RENAME TO suppliers;
```

把一个索引移动到一个不同的表空间：

```
ALTER INDEX distributors SET TABLESPACE fasttablespace;
```

更改一个索引的填充因子（假设该索引方法支持填充因子）：

```
ALTER INDEX distributors SET (fillfactor = 75);
REINDEX INDEX distributors;
```

为一个表达式索引设置统计信息收集目标：

```
CREATE INDEX coord_idx ON measured (x, y, (z + t));
ALTER INDEX coord_idx ALTER COLUMN 3 SET STATISTICS 1000;
```

兼容性

ALTER INDEX是一种 UXDB扩展。

另见

CREATE INDEX(7), REINDEX(7)

名称

ALTER LANGUAGE — 更改一种过程语言的定义

大纲

```
ALTER [ PROCEDURAL ] LANGUAGE name RENAME TO new_name
ALTER [ PROCEDURAL ] LANGUAGE name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
```

描述

ALTER LANGUAGE更改一种过程语言的定义。唯一 的功能是重命名该语言或者为它赋予一个新的拥有者。要使用 **ALTER LANGUAGE**，必须是一个超级用户或者该 语言的拥有者。

参数

name

语言的名称

new_name

该语言的新名称

new_owner

该语言的新拥有者

兼容性

在 SQL 标准中没有**ALTER LANGUAGE**语句。

另见

CREATE LANGUAGE(7), DROP LANGUAGE(7)

名称

ALTER LARGE OBJECT — 更改一个大对象的定义

大纲

```
ALTER LARGE OBJECT large_object_oid OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
```

描述

ALTER LARGE OBJECT更改一个大对象的定义。

您必须拥有大对象才能使用**ALTER LARGE OBJECT**。要更改所有者，您还必须是新所有者的直接或间接成员。（不过，超级用户仍然可以更改任何大对象。）当前，唯一的功能是分配新所有者，因此两者的约束都始终适用。

参数

large_object_oid

要被修改的大对象的 OID

new_owner

该大对象的新拥有者

兼容性

在 SQL 标准中没有**ALTER LARGE OBJECT** 语句。

名称

ALTER MATERIALIZED VIEW — 更改一个物化视图的定义

大纲

```
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    action [, ... ]
ALTER MATERIALIZED VIEW name
    DEPENDS ON EXTENSION extension_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME [ COLUMN ] column_name TO new_column_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    RENAME TO new_name
ALTER MATERIALIZED VIEW [ IF EXISTS ] name
    SET SCHEMA new_schema
ALTER MATERIALIZED VIEW ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
    SET TABLESPACE new_tablespace [ NOWAIT ]
```

其中 *action*是下列之一：

```
    ALTER [ COLUMN ] column_name SET STATISTICS integer
    ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
    ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
    ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
 MAIN }
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

ALTER MATERIALIZED VIEW更改一个现有物化视图的 多个辅助属性。

要使用**ALTER MATERIALIZED VIEW**，必须拥有该物化视图。要 更改一个物化视图的模式，还必须拥有新模式上的CREATE特权。要更 改拥有者，还必须是新拥有角色的一个直接或者间接成员，并且该角色必须拥有该 物化视图所在模式上的CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重建该物化视图做不到的事情。不过，一个超级用户怎么都能更改 任何视图的所有权。）。

DEPENDS ON EXTENSION形式把该物化视图标记为依赖于一个 扩展，这样该扩展被删除时会自动地删除掉这个物化视图。

可用于**ALTER MATERIALIZED VIEW**的语句形式和动作是 **ALTER TABLE**的一个子集，并且在用于物化视图时具有相 同的含义。详见ALTER TABLE(7)的描述。

参数

name

一个现有物化视图的名称（可以是模式限定的）。

column_name

一个新的或者现有的列的名称。

extension_name

该物化视图所依赖的扩展的名称。

new_column_name

一个现有列的新名称。

new_owner

该物化视图的新拥有者的用户名。

new_name

该物化视图的新名称。

new_schema

该物化视图的新模式。

示例

把物化视图foo重命名为 bar：

```
ALTER MATERIALIZED VIEW foo RENAME TO bar;
```

兼容性

ALTER MATERIALIZED VIEW是一种 UXDB扩展。

另见

CREATE MATERIALIZED VIEW(7), DROP MATERIALIZED VIEW(7), REFRESH MATERIALIZED VIEW(7)

名称

ALTER OPERATOR — 更改一个操作符的定义

大纲

```
ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
  OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
  SET SCHEMA new_schema

ALTER OPERATOR name ( { left_type | NONE } , { right_type | NONE } )
  SET ( {  RESTRICT = { res_proc | NONE }
         | JOIN = { join_proc | NONE }
       } [, ... ] )
```

描述

ALTER OPERATOR更改一个操作符的定义。

要使用**ALTER OPERATOR**，必须拥有该操作符。要更改拥有者， 还必须是新拥有角色的一个直接或者间接成员，并且该角色必须具有该操作符所在 模式上的CREATE特权（这些限制强制修改拥有者不能做一些通过 删除和重建操作符做不到的事情。不过，一个超级用户怎么都能更改任何操作符的所 有权。）。

参数

name

一个现有操作符的名称（可以是模式限定的）。

left_type

该操作符左操作数的数据类型，如果该操作符没有左操作数可以写成 NONE。

right_type

该操作符右操作数的数据类型，如果该操作符没有右操作数可以写成 NONE。

new_owner

该操作符的新拥有者。

new_schema

该操作符的新模式。

res_proc

这个操作符的约束选择度估算器函数，写成 NONE 可以移除现有的选择度估算器。

join_proc

这个操作符的连接选择度估算器函数，写成 NONE 可以移除现有的选择度估算器。

示例

更改类型text的一个自定义操作符a @@ b 的拥有者：

```
ALTER OPERATOR @@ (text, text) OWNER TO joe;
```

更改类型int[]的自定义操作符a && b的 约束和连接选择度估算器函数：

```
ALTER OPERATOR && (_int4, _int4) SET (RESTRICT = _int_contsel, JOIN = _int_contjoinsel);
```

兼容性

在 SQL 标准中没有**ALTER OPERATOR**语句。

另见

CREATE OPERATOR(7), DROP OPERATOR(7)

名称

ALTER OPERATOR CLASS — 更改一个操作符类的定义

大纲

```
ALTER OPERATOR CLASS name USING index_method
  RENAME TO new_name

ALTER OPERATOR CLASS name USING index_method
  OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR CLASS name USING index_method
  SET SCHEMA new_schema
```

描述

ALTER OPERATOR CLASS更改一个操作符类的定义。

要使用**ALTER OPERATOR CLASS**，必须拥有该操作符类。要 修改拥有者，还必须是新拥有角色的一个直接或间接成员，并且该角色必须具有 该操作符类所在模式上的CREATE特权（这些限制强制修改拥有 者不能做一些通过删除和重建操作符类做不到的事情。不过，一个超级用户怎么都 能更改任何操作符类的所有权。）。

参数

name

一个现有操作符类的名称（可以是模式限定的）。

index_method

这个操作符类所服务的索引方法的名称。

new_name

该操作符类的新名称。

new_owner

该操作符类的新拥有者。

new_schema

该操作符类的新模式。

兼容性

在 SQL 标准中没有**ALTER OPERATOR CLASS**语句。

另见

CREATE OPERATOR CLASS(7), DROP OPERATOR CLASS(7), ALTER OPERATOR FAMILY(7)

名称

ALTER OPERATOR FAMILY — 更改一个操作符族的定义

大纲

```
ALTER OPERATOR FAMILY name USING index_method ADD
  {  OPERATOR strategy_number operator_name ( op_type, op_type )
           [ FOR SEARCH | FOR ORDER BY sort_family_name ]
  | FUNCTION support_number [ ( op_type [ , op_type ] ) ]
           function_name [ ( argument_type [, ...] ) ]
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method DROP
  {  OPERATOR strategy_number ( op_type [ , op_type ] )
  | FUNCTION support_number ( op_type [ , op_type ] )
  } [, ... ]

ALTER OPERATOR FAMILY name USING index_method
    RENAME TO new_name

ALTER OPERATOR FAMILY name USING index_method
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }

ALTER OPERATOR FAMILY name USING index_method
    SET SCHEMA new_schema
```

描述

ALTER OPERATOR FAMILY更改一个操作符族 的定义。能增加操作符以及支持函数到该家族、从该族中移除它们或者更改 该族的名称或者拥有者。

在用**ALTER OPERATOR FAMILY**增加操作符和 支持函数到一个族中时，它们不是族内任何特定操作符类的组成部分，而只是 “松散”地存在于该族中。这表示这些操作符和函数与该族的语义兼 容，但是没有被任何特定索引的正确功能所要求（所要求的操作符和函数应该 被作为一个操作符类的一部分声明，见CREATE OPERATOR CLASS(7)）。 UXDB将允许一个族的松散成员在 任何时候被从该族中删除，但是在删除一个操作符类的成员之前，必须已经删 除整个类以及依赖于该成员的索引。具有代表性的是，单一数据类型操作符和 函数是操作符类的一部分，因为在特定数据类型上的索引需要它们的支持。而 多数据类型操作符和函数则被作为该族的松散成员。

要使用**ALTER OPERATOR FAMILY**，必须是超级用户（ 做这样的限制是因为一个错误的操作符族定义可能会迷惑服务器甚至让它崩溃）。

ALTER OPERATOR FAMILY目前不检测操作符族 定义是否包括该索引方法所要求的所有操作符和函数，也不检查操作符和函数是 否形成了一个有理的集合。定义一个合法的操作符族是用户的责任。

参数

name

一个现有操作符族的名称（可以是模式限定的）。

index_method

这个操作符族所应用的索引方法的名称。

strategy_number

与该操作符族相关的一个操作符的索引方法策略号。

operator_name

与该操作符族相关的一个操作符的名称（可以是模式限定的）。

op_type

在一个OPERATOR子句中指定该操作符的操作数数据类型， 或者用NONE来表示一个左一元或者右一元操作符。不同于 CREATE OPERATOR CLASS中类似的语法，操作数数据 类型总是必须被指定。

在一个ADD FUNCTION子句中指定该函数意图支持的操作数 数据类型（如果不同于该函数的输入数据类型）。对于 B-树比较函数和哈希 函数，有必要指定*op_type*，因为该函数的输入数据类型 总是正确的。对于 B-树排序支持函数和 GiST、SP-GiST 和 GIN 操作符类中 的所有函数，有必要指定该函数要使用的操作数数据类型。

在一个DROP FUNCTION子句中，必须指定该函数要支持的操 作数数据类型。

sort_family_name

一个现有btree操作符族的名称（可能是模式限定的）， 它描述与一个排序操作符相关的排序顺序。

如果既没有指定FOR SEARCH也没有指定FOR ORDER BY， 默认值是FOR SEARCH。

support_number

一个与该操作符族相关的函数的索引方法支持过程编号。

function_name

作为该操作符族的一种索引方法支持函数的函数名称（可以是模式限定的）。 如果没有指定参数列表，则该名称必须在其模式中唯一。

argument_type

该函数的参数数据类型。

new_name

该操作符族的新名称。

new_owner

该操作符族的新拥有者。

new_schema

该操作符族的新模式。

OPERATOR和FUNCTION子句可以以任何顺序出现。

注解

注意DROP语法只通过策略或者支持号以及输入数据类型指定该 操作符族中的“slot”。占用这个槽的操作符或函数的名称不会被提及。 还有，对于DROP FUNCTION，要指定的类型是该函数意图支持 的输入数据类型。对于 GiST、SP-GiST 以及 GIN 索引，可能无需对该函数的 实际输入参数类型做任何事情。

因为索引机制在使用函数之前不会检查其上的访问权限，包括一个操作符族中的 函数或操作符都等同于授予了其上的公共执行权限。这对于操作符族中很有用的 这类函数来说，这通常不成问题。

操作符应该由 SQL 函数定义。一个 SQL 函数很可能被内联到调用查询中，这将 阻止优化器识别出该查询匹配一个索引。

OPERATOR子句可以包括一个RECHECK选项。这不再被支持，因为一个索引操作符是否为“lossy”现在会在运行时即时决定。这允许高效地处理一个操作符可能或者不可能为有损的情况。

示例

下列示例命令为一个操作符族增加跨数据类型的操作符和支持函数，该操 作符族已经包含用于数据类型int4以及int2的 B-树 操作符类。

```
ALTER OPERATOR FAMILY integer_ops USING btree ADD

  -- int4 vs int2
  OPERATOR 1 < (int4, int2) ,
  OPERATOR 2 <= (int4, int2) ,
  OPERATOR 3 = (int4, int2) ,
  OPERATOR 4 >= (int4, int2) ,
  OPERATOR 5 > (int4, int2) ,
  FUNCTION 1 btint42cmp(int4, int2) ,

  -- int2 vs int4
  OPERATOR 1 < (int2, int4) ,
  OPERATOR 2 <= (int2, int4) ,
  OPERATOR 3 = (int2, int4) ,
  OPERATOR 4 >= (int2, int4) ,
  OPERATOR 5 > (int2, int4) ,
  FUNCTION 1 btint24cmp(int2, int4) ;
```

再次移除这些项：

```
ALTER OPERATOR FAMILY integer_ops USING btree DROP

  -- int4 vs int2
  OPERATOR 1 (int4, int2) ,
  OPERATOR 2 (int4, int2) ,
  OPERATOR 3 (int4, int2) ,
  OPERATOR 4 (int4, int2) ,
  OPERATOR 5 (int4, int2) ,
  FUNCTION 1 (int4, int2) ,
```

```
-- int2 vs int4
OPERATOR 1 (int2, int4) ,
OPERATOR 2 (int2, int4) ,
OPERATOR 3 (int2, int4) ,
OPERATOR 4 (int2, int4) ,
OPERATOR 5 (int2, int4) ,
FUNCTION 1 (int2, int4) ;
```

兼容性

在 SQL 标准中没有 **ALTER OPERATOR FAMILY**语句。

另见

CREATE OPERATOR FAMILY(7), DROP OPERATOR FAMILY(7), CREATE OPERATOR CLASS(7), ALTER OPERATOR CLASS(7), DROP OPERATOR CLASS(7)

名称

ALTER POLICY — 更改一条行级安全性策略的定义

大纲

```
ALTER POLICY name ON table_name RENAME TO new_name

ALTER POLICY name ON table_name
  [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
  [ USING ( using_expression ) ]
  [ WITH CHECK ( check_expression ) ]
```

描述

ALTER POLICY更改一条现有行级安全性策略的定义。 请注意，**ALTER POLICY**只允许修改策略所应用的角色集合， 和要修改的USING和WITH CHECK表达式。 要更改策略的其他属性，例如其应用的命令，或者是允许还是限制， 则必须删除并重新创建策略。

要使用**ALTER POLICY**，必须拥有该策略所适用的 表。

在**ALTER POLICY**的第二种形式中，如果指定了角色列表、 *using_expression*以及*check_expression*， 它们会被独立地替换。当这些子句之一被省略时，策略的对应部分不会被更改。

参数

name

要更改的现有策略的名称。

table_name

该策略所在的表的名称（可以被模式限定）。

new_name

该策略的新名称。

role_name

该策略适用的角色。可以一次指定多个角色。要把该策略 应用于所有角色，可使用PUBLIC。

using_expression

该策略的USING表达式。详见 CREATE POLICY(7)。

check_expression

该策略的WITH CHECK表达式。详见 CREATE POLICY(7)。

兼容性

ALTER POLICY是一种UXDB扩展。

另见

CREATE POLICY(7), DROP POLICY(7)

名称

ALTER PROCEDURE — change the definition of a procedure

大纲

```
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER PROCEDURE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中*action*是下列之一：

```
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

简介

ALTER PROCEDURE更改一个过程的定义。

要使用**ALTER PROCEDURE**，必须拥有该过程。要更改一个过程的方案，还必须有新方案上的CREATE特权。要更改拥有者，还必须是新拥有角色的直接或间接成员，并且那个角色在该过程的方案上拥有CREATE特权（这些限制强制更新拥有者无法做到通过删除和重建该过程无法做到的事情。不过，超级用户总是能够更改任何过程的拥有关系）。

参数

name

一个现有的过程的名字（可以被方案限定）。如果没有指定参数列表，这个名字必须在其方案中唯一。

argmode

参数的模式：IN或VARIADIC。如果被省略，默认是IN。

argname

参数的名字。注意**ALTER PROCEDURE**实际上并不关心参数名，因为只需要参数的数据类型来确定过程的身份。

argtype

如果该过程有参数，这是参数的数据类型（可以被方案限定）。

new_name

该过程的新名字。

new_owner

该过程的新拥有者。注意，如果这个过程被标记为SECURITY DEFINER，接下来它将被作为新拥有者执行。

new_schema

该过程的新方案。

extension_name

该过程所依赖的扩展的名称。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

更改该过程是否为一个安全性定义器。关键词EXTERNAL由于SQL符合性的原因被忽略。更多有关这个能力的信息请见CREATE PROCEDURE(7)。

configuration_parameter
value

增加或者更改在调用该过程时，要对一个配置参数做的赋值。如果*value*是DEFAULT或者等效的值，则会使用RESET，过程本地的设置会被移除，这样该过程的执行就会使用其所处环境中的值。使用RESET ALL可以清除所有的过程本地设置。SET FROM CURRENT会把ALTER PROCEDURE执行时该参数的当前值保存为进入该过程时要被应用的值。

关于允许的参数名和参数值的更多信息请见SET(7)。

RESTRICT

为了符合SQL标准会被忽略。

示例

要重命名具有两个integer类型参数的过程insert_data为insert_record：

```
ALTER PROCEDURE insert_data(integer, integer) RENAME TO insert_record;
```

要把具有两个integer类型参数的过程insert_data的拥有者改为joe：

```
ALTER PROCEDURE insert_data(integer, integer) OWNER TO joe;
```

要重把具有两个integer类型参数的过程insert_data的方案改为accounting：

```
ALTER PROCEDURE insert_data(integer, integer) SET SCHEMA accounting;
```

把过程insert_data(integer, integer)标记为依赖于扩展myext：

ALTER PROCEDURE insert_data(integer, integer) DEPENDS ON EXTENSION myext;

要调整一个过程自动设置的搜索路径：

ALTER PROCEDURE check_password(text) SET search_path = admin, ux_temp;

要为一个过程禁用search_path的自动设置：

ALTER PROCEDURE check_password(text) RESET search_path;

现在这个过程将用其调用者所使用的任何搜索路径执行。

兼容性

这个语句与SQL标准中的**ALTER PROCEDURE**语句部分兼容。标注允许修改一个过程的更多性质，但是不提供重命名过程、让过程成为安全性定义器、为过程附加配置参数值或者更改过程的拥有者、方案或者可变性的能力。标准还要求RESTRICT关键字，而它在UXDB中是可选的。

另见

CREATE PROCEDURE(7), DROP PROCEDURE(7), ALTER FUNCTION(7), ALTER ROUTINE(7)

名称

ALTER PUBLICATION — 修改发布的定义

大纲

```
ALTER PUBLICATION name ADD TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name DROP TABLE [ ONLY ] table_name [ * ] [, ...]
ALTER PUBLICATION name SET ( publication_parameter [= value] [, ... ] )
ALTER PUBLICATION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER PUBLICATION name RENAME TO new_name
```

描述

命令**ALTER PUBLICATION**可以更改发布的属性。

前三个语句更改哪些表是该发布的一部分。SET TABLE 子句用指定的表替换发布中的表的列表。ADD TABLE和 DROP TABLE子句将从发布中添加和删除一个或多个表。 请注意，将表添加到已订阅的发布中将需要在订阅端执行ALTER SUBSCRIPTION ... REFRESH PUBLICATION操作才能生效。

第四条语句可以改变在CREATE PUBLICATION(7)中指定的所有发布属性。 该命令中未提及的属性保留其先前的设置。

其余语句更改所有者和发布的名称。

必须拥有该发布才能使用**ALTER PUBLICATION**。要改变所有者， 也必须是新所有者角色的直接或间接成员。新的所有者必须在数据库上拥有 CREATE权限。此外，FOR ALL TABLES 发布的新所有者必须是超级用户。但是， 超级用户可以在避开这些限制的情况下更改发布的所有权。

参数

name

要修改定义的现有发布的名称。

table_name

现有表的名称。如果在表名之前指定了ONLY，则只有该表受到影响。 如果没有指定ONLY，则该表及其所有后代表（如果有的话）都会受到影响。 可选地，可以在表名之后指定*以明确指示包含后代表。

SET (*publication_parameter* [= *value*] [, ...])

该子句修改最初由CREATE PUBLICATION(7)设置的发布参数。

new_owner

发布的新所有者的用户名。

new_name

发布的新名称。

示例

将发布修改为只发布删除和更新： Change the publication to publish only deletes and updates:

ALTER PUBLICATION noinsert SET (publish = 'update, delete');

给发布添加一些表：

ALTER PUBLICATION mypublication ADD TABLE users, departments;

兼容性

ALTER PUBLICATION是UXDB的一个扩展。

又见

CREATE PUBLICATION(7), DROP PUBLICATION(7), CREATE SUBSCRIPTION(7), ALTER SUBSCRIPTION(7)

名称

ALTER ROLE — 更改一个数据库角色

大纲

ALTER ROLE *role_specification* [WITH] *option* [...]

其中*option*可以是：

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT connlimit
  | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
  | VALID UNTIL 'timestamp'
```

ALTER ROLE *name* RENAME TO *new_name*

```
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter FROM CURRENT
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER ROLE { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中*role_specification*可以是：

```
   role_name
 | CURRENT_USER
 | SESSION_USER
```

描述

ALTER ROLE更改一个 UXDB角色的属性。

前面列出的这个命令的第一种变体能够更改CREATE ROLE(7)中 指定的很多角色属性（覆盖了所有可能的属性，不过没有增加和移除成员关系的选项， 如果要增加和移除成员关系可使用GRANT(7)和 REVOKE(7)）。该命令中没有提到的属性保持它们之前的设置。 数据库超级用户能够更改任何角色的任何这些设置。具有CREATEROLE 特权的角色能够更改任何这些设置，但是只能为非超级用户和非复制角色修改。普通 角色只能更改它们自己的口令。

第二种变体更改该角色的名称。数据库超级用户能重命名任何角色。具有 CREATEROLE特权的角色能够重命名任何非超级用户角色。当前的会话 用户不能被重命名（如果需要这样做，请以一个不同的用户连接）。由于 MD5加密的口令使用角色名作为 salt，因此如果一个角色的口令是 MD5加密的，重命名该角色会清空其口令。

其余的变体用于更改一个角色的配置变量的会话默认值，可以为所有数据库设置，或者 只为IN DATABASE中指定的数据库设置。如果指定的是 ALL而不是一个角色名，将会为所有角色更改该设置。把 ALL和IN DATABASE一起使用实际上和使用命 令ALTER DATABASE ... SET ...相同。

只要改角色后续开始一个新会话，指定的值将会成为该会话的默认值，并且会覆盖 uxsinodb.conf中存在的值或者从 uxdb命令行收到的值。这只在登录时发生，执行 SET ROLE(7)或者 SET SESSION AUTHORIZATION(7)不会导致新的配置值被设置。 对于所有数据库设置的值会被附加到一个角色的数据库相关的设置所覆盖。特定数 据库或角色的设置会覆盖为所有角色所作的设置。

超级用户能够更改任何人的会话默认值。具有CREATEROLE特权的角色 能够更改非超级用户的默认值。普通角色只能为它们自己设置默认值。某些配置变量 不能以这种方式设置，或者只能由一个超级用户发出的命令设置。只有超级用户能够 更改所有角色在所有数据库中的设置。

参数

name

要对其属性进行修改的角色的名称。

CURRENT_USER

修改当前用户而不是一个显式标识的角色。

SESSION_USER

修改当前会话用户而不是一个显式标识的角色。

SUPERUSER
NOSUPERUSER
CREATEDB
NOCREATEDB
CREATEROLE
NOCREATEROLE
INHERIT
NOINHERIT
LOGIN
NOLOGIN
REPLICATION
NOREPLICATION
BYPASSRLS
NOBYPASSRLS
CONNECTION LIMIT *connlimit*
[ENCRYPTED] PASSWORD '*password*'
PASSWORD NULL
VALID UNTIL '*timestamp*'

这些子句修改原来有CREATE ROLE(7) 设置的属性。更多信息请见 CREATE ROLE参考页。

new_name

该角色的新名称。

database_name

要在其中设置该配置变量的数据库名称。

configuration_parameter
value

把这个角色的指定配置参数的会话默认值设置为给定值。如果 *value*为DEFAULT 或者等效地使用了RESET，角色相关的变量 设置会被移除，这样该角色将会在新会话中继承系统范围的默认 设置。使用RESET ALL可清除所有角色相关的 设置。SET FROM CURRENT可以把会话中该参数的 当前值保存为角色相关的值。如果指定了 IN DATABASE，只会为给定的角色和数据库 设置或者移除该配置参数。

角色相关的变量设置只在登录时生效， SET ROLE(7)以及 SET SESSION AUTHORIZATION(7)不会处理角色 相关的变量设置。

关于允许的参数名称和值详见SET(7)。

注解

使用CREATE ROLE(7)增加新角色，使用 DROP ROLE(7)移除一个角色。

ALTER ROLE无法更改一个角色成员关系。 可以使用GRANT(7)和 REVOKE(7)来实现。

在使用这个命令指定一个未加密口令时要多加注意。该口令将会以明文传送到服务器，并且它还可能会被记录在客户端的命令历史或者服务器日志中。psql包含了一个命令\password，它可以被用来更改一个角色的口令而不暴露明文口令。

也可以把一个会话默认值绑定到一个指定的数据库而不是一个角色，详见 ALTER DATABASE(7)。如果出现冲突，数据库角色相关 的设置会覆盖角色相关的设置，角色相关的又会覆盖数据库相关的设置。

示例

更改一个角色的口令：

```
ALTER ROLE davide WITH PASSWORD 'hu8jmn3';
```

移除一个角色的口令：

```
ALTER ROLE davide WITH PASSWORD NULL;
```

更改一个口令的失效日期，指定该口令应该在 2015 年 5 月 4 日中午 （在一个比UTC快 1 小时的时区）过期：

```
ALTER ROLE chris VALID UNTIL 'May 4 12:00:00 2015 +1';
```

让一个口令永远有效：

```
ALTER ROLE fred VALID UNTIL 'infinity';
```

让一个角色能够创建其他角色和新的数据库：

ALTER ROLE miriam CREATEROLE CREATEDB;

为一个角色指定maintenance_work_mem参数的非默认设置：

ALTER ROLE worker_bee SET maintenance_work_mem = 100000;

为一个角色指定maintenance_work_mem参数的数据库相关的非 默认设置：

ALTER ROLE fred IN DATABASE devel SET client_min_messages = DEBUG;

兼容性

ALTER ROLE语句是一个 UXDB扩展。

另见

CREATE ROLE(7), DROP ROLE(7), ALTER DATABASE(7), SET(7)

名称

ALTER ROUTINE — 更改一个例程的定义

大纲

```
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    action [ ... ] [ RESTRICT ]
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    RENAME TO new_name
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    SET SCHEMA new_schema
ALTER ROUTINE name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ]
    DEPENDS ON EXTENSION extension_name
```

其中*action*是下列之一：

```
    IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    PARALLEL { UNSAFE | RESTRICTED | SAFE }
    COST execution_cost
    ROWS result_rows
    SET configuration_parameter { TO | = } { value | DEFAULT }
    SET configuration_parameter FROM CURRENT
    RESET configuration_parameter
    RESET ALL
```

描述

ALTER ROUTINE更改一个例程的定义，它可以是聚集函数、普通函数或者过程。参数的描述、更多的例子以及进一步的细节请参考ALTER AGGREGATE(7)、ALTER FUNCTION(7)以及ALTER PROCEDURE(7)。

示例

将类型integer的例程foo重命名为foobar：

```
ALTER ROUTINE foo(integer) RENAME TO foobar;
```

不管foo是聚集、函数还是过程，这个命令都能使用。

兼容性

这个语句与SQL标准中的**ALTER ROUTINE**语句部分兼容。更多细节请参考ALTER FUNCTION(7)和ALTER PROCEDURE(7)。允许例程名称引用聚集函数是一种UXDB的扩展。

另见

ALTER AGGREGATE(7), ALTER FUNCTION(7), ALTER PROCEDURE(7), DROP ROUTINE(7)

注意没有CREATE ROUTINE命令。

名称

ALTER RULE — 更改一个规则定义

大纲

ALTER RULE *name* ON *table_name* RENAME TO *new_name*

描述

ALTER RULE更改一条现有规则的定义。当前，唯一可用的 动作是更改规则的名称。

要使用**ALTER RULE**，必须拥有该规则适用的表或者视图。

参数

name

要修改的一条现有规则的名称。

table_name

该规则适用的表或视图的名称（可以是模式限定的）。

new_name

该规则的新名称。

示例

要重命名一条现有的规则：

```
ALTER RULE notify_all ON emp RENAME TO notify_me;
```

兼容性

ALTER RULE是一种 UXDB的语言扩展，整个查询重写系统也是。

另见

CREATE RULE(7), DROP RULE(7)

名称

ALTER SCHEMA — 更改一个模式的定义

大纲

```
ALTER SCHEMA name RENAME TO new_name
ALTER SCHEMA name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
```

描述

ALTER SCHEMA更改一个模式的定义。

要使用**ALTER SCHEMA**，必须拥有该模式。要重命名一个模式， 还必须拥有该数据库的CREATE特权。要更改拥有者，还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该数据库上的 CREATE特权（注意超级用户自动拥有所有这些特权）。

参数

name

一个现有模式的名称。

new_name

该模式的新名称。新名称不能以ux_开始，因为这些名称被 保留用于系统模式。

new_owner

该模式的新拥有者。

兼容性

在 SQL 标准中没有**ALTER SCHEMA**语句。

另见

CREATE SCHEMA(7), DROP SCHEMA(7)

名称

ALTER SEQUENCE — 更改一个序列发生器的定义

大纲

```
ALTER SEQUENCE [ IF EXISTS ] name
    [ AS data_type ]
    [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ]
    [ RESTART [ [ WITH ] restart ] ]
    [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
ALTER SEQUENCE [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER SEQUENCE [ IF EXISTS ] name RENAME TO new_name
ALTER SEQUENCE [ IF EXISTS ] name SET SCHEMA new_schema
```

描述

ALTER SEQUENCE更改一个现有序列发生器的参数。 任何没有在ALTER SEQUENCE命令中明确设置的参数 保持它们之前的设置。

要使用ALTER SEQUENCE，必须拥有该序列。要更改一个序列 的模式，还必须拥有新模式上的CREATE特权。要更改拥有者，还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该域的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建该序列做不到的事情。不过，一个超级用户怎么都能更改任何序列的所有权。）。

参数

name

要修改的序列的名称（可以是模式限定的）。

IF EXISTS

在序列不存在时不要抛出一个错误。这种情况下会发出一个提示。

data_type

可选子句AS *data_type* 改变序列的数据类型。有效类型是smallint、integer 和bigint。

当且仅当先前的最小值和最大值是旧数据类型的最小值或最大值时（换句话说， 如果序列是使用NO MINVALUE或NO MAXVALUE， 隐式或显式创建的），则更改数据类型会自动更改序列的最小值和最大值。 否则，将保留最小值和最大值，除非将新值作为同一命令的一部分给出。 如果最小值和最大值不符合新的数据类型，则会生成错误。

increment

子句INCREMENT BY *increment*是可选的。 一个正值将产生一个上升序列，一个负值会产生一个下降序列。如果 没有指定，旧的增量值将被保持。

minvalue
NO MINVALUE

可选的子句MINVALUE *minvalue*决定一个序列 能产生的最小值。如果指定了NO MINVALUE，上升序列和下降序列的默认值分别是 1 和 数据类型的最小值。如果这些选项都没有被指定，将保持当前的 最小值。

maxvalue
NO MAXVALUE

可选的子句MAXVALUE *maxvalue*决定一个序列 能产生的最大值。如果指定了NO MAXVALUE，上升序列和下降序列的默认值分别是 数据类型的最大值和 -1。如果这些选项都没有被指定，将保持当前的 最大值。

start

可选的子句START WITH *start*更改该序列被记录的开始值。 这对于当前序列值没有影响，它会简单地设置 未来ALTER SEQUENCE RESTART命令将会使用的值。

restart

可选的子句RESTART [WITH *restart*]更改该序列的 当前值。这类似于用is_called = false 调用setval函数：被指定的值将会被 下一次nextval调用返回。写上没有 *restart*值的 RESTART等效于提供被 CREATE SEQUENCE记录的或者上一次被 ALTER SEQUENCE START WITH设置的开始值。

与setval调用相比，序列上的RESTART 操作是事务性的并阻止并发事务从同一序列中获取数字。 如果这不是所需的操作模式，则应使用setval。

cache

子句CACHE *cache*使得序列数字被预先 分配并且保存在内存中以便更快的访问。最小值是1（每次只产生一个值，即 无缓存）。如果没有指定，旧的缓冲值将被保持。

CYCLE

可选的CYCLE关键词可以被用来允许该序列在达到 *maxvalue*（上升序列）或 *minvalue*（下降序列）时 回卷。如果到达该限制，下一个被产生的数字将分别是 *minvalue*或者 *maxvalue*。

NO CYCLE

如果指定了可选的NO CYCLE关键词，任何在该 序列到达其最大值后的nextval调用将会返回一个错误。如果既没有指定CYCLE也没有指定 NO CYCLE，旧的循环行为将被保持。

OWNED BY *table_name*. *column_name*
OWNED BY NONE

OWNED BY选项导致该序列与一个特定的表列相关联， 这样如果该列（或者整个表）被删除，该序列也会被自动删除。如果指定， 这种关联会替代之前为该序列指定的任何关联。被指定的表必须具有相同的 拥有者并且与该序列在同一个模式中。指定 OWNED BY NONE可以移除任何现有的关联，让该序列 “自立”。

new_owner

该序列的新拥有者的用户名。

new_name

该序列的新名称。

new_schema

该序列的新模式。

注解

ALTER SEQUENCE将不会立即影响除当前后端外其他后端中的nextval结果，因为它们有预分配（缓存）的序列值。在注意到序列生成参数被更改之前它们将用尽所有缓存的值。当前后端将被立刻影响。

ALTER SEQUENCE不会影响该序列的currval状态。

ALTER SEQUENCE阻塞并发nextval、 currval、lastval和 setval调用。

由于历史原因，ALTER TABLE也可以被用于序列， 但是只有等效于上述形式的ALTER TABLE变体才被 允许用于序列。

示例

在 105 重启一个被称为serial的序列：

```
ALTER SEQUENCE serial RESTART WITH 105;
```

兼容性

ALTER SEQUENCE符合SQL 标准，不过AS、START WITH、 OWNED BY、OWNER TO、RENAME TO 以及SET SCHEMA子句是 UXDB扩展。

另见

CREATE SEQUENCE(7), DROP SEQUENCE(7)

名称

ALTER SERVER — 更改一个外部服务器的定义

大纲

```
ALTER SERVER name [ VERSION 'new_version' ]
    [ OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] ) ]
ALTER SERVER name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SERVER name RENAME TO new_name
```

描述

ALTER SERVER更改一个外部服务器的定义。 第一种形式更改该服务器的版本字符串或者该服务器的一般选项（至少要求 一个子句）。第二种形式更改该服务器的拥有者。

要修改该服务器，必须是它的拥有者。此外为了修改拥有者，必须拥有 该服务器并且是新拥有角色的一个直接或者间接成员，并且必须具有该服 务器的外部数据包装器上的USAGE特权（注意超级用户自动 满足所有这些政策）。

参数

name

一个现有服务器的名称。

new_version

新的服务器版本。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

更改该服务器的选项。ADD、SET和 DROP指定要执行的动作。如果没有显式地指定操作，将会假定为ADD。选项名称必须唯一，名称和值也会 使用该服务器的外部数据包装器库进行验证。

new_owner

该外部服务器的新拥有者的用户名。

new_name

该外部服务器的新名称。

示例

修改服务器foo，增加连接选项：

```
ALTER SERVER foo OPTIONS (host 'foo', dbname 'foodb');
```

修改服务器foo，更改版本、更改host选项：

```
ALTER SERVER foo VERSION '8.4' OPTIONS (SET host 'baz');
```

兼容性

ALTER SERVER符合 ISO/IEC 9075-9 (SQL/MED)。 OWNER TO和RENAME形式是 UXDB 扩展。

另见

CREATE SERVER(7), DROP SERVER(7)

名称

ALTER STATISTICS — 更改扩展统计对象的定义

大纲

```
ALTER STATISTICS name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER STATISTICS name RENAME TO new_name
ALTER STATISTICS name SET SCHEMA new_schema
```

描述

ALTER STATISTICS更改现有扩展统计对象的参数。 任何在**ALTER STATISTICS**命令中没有明确设定的参数保持它们之前的设置。

您必须拥有统计对象才能使用**ALTER STATISTICS**。 要更改统计对象的模式，还必须在新模式上具有CREATE权限。 要更改所有者，还必须是新所有者角色的直接或间接成员， 且该角色在统计对象的模式上必须具有CREATE权限。 （这些限制强制了通过删除和重新创建统计对象来改变所有者不会做任何不能做的事情， 但是超级用户可以改变任何统计对象的所有权。）

参数

name

要修改的统计对象的名称（可能有模式修饰）。

new_owner

统计对象的新所有者的用户名。

new_name

统计对象的新名称。

new_schema

统计对象的新模式。

兼容性

SQL标准中没有**ALTER STATISTICS**命令。

又见

CREATE STATISTICS(7), DROP STATISTICS(7)

名称

ALTER SUBSCRIPTION — 修改订阅的定义

大纲

```
ALTER SUBSCRIPTION name CONNECTION 'conninfo'
ALTER SUBSCRIPTION name SET PUBLICATION publication_name [, ...] [ WITH
( set_publication_option [= value] [, ... ] ) ]
ALTER SUBSCRIPTION name REFRESH PUBLICATION [ WITH ( refresh_option [= value] [, ... ] ) ]
ALTER SUBSCRIPTION name ENABLE
ALTER SUBSCRIPTION name DISABLE
ALTER SUBSCRIPTION name SET ( subscription_parameter [= value] [, ... ] )
ALTER SUBSCRIPTION name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER SUBSCRIPTION name RENAME TO new_name
```

描述

ALTER SUBSCRIPTION可以修改大部分可以在 CREATE SUBSCRIPTION(7)中指定的订阅属性。

要使用**ALTER SUBSCRIPTION**，必须拥有该订阅。要改变所有者， 也必须是新所有者的直接或间接成员。新所有者必须是超级用户。（目前， 所有的订阅所有者必须是超级用户，所以所有者的检查将在实践中被绕过， 但这可能在未来发生变化。）

参数

name

要修改属性的订阅的名称。

CONNECTION '*conninfo*'

该子句修改最初由CREATE SUBSCRIPTION(7)设置的连接属性。

SET PUBLICATION *publication_name*

更改订阅发布的列表。参阅CREATE SUBSCRIPTION(7) 获取更多信息。默认情况下，此命令也将像REFRESH PUBLICATION一样工作。

*set_publication_option*指定了这个操作的附加选项。 支持的选项是：

refresh (boolean)

如果为false，则该命令将不会尝试刷新表信息。然后应单独执行 REFRESH PUBLICATION。默认值是true。

此外，可以指定REFRESH PUBLICATION下描述的刷新选项。

REFRESH PUBLICATION

从发布者获取缺少的表信息。这将开始复制自上次调用**REFRESH PUBLICATION** 或从**CREATE SUBSCRIPTION**以来添加到订阅发布中的表。

*refresh_option*指定了刷新操作的附加选项。 支持的选项有：

copy_data (boolean)

指定在复制启动后是否应复制正在订阅的发布中的现有数据。 默认值是true。（以前订阅的表不会被复制。）

ENABLE

启用先前禁用的订阅，在事务结束时启动逻辑复制工作。

DISABLE

禁用正在运行的订阅，在事务结束时停止逻辑复制工作。

SET (*subscription_parameter* [= *value*] [, ...])

该子句修改原先由CREATE SUBSCRIPTION(7)设置的参数。 允许的选项是slot_name和synchronous_commit。

new_owner

订阅的新所有者的用户名。

new_name

订阅的新名称。

示例

将订阅的发布更改为insert_only：

```
ALTER SUBSCRIPTION mysub SET PUBLICATION insert_only;
```

禁用（停止）订阅：

```
ALTER SUBSCRIPTION mysub DISABLE;
```

兼容性

ALTER SUBSCRIPTION是UXDB 的一个扩展。

又见

CREATE SUBSCRIPTION(7), DROP SUBSCRIPTION(7), CREATE PUBLICATION(7), ALTER PUBLICATION(7)

名称

ALTER SYSTEM — 更改一个服务器配置参数

大纲

ALTER SYSTEM SET *configuration_parameter* { TO | = } { *value* | '*value*' | DEFAULT }

ALTER SYSTEM RESET *configuration_parameter*
ALTER SYSTEM RESET ALL

描述

ALTER SYSTEM被用来在整个数据库集簇范围内更改 服务器配置参数。它比传统的手动编辑uxsinodb.conf 文件的方法更方便。ALTER SYSTEM会把给出的参数 设置写入到uxsinodb.auto.conf文件中，该文件会随着 uxsinodb.conf一起被读入。把一个参数设置为DEFAULT或者使用RESET变体可以 把该配置项从uxsinodb.auto.conf文件中移除。使用 RESET ALL可以移除所有这类配置项。

用ALTER SYSTEM设置的值将在下一次重载服务器 配置后生效，那些只能在服务器启动时更改的参数则会在下一次服务器重启后生效。 重载服务器配置可以通过以下做法实现：调用 SQL 函数ux_reload_conf()， 运行ux_ctl reload或者向主服务器进程发送一个SIGHUP信号。

只有超级用户能够使用ALTER SYSTEM。还有，由于 这个命令直接作用于文件系统并且不能被回滚，不允许在一个事务块或者函数中使用它。

参数

configuration_parameter

一个可设置配置参数的名称。

value

该参数的新值。值可以被指定为字符串常量、标识符、数字或者以上这些构成的 逗号分隔的列表，值的具体形式取决于特定的参数。写上 DEFAULT可以用来把该参数及其值从uxsinodb.auto.conf中移除。

注解

不能用这个命令来设置data_directory以及 uxsinodb.conf中不被允许的参数（例如 preset options）。

示例

设置wal_level：

ALTER SYSTEM SET wal_level = replica;

撤销以上的设置，恢复uxsinodb.conf中有效的设置：

```
ALTER SYSTEM RESET wal_level;
```

兼容性

ALTER SYSTEM语句是一种 UXDB扩展。

另见

SET(7), SHOW(7)

名称

ALTER TABLE — 更改一个表的定义

大纲

```
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
  action [, ... ]
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
  [DISABLE | ENABLE]  [ VALIDATE | NOVALIDATE ]  CONSTRAINT constraint_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
  RENAME [ COLUMN ] column_name TO new_column_name
ALTER TABLE [ IF EXISTS ] [ ONLY ] name [ * ]
  RENAME CONSTRAINT constraint_name TO new_constraint_name
ALTER TABLE [ IF EXISTS ] name
  RENAME TO new_name
ALTER TABLE [ IF EXISTS ] name
  SET SCHEMA new_schema
ALTER TABLE ALL IN TABLESPACE name [ OWNED BY role_name [, ... ] ]
  SET TABLESPACE new_tablespace [ NOWAIT ]
ALTER TABLE [ IF EXISTS ] name
  ATTACH PARTITION partition_name { FOR VALUES partition_bound_spec | DEFAULT }
ALTER TABLE [ IF EXISTS ] name
  DETACH PARTITION partition_name
```

其中*action* 是以下之一：

```
  ADD [ COLUMN ] [ IF NOT EXISTS ] column_name data_type [ COLLATE collation ]
[ column_constraint [ ... ] ]
  DROP [ COLUMN ] [ IF EXISTS ] column_name [ RESTRICT | CASCADE ]
  ALTER [ COLUMN ] column_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
[ USING expression ]
  ALTER [ COLUMN ] column_name SET DEFAULT expression
  ALTER [ COLUMN ] column_name DROP DEFAULT
  ALTER [ COLUMN ] column_name { SET | DROP } NOT NULL
  ALTER [ COLUMN ] column_name ADD GENERATED { ALWAYS | BY DEFAULT } AS
IDENTITY [ ( sequence_options ) ]
  ALTER [ COLUMN ] column_name { SET GENERATED { ALWAYS | BY DEFAULT } |
SET sequence_option | RESTART [ [ WITH ] restart ] } [...]
  ALTER [ COLUMN ] column_name DROP IDENTITY [ IF EXISTS ]
  ALTER [ COLUMN ] column_name SET STATISTICS integer
  ALTER [ COLUMN ] column_name SET ( attribute_option = value [, ... ] )
  ALTER [ COLUMN ] column_name RESET ( attribute_option [, ... ] )
  ALTER [ COLUMN ] column_name SET STORAGE { PLAIN | EXTERNAL | EXTENDED |
MAIN }
  ADD table_constraint [ NOT VALID ]
  ADD table_constraint_using_index
  ADD [ CONSTRAINT [constraint_name] ]...
  ALTER CONSTRAINT constraint_name [ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY
DEFERRED | INITIALLY IMMEDIATE ]
  VALIDATE CONSTRAINT constraint_name
  DROP CONSTRAINT [ IF EXISTS ]  constraint_name [ RESTRICT | CASCADE ]
```

```
    DISABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE TRIGGER [ trigger_name | ALL | USER ]
    ENABLE REPLICA TRIGGER trigger_name
    ENABLE ALWAYS TRIGGER trigger_name
    DISABLE RULE rewrite_rule_name
    ENABLE RULE rewrite_rule_name
    ENABLE REPLICA RULE rewrite_rule_name
    ENABLE ALWAYS RULE rewrite_rule_name
    DISABLE ROW LEVEL SECURITY
    ENABLE ROW LEVEL SECURITY
    FORCE ROW LEVEL SECURITY
    NO FORCE ROW LEVEL SECURITY
    CLUSTER ON index_name
    SET WITHOUT CLUSTER
    SET WITHOUT OIDS
    SET TABLESPACE new_tablespace
    SET { LOGGED | UNLOGGED }
    SET ( storage_parameter = value [, ... ] )
    RESET ( storage_parameter [, ... ] )
    INHERIT parent_table
    NO INHERIT parent_table
    OF type_name
    NOT OF
    OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
    REPLICA IDENTITY { DEFAULT | USING INDEX index_name | FULL | NOTHING }
```

and *partition_bound_spec* is:

```
IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )
```

and *column_constraint* is:

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

而*table_constraint*是：

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
```

PRIMARY KEY (*column_name* [, ...]) *index_parameters* |
EXCLUDE [USING *index_method*] (*exclude_element* WITH *operator* [, ...]) *index_parameters*
[WHERE (*predicate*)] |
FOREIGN KEY (*column_name* [, ...]) REFERENCES *reftable* [(*refcolumn* [, ...])]
[MATCH FULL | MATCH PARTIAL | MATCH SIMPLE] [ON DELETE *action*] [ON
UPDATE *action*] }
[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

并且 *table_constraint_using_index* 是：

[CONSTRAINT *constraint_name*]
{ UNIQUE | PRIMARY KEY } USING INDEX *index_name*
[DEFERRABLE | NOT DEFERRABLE] [INITIALLY DEFERRED | INITIALLY IMMEDIATE]

UNIQUE、PRIMARY KEY以及EXCLUDE约束中的*index_parameters*是：

[INCLUDE (*column_name* [, ...])]
[WITH (*storage_parameter* [= *value*] [, ...])]
[USING INDEX TABLESPACE *tablespace_name*]

exclude_element in an EXCLUDE constraint is:

{ *column_name* | (*expression*) } [*opclass*] [ASC | DESC] [NULLS { FIRST | LAST }]

描述

ALTER TABLE更改一个现有表的定义。下文描述了几种形式。注意每一种形式所要求的锁级别可能不同。如果没有明确说明，将会持有一个ACCESS EXCLUSIVE所。当列出多个子命令时，所持有的锁将是子命令所要求的最严格的那一个。

ADD COLUMN [IF NOT EXISTS]

这种形式向该表增加一个新列，使用与CREATE TABLE(7)相同的语法。如果指定了IF NOT EXISTS并且使用这个名字的列已经存在，则 不会抛出错误。

DROP COLUMN [IF EXISTS]

这种形式从表删除一列。涉及到该列的索引和表约束也将会被自动删除。如果该列的移除会导致引用它的多元统计信息仅包含单一列的数据，则该多元统计信息也将被移除。如果在该表之外有任何东西（例如外键引用或者视图）依赖于该列，将需要用到CASCADE。如果指定了IF EXISTS但该列不存在，则不会抛出错误。这种情况中会发出一个提示。

ADD [CONSTRAINT [name]]...

约束的数据类型是一种限制能够存储在表中数据类别的方法。ADD CONSTRAINT子句用于添加表级约束。表级约束包括：PRIMARY KEY约束、UNIQUE约束和检查约束(CHECK)等。添加表级约束时可以带有约束名，系统中同一模式下的约束名不得重复，如果不带约束名，系统自动为此约束命名。

约束和约束名都是可以省略的。但是对于创建表时，如果带有CONSTRAINT关键字的话，约束名不可以省略。

支持检查约束、唯一约束、主键约束、外键约束、排他约束。

对表在约束命省略情况下添加不同约束，均可添加成功，如下所示。

alter table test1 add constraint check(id>0);
alter table test2 add constraint unique(id);
alter table test3 add constraint primary key(id);
alter table test4 add constraint foreign_id foreign key(id) references test2(id);
alter table test5 add constraint EXCLUDE USING gist (c WITH &&);

SET DATA TYPE

这种形式更改表中一列的类型。涉及到该列的索引和简单表约束将通过重新解析最初提供的表达式被自动转换为使用新的列类型。可选的COLLATE子句为新列指定一种排序规则，如果被省略，排序规则会是新列类型的默认排序规则。可选的USING子句指定如何从旧的列值计算新列值，如果被省略，默认的转换和从旧类型到新类型的赋值造型一样。如果没有从旧类型到新类型的隐式或者赋值造型，则必须提供一个USING子句。

SET/DROP DEFAULT

这些形式为一列设置或者移除默认值。默认值只在后续的 INSERT或UPDATE命令中生效， 它们不会导致已经在表中的行改变。

SET/DROP NOT NULL

这些形式更改一列是否被标记为允许空值或者拒绝空值。

SET NOT NULL 只能应用于列，前提是表中没有任何记录包含该列的NULL值。 通常，这一点在ALTER TABLE全表扫描时来检查；但是，如果找到有效的CHECK约束证明不存在NULL，则跳过表扫描。

如果这个表是一个分区，对于在父表中被标记为NOT NULL的列，不能在其上执行DROP NOT NULL。要从所有的分区中删除NOT NULL约束，可以在父表上执行DROP NOT NULL。即使在父表上没有NOT NULL约束，这样的约束还是能被增加到分区上。也就是说，即便父表允许空值，子表也可以不允许空值，但反过来不行。

ADD GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY
SET GENERATED { ALWAYS | BY DEFAULT }
DROP IDENTITY [IF EXISTS]

这些形式更改一列是否是一个标识列，或者是更改一个已有的标识列的产生属性。详情请参考CREATE TABLE(7)。

如果DROP IDENTITY IF EXISTS被指定并且该列不是一个标识列，则不会有错误被抛出。在这种情况下会发出一个提示。

SET *sequence_option*
RESTART

这些形式修改位于一个现有标识列之下的序列。*sequence_option*是一个ALTER SEQUENCE(7)所支持的选项，例如INCREMENT BY。

SET STATISTICS

这种形式为后续的ANALYZE(7)操作设置针对每列 的统计收集目标。目标可以被设置在范围 0 到 10000 之间，还可以 把它设置为 -1 来恢复到使用系统默认的统计目标。更多有关 UXDB 查询规划器使用统计 信息的内容可见第 11.2 节 “规划器使用的统计信息”

SET STATISTICS要求一个SHARE UPDATE EXCLUSIVE锁。

SET (*attribute_option* = *value* [, ...])
RESET (*attribute_option* [, ...])

这种形式设置或者重置每个属性的选项。当前，已定义的针对每个属性的选项只有n_distinct和n_distinct_inherited，它们会覆盖后续ANALYZE(7)操作所得到的可区分值数量估计。n_distinct影响该表本身的统计信息，而n_distinct_inherited影响为该表外加其继承子女收集的统计信息。当被设置为一个正值时，ANALYZE将假定该列刚好包含指定数量的可区分非空值。当被设置为一个负值（必须大于等于 -1）时，ANALYZE将假定可区分非空值的数量与表的尺寸成线性比例，确切的计数由估计的表尺寸乘以给定数字的绝对值计算得到。例如，值 -1 表示该列中所有的值都是可区分的，而值 -0.5 则表示每一个值平均出现两次。当表的尺寸随时间变化时，这会有所帮助，因为这种计算只有在查询规划时才会被执行。指定值为 0 将回到正常的估计可区分值数量的做法。更多有关 UXDB查询规划器使用统计信息的内容可见第 11.2 节 “规划器使用的统计信息”。

更改针对每个属性的选项要求一个 SHARE UPDATE EXCLUSIVE锁。

SET STORAGE

这种形式为一列设置存储模式。这会控制这列是会被保持在线内还是放在一个二级TOAST表中，以及数据是否应被压缩。对于integer之类的定长、线内、未压缩值必须使用PLAIN。MAIN用于线内、可压缩的数据。EXTERNAL用于外部的、未压缩数据。而EXTENDED用于外部的、压缩数据。对于大部分支持非-PLAIN存储的数据类型，EXTENDED 是默认值。使用EXTERNAL将会让很大的 text和bytea之上的子串操作运行得更快，但是代价是存储空间会增加。注意SET STORAGE本身并不改变表中的任何东西，它只是设置在未来的表更新时要追求的策略。

ADD *table_constraint* [NOT VALID]

这种形式使用和CREATE TABLE(7)相同的语法外加 NOT VALID选项为一个表增加一个新的约束，该选项当前只被允许用于外键和 CHECK 约束。

通常，此窗体将导致对表进行扫描，以验证表中的所有现有行是否满足新约束。但是如果使用了 NOT VALID选项 ，则跳过此可能很漫长的扫描。该约束仍将被强制到后续的插入和删除上（也就是说，在外键的情况下如果在被引用表中没有一个匹配的行，操作会失败；或者如果新行不匹配指定的检查条件，操作也会失败）。但是数据库不会假定约束对该表中的所有行都成立，直到通过使用VALIDATE CONSTRAINT选项对它进行验证。参见下述“注意”一节以了解关于使用NOT VALID选项的更多信息。

除了接收约束的表上的锁外，外键约束的增加要求在被引用表上的一个SHARE ROW EXCLUSIVE锁。

当唯一或者主键约束被添加到分区表时，会有额外的限制，请参考CREATE TABLE(7)。此外，当前分区表上的外键约束不能被声明为NOT VALID。

ADD *table_constraint_using_index*

这种形式基于一个已有的唯一索引为一个表增加新的 PRIMARY KEY或UNIQUE约束。该索引中的所有列将被包括在约束中。

该索引不能有表达式列或者是一个部分索引。还有，它必须是一个带有默认排序顺序的 B-树索引。这些限制确保该索引等效于使用常规 ADD PRIMARY KEY或者ADD UNIQUE命令时创建的索引。

如果PRIMARY KEY被指定，并且该索引的列没有被标记 NOT NULL，那么这个命令将尝试对每一个这样的列做 ALTER COLUMN SET NOT NULL。这需要一次全表扫描 来验证这些列不包含空值。在所有其他情况中，这都是一种很快的操作。

如果提供了一个约束名，那么该索引将被重命名以匹配该约束名。否则 该约束将被命名成索引的名称。

这个命令被执行后，该索引被增加的约束“拥有”，这和用常规 ADD PRIMARY KEY或ADD UNIQUE命令 创建的索引一样。特别地，删掉该约束将会导致该索引也消失。

当前在分区表上不支持这种形式。

> **注意**
>
> 如果需要增加一个新的约束但是不希望长时间阻塞表更新，那么使用现有索引增加约束会有所帮助。要这样做，用 `CREATE INDEX CONCURRENTLY`创建该索引，并且 接着使用这种语法把它安装为一个正式的约束。例子见下文。

ALTER CONSTRAINT

这种形式修改之前创建的一个约束的属性。当前只能修改外键约束。

VALIDATE CONSTRAINT

这种形式验证之前创建为NOT VALID的外键或检查约束， 它会扫描表来确保对于该约束没有行不满足约束。如果约束已经被标记为合法，则什么也不会发生。（参见下述 “注意”一节 以了解此命令用途的说明。）

DROP CONSTRAINT [IF EXISTS]

这种形式在一个表上删除指定的约束，还有位于该约束之下的任何索引。如果IF EXISTS 被指定并且该约束不存在，不会抛出错误。在这种情况下会发出一个提示。

DISABLE/ENABLE [REPLICA | ALWAYS] TRIGGER

这些形式配置属于该表的触发器的触发设置。系统仍然知道被禁用触发器的存在，但是即使它的触发事件发生也不会执行它。对于一个延迟触发器，会在事件发生时而不是触发器函数真正被执行时检查其启用状态。可以禁用或者启用用名称指定的单个触发器、表上的所有触发器、用户拥有的触发器（这个选项会排除内部生成的约束触发器，例如用来实现外键约束或可延迟唯一和排除约束）。禁用或者启用内部生成的约束触发器要求超级用户特权，这样做要注意，因为如果这类触发器不被执行，约束的完整性当然无法保证。

触发器引发机制也受到配置变量 session_replication_role的影响。当复制角色是“origin”（默认）或者“local”时，被简单启用的 触发器将被触发。被配置为ENABLE REPLICA的触发 器只有在会话处于“replica”模式时才将被触发。被配置为 ENABLE ALWAYS的触发器的触发不会考虑当前复制 角色。

这种机制的效果就是，在默认配置中，触发器不会在复制体上引发。这种效果很有用，因为如果一个触发器在源头上被用来在表之间传播数据，那么复制系统也将复制被传播的数据，并且触发器不应该在复制体上引发第二次，因为那会导致重复。不过，如果一个触发器被用于另一种目的（例如创建外部告警），那么将它设置为ENABLE ALWAYS可能更加合适，这样它在复制体上也会被引发。

这个命令要求一个SHARE ROW EXCLUSIVE锁。

DISABLE/ENABLE [REPLICA | ALWAYS] RULE

这些形式配置属于表的重写规则的触发设置。系统仍然知道一个被禁用规则的　存在，但在查询重写时不会应用它。其语义与禁用的/启用的触发器的一样。 对于ON SELECT规则会忽略这个配置，即使当前会话处于　一种非默认的复制角色，这类规则总是会被应用以保持视图工作正常。

规则引发机制也受到配置变量session_replication_role的影响，这和上述的触发器类似。

DISABLE/ENABLE ROW LEVEL SECURITY

这些形式控制属于该表的行安全性策略的应用。如果被启用并且该表上　不存在策略，则将应用一个默认否定的策略。注意即使行级安全性被禁　用，在表上还是可以存在策略。在这种情况下，这些策略将 不 会被应用 并且会被忽略。另见CREATE POLICY(7)。

[DISABLE | ENABLE] [VALIDATE | NOVALIDATE] CONSTRAINT

普通表和分区表的主键、外键约束如下。

表　2.　普通表约束

约束状态	主键约束	外键约束
disable	默认为novalidate。修改主键索引的唯一性字段为false，对应主键列的非空约束为false，校验字段为false。	默认为novalidate。修改对应子表和主表的触发器开关为D表示触发器禁用，修改子表约束的conindid（指向reference表的主键索引oid）字段为0。还有对应的约束开关和校验开关字段。
enable	默认为validate。执行校验->重建原有主键索引，重建成功表示旧表内所有数据符合唯一性，校验旧表内是否有空值，校验成功后 统一修改该主键索引唯一性字段为true，主键列非空约束为true。	默认为validate。先执行数据校验，校验成功之后再修改对应字段，包括触发器，子表约束conindid修改为reference表的主键索引oid。
disable validate	关闭主键约束，开关置为true，但需执行数据校验，若表此时的约束状态为这个，则禁止对表作任何修改操作（插入、删除、更新）。	先执行校验，校验成功之后修改相应的开关字段，触发器开关为D，若表此时的约束状态为disable validate，则禁止对表作任何修改操作（插入、删除、更新）。
disable novalidate	同单独指定disable。	同单独指定disable。
enable validate	同单独指定enable。	同单独指定enable。
enable novalidate	对于主键不论指定校验或非校验，只要是打开约束开关就都执行校验。	仅打开约束开关（触发器开关，conindid字段），对于旧数据不执行校验。

分区表：主表实际不存储数据，但存在唯一性索引，所以对主表不执行数据校验，仅修改索引唯一性字段和主键列的非空约束还有约束状态然后对于每个子表重复普通表的操作（每个子表都有一个主键约束）。

约束示例如下所示。

alter table table_name enable constraint constraint_name;
alter table table_name disable constraint constraint_name;
alter table table_name enable validate constraint constraint_name;
alter table table_name enable novalidate constraint constraint_name;
alter table table_name disable validate constraint constraint_name;
alter table table_name disable novalidate constraint constraint_name;
alter table table_name validate constraint constraint_name;

NO FORCE/FORCE ROW LEVEL SECURITY

这些形式控制当用户是表拥有者时表上的行安全性策略的应用。如果被启用， 当用户是表拥有者时，行级安全性策略将被应用。如果被禁用（默认），则 当用户是表拥有者时，行级安全性将不会被应用。另见 CREATE POLICY(7)。

CLUSTER ON

这种形式为未来的CLUSTER(7)操作选择默认的索引。 它不会真正地对表进行聚簇。

改变聚簇选项要求一个SHARE UPDATE EXCLUSIVE锁。

SET WITHOUT CLUSTER

这种形式从表中移除最近使用的 CLUSTER(7)索引说明。这会影响未来的不指定索引 的聚簇操作。

改变聚簇选项要求一个SHARE UPDATE EXCLUSIVE锁。

SET WITHOUT OIDS

向后兼容的语法，用于删除oid系统列。由于oid系统列无法再添加，所以不会有实际效果。

SET TABLESPACE

这种形式把该表的表空间更改为指定的表空间并且把该表相关联的数据文件 移动到新的表空间中。表上的索引（如果有）不会被移动，但是它们可以用 额外的SET TABLESPACE命令单独移动。 当应用于分区表时，不会移动，但之后用CREATE TABLE PARTITION OF创建的任何分区将使用该表空间，除非TABLESPACE子句用于重写它。

当前数据库在一个表空间中的所有表可以用ALL IN TABLESPACE形式 移动，这将会首先锁住所有将被移动的表然后逐个移动。这种形式也支持 OWNED BY，它将只移动指定角色所拥有的表。如果指 定了NOWAIT选项，则命令将在无法立刻获得所有所需 要的锁时失败。注意这个命令不移动系统目录；如果想要移动系统目录，应 该用ALTER DATABASE或者显式的ALTER TABLE调用。对于这种形式来说， information_schema关系不被认为是系统目录的一部分， 因此它们将会被移动。另见CREATE TABLESPACE(7)。

SET { LOGGED | UNLOGGED }

This form changes the table from unlogged to logged or vice-versa (see UNLOGGED). It cannot be applied to a temporary table.

SET (*storage_parameter* = *value* [, ...])

这种形式为该表更改一个或者更多存储参数。可用的参数请见 “存储参数”一节。注意这个 命令将不会立刻修改表内容，这取决于重写表以得到想要的结果可能需要的 参数。可以

用VACUUM FULL、CLUSTER(7)或者 ALTER TABLE的一种形式来强制一次表重写。对于规划器相关的参数，更改将从该表下一次被锁定开始生效，因此当前执行的查询不会受到影响。

对fillfactor、toast以及autovacuum存储参数，将会拿取SHARE UPDATE EXCLUSIVE锁，就像计划器参数 parallel_workers。

RESET (*storage_parameter* [, ...])

这种形式把一个或者更多存储参数重置到它们的默认值。和 SET一样，可能需要一次表重写来更新整个表。

INHERIT *parent_table*

这种形式把目标表增加为指定父表的一个新子女。随后，针对父亲的查询将 包括目标表中的记录。要被增加为一个子女，目标表必须已经包含和父表完 全相同的列（也可以有额外的列）。这些列必须具有匹配的数据类型，并且 如果它们在父表中具有NOT NULL约束，它们在子表中 也必须有NOT NULL约束。

也必须把子表约束与所有父表的CHECK约束进行匹配， 不过父表中那些被标记为非可继承（也就是用ALTER TABLE ... ADD CONSTRAINT ... NO INHERIT 创建的）除外，它们会被忽略。所有匹配得上的子表约束不能被标记为不可 继承。当前，UNIQUE、PRIMARY KEY以及FOREIGN KEY约束没有被考虑，但是这种情况可能 会在未来发生变化。

NO INHERIT *parent_table*

这种形式把目标表从指定父表的子女列表中移除。针对父表的查询将不再包括 来自目标表的记录。

OF *type_name*

这种形式把该表链接到一种组合类型，就好像CREATE TABLE OF所做的那样。该表的列名和类型列表必须精确地匹配该组合类型。 该表必须不从任何其他表继承。这些限制确保CREATE TABLE OF能允许一个等价的表定义。

NOT OF

这种形式解除一个有类型的表和其类型之间的关联。

OWNER TO

这种形式把表、序列、视图、物化视图或外部表的拥有者改为指定用户。

REPLICA IDENTITY

这种形式更改被写入到预写式日志来标识被更新或删除行的信息。除非使用逻辑复制， 这个选项将不会产生效果。DEFAULT（非系统表的默认值）记录主键列 （如果有）的旧值。USING INDEX记录被所提到的索引所覆盖的列的 旧值，该索引必须是唯一索引、不是部分索引、不是可延迟索引并且只包括被标记成 NOT NULL的列。FULL记录行中所有列的旧值。 NOTHING不记录有关旧行的任何信息（这是系统表的默认值）。在所 有情况下，除非至少有一个要被记录的列在新旧行版本之间发生变化，将不记录旧值。

RENAME

RENAME形式更改一个表（或者一个索引、序列、视图、物化视图 或者外部表）的名称、表中一个列的名称或者表的一个约束的名称。在重命名一个具有底层索引的约束时，该索引也会被重命名。它对已存储的数据 没有影响。

SET SCHEMA

这种形式把该表移动到另一个模式中。相关的该表列拥有的索引、约束和序列也会被 移动。

ATTACH PARTITION *partition_name* { FOR VALUES *partition_bound_spec* | DEFAULT }

这种形式把一个已有表（自身也可能被分区）作为一个分区挂接到目标表。该表可以为特定的值使用FOR VALUES挂接为分区，或者用DEFAULT挂接为一个默认分区。 对于目标表中的每一个索引，在被挂接的表上都将创建一个响应的索引，如果已经存在等效的索引，该索引将被挂接到目标表的索引，就像执行了ALTER INDEX ATTACH PARTITION一样。 请注意，如果现有表是外表，如果目标表上有UNIQUE索引，则当前不允许将表作为目标表的分区附加。（参见 CREATE FOREIGN TABLE(7)。）

一个使用FOR VALUES的分区使用与CREATE TABLE(7)中*partition_bound_spec*相同的语法。分区边界说明必须对应于目标表的分区策略以及分区键。要被挂接的表必须具有和目标表完全相同的所有列，并且不能有多出来的列，而且列的类型也必须匹配。此外，它必须有目标表上所有的NOT NULL以及CHECK约束。当前不考虑FOREIGN KEY约束。来自于父表的UNIQUE和PRIMARY KEY约束将被创建在分区上（如果它们还不存在）。如果被挂接的表上的任何CHECK约束被标记为NO INHERIT，则命令将失败，这类约束必须被重建且重建时不能有NO INHERIT子句。

如果新分区是一个常规表，会执行一次全表扫描来检查表中现有行没有违背分区约束。 可以通过对表增加一个有效的CHECK约束来避免这种扫描，该约束可以在运行这个命令之前仅允许满足所需分区约束的行。 CHECK约束可以解决让表无需被扫描就能验证分区约束。但是，如果任一分区键是一个表达式并且该分区不接受NULL值，这种方式就无效了。 如果挂接一个不接受NULL值的列表分区，还应该为分区键列增加NOT NULL约束，除非它是一个表达式。

如果新分区是一个外部表，则不需要验证该外部表中的所有行遵守分区约束（有关外部表上的约束请参考CREATE FOREIGN TABLE(7)中的讨论）。

当一个表有默认分区时，定义新分区会更改默认分区的分区约束。默认分区不能包含任何需要被移动到新分区中的行，并且将被扫描以验证不存在那样的行。如果一个合适的CHECK约束存在，这种扫描（和新分区的扫描一样）可以被避免。还是和新分区的扫描一样，当默认分区是外部表时这种扫描总是会被跳过。

在父表上附加一个分区获得一个SHARE UPDATE EXCLUSIVE锁，除了要附加的表和默认分区上的ACCESS EXCLUSIVE 锁（如果有）。

DETACH PARTITION *partition_name*

这种形式会分离目标表的指定分区。被分离的分区继续作为独立的表存在，但是与它之前挂接的表不再有任何联系。任何被挂接到目标表索引的索引也会被分离。

除了RENAME、SET SCHEMA、ATTACH PARTITION和DETACH PARTITION之外，所有形式的ALTER TABLE都作用在单个表上，前面这些形式可以被组合成一个多修改的列表被一起应用。例如，可以在一个命令中增加多个列并且/或者修改多个列的类型。对于大型表来说这会特别有用，因为只需要对表做一趟操作。

要使用ALTER TABLE，必须拥有该表。要更改一个表的 模式或者表空间，还必须拥有新模式或表空间上的 CREATE特权。要把一个表作为一个父表的新子表加入， 必须也拥有该父表。此外，要把一个表挂接为另一个表的新分区，必须拥有被挂接的表。要更改拥有者，还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该表的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建表做不到的事情。不过，一个超级用户怎么都能更

改任何表的所有权。）。 要增加一个列、修改一列的类型或者使用OF子句，还必 须具有该数据类型上的USAGE特权。

参数

IF EXISTS

如果表不存在则不要抛出一个错误。这种情况下会发出一个提示。

name

要修改的一个现有表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，则只会修改该表。如果没有指定ONLY， 该表及其所有后代表（如果有）都会被修改。可选地，在表名后面可以指定 *用来显式地指示包括后代表。

column_name

一个新列或者现有列的名称。

new_column_name

一个现有列的新名称。

new_name

该表的新名称。

data_type

一个新列的数据类型或者一个现有列的新数据类型。

table_constraint

该表的新的表约束。

constraint_name

一个新约束或者现有约束的名称。

CASCADE

自动删除依赖于被删除列或约束的对象（例如引用该列的视图）， 并且接着删除依赖于那些对象的 所有对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何依赖对象时拒绝删除列或者约束。这是默认行为。

trigger_name

一个要禁用或启用的触发器的名称。

ALL

禁用或者启用属于该表的所有触发器（如果有任何触发器是内部产生的约 束触发器则需要超级用户特权，例如那些被用来实现外键约束或者可延迟 一致性和排他约束的触发器）。

USER

禁用或者启用属于该表的所有触发器，内部产生的约束触发器（例如那些 被用来实现外键约束或者可延迟一致性和排他约束的触发器）除外。

index_name

一个现有索引的名称。

storage_parameter

一个表存储参数的名称。

value

一个表存储参数的新值。根据该参数，该值可能是一个数字或者一个词。

parent_table

要与这个表关联或者解除关联的父表。

new_owner

该表的新拥有者的用户名。

new_tablespace

要把该表移入其中的表空间的名称。

new_schema

要把该表移入其中的模式的名称。

partition_name

要被作为新分区附着到这个表或者从这个表上分离的表的名称。

partition_bound_spec

新分区的分区边界说明。更多细节请参考CREATE TABLE(7)中相同的语法。

注意

关键词COLUMN是噪声，可以被省略。

在使用ADD COLUMN增加一列并且指定了一个非易失性DEFAULT时，默认值会在该语句执行时计算并且结果会被保存在表的元数据中。这个值将被用于所有现有行的该列。如果没有指定DEFAULT，则使用NULL。在两种情况下都不需要重写表。

增加一个带有非易失性DEFAULT子句的列或者更改一个现有列的类型将 要求重写整个表及其索引。在更改一个现有列的类型时有一种例外：如果 USING子句不更改列的内容并且旧类型在二进制上与新类型可 强制转换或者是新类型上的一个未约束域，则不需要重写表。但是受影响列上 的任何索引仍必须被重建。对于一个大型表，表和/或索引重建可能会消耗相当多的时间，并且会临时要求差不多两倍的磁盘空间。

增加一个CHECK或者NOT NULL约束要求扫描 表以验证现有行符合该约束，但是不要求一次表重写。

类似地，在挂接一个新分区时，它需要被扫描以验证现有行满足该分区约束。

提供在一个ALTER TABLE中指定多个更改的选项的主要 原因就是多次表扫描或者重写可以因此被整合成一次。

扫描大型表以验证新的外键或检查约束可能需要很长时间，并且对表的其他更新将锁定，直到ALTER TABLE ADD CONSTRAINT命令被提交。 NOT VALID约束选项的主要目的是减少对并发更新添加约束的影响。 使用 NOT VALID，ADD CONSTRAINT命令不扫描表，可以立即提交。 在之后，VALIDATE CONSTRAINT命令会被发出以验证现有行是否满足约束。 验证步骤不需要锁定并发更新，因为它知道其他事务将强制执行它们插入或更新的行的约束;只有预先存在的行需要检查。 因此，验证在被更改的表上仅获得一个SHARE UPDATE EXCLUSIVE锁。 （如果约束是外键，则 ROW SHARE锁也需要约束的表引用。） 除了改进并发性外，在已知该表包含预先存在的违规行为的情况下使用 NOT VALID 和 VALIDATE CONSTRAINT也能有作用。 一旦约束就位，就不能再插入新的违规，，并且现有问题可以在空闲时纠正，直到VALIDATE CONSTRAINT最终完成。

DROP COLUMN形式不会在物理上移除列，而只是简 单地让它对 SQL 操作不可见。后续该表中的插入和更新操作将为该列存储 一个空值。因此，删除一个列很快，但是它不会立刻减少表所占的磁盘空间， 因为被删除列所占用的空间还没有被回收。随着现有列被更新，空间将被逐渐回收。

要强制立即回收被已删除列占据的空间，可以执行一种能导致全表重写的 ALTER TABLE形式。这种形式会导致重新构造每一个把被 删除列替换为空值的行。

ALTER TABLE的重写形式对于 MVCC 是不安全的。 在一次表重写之后，如果并发事务使用的是一个在重写发生前取得的 快照，该表将对这些并发事务呈现出空表的形态。详见 第 10.5 节 “提醒”。

SET DATA TYPE的USING选项能实际指定 涉及该列旧值的任何表达式。也就是说，它可以不但可以引用要被转换的列， 还可以引用其他列。这允许使用SET DATA TYPE语法完成十分 普遍的转换。由于这种灵活性，USING表达式不适合于列 的默认值（如果有），结果可能不是一个默认值所需的常量表达式。这意味着 在没有从旧类型到新类型的隐式或者赋值造型时，即便提供了一个 USING子句，SET DATA TYPE还是可能无法 转换默认值。在这种情况下，用DROP DEFAULT删除该默认值， 执行ALTER TYPE并且接着使用SET DEFAULT增加 一个合适的新默认值。类似的考虑也适用于涉及该列的索引和约束。

如果一个表有任何后代表，在不对后代表做相同操作的情况下，不允许在父表中增加列、重命名列或者更改列的类型。这确保了后代总是具有和父表匹配的列。类似地，如果不对所有后代上的CHECK约束进行重命名，就不能在父表中重命名该CHECK约束，这样CHECK约束也能在父表及其后代之间保持匹配（不过，这个限制不适用于基于索引的约束）。此外，因为从父表中选择也会从其后代中选择，父表上的约束不能被标记为有效，除非它在那些后代上也被标记为有效。在所有这些情况下，ALTER TABLE ONLY都将被拒绝。

只有当一个后代表的列不是从任何其他父表继承而来并且没有该列的独立定义时， 一次递归的DROP COLUMN操作才会移除该列。一次非递归 的DROP COLUMN（即 ALTER TABLE ONLY ... DROP COLUMN）不会移除 任何后代列，而是会把它们标记成独立定义的列。对于一个分区表，一个非递归的DROP COLUMN命令将会失败，因为一个表的所有分区都必须有和分区根节点相同的列。

标识列的动作（ADD GENERATED、SET等、DROP IDENTITY）以及动作TRIGGER、CLUSTER、OWNER和TABLESPACE不会递归到后代表上，也就是说它们执行时总是好像指定了ONLY一样。增加约束的动作仅对没有标记为NO INHERIT的CHECK约束递归。

不允许更改一个系统目录表的任何部分。

可用参数的进一步描述请见CREATE TABLE(7)。 第 2 章 数据定义有关于继承的进一步信息。

示例

要向一个表增加一个类型为varchar的列：

```
ALTER TABLE distributors ADD COLUMN address varchar(30);
```

要从表中删除一列：

```
ALTER TABLE distributors DROP COLUMN address RESTRICT;
```

要在一个操作中更改两个现有列的类型：

```
ALTER TABLE distributors
  ALTER COLUMN address TYPE varchar(80),
  ALTER COLUMN name TYPE varchar(100);
```

通过一个USING子句更改一个包含 Unix 时间戳的整数列为 timestamp with time zone：

```
ALTER TABLE foo
  ALTER COLUMN foo_timestamp SET DATA TYPE timestamp with time zone
  USING
    timestamp with time zone 'epoch' + foo_timestamp * interval '1 second';
```

同样的，当该列具有一个不能自动造型成新数据类型的默认值表达式时：

```
ALTER TABLE foo
  ALTER COLUMN foo_timestamp DROP DEFAULT,
  ALTER COLUMN foo_timestamp TYPE timestamp with time zone
  USING
    timestamp with time zone 'epoch' + foo_timestamp * interval '1 second',
  ALTER COLUMN foo_timestamp SET DEFAULT now();
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

重命名一个现有的表：

```
ALTER TABLE distributors RENAME TO suppliers;
```

重命名一个现有的约束：

```
ALTER TABLE distributors RENAME CONSTRAINT zipchk TO zip_check;
```

为一列增加一个非空约束：

ALTER TABLE distributors ALTER COLUMN street SET NOT NULL;

从一列移除一个非空约束：

ALTER TABLE distributors ALTER COLUMN street DROP NOT NULL;

向一个表及其所有子女增加一个检查约束：

ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5);

只向一个表增加一个检查约束（不为其子女增加）：

ALTER TABLE distributors ADD CONSTRAINT zipchk CHECK (char_length(zipcode) = 5) NO INHERIT;

（该检查约束也不会被未来的子女继承）。

从一个表及其子女移除一个检查约束：

ALTER TABLE distributors DROP CONSTRAINT zipchk;

只从一个表移除一个检查约束：

ALTER TABLE ONLY distributors DROP CONSTRAINT zipchk;

（该检查约束仍为子女表保留在某个地方）。

为一个表增加一个外键约束：

ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (address);

为一个表增加一个外键约束，并且尽量不要影响其他工作：

ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY (address) REFERENCES addresses (address) NOT VALID;
ALTER TABLE distributors VALIDATE CONSTRAINT distfk;

为一个表增加一个（多列）唯一约束：

ALTER TABLE distributors ADD CONSTRAINT dist_id_zipcode_key UNIQUE (dist_id, zipcode);

为一个表增加一个自动命名的主键约束，注意一个表只能拥有一个主键：

ALTER TABLE distributors ADD PRIMARY KEY (dist_id);

把一个表移动到一个不同的表空间：

ALTER TABLE distributors SET TABLESPACE fasttablespace;

把一个表移动到一个不同的模式：

```
ALTER TABLE myschema.distributors SET SCHEMA yourschema;
```

重建一个主键约束，并且在重建索引期间不阻塞更新：

```
CREATE UNIQUE INDEX CONCURRENTLY dist_id_temp_idx ON distributors (dist_id);
ALTER TABLE distributors DROP CONSTRAINT distributors_pkey,
   ADD CONSTRAINT distributors_pkey PRIMARY KEY USING INDEX dist_id_temp_idx;
```

要把一个分区挂接到一个范围分区表上：

```
ALTER TABLE measurement
   ATTACH PARTITION measurement_y2016m07 FOR VALUES FROM ('2016-07-01') TO
 ('2016-08-01');
```

要把一个分区挂接到一个列表分区表上：

```
ALTER TABLE cities
   ATTACH PARTITION cities_ab FOR VALUES IN ('a', 'b');
```

要把一个分区挂接到一个哈希分区表上：

```
ALTER TABLE orders
   ATTACH PARTITION orders_p4 FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

要把一个默认分区挂接到一个分区表上：

```
ALTER TABLE cities
   ATTACH PARTITION cities_partdef DEFAULT;
```

从一个分区表分离一个分区：

```
ALTER TABLE measurement
   DETACH PARTITION measurement_y2015m12;
```

兼容性

形式ADD（没有USING INDEX）、 DROP [COLUMN]、DROP IDENTITY、RESTART、 SET DEFAULT、SET DATA TYPE（没有USING）、 SET GENERATED以及SET *sequence_option*服从SQL标准。其他形式都是UXDB对SQL标准的扩展。此外，在单个**ALTER TABLE**命令中指定多个操作的能力是一种扩展。

ALTER TABLE DROP COLUMN可以被用来删除一个表的唯一的 列，从而留下一个零列的表。这是一种 SQL 的扩展，SQL 中不允许零列的表。

另见

CREATE TABLE(7)

名称

ALTER TABLESPACE — 更改一个表空间的定义

大纲

```
ALTER TABLESPACE name RENAME TO new_name
ALTER TABLESPACE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TABLESPACE name SET ( tablespace_option = value [, ... ] )
ALTER TABLESPACE name RESET ( tablespace_option [, ... ] )
```

描述

ALTER TABLESPACE可以被用于更改一个 表空间的定义。

要更改一个表空间的定义，必须拥有它。要修改拥有者，还必须是 新拥有角色的一个直接或间接成员（注意超级用户自动拥有这些特权）。

参数

name

一个现有表空间的名称。

new_name

该表空间的新名称。新名称不能以ux_开始，因为这类名称被 保留用于系统表空间。

new_owner

该表空间的新拥有者。

tablespace_option

要设置或者重置的一个表空间参数。当前，唯一可用的参数是 seq_page_cost、random_page_cost和effective_io_concurrency。 为一个特定表空间设定这两个参数值将覆盖规划器对从该表空间中的表读取 页面代价的估计值，这些估计值由具有相同名称配置参数建立。如果一个表空间位于一个比 其余 I/O 子系统更快或者更慢的磁盘上时，这些参数就能派上用场。

示例

将表空间index_space重命名为fast_raid：

```
ALTER TABLESPACE index_space RENAME TO fast_raid;
```

更改表空间index_space的拥有者：

```
ALTER TABLESPACE index_space OWNER TO mary;
```

兼容性

在 SQL 标准中没有 **ALTER TABLESPACE**语句。

另见

CREATE TABLESPACE(7), DROP TABLESPACE(7)

名称

ALTER TEXT SEARCH CONFIGURATION — 更改一个文本搜索配置的定义

大纲

```
ALTER TEXT SEARCH CONFIGURATION name
  ADD MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
  ALTER MAPPING FOR token_type [, ... ] WITH dictionary_name [, ... ]
ALTER TEXT SEARCH CONFIGURATION name
  ALTER MAPPING REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
  ALTER MAPPING FOR token_type [, ... ] REPLACE old_dictionary WITH new_dictionary
ALTER TEXT SEARCH CONFIGURATION name
  DROP MAPPING [ IF EXISTS ] FOR token_type [, ... ]
ALTER TEXT SEARCH CONFIGURATION name RENAME TO new_name
ALTER TEXT SEARCH CONFIGURATION name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER TEXT SEARCH CONFIGURATION name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH CONFIGURATION 更改一个文本搜索配置的定义。可以修改其从记号类型到词典的映射 或者更改该配置的名称或者拥有者。

要使用ALTER TEXT SEARCH CONFIGURATION， 必须是该配置的拥有者。

参数

name

一个现有文本搜索配置的名称（可以是模式限定的）。

token_type

由该配置的解析器发出的记号类型的名称。

dictionary_name

在其中查阅指定记号类型的文本搜索字典的名称。如果列出了 多个字典，会按照指定的顺序查阅它们。

old_dictionary

在映射中要替换的文本搜索字典的名称。

new_dictionary

被用来替代*old_dictionary* 的文本搜索字典的名称。

new_name

该文本搜索配置的新名称。

new_owner

该文本搜索配置的新拥有者。

new_schema

该文本搜索配置的新模式。

ADD MAPPING FOR形式会安装一些词典（用列表列出）用于在其中 查阅指定的记号类型。如果对任一记号类型已经有一个映射，则会发生错误。 ALTER MAPPING FOR形式做同样的事情，但是首先会移除这些记号 类型的任何现有映射。ALTER MAPPING REPLACE形式用*new_dictionary*来替换任何位 置上的*old_dictionary*。当出 现FOR时，只会为指定的记号类型做这样的事情。如果不出现 FOR，则会为该配置中所有的映射都这样做。 DROP MAPPING形式会移除指定记号类型的所有字典，导致该文本 搜索配置忽略这些类型。除非出现IF EXISTS，在那些记号类型没有 任何映射时会发生错误。

示例

下面的例子把my_config中任何位置上的english字典 替换为swedish字典。

```
ALTER TEXT SEARCH CONFIGURATION my_config
  ALTER MAPPING REPLACE english WITH swedish;
```

兼容性

在 SQL 标准中没有 `ALTER TEXT SEARCH CONFIGURATION` 语句。

另见

, DROP TEXT SEARCH CONFIGURATION(7)

名称

ALTER TEXT SEARCH DICTIONARY — 更改一个文本搜索字典的定义

大纲

```
ALTER TEXT SEARCH DICTIONARY name (
    option [ = value ] [, ... ]
)
ALTER TEXT SEARCH DICTIONARY name RENAME TO new_name
ALTER TEXT SEARCH DICTIONARY name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER TEXT SEARCH DICTIONARY name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH DICTIONARY更改一个 文本搜索字典的定义。可以更改该字典的与模板相关的选项，或者更改该 字典的名称或者拥有者。

要使用 **ALTER TEXT SEARCH DICTIONARY**，必须是超级用户。

参数

name

一个现有的文本搜索字典的名称（可以是模式限定的）。

option

要为这个字典设置的与模板相关的选项的名称。

value

用于一个模板相关选项的新值。如果等号和值被忽略，则会从该字典 中移除该选项之前的设置而允许使用默认值。

new_name

该文本搜索字典的新名称。

new_owner

该文本搜索字典的新拥有者。

new_schema

该文本搜索字典的新模式。

模板相关的选项可以以任何顺序出现。

示例

下面的命令更改一个基于 Snowball 的字典的停用词列表。其他参数保持不变。

ALTER TEXT SEARCH DICTIONARY my_dict (StopWords = newrussian);

下面的命令更改语言选项为dutch，并且完全移除停用词选项。

ALTER TEXT SEARCH DICTIONARY my_dict (language = dutch, StopWords);

下面的命令“更新”该字典的定义，但是实际没有做任何更改。

ALTER TEXT SEARCH DICTIONARY my_dict (dummy);

（之所以能这样做是因为选项移除代码在选项不存在时也不会抱怨）。 这种技巧在为该字典更改配置文件时有用：ALTER 将强制现有的数据库会话重读配置文件，否则如果会话之前已经读取过 就不会再次读取。

兼容性

在 SQL 标准中没有 ALTER TEXT SEARCH DICTIONARY语句。

另见

CREATE TEXT SEARCH DICTIONARY(7), DROP TEXT SEARCH DICTIONARY(7)

名称

ALTER TEXT SEARCH PARSER — 更改一个文本搜索解析器的定义

大纲

```
ALTER TEXT SEARCH PARSER name RENAME TO new_name
ALTER TEXT SEARCH PARSER name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH PARSER更改一个文本 搜索解析器的定义。当前，唯一支持的功能是更改该解析器的名称。

要使用**ALTER TEXT SEARCH PARSER**，必须是超级用户。

参数

name

一个现有文本搜索解析器的名称（可以是模式限定的）。

new_name

该文本搜索解析器的新名称。

new_schema

该文本搜索解析器的新模式。

兼容性

在 SQL 标准中没有 **ALTER TEXT SEARCH PARSER**语句。

另见

CREATE TEXT SEARCH PARSER(7), DROP TEXT SEARCH PARSER(7)

名称

ALTER TEXT SEARCH TEMPLATE — 更改一个文本搜索模板的定义

大纲

```
ALTER TEXT SEARCH TEMPLATE name RENAME TO new_name
ALTER TEXT SEARCH TEMPLATE name SET SCHEMA new_schema
```

描述

ALTER TEXT SEARCH TEMPLATE更改一个 文本搜索模板的定义。当前唯一支持的功能是更改该模板的名称。

要使用**ALTER TEXT SEARCH TEMPLATE**，必须是超级用户。

参数

name

一个现有文本搜索模板的名称（可以是模式限定的）。

new_name

该文本搜索模板的新名称。

new_schema

该文本搜索模板的新模式。

兼容性

在 SQL 标准中没有 **ALTER TEXT SEARCH TEMPLATE**语句。

另见

CREATE TEXT SEARCH TEMPLATE(7), DROP TEXT SEARCH TEMPLATE(7)

名称

ALTER TRIGGER — 更改一个触发器的定义

大纲

```
ALTER TRIGGER name ON table_name RENAME TO new_name
ALTER TRIGGER name ON table_name DEPENDS ON EXTENSION extension_name
```

描述

ALTER TRIGGER更改一个现有触发器的属性。 RENAME子句更改给定触发器的名称而不更改其定义。 DEPENDS ON EXTENSION子句把该触发器标记为依赖于 一个扩展，这样如果扩展被删除，该触发器也会被自动删除。

要更改一个触发器的属性，必须拥有该触发器所作用的表。

参数

name

要修改的一个现有触发器的名称。

table_name

这个触发器所作用的表的名称。

new_name

该触发器的新名称。

extension_name

该触发器所依赖的扩展的名称。

注解

临时启用或者禁用一个触发器的功能由ALTER TABLE(7)而不是 ALTER TRIGGER提供，因为ALTER TRIGGER 无法表示一次性启用或者禁用一个表上所有触发器的选项。

示例

要重命名一个现有的触发器：

```
ALTER TRIGGER emp_stamp ON emp RENAME TO emp_track_chgs;
```

要把一个触发器标记为依赖于一个扩展：

```
ALTER TRIGGER emp_stamp ON emp DEPENDS ON EXTENSION emplib;
```

兼容性

ALTER TRIGGER是一种 UXDB的 SQL 标准扩展。

另见

ALTER TABLE(7)

名称

ALTER TYPE — 更改一个类型的定义

大纲

```
ALTER TYPE name action [, ... ]
ALTER TYPE name OWNER TO { new_owner | CURRENT_USER | SESSION_USER }
ALTER TYPE name RENAME ATTRIBUTE attribute_name TO new_attribute_name [ CASCADE |
 RESTRICT ]
ALTER TYPE name RENAME TO new_name
ALTER TYPE name SET SCHEMA new_schema
ALTER TYPE name ADD VALUE [ IF NOT EXISTS ] new_enum_value [ { BEFORE |
 AFTER } neighbor_enum_value ]
ALTER TYPE name RENAME VALUE existing_enum_value TO new_enum_value
```

这里*action* 是以下之一：

```
   ADD ATTRIBUTE attribute_name data_type [ COLLATE collation ] [ CASCADE | RESTRICT ]
   DROP ATTRIBUTE [ IF EXISTS ] attribute_name [ CASCADE | RESTRICT ]
   ALTER ATTRIBUTE attribute_name [ SET DATA ] TYPE data_type [ COLLATE collation ]
[ CASCADE | RESTRICT ]
```

描述

ALTER TYPE更改一种现有类型的定义。 它有几种形式：

ADD ATTRIBUTE

这种形式为一种组合类型增加一个新属性，使用的语法和 CREATE TYPE(7)相同。

DROP ATTRIBUTE [IF EXISTS]

这种形式从一种组合类型删除一个属性。如果指定了 IF EXISTS并且该属性不存在，则不会抛出错误。 这种情况下会发出一个提示。

SET DATA TYPE

这种形式更改一种组合类型的一个属性类型。

OWNER

这种形式更改该类型的拥有者。

RENAME

这种形式更改该类型的名称或者一种组合类型的一个属性的名称。

SET SCHEMA

这种形式把该类型移动到另一个模式中。

ADD VALUE [IF NOT EXISTS] [BEFORE | AFTER]

这种形式为一种枚举类型增加一个新值。可以用BEFORE或者 AFTER一个现有值来指定新值在枚举顺序中的位置。 否则，新项会被增加在值列表的最后。

如果指定了IF NOT EXISTS，该类型已经包含新值时不会发生 错误：会发出一个提示但是不采取其他行动。否则，如果新值已经存在会发生错误。

RENAME VALUE

该形式重命名枚举类型的值。该值在枚举排序中的位置不受影响。 如果指定的值不存在或新名称已存在，则会发生错误。

ADD ATTRIBUTE、DROP ATTRIBUTE和ALTER ATTRIBUTE动作 可以被整合到一个多个修改组成的列表中，以便被平行应用。例如， 可以在一个命令中增加多个属性并且/或者修改多个属性的类型。

要使用ALTER TYPE，必须拥有该类型。要更改 一个类型的模式，还必须拥有新模式上的CREATE特权。要更改拥有者，还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该类型的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重 建该类型做不到的事情。不过，一个超级用户怎么都能更改任何类型的所有权。）。 要增加一个属性或者修改一个属性类型，还必须具有该数据类型上的 USAGE特权。

参数

name

要修改的一个现有类型的名称（可能被模式限定）。

new_name

该类型的新名称。

new_owner

该类型新拥有者的用户名。

new_schema

该类型的新模式。

attribute_name

要增加、修改或者删除的属性名称。

new_attribute_name

要被重命名的属性的新名称。

data_type

要增加的属性的数据类型，或者是要修改的属性的新类型。

new_enum_value

要被增加到一个枚举类型的值列表的新值，或将赋予现有值的新名称。 和所有枚举文本一样，它需要被引号引用。

neighbor_enum_value

一个现有枚举值，新值应该被增加在紧接着该枚举值之前或者 之后的位置上。和所有枚举文本一样，它需要被引号引用。

existing_enum_value

现有的应该重命名的枚举值。和所有的枚举文本一样，它需要被引号引用。

CASCADE

自动将操作传播到被更改类型的类型表及其后代。

RESTRICT

如果被更改的类型是类型表的类型，则拒绝该操作。这是默认设置。

注解

如果ALTER TYPE ... ADD VALUE（增加一个新值到枚举类型的形式） 不能在一个事务块中执行，新值不能被使用直到事务被提交之后。

涉及到一个新增加枚举值的比较有时会被只涉及原始枚举值的比较更慢。这通常 只会在利用BEFORE或者AFTER来把新值的 排序位置设置为非列表结尾的地方时发生。不过，有时候即使把新值增加到最后时 也会发生这种情况（如果创建了该枚举类型之后出现过 OID 计数器 “回卷”，就会发生这种情况）。这种减速通常不明显，但是如果它确实 带来了麻烦，通过删除并且重建该枚举类型或者转储并且重载整个数据库可以重新 得到最优性能。

示例

要重命名一个数据类型：

```
ALTER TYPE electronic_mail RENAME TO email;
```

把类型email的拥有者改为 joe：

```
ALTER TYPE email OWNER TO joe;
```

把类型email的模式改为 customers：

```
ALTER TYPE email SET SCHEMA customers;
```

增加一个新属性到一个类型：

```
ALTER TYPE compfoo ADD ATTRIBUTE f3 int;
```

在一个特定的排序位置上为一个枚举类型增加一个新值：

```
ALTER TYPE colors ADD VALUE 'orange' AFTER 'red';
```

重命名一个枚举值：

```
ALTER TYPE colors RENAME VALUE 'purple' TO 'mauve';
```

兼容性

增加和删除属性的变体是 SQL 标准的一部分，而其他变体是 UXDB 扩展。

另见

CREATE TYPE(7), DROP TYPE(7)

名称

ALTER USER — 更改一个数据库角色

大纲

ALTER USER *role_specification* [WITH] *option* [...]

其中 *option* 可以是：

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT connlimit
  | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
  | VALID UNTIL 'timestamp'
```

ALTER USER *name* RENAME TO *new_name*

```
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter { TO | = } { value | DEFAULT }
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
 SET configuration_parameter FROM CURRENT
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ]
 RESET configuration_parameter
ALTER USER { role_specification | ALL } [ IN DATABASE database_name ] RESET ALL
```

其中 *role_specification* 可以是：

```
   role_name
  | CURRENT_USER
  | SESSION_USER
```

描述

ALTER USER现在是 ALTER ROLE(7)的一种别名。

兼容性

ALTER USER语句是一种 UXDB扩展。SQL 标准把用户的定义留给 实现来处理。

另见

ALTER ROLE(7)

名称

ALTER USER MAPPING — 更改一个用户映射的定义

大纲

```
ALTER USER MAPPING FOR { user_name | USER | CURRENT_USER | SESSION_USER |
 PUBLIC }
   SERVER server_name
   OPTIONS ( [ ADD | SET | DROP ] option ['value'] [, ... ] )
```

描述

ALTER USER MAPPING更改一个用户映射的定义。

一个外部服务器的拥有者可以为任何用户修改用于该服务器的用户映射。还有， 如果一个用户被授予了外部服务器上的USAGE特权，它就能为它们自 己的用户名修改一个用户映射。

参数

user_name

该映射的用户名。CURRENT_USER和USER匹配当前 用户的名称。PUBLIC被用来匹配系统中所有当前以及未来的用户名。

server_name

该用户映射的服务器名。

OPTIONS ([ADD | SET | DROP] *option* ['*value*'] [, ...])

为该用户映射更改选项。新选项会覆盖任何之前指定的选项。ADD、 SET和DROP指定要被执行的动作。如果没有显式地指定 操作，将假定为ADD。选项名称必须为唯一，该服务器的外部数据包装 器也会验证选项。

示例

为服务器 foo的用户映射bob更改口令：

```
ALTER USER MAPPING FOR bob SERVER foo OPTIONS (SET password 'public');
```

兼容性

ALTER USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。有一点细微的语法问题：该标准会忽略FOR关键字。 由于CREATE USER MAPPING以及 DROP USER MAPPING都在类似的位置上使用 FOR，并且 IBM DB2 （作为其他主要的 SQL/MED 实现）也在 ALTER USER MAPPING中要求该关键字，因此为了一致性和互 操作性，UXDB 在这里的实现与标准不同。

另见

CREATE USER MAPPING(7), DROP USER MAPPING(7)

名称

ALTER VIEW — 更改一个视图的定义

大纲

```
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name SET DEFAULT expression
ALTER VIEW [ IF EXISTS ] name ALTER [ COLUMN ] column_name DROP DEFAULT
ALTER VIEW [ IF EXISTS ] name OWNER TO { new_owner | CURRENT_USER |
 SESSION_USER }
ALTER VIEW [ IF EXISTS ] name RENAME TO new_name
ALTER VIEW [ IF EXISTS ] name SET SCHEMA new_schema
ALTER VIEW [ IF EXISTS ] name SET ( view_option_name [= view_option_value] [, ... ] )
ALTER VIEW [ IF EXISTS ] name RESET ( view_option_name [, ... ] )
```

描述

ALTER VIEW更改一个视图的多种辅助属性（如果想要 修改视图的查询定义，应使用CREATE OR REPLACE VIEW）。

要使用ALTER VIEW，必须拥有该视图。要更改一个视图的模式， 还必须具有新模式上的CREATE特权。要更改拥有者，还必须 是新拥有角色的一个直接或者间接成员，并且该角色必须具有该视图的模式上的 CREATE特权（这些限制强制修改拥有者不能做一些通过删除和重建视图做不到的事情。不过，一个超级用户怎么都能更改任何视图的所有权。）。

参数

name

一个现有视图的名称（可以是模式限定的）。

IF EXISTS

该视图不存在时不要抛出一个错误。这种情况下会发出一个提示。

SET/DROP DEFAULT

这些形式为一个列设置或者移除默认值。对于任何在该视图上的 INSERT或者UPDATE命令，一个视图列的默认值 会在引用该视图的任何规则或触发器之前被替换进来。因此，该视图的默认值将会 优先于来自底层关系的任何默认值。

new_owner

该视图的新拥有者的用户名。

new_name

该视图的新名称。

new_schema

该视图的新模式。

SET (*view_option_name* [= *view_option_value*] [, ...])
RESET (*view_option_name* [, ...])

设置或者重置一个视图选项。当前支持的选项有：

check_option（string）

更改该视图的检查选项。值必须是local 或者cascaded。

security_barrier（boolean）

更改该视图的安全屏障属性。值必须是一个布尔值，如 true或者false。

注解

由于历史原因，**ALTER TABLE**也可以用于视图，但是 只允许等效于以上形式的**ALTER TABLE**变体用于视图。

示例

把视图foo重命名为 bar：

```
ALTER VIEW foo RENAME TO bar;
```

要为一个可更新视图附加一个默认列值：

```
CREATE TABLE base_table (id int, ts timestamptz);
CREATE VIEW a_view AS SELECT * FROM base_table;
ALTER VIEW a_view ALTER COLUMN ts SET DEFAULT now();
INSERT INTO base_table(id) VALUES(1);  -- ts will receive a NULL
INSERT INTO a_view(id) VALUES(2);  -- ts will receive the current time
```

兼容性

ALTER VIEW是一种UXDB 的 SQL 标准扩展。

另见

CREATE VIEW(7), DROP VIEW(7)

名称

ANALYZE — 收集有关一个数据库的统计信息

大纲

ANALYZE [(*option* [, ...])] [*table_and_columns* [, ...]]
ANALYZE [VERBOSE] [*table_and_columns* [, ...]]

其中*option*可以是：

VERBOSE [*boolean*]
SKIP_LOCKED [*boolean*]

*table_and_columns*是：

table_name [(*column_name* [, ...])]

描述

ANALYZE收集一个数据库中的表的内容的统计信息，并且将结果存储在ux_statistic系统目录中。接下来，查询规划器会使用这些统计信息来帮助确定查询最有效的执行计划。

如果没有*table_and_columns*列表，则**ANALYZE**处理当前用户有权分析的当前数据库中的每个表和物化视图。使用列表，**ANALYZE**仅处理那些表。还可以给出表的列名列表，在这种情况下，仅收集这些列的统计信息。

当选项列表用括号括起来时，选项可以按任何顺序来写。 带括号的语法是在UXDB 2.1.1.2版本中添加的；不带括号的语法已弃用。

参数

VERBOSE

允许显示进度消息。

SKIP_LOCKED

规定**ANALYZE**在开始处理一个关系时不应等待任何冲突的锁被释放:如果关系不能无需等待立即锁定，则跳过该关系。 请注意即使采用此选项，**ANALYZE**在打开关系的索引或从分区、表继承子级和某些类型的外表获取样本行时仍可能阻塞。 此外，当**ANALYZE**通常处理指定分区表的所有分区时，如果分区表上有一个冲突的锁，这个选项将导致**ANALYZE**跳过所有分区表。

boolean

规定所选的选项打开或关闭。 您可以写TRUE、ON或1以启用该选项，或者是FALSE、OFF或0来禁用它。 *boolean* 值可以被省略，在假定为TRUE的情况下。

table_name

要分析的一个指定表的名称（可以是模式限定的）。如果省略，则分析当前数据库中的所有常规表、分区表和物化视图（但不包含外部表）。 如果指定的表是分区表，则整个分区表的继承统计信息和各个分区的统计信息都将更新。

column_name

要分析的一个指定列的名称。默认是所有列。

输出

当指定了VERBOSE时，ANALYZE会发出进度消息来指示当前正在处理哪个表。还会打印有关那些表的多种统计信息。

注解

要分析表，通常必须是该表的所有者或超级用户。不过，数据库所有者可以分析数据库中的所有表，共享目录除外。 （对共享目录的限制意味着真正的数据库范围的ANALYZE只能由超级用户执行。） ANALYZE将跳过调用用户没有分析权限的任何表。

只有被显式选中时才会分析外部表。并非所有外部数据包装器都支持ANALYZE。如果表的包装器不支持ANALYZE，该命令会打印一个警告并且什么也不做。

在默认的UXDB配置中，自动清理守护进程会在表第一次载入数据或者用常规操作改变时负责表的自动分析。当启用自动清理时，定期运行ANALYZE是个好主意，或者可以在表内容做了大的修改后运行ANALYZE。准确的统计信息将帮助规划器选择最合适的查询计划，从而提升查询处理的速度。主读数据库的一般策略是在一天中使用量最低时运行一次VACUUM(7)和ANALYZE（如果有大量的更新动作则是不够的）。

ANALYZE只要求目标表上的一个读锁，因此它可以和表上的其他动作并行。

ANALYZE收集的统计信息通畅包括每列中最常见值的列表以及展示每列中近似数据分布的一个直方图。如果ANALYZE认为这些东西无趣（例如在一个唯一键列中，没有共同值）或者该列的数据类型不支持合适的操作符，以上工作都会被省略。

对于大型的表，ANALYZE会对表内容做随机采样而不是检查每一行。这允许在很少的时间内完成对大型表的分析。不过要注意，这些统计信息只是近似值，并且即使实际表内容没有改变，每次运行ANALYZE时统计信息都会有微小地改变。这可能会导致EXPLAIN(7)显示的规划器估算代价有小的改变。在很少的情况下，这会非决定性地导致规划器的查询计划选择在ANALYZE运行后改变。为了避免这种情况，可以按照下文所述提高ANALYZE所收集的统计信息量。

通过调整default_statistics_target配置变量可以控制分析量，对每个列可以用ALTER TABLE ... ALTER COLUMN ... SET STATISTICS设置每列的统计信息目标（见ALTER TABLE(7)）。目标值会设置最常用值列表中的最大项数以及直方图中的最大容器数。默认目标值是 100，可以把它调大或者调小在规划器估计值精度和ANALYZE花费的时间以及ux_statistic所占空间之间做出平衡。特别地，将统计信息目标设置为零会禁用该列的统计信息收集。在查询的WHERE、GROUP BY或者ORDER BY子句中从不出现的列上这样做会有所帮助，因为规划器用不上这些列上的统计信息。

被分析的列中最大的统计信息目标决定了为准备统计信息要采样的表行数。增加该目标会导致做ANALYZE所需的时间和空间成比例增加。

ANALYZE所估算的值之一是出现在每个列中的可区分值。因为只会检查行的一个子集，即便使用最大的统计信息目标，这种估计有时也可能很不精确。如果这种不精确导致不好的查询计划，可以手工确定一个更精确的值并且用ALTER TABLE ... ALTER COLUMN ... SET (n_distinct = ...)设置该值（见ALTER TABLE(7)）。

如果被分析的表有一个或者更多子女，ANALYZE将会收集两次统计信息：一次只对父表的行收集，第二次则在父表及其所有子女表的行上收集。在规划需要遍历整个继承树的查询时需要第二个统计信息集。不过，在决定是否触发表上的自动分析时，自动清理后台进程将只考虑父表本身

上的插入和更新。如果该表很少被插入或者更新，只有手工运行**ANALYZE**时才会把继承统计信息更新到最新。

如果任何子表是外部表并且其外部数据包装器不支持**ANALYZE**，在收集继承统计信息时会忽略那些子表。

如果被分析的表不完全为空，**ANALYZE**将不会为该表记录新统计信息。任何现有统计信息将会被保留。

兼容性

SQL 标准中没有**ANALYZE**语句。

另见

VACUUM(7)

名称

BEGIN — 开始一个事务块

大纲

BEGIN [WORK | TRANSACTION] [*transaction_mode* [, ...]]

其中 *transaction_mode* 是以下之一：

```
  ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
  READ WRITE | READ ONLY
  [ NOT ] DEFERRABLE
```

描述

BEGIN开始一个事务块，也就是说所有 **BEGIN**命令之后的所有语句将被在一个 事务中执行，直到给出一个显式的COMMIT(7)或 者ROLLBACK(7)。默认情况下（没有 **BEGIN**）， UXDB在 “自动提交”模式中执行事务，也就是说每个语句都 在自己的事务中执行并且在语句结束时隐式地执行一次提交（如果执 行成功，否则会完成一次回滚）。

在一个事务块内的语句会执行得更快，因为事务的开始/提交也要求可观 的 CPU 和磁盘活动。在进行多个相关更改时，在一个事务内执行多个语 句也有助于保证一致性：在所有相关更新还没有完成之前，其他会话将不 能看到中间状态。

如果指定了隔离级别、读/写模式或者延迟模式，新事务也会有那些特性， 就像执行了SET TRANSACTION(7)一样。

参数

WORK
TRANSACTION

可选的关键词。它们没有效果。

这个语句其他参数的含义请参考 SET TRANSACTION(7)。

注解

START TRANSACTION(7)具有和**BEGIN** 相同的功能。

使用COMMIT(7)或者 ROLLBACK(7)来终止一个事务块。

在已经在一个事务块中时发出**BEGIN**将惹出一个警告 消息。事务状态不会被影响。要在一个事务块中嵌套事务，可以使用保 存点（见SAVEPOINT(7)）。

由于向后兼容的原因，连续的 *transaction_modes* 之间的逗号可以被省略。

示例

开始一个事务块：

BEGIN;

兼容性

BEGIN是一种 UXDB语言扩展。它等效于 SQL 标准的命令START TRANSACTION(7)，它的参考页 包含额外的兼容性信息。

DEFERRABLE *transaction_mode* 是一种UXDB语言扩展。

附带地，BEGIN关键词被用于嵌入式 SQL 中的一种不同目的。在移植数据库应用时，应注意对待事务语义。

另见

COMMIT(7)， ROLLBACK(7)， START TRANSACTION(7)， SAVEPOINT(7)

名称

CALL — 调用一个过程

大纲

CALL *name* ([*argument*] [, ...])

简介

CALL执行一个过程。

如果过程有任何输出参数，则会返回一个结果行，返回这些参数的值。

参数

name

过程的名称（可以被方案限定）。

argument

过程调用的一个输入参数。函数和过程调用语法的完整细节（包括参数的使用）请参考第 1.3 节 “调用函数”。

注解

用户必须有过程上的EXECUTE特权才能调用它。

要调用一个函数（不是过程），应使用**SELECT**。

如果在事务块中执行**CALL**，那么被调用的过程不能执行事务控制语句。只有当**CALL**在其自身的事务中执行时，才允许过程执行事务控制语句。

PL/uxSQL 以不同的方式处理**CALL**中的输出参数。

示例

```
CALL do_db_maintenance();
```

兼容性

CALL符合SQL标准。

另见

CREATE PROCEDURE(7)

名称

CHECKPOINT — 强制一个事务日志检查点

大纲

CHECKPOINT

描述

一个检查点是事务日志序列中的一个点，在该点上所有数据文件　都已经被更新为反映日志中的信息。所有数据文件将被刷写到磁盘。

CHECKPOINT命令在发出时强制一个 立即的检查点，而不用等待由系统规划的常规检查点（由 设置控制）。 CHECKPOINT不是用来在普通操作中 使用的命令。

如果在恢复期间执行，CHECKPOINT 命令将强制一个重启点 而不是写一个新检查点。

只有超级用户能够调用CHECKPOINT。

兼容性

CHECKPOINT命令是一种 UXDB语言扩展。

名称

CLOSE — 关闭一个游标

大纲

CLOSE { *name* | ALL }

描述

CLOSE释放与一个已打开游标相关 的资源。在游标被关闭后，不允许在其上做后续的操作。当不再 需要使用一个游标时应该关闭它。

当一个事务被**COMMIT**或者 **ROLLBACK**终止时，每一个非可保持 的已打开游标会被隐式地关闭。当创建一个可保持游标的事务通过 **ROLLBACK**中止时，该可保持游标会 被隐式地关闭。如果该创建事务成功地提交，可保持游标会保持打 开，直至执行一个显式的**CLOSE**或者 客户端连接断开。

参数

name

要关闭的已打开游标的名称。

ALL

关闭所有 已打开的游标。

注解

UXDB没有一个显式的 **OPEN**游标语句，一个游标在被声明时 就被认为是打开的。使用DECLARE(7)语句可以声明游标。

通过查询ux_cursors 系统视图可以看到所有可用的游标。

如果一个游标在一个保存点之后关闭，并且后来回滚到了这个保存点， 那么**CLOSE**不会被回滚，也就是说回滚 后游标仍然保持关闭。

示例

关闭游标liahona：

```
CLOSE liahona;
```

兼容性

CLOSE完全服从 SQL 标准。 **CLOSE ALL**是一种UXDB 扩展。

另见

DECLARE(7), FETCH(7), MOVE(7)

名称

CLUSTER — 根据一个索引聚簇一个表

大纲

CLUSTER [VERBOSE] *table_name* [USING *index_name*]
CLUSTER [VERBOSE]

描述

CLUSTER指示UXDB 基于*index_name* 所指定的索引来聚簇 *table_name* 所指定的表。该索引必须已经定义在 *table_name*上。

当一个表被聚簇时，会基于索引信息对它进行物理上的排序。聚簇是一种 一次性的操作：当表后续被更新时，更改没有被聚簇。也就是说，不会尝 试根据新行或者被更新行的索引顺序来存储它们（如果想这样做，可以周 期性地通过发出该命令重新聚簇。还有，把表的 fillfactor存储参数设置为小于 100% 有助于在更 新期间保持聚簇顺序，因为如果空间足够会把被更新行保留在同一个页面 中）。

当一个表被更新时，UXDB 会记住它是按照哪个索引聚簇的。形式 **CLUSTER** *table_name* 会使用前面所用的同一个索引对表重新聚簇。也可以使用 CLUSTER或者ALTER TABLE(7) 的SET WITHOUT CLUSTER形式把索引设置为可用于 未来的聚簇操作，或者清除任何之前的设置。

不带任何参数的**CLUSTER**会重新聚簇调用用 户所拥有的当前数据库中已经被聚簇过的表（如果是超级用户调用，则是 所有已被聚簇过的表）。这种形式的 **CLUSTER**不能在一个事务块内执行。

当一个表被聚簇时，会在其上要求一个ACCESS EXCLUSIVE锁。这会阻止任何其他数据库操作（包括读和写） 在**CLUSTER**结束前在该表上操作。

参数

table_name

一个表的名称（可能是模式限定的）。

index_name

一个索引的名称。

VERBOSE

在每一个表被聚簇时打印一个进度报告。

注解

在随机访问一个表中的行时，表中数据的实际顺序是无关紧要的。 不过，如果想要更多地访问其中一些数据，并且有一个索引把它 们分组在一起，使用**CLUSTER**就会带 来好处。如果从一个表中要求一个范围的被索引值或者多行都匹 配的一个单一值，**CLUSTER**就会有所 帮助，因为一旦该索引标识出了第一个匹配行所在的表页，所有其 他匹配行很可能就在同一个表页中，并且因此节省了磁盘访问并且 提高了查询速度。

CLUSTER可以使用指定索引上的一次索引扫描 或者遵循排序的一次顺序扫描（如果索引是 B树）对表重新排序。 它将会基于规划器代价参数以及可用的统计信息来选择较快的方法。

在使用索引扫描时，会创建该表的一份临时拷贝，其中包含按索引顺序 排列的表数据。该表上每一个索引的临时拷贝也会被创建。因此，在磁 盘上需要至少等于表尺寸加上索引尺寸的综合的空闲空间。

在使用顺序扫描以及排序时，也会创建一个临时排序文件，因此临时空 间需求的峰值也就是表尺寸的两倍外加索引尺寸。这种方法通常比索引 扫描方法更快，但是如果磁盘空间需求是不能接受的，可以通过临时 地把enable_sort设置为off来禁 用这种选择。

建议在聚簇前把 maintenance_work_mem设 置为一个合理地比较大的值（但是不能超过可以用于CLUSTER操作的 RAM 容量）。

因为规划器会记录有关表顺序的统计信息，建议在新近被聚簇的表上 运行ANALYZE(7)。否则，规划器可能会产生很差 的查询计划。

因为CLUSTER会记住哪些索引被聚簇， 我们可以第一次手动聚簇想要聚簇的表，然后设置一个定期运行的维护 脚本，其中执行不带任何参数的CLUSTER，这样那些 表就会被周期性地重新聚簇。

示例

基于索引employees_ind聚簇表 employees：

```
CLUSTER employees USING employees_ind;
```

使用之前用过的同一个索引聚簇employees表：

```
CLUSTER employees;
```

对数据库中以前被聚簇过的所有表进行聚簇：

```
CLUSTER;
```

兼容性

在 SQL 标准中没有CLUSTER语句。

名称

COMMENT — 定义或者更改一个对象的注释

大纲

```
COMMENT ON
{
 ACCESS METHOD object_name |
 AGGREGATE aggregate_name ( aggregate_signature ) |
 CAST (source_type AS target_type) |
 COLLATION object_name |
 COLUMN relation_name.column_name |
 CONSTRAINT constraint_name ON table_name |
 CONSTRAINT constraint_name ON DOMAIN domain_name |
 CONVERSION object_name |
 DATABASE object_name |
 DOMAIN object_name |
 EXTENSION object_name |
 EVENT TRIGGER object_name |
 FOREIGN DATA WRAPPER object_name |
 FOREIGN TABLE object_name |
 FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
 INDEX object_name |
 LARGE OBJECT large_object_oid |
 MATERIALIZED VIEW object_name |
 OPERATOR operator_name (left_type, right_type) |
 OPERATOR CLASS object_name USING index_method |
 OPERATOR FAMILY object_name USING index_method |
 POLICY policy_name ON table_name |
 [ PROCEDURAL ] LANGUAGE object_name |
 PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
 PUBLICATION object_name |
 ROLE object_name |
 ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
 RULE rule_name ON table_name |
 SCHEMA object_name |
 SEQUENCE object_name |
 SERVER object_name |
 STATISTICS object_name |
 SUBSCRIPTION object_name |
 TABLE object_name |
 TABLESPACE object_name |
 TEXT SEARCH CONFIGURATION object_name |
 TEXT SEARCH DICTIONARY object_name |
 TEXT SEARCH PARSER object_name |
 TEXT SEARCH TEMPLATE object_name |
 TRANSFORM FOR type_name LANGUAGE lang_name |
 TRIGGER trigger_name ON table_name |
 TYPE object_name |
 VIEW object_name
} IS 'text'
```

其中 *aggregate_signature* 是：

* |
[*argmode*] [*argname*] *argtype* [, ...] |
[[*argmode*] [*argname*] *argtype* [, ...]] ORDER BY [*argmode*] [*argname*] *argtype* [, ...]

描述

COMMENT存储关于一个数据库对象的注释。

对每一个对象只保存一个注释字符串，因此为了修改一段注释，对同一个对象 发出一个新的**COMMENT**命令。要移除一段注释，可在文 本字符串的位置上写上NULL。当对象被删除时，其注释 也会被自动删除。

对大部分类型的对象，只有对象的拥有者可以设置注释。角色没有拥有者，因此 COMMENT ON ROLE的规则是必须作为一个超级用户来对一个 超级用户角色设置注释，或者具有CREATEROLE特权来对非超级用 户角色设置注释。同样的，访问方法也没有拥有者，必须作为一个超级用户来 对一个访问方法设置注释。当然，一个超级用户可以对任何东西设置注释。

使用uxsql的**\d** 命令家族可以查看注释。其他检索注释的用户接口可以构建在 uxsql使用的内建函数之上，即 obj_description、col_description 以及shobj_description （见表 6.81 “注释信息函数”）。

参数

object_name
relation_name.*column_name*
aggregate_name
constraint_name
function_name
operator_name
policy_name
procedure_name
routine_name
rule_name
trigger_name

要被注释的对象的名称。表、聚集、排序方式、转换、域、外部表、函数、 索引、操作符、操作符类、操作符族、存储过程、例程、序列、统计信息、文本搜索对象、类型和视图的名称可以被模式限定。在注释一列时， *relation_name*必须 引用一个表、视图、组合类型或者外部表。

table_name
domain_name

当在一个约束、触发器、规则或者策略上创建一段注释时，这些参数指定在其上定义 该对象的表或域的名称。

source_type

造型的源数据类型的名称。

target_type

造型的目标数据类型的名称。

argmode

一个函数，存储过程或者聚集函数的参数的模式：IN、 OUT、INOUT或者VARIADIC。 如果被省略，默认值是IN。注意 **COMMENT**并不真正关心 OUT参数，因为决定函数的身份只需要输入参数。因此 列出IN、INOUT和VARIADIC 参数就足够了。

argname

一个函数，存储过程或者聚集函数参数的名称。注意 **COMMENT**并不真正关心参数名称， 因为决定函数的身份只需要参数数据类型。

argtype

一个函数，存储过程或者聚集函数参数的数据类型。

large_object_oid

大对象的 OID。

left_type
right_type

操作符的参数的数据类型（可以是模式限定的）。对一个前缀后后缀操作符 的缺失参数可以写NONE。

PROCEDURAL

这是一个噪声词。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

text

写成一个字符串的新注释。如果要删除注释，写成NULL。

注解

当前对查看注释没有安全机制：任何连接到一个数据库的用户能够看到 该数据库中所有对象的注释。对于数据库、角色、表空间这类共享对象， 注释被全局存储，因此连接到集簇中任何数据库的任何用户可以看到共 享对象的所有注释。因此，不要在注释中放置有安全性风险的信息。

示例

为表mytable附加一段注释：

```
COMMENT ON TABLE mytable IS 'This is my table.';
```

移除它：

```
COMMENT ON TABLE mytable IS NULL;
```

更多的一些例子：

```
COMMENT ON ACCESS METHOD rtree IS 'R-Tree access method';
COMMENT ON AGGREGATE my_aggregate (double precision) IS 'Computes sample variance';
COMMENT ON CAST (text AS int4) IS 'Allow casts from text to int4';
COMMENT ON COLLATION "fr_CA" IS 'Canadian French';
COMMENT ON COLUMN my_table.my_column IS 'Employee ID number';
COMMENT ON CONVERSION my_conv IS 'Conversion to UTF8';
COMMENT ON CONSTRAINT bar_col_cons ON bar IS 'Constrains column col';
COMMENT ON CONSTRAINT dom_col_constr ON DOMAIN dom IS 'Constrains col of domain';
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON DOMAIN my_domain IS 'Email Address Domain';
COMMENT ON EXTENSION hstore IS 'implements the hstore data type';
COMMENT ON FOREIGN DATA WRAPPER mywrapper IS 'my foreign data wrapper';
COMMENT ON FOREIGN TABLE my_foreign_table IS 'Employee Information in other database';
COMMENT ON FUNCTION my_function (timestamp) IS 'Returns Roman Numeral';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee ID';
COMMENT ON LANGUAGE plpython IS 'Python support for stored procedures';
COMMENT ON LARGE OBJECT 346344 IS 'Planning document';
COMMENT ON MATERIALIZED VIEW my_matview IS 'Summary of order history';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two texts';
COMMENT ON OPERATOR - (NONE, integer) IS 'Unary minus';
COMMENT ON OPERATOR CLASS int4ops USING btree IS '4 byte integer operators for btrees';
COMMENT ON OPERATOR FAMILY integer_ops USING btree IS 'all integer operators for btrees';
COMMENT ON POLICY my_policy ON mytable IS 'Filter rows by users';
COMMENT ON PROCEDURE my_proc (integer, integer) IS 'Runs a report';
COMMENT ON ROLE my_role IS 'Administration group for finance tables';
COMMENT ON RULE my_rule ON my_table IS 'Logs updates of employee records';
COMMENT ON SCHEMA my_schema IS 'Departmental data';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON SERVER myserver IS 'my foreign server';
COMMENT ON STATISTICS my_statistics IS 'Improves planner row estimations';
COMMENT ON TABLE my_schema.my_table IS 'Employee Information';
COMMENT ON TABLESPACE my_tablespace IS 'Tablespace for indexes';
COMMENT ON TEXT SEARCH CONFIGURATION my_config IS 'Special word filtering';
COMMENT ON TEXT SEARCH DICTIONARY swedish IS 'Snowball stemmer for Swedish
 language';
COMMENT ON TEXT SEARCH PARSER my_parser IS 'Splits text into words';
COMMENT ON TEXT SEARCH TEMPLATE snowball IS 'Snowball stemmer';
COMMENT ON TRANSFORM FOR hstore LANGUAGE plpythonu IS 'Transform between hstore and
 Python dict';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for RI';
COMMENT ON TYPE complex IS 'Complex number data type';
COMMENT ON VIEW my_view IS 'View of departmental costs';
```

兼容性

SQL 标准中没有**COMMENT**命令。

名称

COMMIT — 提交当前事务

大纲

COMMIT [WORK | TRANSACTION] [AND [NO] CHAIN]

描述

COMMIT提交当前事务。所有由该 事务所作的更改会变得对他人可见并且被保证在崩溃发生时仍能 持久。

参数

WORK
TRANSACTION

可选的关键词。它们没有效果。

AND CHAIN

如果指定了AND CHAIN，则立即启动与刚刚完成的事务具有相同事务特征（参见SET TRANSACTION(7)）的新事务。 否则，没有新事务被启动。

注解

使用ROLLBACK(7)中止一个事务。

当不在一个事务内时发出**COMMIT**不会 产生危害，但是它会产生一个警告消息。当**COMMIT AND CHAIN** 不在事务内时是一个错误。

示例

要提交当前事务并且让所有更改持久化：

```
COMMIT;
```

兼容性

命令**COMMIT**符合 SQL 标准。 表单COMMIT TRANSACTION为UXDB扩展。

另见

BEGIN(7), ROLLBACK(7)

名称

COMMIT PREPARED — 提交一个早前为两阶段提交预备的事务

大纲

COMMIT PREPARED *transaction_id*

描述

COMMIT PREPARED提交一个处于预备状态的事务。

参数

transaction_id

要被提交的事务的事务标识符。

注解

要提交一个预备的事务，必须是原先执行该事务的同一用户或者超级用户。 但是不需要处于执行该事务的同一会话中。

这个命令不能在一个事务块中执行。该预备事务将被立刻提交。

ux_prepared_xacts 系统视图中列出了所有当前可用的预备事务。

例子

提交由事务标识符foobar标识的事务：

```
COMMIT PREPARED 'foobar';
```

兼容性

COMMIT PREPARED是一种 UXDB扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA）， 但是那些系统的 SQL 方面未被标准化。

另见

PREPARE TRANSACTION(7), ROLLBACK PREPARED(7)

名称

COPY — 在一个文件和一个表之间复制数据

大纲

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | PROGRAM 'command' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]
    [ WHERE condition ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | PROGRAM 'command' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```

其中 *option* 可以是下列之一：

```
    FORMAT format_name
    FREEZE [ boolean ]
    DELIMITER 'delimiter_character'
    NULL 'null_string'
    HEADER [ boolean ]
    QUOTE 'quote_character'
    ESCAPE 'escape_character'
    FORCE_QUOTE { ( column_name [, ...] ) | * }
    FORCE_NOT_NULL ( column_name [, ...] )
    FORCE_NULL ( column_name [, ...] )
    ENCODING 'encoding_name'
```

描述

COPY在 UXDB表和标准文件系统文件之间 移动数据。COPY TO把一个表的内容复制 到一个文件，而COPY FROM 则从一个文件复制数据到一个表（把数据追加到表中原有数 据）。COPY TO也能复制一个 SELECT查询的结果。

如果指定了一个列列表，COPY TO将只把指定列的数据复制到文件。对于COPY FROM，文件中的每个字段将按顺序插入到指定列中。COPY FROM命令的列列表中没有指定的表列则会采纳其默认值。

带一个文件名的COPY指示 UXDB服务器直接从一个文件读取 或者写入到一个文件。该文件必须是 UXDB用户（运行服务器的用户 ID） 可访问的并且应该以服务器的视角来指定其名称。当指定了 PROGRAM时，服务器执行给定的命令并且从该程序的标准 输出读取或者写入到该程序的标准输入。该程序必须以服务器的视角指定，并且 必须是UXDB用户可执行的。在指定 STDIN或者STDOUT时，数据会通过客 户端和服务器之间的连接传输。

参数

table_name

一个现有表的名称（可以是模式限定的）。

column_name

可选的要被复制的列列表。如果没有指定列列表，则该表的所有列除了生成的列都会被复制。

query

其结果要被复制的SELECT(7)、 VALUES(7)、 INSERT(7)、UPDATE(7)或者 DELETE(7)命令。注意查询周围的圆括号是必要的。

对于INSERT、UPDATE以及 DELETE查询，必须提供一个 RETURNING 子句并且 目标关系不能具有会扩展成多条语句的条件规则、 ALSO规则或者INSTEAD规则。

filename

输入或者输出文件的路径名。一个输入文件的名称可以是一个绝对或相对路径， 但一个输出文件的名称必须是绝对路径。Windows 用户可能需要使用一个 E"字符串并且双写路径名称中使用的任何反斜线。

PROGRAM

一个要执行的命令。在COPY FROM中，输入 将从该命令的标准输出读取，而在COPY TO中，输出会 写入到该命令的标准输入。

注意该命令是由 shell 调用，因此如果需要传递任何来自不可信来源的参数给 shell 命令，必须剥离那些可能对 shell 有特殊意义的特殊字符。出于安全原因，最好使用一个固定的命令字符串，或者至少避免传递任何用户输入到其中。

STDIN

指定输入来自客户端应用。

STDOUT

指定输出会去到客户端应用。

boolean

指定选中的选项是应该被关闭还是打开。可以写TRUE、 ON或1来启用选项，写FALSE、OFF或0禁用它。 *boolean*值也可以被省略， 那样会假定为TRUE。

FORMAT

选择要读取或者写入的数据格式： text、 csv（逗号分隔值）或者binary。 默认是text。

FREEZE

请求复制已经完成了行冻结的数据，就好像在运行 VACUUM FREEZE命令之后复制。这是为了初始 数据载入的性能而设计的。只有被载入表已经在当前子事务中被创建 或截断、该事务中没有游标打开并且该事务没有持有更旧的快照时， 行才会被冻结。目前无法在分区表上执行COPY FREEZE。

注意一旦成功地载入，所有其他会话将能立即看到该数据。这违背了 普通的 MVCC 可见性规则，指定该选项的用户应该注意这可能会导致 的潜在问题。

DELIMITER

指定分隔文件每行中各列的字符。文本格式中默认是一个制表符， 而CSV格式中默认是一个逗号。这必须是一个单一 的单字节字符。使用binary格式时不允许这个选项。

NULL

指定表示一个空值的字符串。文本格式中默认是 \N（反斜线-N），CSV格式中默认 是一个未加引用的空串。在不想区分空值和空串的情况下，即使在文本 格式中也可能更喜欢空串。使用binary格式时不允许这 个选项。

> **注意**
>
> 在使用COPY FROM时，任何匹配 这个串的数据项将被存储为空值，因此应该确定使用的是和 COPY TO时相同的串。

HEADER

指定文件包含标题行，其中有每一列的名称。在输出时，第一行包含 来自表的列名。在输入时，第一行会被忽略。只有使用 CSV格式时才允许这个选项。

QUOTE

指定一个数据值被引用时使用的引用字符。默认是双引号。 这必须是一个单一的单字节字符。只有使用 CSV格式时才允许这个选项。

ESCAPE

指定应该出现在一个匹配QUOTE值的数据字符之前 的字符。默认和QUOTE值一样（这样如果引用字符 出现在数据中，它会被双写）。这必须是一个单一的单字节字符。 只有使用CSV格式时才允许这个选项。

FORCE_QUOTE

强制必须对每个指定列中的所有非NULL值使用引用。 NULL输出不会被引用。如果指定了*， 所有列的非NULL值都将被引用。只有在 COPY TO中使用CSV格式时才允许 这个选项。

FORCE_NOT_NULL

不要把指定列的值与空值串匹配。在空值串就是空串的默认情况下， 这意味着空串将被读作长度为零的字符串而不是空值（即使它们没有 被引用）。只有在COPY FROM中使用 CSV格式时才允许这个选项。

FORCE_NULL

将指定列的值与空值串匹配（即使它已经被加上引号），并且在找到 匹配时将该值设置为NULL。在空值串就是空串的默认 情况下，这会把一个被引用的空串转换为 NULL。 只有在COPY FROM中使用 CSV格式时才允许这个选项。

ENCODING

指定文件被以*encoding_name*编码。如果省略 这个选项，将使用当前的客户端编码。详见下文的注解。

WHERE

WHERE子句是可选的，其一般形式是：

WHERE *condition*

其中*condition*是计算结果为boolean类型的任意表达式。任何不满足此条件的行都不会插入到表中。在用实际的行值替换任何变量引用时，如果该行返回true，则该行满足条件。

目前，在WHERE表达式中不允许使用子查询，并且值的计算不会看到COPY本身所做的任何更改（当表达式包含对VOLATILE函数的调用时，这一点很重要）。

输出

在成功完成时，一个COPY命令会返回一个形为

COPY *count*

的命令标签。 *count*是被复制 的行数。

> **注意**
>
> 如果命令不是COPY ... TO STDOUT或者等效的 uxsql元命令\copy ... to stdout， uxsql将只打印这个命令标签。这是为了防止弄混 命令标签和刚刚打印的数据。

注解

COPY TO只能被用于纯粹的表，不能用于视图。 不过可以写COPY (SELECT * FROM *viewname*) TO ...来拷贝一个视图的当前内容。

COPY FROM可以被用于普通表、外部表、分区表或者具有INSTEAD OF INSERT触发器的视图。

COPY只处理提到的表，它不会从子表复制 数据或者复制数据到子表中。例如 COPY *table* TO 会显示与SELECT * FROM ONLY *table*相同的数据。而COPY (SELECT * FROM *table*) TO ... 可以被用来转储一个继承层次中的所有数据。

必须拥有被COPY TO读取的表上的选择特权， 以及被COPY FROM插入的表上的插入特权。 拥有在命令中列出的列上的特权就足够了。

如果对表启用了行级安全性，相关的SELECT策略将应用于COPY *table* TO语句。当前，有行级安全性的表不支持COPY FROM。不过可以使用等效的INSERT语句。

COPY命令中提到的文件会被服务器（而不是 客户端应用）直接读取或写入。因此它们必须位于数据库服务器（不是客户 端）的机器上或者是数据库服务器可以访问的。它们必须是 UXDB用户（运行服务器的用户 ID）可访问的并且是可读或者可写的。类似地，用PROGRAM 指定的命令也会由服务器（不是客户端应用）直接执行，它也必须是 UXDB用户可以执行的。 只允许数据库超级用户或者授予了默认角色ux_read_server_files、ux_write_server_files及ux_execute_server_program之一的用户COPY一个文件或者命令， 因为它允许读取或者写入服务器有特权访问的任何文件或者运行服务器有特权访问的程序。

不要把COPY和 uxsql指令 \copy 弄混。\copy会调用 COPY FROM STDIN或者COPY TO STDOUT，然后读取/存储一个 uxsql客户端可访问的文件中的数据。 因此，在使用\copy时，文件的可访 问性和访问权利取决于客户端而不是服务器。

我们推荐在COPY中使用的文件名总是 指定为一个绝对路径。在COPY TO的 情况下服务器会强制这一点，但是对于 COPY FROM可以选择从一个用相对 路径指定的文件中读取。该路径将根据服务器进程（而不是客户端） 的工作目录（通常是集簇的数据目录）解释。

用PROGRAM执行一个命令可能会受到操作系统 的访问控制机制（如 SELinux）的限制。

COPY FROM将调用目标表上的任何触发器 和检查约束。但是它不会调用规则。

对于标识列，COPY FROM命令将总是写上输入数据中提供的列值，这和INSERT的选项OVERRIDING SYSTEM VALUE的行为一样。

COPY输入和输出受到 DateStyle的影响。为了确保到其他 可能使用非默认DateStyle设置的 UXDB安装的可移植性，在使用 COPY TO前应该把 DateStyle设置为ISO。避免转储把 IntervalStyle设置为 sql_standard的数据也是一个好主意，因为负的区间值可能会 被具有不同IntervalStyle设置的服务器解释错误。

即使数据会被服务器直接从一个文件读取或者写入一个文件而不通过 客户端，输入数据也会被根据ENCODING选项或者当前 客户端编码解释，并且输出数据会被根据ENCODING或 者当前客户端编码进行编码。

COPY会在第一个错误处停止操作。这在 COPY TO的情况下不会导致问题，但是 在COPY FROM中目标表将已经收到了一 些行。这些行将不会变得可见或者可访问，但是它们仍然占据磁盘空间。 如果在一次大型的复制操作中出现错误，这可能浪费相当可观的磁盘空间。 可能希望调用VACUUM来恢复被浪费的 空间。

FORCE_NULL和FORCE_NOT_NULL可以被同时 用在同一列上。这会导致把已被引用的空值串转换为空值并且把未引用的空值 串转换为空串。

文件格式

文本格式

在使用text格式时，读取或写入的是一个文本文件， 其中每一行就是表中的一行。一行中的列被定界字符分隔。列值 本身是由输出函数产生的或者是可被输入函数接受的属于每个属性 数据类型的字符串。在为空值的列的位置使用指定的空值串。如果 输入文件的任何行包含比预期更多或者更少的列， COPY FROM将会抛出一个错误。

数据的结束可以表示为一个只包含反斜线-点号（\.）的 单一行。从一个文件读取时，数据结束标记并不是必要的，因为文件 结束符就已经足够用了。只有使用 3.0 客户端协议之前的客户端应用 复制数据时才需要它。

反斜线字符（\）可以被用在 COPY数据中来引用被用作行或者列定界符的 字符。特别地，如果下列字符作为一个列值的一部分出现，它们 必须被前置一个反斜线：反斜线本身、新行、回车以及 当前的定界符字符。

COPY TO会不加任何反斜线返回指定的空值串。 相反，COPY FROM会在移除反斜线之前把输入与空值串相匹配。因此，一个空值串（例如\N）不会与实 际的数据值\N（它会被表示为\\N）搞混。

COPY FROM识别下列特殊的反斜线序列：

序列	表示
\b	退格 (ASCII 8)

序列	表示
\f	换页 (ASCII 12)
\n	新行 (ASCII 10)
\r	回车 (ASCII 13)
\t	制表 (ASCII 9)
\v	纵向制表 (ASCII 11)
digits	反斜线后跟一到三个十进制位表示该数字代码对应的字符
\x*digits*	反斜线加x后跟一到三个十六进制位表示该数字代码对应的字符

当前，COPY TO不会发出一个十进制或十六进制位 反斜线序列，但是它确实把上面列出的其他序列用于那些控制字符。

任何上述表格中没有提到的其他反斜线字符将被当作表示其本身。不过，要注意 增加不必要的反斜线，因为那可能意外地产生一个匹配数据结束标记（ \.）或者空值串（默认是\N）的字符串。这些字符串 将在完成任何其他反斜线处理之前被识别。

强烈建议产生COPY数据的应用把数据新行和回车分别 转换为\n和\r序列。当前可以把一个数据回车表示为 一个反斜线和回车，把一个数据新行表示为一个反斜线和新行。不过，未来的发行 可能不会接受这些表示。如果在不同的机器之间（例如从 Unix 到 Windows） 传输COPY文件，它们也很容易受到破坏。

COPY TO将用一个 Unix 风格的新行（ “\n”）终止每一行。运行在 Microsoft Windows 上的服务器则会输出回车/新行（“\r\n”），不过只对 COPY到一个服务器文件这样做。为了做到跨平台一致， COPY TO STDOUT总是发送“\n”而 不管服务器平台是什么。COPY FROM能够处理以 新行、回车或者回车/新行结尾的行。为了减少由作为数据的未加反斜线的新行 或者回车带来的风险，如果输出中的行结束并不完全相似， COPY FROM将会抱怨。

CSV 格式

这种格式选项被用于导入和导出很多其他程序（例如电子表格）使用的逗号 分隔值（CSV）文件格式。不同于 UXDB标准文本格式使用的转义 规则，它产生并且识别一般的 CSV 转义机制。

每个记录中的值用DELIMITER字符分隔。如果值包含 定界符字符、QUOTE字符、NULL字符串、 一个回车或者换行字符，那么整个值会被加上QUOTE字符 作为前缀或者后缀，并且在该值内QUOTE字符或者 ESCAPE字符的任何一次出现之前放上转义字符。在输出 指定列中非NULL值时，还可以使用 FORCE_QUOTE来强制加上引用。

CSV格式没有标准方式来区分NULL值和空字符串。 UXDB的COPY用引用来处理 这种区分工作。NULL被按照NULL参数字符串输出 并且不会被引用，而匹配NULL参数字符串的非NULL 值会被加上引用。例如，使用默认设置时，NULL被写作一个未 被引用的空字符串，而一个空字符串数据值会被写成带双引号（""）。 值的读取遵循类似的规则。可以用FORCE_NOT_NULL来防止 对指定列的NULL输入比较。还可以使用 FORCE_NULL把带引用的空值字符串数据值转换成NULL。

因为反斜线在CSV格式中不是一种特殊字符，数据结束标记 \.也可以作为一个数据值出现。为了避免任何解释误会，在 一行上作为孤项出现的\.数据值输出时会自动被引用，并且 输入时如果被引用，则不会被解释为数据结束标记。如果正在载入一个由 另一个应用创建的文件并且其中具有一个未被引用的列且可能具有 \.值，可能需要在输入文件中引用该值。

> **注意**
>
> CSV格式中，所有字符都是有意义的。一个被空白或者其他 非DELIMITER字符围绕的引用值将包括那些字符。在导入 来自用空白填充CSV行到固定长度的系统的数据时，这可能 会导致错误。如果出现这种情况，在导入数据到 UXDB之前，可能需要预处理该 CSV文件以移除拖尾的空白。

> **注意**
>
> CSV 格式将识别并且产生带有包含嵌入的回车和换行的引用值的 CSV 文件。因此文件并不限于文本格式文件的每个表行一行的形式。

> **注意**
>
> 很多程序会产生奇怪的甚至偶尔是不合常理的 CSV 文件，因此该文件 格式更像是一种习惯而不是标准。因此可能会碰到一些无法使用这种 机制导入的文件，并且COPY也可能产生其他程序无 法处理的文件。

二进制格式

binary格式选项导致所有数据被以二进制格式 而不是文本格式存储/读取。它比文本和CSV格式要 快一些，但是二进制格式文件在不同的机器架构和 UXDB版本之间的可移 植性要差些。还有，二进制格式与数据格式非常相关。例如不能从 一个smallint列中输出二进制数据并且把它读入到一个 integer列中，虽然这样做在文本格式中是可行的。

binary文件格式由一个文件头、零个或者更多个包含 行数据的元组以及一个文件尾构成。头部和数据都以网络字节序表示。

> **注意**
>
> UXDB发行使用一种不同的二进制文件格式。

文件头

文件头由 15 字节的固定域构成，后面跟着一个变长的头部扩展区。 固定域有：

签名

11-字节的序列UXCOPY\n\377\r\n\0 — 注意零字节是签名的一个必要的部分（该签名是为了能容易地发现文件被无法正确处理 8 位字符编码的传输所破坏。这个签名将被行尾翻译过滤器、删除零字节、删除高位或者奇偶修改等改变）。

标志域

32-位整数位掩码，用以表示该文件格式的重要方面。位被编号为 从 0 （LSB）到 31（MSB）。 注意这个域以网络字节序存放（最高有效位在前），所有该文件格式 中使用的整数域都是这样。16-31 位被保留用来表示严重的文件格式 问题， 读取者如果在这个范围内发现预期之外的被设置位，它应该 中止。0-15 位被保留用来表示向后兼容的格式问题，读取者应该简单 地略过这个范围内任何预期之外的被设置位。当前只定义了一个标志 位，其他位必须为零：

位 16

如果为 1，表示数据中包含 OID；如果为 0，则不包含。UXDB不再支持Oid系统列，但是格式仍然包含该指示符。

头部扩展区长度

32-为整数，表示头部剩余部分的以字节计的长度，不包括其本身。 当前，这个长度为零，并且其后就紧跟着第一个元组。未来对该 格式的更改可能会允许在头部中表示额外的数据。如果读取者不知 道要对头部扩展区数据做什么，可以安静地跳过它。

头部扩展区域被预期包含一个能自我解释的块的序列。 该标志域并不想告诉读取者扩展数据是什么。详细的 头部扩展内容的设计留给后来的发行去做。

这种设计允许向后兼容的头部增加（增加头部扩展块或者设置低位标志位）以及 非向后兼容的更改（设置高位标志位来表示这类更改并且在需要时向扩展区域 中增加支持数据）。

元组

每一个元组由一个表示元组中域数量的 16 位整数计数开始（当前，一个表中 的所有元组都应该具有相同的计数，但是这可能不会总是为真）。然后是元组 中的每一个域，它是一个 32 位的长度字，后面则跟随着这么多个字节的域数 据（长度字不包括其本身，并且可以是零）。作为一种特殊情况，-1 表示一个 NULL 域值。在 NULL 情况下，后面不会跟随值字节。

在域之间没有对齐填充或者任何其他额外的数据。

当前，一个二进制格式文件中的所有数据值都被假设为二进制格式（格式代码一）。 可以预见未来的扩展可能会增加一个允许独立指定各列的格式代码的头部域。

要为实际的元组数据决定合适的二进制格式，应该参考 UXDB源码，特别是用于各列 数据类型的*send和*recv函数（通常可 以在源码的src/backend/utils/adt/目录中找到 这些函数）。

如果文件中包含 OID，OID 域会紧跟在域计数字之后。它是一个普通域， 不过它没有被包含在域计数中。注意UXDB当前版本不支持oid系统列。

文件尾

文件位由一个包含 -1 的 16 位整数字组成。这很容易与一个 元组的域计数字区分开。

如果一个域计数字不是 -1 也不是期望的列数，读取者应该报告错误。 这提供了一种针对某种数据不同步的额外检查。

示例

下面的例子使用竖线（|）作为域定界符把一个表复制到客户端：

```
COPY country TO STDOUT (DELIMITER '|');
```

从一个文件中复制数据到country表中：

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

只把名称以 ’A’ 开头的国家复制到一个文件中：

COPY (SELECT * FROM country WHERE country_name LIKE 'A%') TO '/usr1/proj/bray/sql/a_list_countries.copy';

要复制到一个压缩文件中，可以用管道把输出导到一个外部压缩程序：

COPY country TO PROGRAM 'gzip > /usr1/proj/bray/sql/country_data.gz';

这里是一个适合于从STDIN复制到表中的数据：

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
ZM      ZAMBIA
ZW      ZIMBABWE
```

注意每一行上的空白实际是一个制表符。

下面是用二进制格式输出的相同数据。该数据是用 Unix 工具 `od -c`过滤后显示的。该表具有三列， 第一列类型是char(2)，第二列类型是text， 第三列类型是integer。所有行在第三列都是空值。

```
0000000   P   G   C   O   P   Y  \n 377  \r  \n  \0  \0  \0  \0  \0  \0
0000020  \0  \0  \0  \0 003  \0  \0  \0 002   A   F  \0  \0  \0 013   A
0000040   F   G   H   A   N   I   S   T   A   N 377 377 377 377  \0 003
0000060  \0  \0  \0 002   A   L  \0  \0  \0 007   A   L   B   A   N   I
0000100   A 377 377 377 377  \0 003  \0  \0  \0 002   D   Z  \0  \0  \0
0000120 007   A   L   G   E   R   I   A 377 377 377 377  \0 003  \0  \0
0000140  \0 002   Z   M  \0  \0  \0 006   Z   A   M   B   I   A 377 377
0000160 377 377  \0 003  \0  \0  \0 002   Z   W  \0  \0  \0  \b   Z   I
0000200   M   B   A   B   W   E 377 377 377 377 377 377
```

兼容性

SQL 标准中没有`COPY`语句。

名称

CREATE ACCESS METHOD — 定义一种新的访问方法

大纲

```
CREATE ACCESS METHOD name
  TYPE access_method_type
  HANDLER handler_function
```

简介

CREATE ACCESS METHOD创建一种新的访问方法。

访问方法名称在数据库中必须唯一。

只有超级用户可以定义新的访问方法。

参数

name

要创建的访问方法的名称。

access_method_type

这个子句指定要定义的访问方法的类型。当前只支持TABLE和INDEX。

handler_function

*handler_function*是一个之前已注册的函数的名称（可能被模式限定），该函数表示要创建的访问方法。处理器函数必须被声明为接收一个单一的internal类型的参数，并且它的返回类型取决于访问方法的类型；　　对于TABLE访问方法，它必须是table_am_handler，而对于INDEX访问方法，它必须是index_am_handler。 处理器函数必须实现的 C 级别 API 取决于访问方法的类型。

示例

用处理器函数heptree_handler创建一种索引访问方法heptree：

```
CREATE ACCESS METHOD heptree TYPE INDEX HANDLER heptree_handler;
```

兼容性

CREATE ACCESS METHOD是一种UXDB扩展。

另见

DROP ACCESS METHOD(7), CREATE OPERATOR CLASS(7), CREATE OPERATOR FAMILY(7)

名称

CREATE AGGREGATE — 定义一个新的聚集函数

大纲

```
CREATE [ OR REPLACE ] AGGREGATE name ( [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

CREATE [ OR REPLACE ] AGGREGATE name ( [ [ argmode ] [ argname ] arg_data_type [ , ... ] ]
                    ORDER BY [ argmode ] [ argname ] arg_data_type [ , ... ] ) (
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , INITCOND = initial_condition ]
    [ , HYPOTHETICAL ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)
```

或者旧的语法

```
CREATE [ OR REPLACE ] AGGREGATE name (
    BASETYPE = base_type,
    SFUNC = sfunc,
    STYPE = state_data_type
    [ , SSPACE = state_data_size ]
    [ , FINALFUNC = ffunc ]
    [ , FINALFUNC_EXTRA ]
    [ , FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , COMBINEFUNC = combinefunc ]
```

```
    [ , SERIALFUNC = serialfunc ]
    [ , DESERIALFUNC = deserialfunc ]
    [ , INITCOND = initial_condition ]
    [ , MSFUNC = msfunc ]
    [ , MINVFUNC = minvfunc ]
    [ , MSTYPE = mstate_data_type ]
    [ , MSSPACE = mstate_data_size ]
    [ , MFINALFUNC = mffunc ]
    [ , MFINALFUNC_EXTRA ]
    [ , MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE } ]
    [ , MINITCOND = minitial_condition ]
    [ , SORTOP = sort_operator ]
)
```

描述

CREATE AGGREGATE定义一个新的聚集函数。CREATE OR REPLACE AGGREGATE将定义新的聚合函数或替换现有定义。 在发布中已经包括了一些基本的和常用的聚集函数；它们的文档请见第 6.20 节 “聚集函数”。 如果要定义一个新类型或者需要一个还没有被提供的聚集函数，那么CREATE AGGREGATE就可以被用来提供想要的特性。

在替换现有定义时，参数类型、结果类型和直接参数的数量可能不会更改。 此外，新定义的类型（普通聚合、有序集聚合或假设集聚合）必须与旧定义相同。

如果给定了一个模式名（例如CREATE AGGREGATE myschema.myagg ...），那么该聚集会被创建在指定的模式中。否则它 会被创建在当前模式中。

一个聚集函数需要用它的名称和输入数据类型标识。同一个模式中的两个聚集 可以具有相同的名称，只要它们在不同的输入类型上操作即可。一个聚集的名称 和输入数据类型必须与同一模式中的每一个普通函数区分开。这种行为与普通函 数名的重载完全一样（见CREATE FUNCTION(7)）。

一个简单的聚集函数由一个或者多个普通函数组成： 一个状态转移函数 *sfunc*和一个可选的最终计算函数 *ffunc*。 它们被这样使用：

sfunc(internal-state, next-data-values) ---> next-internal-state
ffunc(internal-state) ---> aggregate-value

UXDB创建一个数据类型 *stype*的临时变量来 保持聚集的当前内部状态。对每一个输入行，聚集参数值会被计算并且状态 转移函数会被调用，它用当前状态值和新参数值计算一个新的内部状态值。 等所有行都被处理完后，最终函数会被调用一次来计算该聚集的返回值。如果 没有最终函数，则最终的状态值会被返回。

一个聚集函数可以提供一个初始条件，即一个用于内部状态值的初始值。它被 作为一个类型text的值指定并且存储在数据库中，但是它必须 是状态值数据类型的一个常量的合法外部表示。如果没有提供，则状态值从空值 开始。

如果状态转移函数被声明为“strict”，那么不能用空值输入来 调用它。如果有这种转移函数，聚集将按照下面的行为执行。带有任何空值的 行会被忽略（函数不被调用并且之前的状态值被保持）。如果初始状态值就是 空值，那么碰到第一个没有空值的行时，状态值会被替换成第一个参数值，并且 对于每一个后续的没有空值的行都会调用该转移函数。这对实现 max这样的聚集很方便。注意只有当 *state_data_type* 和第一个 *arg_data_type*相同时， 这种行为才可用。当这些类型不同时，必须提供一个非空初始条件或者使用 一个非严格转移函数。

如果状态转移函数不是严格的，那么在碰到每个输入行时都将会调用它，并且 它必须自行处理空值输入和空状态值。这允许聚集的作者完全控制该聚集如何 处理空值。

如果最终函数被声明为“strict”，那么当最终状态值为空时将 不会调用它，而是自动地返回一个空结果（当然，这只是严格函数的普通行为）。 在任何情况下最终函数都可以返回一个空值。例如，avg的最终函数会在看到零个 输入行时返回空。

有时候把最终函数声明成不仅采用状态值还采用对应于聚集输入值的额外参数 是有用的。这样做的主要原因是，如果最终函数是多态的，那么状态值的数据 类型将不适合于用来确定结果类型。这些额外的参数总是以 NULL 形式传递 （因此使用FINALFUNC_EXTRA选项时，最终函数不能是严格的）， 但尽管如此它们都是合法参数。例如，最终函数可以利用get_fn_expr_argtype来标识当前调用中的实际参数类型。

一个聚集可以 选择支持moving-aggregate mode。这要求指定 MSFUNC、MINVFUNC以及 MSTYPE参数，并且参数MSSPACE、 MFINALFUNC、MFINALFUNC_EXTRA和MINITCOND是可选的。除了MINVFUNC， 这些参数的工作都和对应的不带M的简单聚集参数相似，它们 定义了包括一个逆向转移函数的聚集的一种独立实现。

在参数列表中带有ORDER BY的语法会创建一种被称为 有序集聚集的特殊聚集类型。如果指定了 HYPOTHETICAL，则会创建一个 假想集聚集。这些聚集以依赖排序的方法在排好序 的值上操作，因此指定一个输入排序顺序是调用过程的重要一环。还有，它们 可以有直接参数，这类参数只对每次聚集计算一次，而不是对 每一个输入行计算一次。假想集聚集是有序集聚集的一个子类，其中一些直接 参数要求在数量和类型上都匹配被聚集的参数列。这允许这些直接参数的值被 当做一个附加的“假想”行被加入到聚集输入行的集合中。

一个聚集可以支持 部分聚集。这要求指定COMBINEFUNC参数。 如果*state_data_type* 为 internal，通常也可以提供SERIALFUNC和 DESERIALFUNC参数，这样可以让并行聚集成为可能。注意， 该聚集还必须被标记为PARALLEL SAFE以启用并行聚集。

行为与MIN或MAX相似的聚集有时可以通过 直接查看一个索引而不是扫描每一个输入行来优化。如果这个聚集可以被这样 优化，请通过指定一个排序操作符来指出。基本要求是，该 聚集必须得出由该操作符产生的排序顺序中的第一个元素，换句话说：

SELECT agg(col) FROM tab;

必须等价于：

SELECT col FROM tab ORDER BY col USING sortop LIMIT 1;

进一步的假定是该聚集忽略空输入，并且当且仅当没有非空输入时它才会返回 一个空结果。通常，一种数据类型的<操作符是 MIN的合适的排序操作符，而>是 MAX的合适的排序操作符。注意，除非指定的操作符是一个 B-树索引操作符类的“小于”或者“大于” 策略成员，优化将永远不会产生实际效果。

要能够创建一个聚集函数，必须具有参数类型、状态类型和返回类型上的 USAGE特权，还有在支持函数上的 EXECUTE特权。

参数

name

要创建的聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或者VARIADIC（聚集函数　　不支持OUT参数）。如果忽略，默认值是IN。 只有最后一个参数能被标记为VARIADIC。

argname

一个参数的名称。当前这只用于文档的目的。如果被忽略，该参数就没有名称。

arg_data_type

这个聚集函数操作的一个输入数据类型。要创建一个零参数的聚集函数，可以 写一个*来替代参数说明的列表（这类聚集的一个例子是 count(*)）。

base_type

在CREATE AGGREGATE的旧语法中，输入数据类型是由 一个basetype参数指定而不是写在聚集名之后。注意这种语法　只允许一个输入参数。要用这种语法定义一个零参数的聚集函数，把 basetype指定为"ANY"（不是*）。 有序集聚集不能用旧语法定义。

sfunc

要为每一个输入行调用的状态转移函数名。对于一个正常的 *N*-参数的聚集函数， *sfunc*必须接收 *N*+1 个参数， 第一个参数的类型是*state_data_type*而其余的参数匹配 该聚集被声明的输入数据类型。该函数必须返回一个类型为 *state_data_type* 的值。这个函数会采用当前的状态值以及当前的输入数据值，并且返回下一个 状态值。

对于有序集（包括假想集）聚集，状态转移函数只接收当前的状态值和聚集参数，　但无需直接参数。否则它就和其他转移函数一样了。

state_data_type

聚集的状态值的数据类型。

state_data_size

聚集的状态值的近似平均尺寸（以字节计）。如果这个参数被忽略或者为零，　将使用一个基于*state_data_type*的默认估计值。规划器　使用这个值来估计一个分组聚集查询所需的内存。只有估计哈希表能够放在 work_mem大小的内存中时，规划器才会对这类查询 使用哈希聚集。因此，对这个参数设置大的值会阻止使用哈希聚集。

ffunc

最终函数的名称，该函数在所有输入行都被遍历之后被调用来计算聚集的结果。　对于一个常规聚集，这个函数必须只接受一个类型为*state_data_type*的单一参数。该聚集　的返回数据类型被定义为这个函数的返回类型。如果没有指定　*ffunc*，则结束状态值　被用作聚集的结果，并且返回类型为*state_data_type*。

对于有序集（包括假想集）聚集，最终函数不仅接收最终状态值，还会接收所　有直接参数的值。

如果指定了FINALFUNC_EXTRA，则除了最终状态值和任何直接　参数之外，最终函数还接收额外的对应于该聚集的常规（聚集）参数的 NULL 值。 这主要用于在定义了一个多态聚集时允许正确地决定聚集的结果类型。

FINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

此选项指定最终函数是否为不会修改参数的纯函数。READ_ONLY表示它不会修改；其他两个值表示它可能会更改迁移状态值。请参见“注解”一节 以获取更多详细信息。除了有序集合的聚合使用默认值READ_WRITE，其他默认值均为READ_ONLY。

combinefunc

*combinefunc*函数可以被 有选择地指定以允许聚集函数支持部分聚集。如果提供这个函数，*combinefunc*必须组合两个 *state_data_type*值，每一个 都包含在输入值某个子集上的聚集结果，它会产生一个新的 *state_data_type*来表示在 两个输入集上的聚集结果。这个函数可以被看做是一个 *sfunc*，和后者在一个个体 输入行上操作并且把它加到运行聚集状态上不同，这个函数是把另一个聚集状态加 到运行状态上。

*combinefunc*必须被声明为 有两个*state_data_type*参数 并且返回一个*state_data_type* 值。这个函数可以有选择性地被标记为“strict”。在被标记的情况下， 当任何一个输入状态为空时，将不会调用该函数，而是把另一个状态当作正确的结果。

对于*state_data_type*为 internal的聚集函数， *combinefunc*不能为 strict。这种情况下，*combinefunc*必须确保 正确地处理空状态并且被返回的状态能被恰当地存储在聚集内存上下文中。

serialfunc

*state_data_type*为 internal的一个聚集函数可以参与到并行聚集中，当且仅当它具有一个*serialfunc*函数，该函数 必须把聚集状态序列化成一个bytea值以传送给另一个进程。这个函数 必须有一个单一的internal类型参数并且返回类型bytea。 相应地也需要一个*deserialfunc*。

deserialfunc

把一个之前序列化后的聚集状态反序列化为 *state_data_type*。这个函数 必须有两个类型分别为bytea和internal的参数，并且产生 一个类型internal的结果（注意：第二个类型为internal的 参数是无用的，但是为了类型安全的原因还是要求有该参数）。

initial_condition

状态值的初始设置。这必须是以数据类型*state_data_type*能够接受的形式出现 的一个字符串常量。如果没有指定，状态值会从空值开始。

msfunc

前向状态转移函数的名称，在移动聚集模式中会为每个输入行调用这个函数。它 非常像常规的转移函数，不过它的第一个参数和结果类型是 *mstate_data_type*，这可能与*state_data_type*不同。

minvfunc

在移动聚集模式中用到的逆向状态转移函数的名称。这个函数与 *msfunc*具有相同的参数和结果类型，但是它被用于从当前聚集 状态中移除一个值，而不是向其中增加一个值。逆向转移函数必须具有和前向状态 转移函数相同的严格性属性。

mstate_data_type

使用移动聚集模式时，用于聚集状态值的数据类型。

mstate_data_size

使用移动聚集模式时，聚集状态值的近似平均尺寸（以字节计）。它的作用和 *state_data_size*相同。

mffunc

使用移动聚集模式时用到的最终函数名称，当所有输入行都被遍历后会调用它来 计算聚集的结果。它的工作和*ffunc*一样，但是它的第一个参 数类型是*mstate_data_type*并且额外的空参数要通过书写 MFINALFUNC_EXTRA来指定。*mffunc* 或者*mstate_data_type*决定的聚集结果类型必须匹配由聚集 的常规实现所确定的类型。

MFINALFUNC_MODIFY = { READ_ONLY | SHAREABLE | READ_WRITE }

此选项类似于FINALFUNC_MODIFY，只是它描述了移动聚集最终函数的行为。

minitial_condition

使用移动聚集模式时，状态值的初始设置。它的作用和 *initial_condition*一样。

sort_operator

一个MIN- 类或者MAX-类聚集的相关 排序操作符。这只是一个操作符名称（可能被模式限定）。这个操作符被 假定为具有和该聚集（必须是一个单一参数的常规聚集）相同的输入数据 类型。

PARALLEL = { SAFE | RESTRICTED | UNSAFE }

PARALLEL SAFE、PARALLEL RESTRICTED和PARALLEL UNSAFE的含义和 CREATE FUNCTION(7)中的相同。如果一个聚集被标记为 PARALLEL UNSAFE（默认）或者PARALLEL RESTRICTED，将不会考虑将它并行化。注意 规划器不会参考聚集的支持函数的并行安全性标记，它只会考虑聚集本身 的这类标记。

HYPOTHETICAL

只用于有序集聚集，这个标志指定聚集参数会被根据假想集聚集的要求进行处理： 即后面的直接参数必须匹配聚集（WITHIN GROUP）参数的数据 类型。HYPOTHETICAL标志在运行时没有任何效果，它只在 命令解析期间对确定数据类型和聚集参数的排序规则有用。

CREATE AGGREGATE的参数可以用任意顺序书写， 而无需遵照以上说明的顺序。

注解

在指定支持函数名的参数中，如果需要可以写一个模式名，例如 SFUNC = public.sum。在这里不能写参数类型 — 支持函数 的参数类型是根据其他参数决定的。

通常，UXDB函数是不要修改输入值的真函数。然而，一个聚合迁移函数在聚合上下文中使用时被允许诈欺并修改已在迁移状态的参数。因为与每次创建一个迁移状态的新的拷贝相比，这样可以提供实质的性能提升。

同样，虽然人们一般不期望聚合最终函数修改它的输入值，但有时回避修改迁移态参数是不切实际的。这种行为必须使用FINALFUNC_MODIFY参数声明。READ_WRITE值表示最终函数以某种未指定的方式修改了迁移状态值。这个值防止将聚合用作窗口函数，并且还可以防止因共用相同的输入值和迁移函数的聚合调用而合并迁移状态。SHAREABLE值表示过渡函数不能在最终功能之后使用用，但多重最终函数调用可以对最终的迁移状态值执行调用。这个值阻止将聚合用作

窗口函数，但允许合并过渡状态。（也就是说，此处所关注的优化不是重复地使用相同的最终函数，而是把不同的最终函数应用到相同的最终迁移状态值。只要所有最终功能都没有标记为READ_WRITE就被允许。）

如果一个聚集支持移动聚集模式，当该聚集被用于一个具有移动帧起点（即帧起点 模式不是UNBOUNDED PRECEDING）的窗口函数时，它将提升计 算效率。在概念上，当从底部进入窗口帧时前向转移函数会把输入值加到聚集的状 态上，而逆向转移函数会在从顶部离开帧时再次移除输入值。因此，当值被移除时， 它们总是按照被加入的相同顺序被移除。无论何时调用逆向转移函数，它都将因此 接收最近增加但是还未被移除的参数值。逆向转移函数可以假定在它移除最旧的行 之后至少有一行保留在当前状态中（当情况不是这样时，窗口函数机制会简单地开 始一次新的聚集，而不是使用逆向转移函数）。

用于移动聚集模式的前向转移函数不允许返回 NULL 作为新的状态值。如果逆向 转移函数返回 NULL，这表明逆向函数无法为这个特定的输入逆转状态计算，并且 该聚集计算因此必须从当前帧的开始位置“从零开始”重新被计算。在一些少见的情 况中，逆转正在计算中的状态值是不切实际的，这种习惯可以允许在此类情形中使用 移动聚集模式。

如果没有提供移动聚集实现，聚集仍然可以被用于移动帧，但是 只要帧起点移动，UXDB都将会重新计算 整个聚集。注意不管聚集有没有支持移动聚集模式， UXDB都能处理一个移动帧结束而无需重 新计算，这可以通过增加新值到聚集状态完成。这就是为什么使用聚合作为窗口函数需要最终函数只读的原因。人们认为最终函数不能破坏聚集的状 态值，这样即使已经为一组帧边界得到了一个聚集结果值，该聚集也能继续下去。

有序集聚集的语法允许为最后一个直接参数以及最后一个聚集（ WITHIN GROUP）参数指定VARIADIC。但是，当前的 实现限制只能以两种方式使用VARIADIC。第一种，有续集聚集只能 使用VARIADIC "any"，不能使用其他可变数组类型。第二种，如果最 后一个直接参数是VARIADIC "any"，那么只能有一个聚集参数并且它 也必须是VARIADIC "any"（在系统目录中使用的表示中，这两个参数 会被合并为一个单一的VARIADIC "any"项，因为 ux_proc无法表示具有超过一个VARIADIC参数的 函数）。如果该聚集是一个假想集聚集，匹配VARIADIC "any"参数的 直接参数就是假想参数。任何在前面的参数表示额外的直接参数，它们不被约束为需 要匹配聚集参数。

当前，有序集聚集无须支持移动聚集模式，因为它们不能被用作窗口函数。

部分（包括并行）聚集当前不被有续集聚集支持。还有，包括 DISTINCT或者ORDER BY子句的聚集调用将不会使用 部分聚集，因为在部分聚集时无法支持那些语义。

兼容性

CREATE AGGREGATE是一种 UXDB的语言扩展。SQL 标准没有提供 用户定义的聚集函数。

另见

ALTER AGGREGATE(7), DROP AGGREGATE(7)

名称

CREATE CAST — 定义一种新的造型

大纲

```
CREATE CAST (source_type AS target_type)
  WITH FUNCTION function_name [ (argument_type [, ...]) ]
  [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
  WITHOUT FUNCTION
  [ AS ASSIGNMENT | AS IMPLICIT ]

CREATE CAST (source_type AS target_type)
  WITH INOUT
  [ AS ASSIGNMENT | AS IMPLICIT ]
```

描述

CREATE CAST定义一种新的造型。 一种造型指定如何在两种数据类型之间执行转换。例如，

```
SELECT CAST(42 AS float8);
```

通过调用一个之前指定的函数（这种情况中是 float8(int4)）把整型常量 42 转换成类型float8（如果没有定义合适的造型， 该转换会失败）。

两种类型可以是二进制可强制，这表示该转换 可以被“免费”执行而不用调用任何函数。这要求相应的 值使用同样的内部表示。例如，类型text和 varchar在双向都是二进制可强制的。二进制可强制性并 不必是一种对称关系。例如，在当前实现中从xml到 text的造型可以被免费执行，但是反向则需要一个函数来 执行至少一次语法检查（两种在双向都二进制值兼容的类型也被称作二进 制兼容）。

通过使用WITH INOUT语法，可以把一种造型定义 成I/O 转换造型。一种 I/O 转换造型执行时，会调用源 数据类型的输出函数，并且把结果字符串传递给目标数据类型的输入函数。 在很多常见情况中，这种特性避免了为转换单独定义一个造型函数。一种 I/O 转换造型表现得和一个常规的基于函数的造型相同，只是实现不同而已。

默认情况下，只有一次显式造型请求才会调用造型， 形式是CAST(*x* AS *typename*) or *x*::*typename*。

如果造型被标记为AS ASSIGNMENT，那么在为一个目标数据 类型的列赋值时会隐式地调用它。例如，假设foo.f1是 一个类型text的列，那么如果从类型integer 到类型text的造型被标记为AS ASSIGNMENT， 则：

```
INSERT INTO foo (f1) VALUES (42);
```

将被允许，否则不会允许（我们通常使用赋值造型 来描述此类造型）。

如果造型被标记为AS IMPLICIT，那么可以在任何上下文 中隐式地调用它，无论是赋值还是在一个表达式内部（我们通常用术语 隐式造型来描述这类造型）。例如，考虑这个 查询：

SELECT 2 + 4.0;

解析器初始会把常量分别标记为类型integer和 numeric。在系统目录中没有integer + numeric 操作符，但是有一个 numeric + numeric操作符。 因此，如果有一种可用的从integer到 numeric的造型且被标记为AS IMPLICIT — 实际上确实有 — 该查询将会成功。解析器将应用该 隐式造型 并且解决该查询，就好像它被写成：

SELECT CAST (2 AS numeric) + 4.0;

现在，系统目录也提供一种从numeric到integer 的造型。如果这种造型被标记为AS IMPLICIT — 实际上并没有 — 那么解析器将面临选择：是用前面介绍的过程， 还是把numeric常量造型成integer并且应用 integer + integer操作符。由于 缺少哪种选择更好的知识，解析器会放弃并且说明查询有歧义。我们能 告诉解析器把一个混合了numeric和integer的 表达式解析成 numeric更好的方法就是只让这两种造型中的 一个是隐式的，没有有关于此的内建知识。

对标记造型为隐式持保守态度是明智的。过多的隐式造型路径可能导致 UXDB以令人吃惊的方式解 释命令，或者由于有多种可能解释而根本无法解析命令。一种好的经验 是让一种造型只对于同一种一般类型分类中的类型间的信息保持转换隐式 可调用。例如，从int2到int4的造型 可以被合理地标记为隐式，但是从float8到 int4的造型可能应该只能在赋值时使用。跨类型分类 的造型（如text到int4）最好只被用于显式使用。

> **注意**
>
> 有时为了可用性或者标准兼容的原因，有必要提供在一个类型集合之间 的多种隐式造型，这会导致上述不可避免的歧义。解析器还有一招基于 类型分类和优先类型的后手，它能帮助 提供这类情况下预期的行为。详见 CREATE TYPE(7)。

要创建一种造型，必须拥有源数据类型和目标数据类型并且具有在其他类型上 的USAGE特权。要创建一种二进制可强制造型，必须是一 个超级用户（这种限制是因为错误的二进制可强制造型转换很容易让服务器崩溃）。

参数

source_type

该造型的源数据类型的名称。

target_type

该造型的目标数据类型的名称。

function_name[(*argument_type* [, ...])]

被用于执行该造型的函数。函数名称可以用模式限定。如果没有被限定， 将在模式搜索路径中查找该函数。函数的结果数据类型必须是该造型的 目标数据类型。它的参数讨论如下。 如果没有指定参数列表，则该函数名称在其模式中必须是唯一的。

WITHOUT FUNCTION

指示源类型可以二进制强制到目标类型，因此执行该造型不需要函数。

WITH INOUT

指示该造型是一种 I/O 转换造型，执行需要调用源数据类型的输出函数， 并且把结果字符串传递给目标数据类型的输入函数。

AS ASSIGNMENT

指示该造型可以在赋值的情况下被隐式调用。

AS IMPLICIT

指示该造型可以在任何上下文中被隐式调用。

造型实现函数可以具有 1 到 3 个参数。第一个参数类型必须等于源类型或者 能从源类型二进制强制得到。第二个参数（如果存在）必须是类型 integer，它接收与目标类型相关联的类型修饰符，如果没有类型 修饰符，它会收到-1。第三个参数（如果存在）必须是类型 boolean，如果该造型是一种显式造型，它会收到 true，否则会收到false（奇怪地是，SQL 标准在 某些情况中对显式和隐式造型要求不同的行为。这个参数被提供给必须实现这 类造型的函数。不推荐在设计自己的数据类型时用它）。

一个造型函数的返回类型必须等于目标类型或者能二进制强制到目标类型。

通常，强制转换必须具有不同的源和目标数据类型。但是，如果它有一个带有多个参数的强制转换实现函数，则可以声明具有相同源类型和目标类型的造型。它用于表示系统目录中特定类型的长度强制函数。命名函数用于将类型的值强制转为其第二个参数提供的类型修饰符的值。

当强制转换具有不同的源类型和目标类型，并且一个函数使用多个参数时，它支持从一种类型转换为另一种类型，并在单个步骤中应用长度强制。如果没有这样的条目，强制转换为使用类型修饰符的类型将涉及两个强制转换步骤，一个是在数据类型之间进行转换，另一个是应用修饰符。

向域类型强制转换或从域类型强制转换当前无效。向域或从域强制转换使用与其基础类型关联的造型。

注解

使用DROP CAST(7)移除用户定义的造型。

记住如果想要能够双向转换类型，需要在两个方向上都 显式声明造型。

通常没有必要创建用户定义类型和标准字符串类型（text、 varchar和char(*n*)， 以及被定义在字符串分类中的用户定义类型）之间的造型。 UXDB会为它们提供自动的 I/O 转换造型。 到字符串类型的自动造型被当做赋值造型，而字符串类型作为源的自动 造型只能是显式的。通过声明自己的造型来替换自动造型可以覆盖这 种行为，但是这样做的唯一原因是想让该转换比标准的设置更容易被 调用。另一种可能的原因是想让该转换的行为与该类型的 I/O 函数不 同，但这种原因足够令人感到意外，应该考虑再三它是不是个好主意 （确实有少量内建类型对转换具有不同的行为，绝大部分是因为 SQL 标准的要求）。

虽然不必要，推荐继续遵循这种在目标数据类型后面命名造型 实现函数的习惯。很多用户习惯于能够使用一种函数风格的记法来造型 数据类型，即*typename*(*x*)。 这种记法正好是对造型实现函数的调用，这里它没有被作为造型特殊对待。 如果转换函数没有被指定支持这种习惯，那么用户会觉得意外。 由于UXDB允许用不同的参数类型重载同一个 函数名，因此存在多个从不同类型到同一目标类型的同名转换函数并不困难。

> **注意**
>
> 实际上前一段过于简化了：有两种情况中一个函数调用结构在没有被匹配到 一个实际函数时将被当作一次造型请求。如果函数调用 *name*(*x*)没有正

好匹配任何现有函数， 但*name*是一种数据类型的名称并且 ux_cast提供了一种从*x*的类型到这种 类型的二进制可强制造型，那么该调用将被翻译为一次二进制可强制造型。 通过这种例外，二进制可强制造型能够以函数语法调用，即便没有该函数。 同样的，如果没有ux_cast项，但是该造型是要造型到一种 字符串类型或者是要从一种字符串类型造型，调用将被翻译成一次 I/O 转换 造型。这种例外允许以函数语法调用 I/O 转换造型。

注意

还有一种例外中的例外：从组合类型到字符串类型的 I/O 转换造型不能使用 函数语法调用，而必须被写成显式造型语法（CAST或者 ::记号）。增加这种例外是因为在引入了自动提供的 I/O 转换 造型之后，在想要引用一个函数或者列时太容易意外地调用这种造型。

示例

要使用函数int4(bigint)创建一种从类型 bigint到类型int4的赋值造型：

```
CREATE CAST (bigint AS int4) WITH FUNCTION int4(bigint) AS ASSIGNMENT;
```

（在系统中这种造型已经被预定义）。

兼容性

CREATE CAST命令符合 SQL标准，不过 SQL 没有为二进制可强制 类型或者实现函数的额外参数做好准备。 AS IMPLICIT也是一种 UXDB扩展。

另见

CREATE FUNCTION(7), CREATE TYPE(7), DROP CAST(7)

名称

CREATE COLLATION — 定义一种新排序规则

大纲

```
CREATE COLLATION [ IF NOT EXISTS ] name (
   [ LOCALE = locale, ]
   [ LC_COLLATE = lc_collate, ]
   [ LC_CTYPE = lc_ctype, ]
   [ PROVIDER = provider, ]
   [ DETERMINISTIC = boolean, ]
   [ VERSION = version ]
)
CREATE COLLATION [ IF NOT EXISTS ] name FROM existing_collation
```

描述

CREATE COLLATION使用指定的操作系统区域 设置或者复制一个现有的排序规则来定义新的排序规则。

要创建一种排序规则，必须拥有目标模式上的 CREATE特权。

参数

IF NOT EXISTS

如果已经存在了同名的排序规则，则不要抛出错误。在这种情况下发出一个通知。 请注意，不保证已经存在的排序规则与要创建的这个类似。

name

排序规则的名字，可以被模式限定。如果没有用模式限定，该排序规则 会被定义在当前模式中。排序规则名称在其所处的模式中必须唯一（系统 目录可以为其他编码包含具有相同名称的排序规则，但数据库编码不匹配 时它们会被忽略）。

locale

这是一种一次设置LC_COLLATE 和LC_CTYPE的快捷方式。如果指定它，就 不能指定那两个参数。

lc_collate

为LC_COLLATE区域分类使用指定的操作系统 区域。

lc_ctype

为LC_CTYPE区域分类使用指定的操作系统 区域。

provider

指定用于与此排序规则相关的区域服务的提供程序。可能的值是： icu、 libc。 默认是libc。 可用的选择取决于操作系统和构建选项。

DETERMINISTIC

规定排序是否应使用确定性比较。默认值为true。确定性比较认为那些在字节上不相等的字符串是不相等的，即使它们在逻辑上被比较认为相等。 UXDB使用字节比较来断开连接。不确定的比较可能会使排序规则大小写不敏感或音调不敏感。 因此，您需要选择一个适当的LC_COLLATE设置并且在此处将排序规则设置为不确定。

非确定性排序规则仅被ICU提供者支持。

version

指定使用该排序规则存储的版本字符串。通常忽略该选项， 这会导致版本从操作系统提供的排序规则实际版本中计算出来。 此选项旨在供ux_uuxrade用于复制现有安装中的版本。

又见ALTER COLLATION(7)获取如何处理排序规则版本错误匹配。

existing_collation

要复制的一种现有的排序规则的名称。新的排序规则将和现有的具有 同样的属性，但是它是一个独立的对象。

注解

CREATE COLLATION 采用SHARE ROW EXCLUSIVE锁，在ux_collation系统分类上，这是自我冲突的，所以一次只能运行一条CREATE COLLATION 命令。

使用DROP COLLATION可移除用户定义的排序规则。

使用libc排序规则提供程序时，语言环境必须适用于当前的数据库编码。 有关精确的规则，请参见CREATE DATABASE(7)。

示例

从操作系统区域fr_FR.utf8创建一种排序规则（假定 当前数据库编码是UTF8）：

```
CREATE COLLATION french (locale = 'fr_FR.utf8');
```

使用German phone book排序顺序使用ICU提供程序创建排序规则：

```
CREATE COLLATION german_phonebook (provider = icu, locale = 'de-u-co-phonebk');
```

从一个现有的排序规则创建一个新的排序规则：

```
CREATE COLLATION german FROM "de_DE";
```

能在应用中使用与操作系统无关的排序规则名称就很方便了。

兼容性

在 SQL 标准中有一个CREATE COLLATION 语句，但是它被限制为只能复制一个现有的排序规则。创建新排序规则的 语法是一种UXDB扩展。

另见

ALTER COLLATION(7), DROP COLLATION(7)

名称

CREATE CONVERSION — 定义一种新的编码转换

大纲

```
CREATE [ DEFAULT ] CONVERSION name
  FOR source_encoding TO dest_encoding FROM function_name
```

描述

CREATE CONVERSION定义一种字符集编码间 新的转换。还有，被标记为DEFAULT的转换将被 自动地用于客户端和服务器之间的编码转换。为了这个目的，必须定义两个 转换（从编码 A 到 B 以及从编码 B 到 A）。

要创建一个转换，必须拥有该函数上的EXECUTE特权 以及目标模式上的CREATE特权。

参数

DEFAULT

DEFAULT子句表示这个转换是从源编码到目标编码的默认 转换。在一个模式中对于每一个编码对，只应该有一个默认转换。

name

转换的名称，可以被模式限定。如果没有被模式限定，该转换被定义在 当前模式中。在一个模式中，转换名称必须唯一。

source_encoding

源编码名称。

dest_encoding

目标编码名称。

function_name

被用来执行转换的函数。函数名可以被模式限定。如果没有，将在路径 中查找该函数。

该函数必须具有一下的特征：

```
conv_proc(
   integer,  -- 源编码 ID
   integer,  -- 目标编码 ID
   cstring,  -- 源字符串（空值终止的 C 字符串）
   internal, -- 目标（用一个空值终止的 C 字符串填充）
   integer   -- 源字符串长度
) RETURNS void;
```

注解

使用DROP CONVERSION可以移除用户定义的转换。

创建转换所要求的特权可能在未来的发行中被更改。

示例

使用myfunc创建一个从编码UTF8到 LATIN1的转换：

```
CREATE CONVERSION myconv FOR 'UTF8' TO 'LATIN1' FROM myfunc;
```

兼容性

CREATE CONVERSION是一种 UXDB扩展。在 SQL 标准中 没有CREATE CONVERSION语句，但是有 一个目的和语法都类似的 CREATE TRANSLATION语句。

另见

ALTER CONVERSION(7), CREATE FUNCTION(7), DROP CONVERSION(7)

名称

CREATE DATABASE — 创建一个新数据库

大纲

```
CREATE DATABASE name
  [ [ WITH ] [ OWNER [=] user_name ]
      [ TEMPLATE [=] template ]
      [ ENCODING [=] encoding ]
      [ LC_COLLATE [=] lc_collate ]
      [ LC_CTYPE [=] lc_ctype ]
      [ TABLESPACE [=] tablespace_name ]
      [ ALLOW_CONNECTIONS [=] allowconn ]
      [ CONNECTION LIMIT [=] connlimit ]
      [ IS_TEMPLATE [=] istemplate ] ]
```

描述

CREATE DATABASE创建一个新的UXDB数据库。

要创建一个数据库，必须是一个超级用户或者具有特殊的CREATEDB特权。见CREATE ROLE(7)。

默认情况下，新数据库将通过克隆标准系统数据库template1被创建。可以通过写TEMPLATE *name*指定一个不同的模板。特别地，通过写TEMPLATE template0可以创建一个干净的数据库，它将只包含UXDB版本所预定义的标准对象。如果希望避免拷贝任何可能被加入到template1中的本地安装对象，这将有所帮助。

参数

name

要创建的数据库名。

user_name

将拥有新数据库的用户的角色名，或者用DEFAULT来使用默认值（即，执行该命令的用户）。要创建一个被另一个角色拥有的数据库，必须是该角色的一个直接或间接成员，或者是一个超级用户。

template

要从其创建新数据库的模板名称，或者用DEFAULT来使用默认模板（template1）。

encoding

要在新数据库中使用的字符集编码。指定一个字符串常量（例如'SQL_ASCII'），或者一个整数编码编号，或者DEFAULT来使用默认的编码（即，模板数据库的编码）。附加限制见下文。

lc_collate

要在新数据库中使用的排序规则顺序（LC_COLLATE）。这会影响应用到字符串的排序顺序，例如在带 ORDER BY 的查询中，以及文本列上索引所使用的顺序。默认是使用模板数据库的排序规则顺序。附加限制见下文。

lc_ctype

要在新数据库中使用的字符分类（LC_CTYPE）。这会影响字符的类别，如小写、大写和数字。默认是使用模板数据库的字符分类。附加限制见下文。

tablespace_name

将与新数据库相关联的表空间名称，或者DEFAULT来使用模板数据库的表空间。这个表空间将是在这个数据库中创建的对象的表空间。详见CREATE TABLESPACE(7)。

allowconn

如果为假，则没有人能连接到这个数据库。默认为真，表示允许连接（除了 被其他机制约束以外，例如GRANT/REVOKE CONNECT）。

connlimit

这个数据库允许多少并发连接。-1 （默认值）表示没有限制。

istemplate

如果为真，则任何具有CREATEDB特权的用户都可以从 这个数据库克隆。如果为假（默认），则只有超级用户或者该数据库的拥有者 可以克隆它。

可选的参数可以被写成任何顺序，不用按照上面说明的顺序。

注解

CREATE DATABASE不能在一个事务块内被执行。

带有一行“不能初始化数据库目录”的错误大部分与在数据目录上权限不足、磁盘满或其他文件系统问题有关。

使用DROP DATABASE(7)移除一个数据库。

程序createdb是这个命令的一个包装器程序，为了使用方便而提供。

不会从模板数据库中复制数据库层面的配置参数（通过ALTER DATABASE(7)设置）。

尽管可以通过指定一个数据库作为模板来从其中而不是template1复制，这（还）不是“COPY DATABASE”功能的一般目的。主要的限制是在模板数据库被拷贝期间其他会话不能连接到它。如果CREATE DATABASE启动时还存在任何其他连接，它将失败。否则，到模板数据库的新连接将被挡在外面直到CREATE DATABASE完成。

为新数据库指定的字符集编码必须与选定的区域设置（LC_COLLATE和LC_CTYPE）相兼容。如果区域是C（或者等效的POSIX），那么所有编码都被允许，但是对于其他区域设置只有一种编码能正确工作（不过，在 Windows 上 UTF-8 编码能够与任何区域一起使用）。CREATE DATABASE将允许超级用户指定SQL_ASCII编码而不管区域设置，但是这种选择已被废弃并且可能在数据与数据库中存储的区域不是编码兼容时让字符串函数行为失当。

编码和区域设置必须匹配模板数据的编码和区域，除非template0被用作模板。这是因为其他数据库可能包含不匹配指定编码的数据，或者可能包含排序顺序受LC_COLLATE和LC_CTYPE影响的索引。拷贝这种数据将导致一个由于该新设置损坏的数据库。不过，template0是不会含有任何可能被影响的数据或索引的。

CONNECTION LIMIT选项大概是唯一会被强制的，如果两个新会话在大约同一时间开始并且那时该数据库只剩有一个连接“槽”，可能两者都会失败。还有，该限制对超级用户或后台工作进程无效。

例子

要创建一个新数据库：

```
CREATE DATABASE lusiadas;
```

要在一个默认表空间salesspace中创建一个被用户salesapp拥有的新数据库sales：

```
CREATE DATABASE sales OWNER salesapp TABLESPACE salesspace;
```

要用不同的语言环境创建数据库music：

```
CREATE DATABASE music
  LC_COLLATE 'sv_SE.utf8' LC_CTYPE 'sv_SE.utf8'
  TEMPLATE template0;
```

在这个例子中，如果指定的语言环境与template1中的语言环境不同，则需要TEMPLATE template0子句。（如果不是，则明确指定区域设置是多余的。）

要用不同的语言环境和不同的字符集编码创建数据库music2：

```
CREATE DATABASE music2
  LC_COLLATE 'sv_SE.iso885915' LC_CTYPE 'sv_SE.iso885915'
  ENCODING LATIN9
  TEMPLATE template0;
```

指定的区域设置和编码设置必须匹配，否则会报告错误。

请注意，区域名称是特定于操作系统的，因此上述命令可能无法在任何地方以相同的方式工作。

兼容性

在 SQL 标准中没有**CREATE DATABASE**语句。数据库等效于目录，而目录的创建由实现定义。

参见

ALTER DATABASE(7), DROP DATABASE(7)

名称

CREATE DATABASE LINK — 创建数据库链接

大纲

```
CREATE [ PUBLIC ] DATABASE LINK linkname
   CONNECT TO username
   IDENTIFIED BY password
   USING 'connstr' | (connstr = values,dbtype = values)
```

描述

uxdb兼容Oracle使用CREATE DATABASE LINK语句创建数据库链接。创建数据库链接后，可以在 SQL 语句中通过附加@dblink到表、视图或 PL/SQL 对象名称来引用其他数据库中的表、视图和 PL/SQL 对象。可以使用该语句查询其他数据库中的表或视图。还可以使用任何INSERT、UPDATE、DELETE或LOCK TABLE语句访问远程表和视图。

UXDB数据库实现CREATE/DROP DATABASE LINK相关语法及功能支持场景如下表所示。

表 3. CREATE/DROP DATABASE LINK相关语法及功能支持

应用场景	Oracle-Oracle	Oracle-其他库	Uxdb-uxdb	Uxdb-oracle
SELECT	支持	支持	支持	支持
INSERT	支持	支持	支持	支持
UPDATE	支持	支持	支持	支持
DELETE	支持	支持	支持	支持
LOCK TABLE	支持	不支持	不支持	不支持
PL/SQL functions	支持	不支持	不支持	不支持
procedures	支持	支持部分（创建本地过程操作远程对象）	不支持	不支持
packages	支持	不支持	不支持	不支持

UXDB数据库通过@dblink操作远程对象的支持场景如下所示。

表 4. 操作远程对象用法汇总

描述	例句	Oracle	UXDB
from阶段带@	select ... from tab@dblink1 where id < 1;	支持	支持
from阶段带@，where子句中[带@dblink]引用远程表列	select ... from tab@dblink1 where tab.id@dblink1< 1; select ... from tab@dblink1 where id< 3;	支持	支持

描述	例句	Oracle	UXDB
from阶段带@，where子句中直接使用远程表名.列（多表连接）	select ... from tab@dblink1,t_b where tab.id = t_b.id;	支持	不支持

本地表和远程表同名，两表连接操作	select ... from t_a@dblink1,t_a where t_a.id = t_a.id;
	
	select ... from t_a@dblink1,t_a where id@dblink1 = t_a.id;
	
	select ... from t_a@dblink1,t_a where id1@dblink1 = t_a.id;
		支持（但列名相同时报错）	例1支持，例2报错，例3支持

表名 @ dblink三者之间可以任意空格	select ... from tab@dblink1; select ... from tab @dblink1; select ... from tab@ dblink1; select ... from tab @ dblink1;	支持	支持

Update远程表时，set后的目标对象支持带@	update stu@dblink set stu.name@dblink = 'ji' where stu.id@dblink = 3;
	
	update stu@dblink set name@dblink = 'ji' where stu.id@dblink = 3;
		支持	支持

Insert远程表后的目标列中使用带@	insert into t_a@DBLINK3(id, t_a.address@dblink3, name) values(4, 'jhfje' , 'lpp');	支持	不支持

参数

linkname

dblink 名称。

username

远程连接用户名（注意标准模式和兼容模式下用户名的大小写）。

password

远程用户登录密码。

connstr

此规则针对oracle数据库的专用连接标识串　如192.71.1.104:1521/orcl，传入的字符串包含远程机器HOST:PORT/SERVICE_NAME。

也可以和oracle一样使用连接描述符<description>。

示例如下所示。

USING '192.71.1.104:1521/orcl'
USING '(DESCRIPTION =(ADDRESS_LIST =(ADDRESS =(PROTOCOL = TCP)(HOST = 192.71.1.104)(PORT = 1521)))(CONNECT_DATA =(SERVICE_NAME = orcl)))'

(connstr=values,dbtype=values)：如果需要指定远程连接数据库类型，则使用此规则。

示例如下所示。

USING (connstr='192.71.1.104:1521/orcl', dbtype='oracle')
USING (connstr='(DESCRIPTION =(ADDRESS_LIST =(ADDRESS =(PROTOCOL = TCP)(HOST = 192.71.1.102)(PORT = 1521)))(CONNECT_DATA =(SERVICE_NAME = orcl)))', dbtype='oracle')
USING (connstr='192.71.1.104:5532/uxdb', dbtype='uxdb') 其中connstr串代表HOST:PORT/DBNAME

使用限制说明

1. 通过dblink访问远程库上的系统表时，需要指定对应的模式名。（uxdb对应为ux_catalog；oracle对应sys）
2. Oracle指定connecting_string有两种方式，可以是配置的网络服务名<tsn_name>；uxdb不支持tsn_name的方式。
3. 远程连接oracle数据库时，版本不高于11g，如果连接高于此版本(如12c)，则会报错（ORA-28040: No matching authentication protocol），原因oracle_fdw对应的ODBC库版本为libclntsh.so.11.1，高版本数据库对应的库版本不一致。
4. 通用户创建dblink需要的权限。

   uxdb用户登录授权，如下所示。

   grant all on FOREIGN data wrapper oracle_fdw|uxdb_fdw to public;
   grant all on DATABASE dbname to PUBLIC;(在创建dblink的时候需要创建本地外部表对应的schema，因此需要对相应的数据库授予public权限)
5. uxdb私有dblink不能创建在public模式下。
6. 当远程库为oracle时，UPDATE|DELETE远程表时，远程表必须有主键(本地外部表必须有主键才可以执行，否则会报错)。
7. Uxdb操作远程oracle视图，执行update|delete操作会报错，原因无论表是否有主键，创建的视图都是不带主键的，因此oracle_fdw会报错。

   针对5、6的情况，当远程表没有主键时，但又需要执行update|delete 操作，可以为远程表对应的本地外部表增加主键，但是本地外部表必须要选择一个值唯一且不为空的列作为主键列。

   执行sql语句为：alter foreign table <foreign_tabname> alter <colname> options (key 'true');

- <foreign_tabname>：外部表名
- <colname>：选择的主键列列名

8. 不支持远程物化视图的刷新。
9. 当远程库为oracle时，对远程表进行批量插入时，可能因意外情况导致序列化错误(ORA-08177)。解决方法如下所示。
   a. 通过脚本调用uxsql执行远程表的insert命名(该方法脚本运行时，第一条insert也会报ORA-08177错误，后续的不会再报错)。
   b. 修改外部服务器的隔离级别为read_committed，但根据oracle_fdw官方文档说明，可能会存在数据不一致风险。

      ALTER SERVER <server_name> OPTIONS (isolation_level 'read_committed');

10. 远端服务器为oracle时，列类型为real类型，通过dblink插入数据时数据最大位数不超过63位。

注解

如果访问中文，如用户名或密码中含有中文，需要注意编码保持一致，Uxdb客户端和服务端编码为UTF8。

Oracle客户端和服务端为AMERICAN_AMERICA.AL32UTF8。

Oracle编码配置方法如下。

- 配置客户端编码

  在~/.bash_profile增加如下内容，保存并退出。

  ```
  vi ~/.bash_profile
  export NLS_LANG=AMERICAN_AMERICA.AL32UTF8
  ```

  执行source ~/.bash_profile使其生效，执行echo $NLS_LANG结果为上面设置的编码这说明配置成功。

- 配置/修改Oracle服务器端的编码

  当前服务端编码为AMERICAN_AMERICA.UTF8，需要和客户端保持一致，因此将服务端编码修改为AMERICAN_AMERICA.AL32UTF8。

  修改方法如下。

  以sys用户登录进去，执行以下命令。

  ```
  shutdown immediate                //关闭数据库
  startup nomount        //以nomount启动数据库
  alter database mount exclusive;          //装载数据为专用的高级模式;
  alter system enable restricted session;     //启用受限制的session模式
  alter system set job_queue_processes=0;   //设置工作队列的最大进程数为0
  alter system set aq_tm_processes=0;
  alter database open;
  alter database character set INTERNAL_USE AL32UTF8; // 修改字符集为 AL32UTF8
  ```

```
shutdown immediate      //关闭数据库
Startup      //启动数据库
select userenv('language') from dual;      //查看编码是否修改成功
```

> **注意**
>
> 需要注意的是，对于之前已经存在的数据，在修改了字符集之后还是会显示为乱码，只有新插入的数值才会生效。

示例

1. 本地数据库实例配置文件预加载database_link插件。

   ```
   vim data1/uxsinodb.conf
   shared_preload_libraries = 'database_link'
   ```

   如果未将插件名写入配置文件，在创建该插件时会报错。

2. 启动本地实例，创建对应的插件。

   ```
   create extension database_link ;
   create extension uxdb_fdw;
   create extension oracle_fdw;
   ```

3. 打开表结构比对开关。

   表结构比对开关默认关闭，使用如下语句进行打开或者关闭。

   ```
   set databaselink.enable_dblink_comparetable = on/off;
   ```

   关闭时，远端表结构发生改变，本地不会感知，也不会更新本地外部表结构。

   打开后，远端表结构发生改变时，使用DML语句会更新本地外部表结构。

4. 创建dblink，并进行应用。

uxdb-uxdb标准模式测试如下。

1. 本地数据库实例配置文件预加载database_link插件，并启动。

   ```
   vim data1/uxsinodb.conf
   shared_preload_libraries = 'orafce,database_link'
   ```

2. 启动本地实例，创建对应的插件。

   ```
   create extension database_link ;
   create extension uxdb_fdw;
   create extension oracle_fdw;
   ```

3. 创建dblink，并进行相关应用测试。

   准备环境本地实例data1，远程实例data2。

   ```
   vim data1/uxsinodb.conf
   shared_preload_libraries = 'orafce,database_link'
   ```

   启动data1 data2。

---远程实例data2 创建测试表
create table t_a (id int, name char(5));
insert into t_a values (1, 'a');
insert into t_a values (2, 'b');
insert into t_a values (3, 'c');
insert into t_a values (4, 'd');

create table t_b (id int, name char(5));
insert into t_b values (1, 'c');
insert into t_b values (2, 'a');
insert into t_b values (3, 'd');
insert into t_b values (4, 'e');

create table t_c (id int, name char(5));
insert into t_c values (1, 'a');
insert into t_c values (2, 'b');
insert into t_c values (3, 'c');
insert into t_c values (4, 'd');

--远程表带默认值
create table stu1(id int, name varchar(4) not null, birthday date default '2015-4-27');
insert into stu1 values(1, 'bbb');

--远程表带引号
create table "Atest" (id int);
insert into "Atest" values (1),(2),(3);
create schema "BBB";
set search_path to "BBB";
create table "BTESt" (id int);
insert into "BTESt" values (1),(2),(3);

--远程创建视图
create view myview as select * from t_c;
CREATE VIEW
select * from myview ;
```
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  4 | d
(4 rows)
```

--远程创建物化视图
CREATE MATERIALIZED VIEW mymatview AS SELECT * FROM stu1;
SELECT 3
select * from mymatview;
```
 id | name |  birthday
----+------+------------
  1 | bbb  | 2015-04-27
  2 | ccc  | 2015-04-27
  3 | ddd  | 2015-04-27
(3 rows)
```

---实例data1 创建dblink
create extension database_link ;
CREATE EXTENSION
create extension uxdb_fdw;
CREATE EXTENSION
create public database link dblink2
connect to 'uxdb' identified by '1'
USING (connstr='127.0.0.1:5532/uxdb', dbtype='uxdb');
CREATE DATABASE LINK

---增删改查测试
select * from t_a@dblink2;
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  4 | d
(4 rows)
insert into t_a@dblink2 values(5, 'hhh');
INSERT 0 1
select * from t_a@dblink2;
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  4 | d
  5 | hhh
(5 rows)
insert into t_a@dblink2 values(6, 'kkkk') returning ctid, *;
 ctid | id | name
-------+----+-------
 (0,6) | 6 | kkkk
(1 row)
INSERT 0 1
update t_a@dblink2 set name = 'ddd' where id = 4;
UPDATE 1
select * from t_a@dblink2;
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  5 | hhh
  4 | ddd
(5 rows)
delete from t_a@dblink2 where id = 3;
DELETE 1
select * from t_a@dblink2;
 id | name
----+-------
  1 | a

```
 2 | b
 5 | hhh
 4 | ddd
(4 rows)

--为带默认值的远程表插入数据
select * from stu1@dblink2;
 id | name |  birthday
----+------+------------
 1 | bbb  | 2015-04-27
(1 row)
insert into stu1@dblink2 values(2,'ccc');
INSERT 0 1
select * from stu1@dblink2;
 id | name |  birthday
----+------+------------
 1 | bbb  | 2015-04-27
 2 | ccc  | 2015-04-27
(2 rows)
insert into stu1@dblink2 values(3,'ddd', '2019-10-08');
INSERT 0 1
select * from stu1@dblink2;
 id | name |  birthday
----+------+------------
 1 | bbb  | 2015-04-27
 2 | ccc  | 2015-04-27
 3 | ddd  | 2019-10-08
(3 rows)
update stu1@dblink2 set birthday=default where id = 3;
UPDATE 1
select * from stu1@dblink2;
 id | name |  birthday
----+------+------------
 1 | bbb  | 2015-04-27
 2 | ccc  | 2015-04-27
 3 | ddd  | 2015-04-27
(3 rows)

--对于创建时加引号的模式名|表名|列名，通过dblink操作时也需要加引号
select * from "Atest"@dblink2;
 id
----
 1
 2
 3
(3 rows)
select * from "BBB"."BTESt"@dblink2;
 id
----
 1
 2
 3
(3 rows)
insert into "BBB"."BTESt"@dblink2 values (5), (6);
```

```
INSERT 0 2
select * from "BBB"."BTESt"@dblink2;
 id
----
  1
  2
  3
  5
  6
(5 rows)

---访问远程视图或物化视图
select * from myview@dblink2;
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  4 | d
(4 rows)
select * from mymatview@dblink2;
 id | name |  birthday
----+------+------------
  1 | bbb  | 2015-04-27
  2 | ccc  | 2015-04-27
  3 | ddd  | 2015-04-27
(3 rows)

--insert|update|delete操作远程视图
insert into myview@dblink2 values(5,'p');
INSERT 0 1
select * from myview@dblink2;
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  4 | d
  5 | p
(5 rows)
update myview@dblink2 set name = 'pp' where id = 4;
UPDATE 1
select * from myview@dblink2;
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  5 | p
  4 | pp
(5 rows)
delete from myview@dblink2 where id = 4;
DELETE 1
select * from myview@dblink2;
```

```
 id | name
----+-------
  1 | a
  2 | b
  3 | c
  5 | p
(4 rows)

--insert|update|delete操作远程物化视图
insert into mymatview@dblink2 values(4,'eee');
ERROR:  cannot change materialized view "mymatview"
CONTEXT:  remote SQL command: INSERT INTO public.mymatview(id, name, birthday)
 VALUES ($1, $2, $3)
update mymatview@dblink2 set name = 'hhh' where id = 2;
ERROR:  cannot change materialized view "mymatview"
CONTEXT:  remote SQL command: UPDATE public.mymatview SET name = 'hhh'::character
 varying(4) WHERE ((id = 2))
delete from mymatview@dblink2 where id = 3;
ERROR:  cannot change materialized view "mymatview"
CONTEXT:  remote SQL command: DELETE FROM public.mymatview WHERE ((id = 3))
select * from mymatview@dblink2;
 id | name |  birthday
----+------+------------
  1 | bbb  | 2015-04-27
  2 | ccc  | 2015-04-27
  3 | ddd  | 2015-04-27
(3 rows)

-----多表连接测试
--本地创建测试表
create table tt (id int, name char(5));
insert into tt values (1, 'g');
insert into tt values (2, 'j');
insert into tt values (3, 'k');
insert into tt values (4, 'l');
\d
      List of relations
 Schema | Name | Type  | Owner
--------+------+-------+-------
 public | tt   | table | uxdb
(1 row)
select * from t_a@dblink2, tt where t_a.id@dblink2 = tt.id;
 id | name  | id | name
----+-------+----+-------
  1 | a     |  1 | g
  2 | b     |  2 | j
  4 | ddd   |  4 | l
(3 rows)

select * from t_a@dblink2, tt where t_a.id@ dblink2 = tt.id;
 id | name  | id | name
----+-------+----+-------
  1 | a     |  1 | g
  2 | b     |  2 | j
```

```
 4 | ddd  | 4 | l
(3 rows)
select * from t_a@dblink2, tt where t_a.id  @ dblink2 = tt.id;
 id | name  | id | name
----+-------+----+-------
  1 | a     | 1 | g
  2 | b     | 2 | j
  4 | ddd   | 4 | l
(3 rows)
select * from t_a@dblink2 left join tt on t_a.id  @ dblink2 = tt.id;
 id | name  | id | name
----+-------+----+-------
  1 | a     | 1 | g
  2 | b     | 2 | j
  4 | ddd   | 4 | l
  5 | hhh   |    |
(4 rows)
select * from t_b@dblink2 right join (select * from t_a@dblink2 left join tt on t_a.id  @
 dblink2 = tt.id) b on t_b.id@dblink2 > 1;
 id | name  | id | name  | id | name
----+-------+----+-------+----+-------
  2 | a     | 1 | a     | 1 | g
  3 | d     | 1 | a     | 1 | g
  4 | e     | 1 | a     | 1 | g
  2 | a     | 2 | b     | 2 | j
  3 | d     | 2 | b     | 2 | j
  4 | e     | 2 | b     | 2 | j
  2 | a     | 4 | ddd   | 4 | l
  3 | d     | 4 | ddd   | 4 | l
  4 | e     | 4 | ddd   | 4 | l
  2 | a     | 5 | hhh   |    |
  3 | d     | 5 | hhh   |    |
  4 | e     | 5 | hhh   |    |
(12 rows)

--其他用户登录通过PUBLIC类型的dblink访问远程对象
create user aaa password '1';
CREATE ROLE
 \c uxdb aaa
Password for user aaa:
You are now connected to database "uxdb" as user "aaa".
select * from "Atest"@dblink2;
 id
----
  1
  2
  3
(3 rows)
insert into "BBB"."BTESt"@dblink2 values (5), (6);
INSERT 0 2
select * from "BBB"."BTESt"@dblink2;
 id
----
  1
```

```
 2
 3
 5
 6
 5
 6
(7 rows)

---创建私有dblink测试私有dblink创建在非public模式下，因此需要先创建对应的schema
create database link dblink3
 connect to 'uxdb' identified by '1'
USING (connstr='127.0.0.1:5532/uxdb', dbtype='uxdb');
ERROR:  Non-public dblink can not be created in "public" schema
create schema stest;
CREATE SCHEMA
set search_path to stest ;
SET
create database link dblink3 connect to 'uxdb' identified by '1' USING
 (connstr='127.0.0.1:5532/uxdb', dbtype='uxdb');
CREATE DATABASE LINK
create database link stest.dblink4 connect to 'uxdb' identified by '1' USING
 (connstr='127.0.0.1:5532/uxdb', dbtype='uxdb');
CREATE DATABASE LINK
select * from "BBB"."BTESt"@dblink3;
 id
----
  1
  2
  3
  5
  6
  5
  6
(7 rows)
select * from "BBB"."BTESt"@dblink4;
 id
----
  1
  2
  3
  5
  6
  5
  6
(7 rows)
delete from "BBB"."BTESt"@dblink4 where id = 6;
DELETE 2
select * from "BBB"."BTESt"@dblink4;
 id
----
  1
  2
  3
  5
```

```
  5
(5 rows)
```

--其他用户登录通过私有dblink访问远程对象会报错
--私有dblink创建在非public模式下，因此首先需要为普通用户授予使用该模式的权限

```
grant usage on schema stest to aaa;
GRANT
\c uxdb aaa
Password for user aaa:
You are now connected to database "uxdb" as user "aaa".
set search_path to stest ;
SET
select * from "BBB"."BTESt"@dblink3;
ERROR:  connection description for remote database not found
LINE 1: select * from "BBB"."BTESt"@dblink3;
                      ^
```

--普通用户创建dblink
--首先需要管理员授予相关权限

grant all on FOREIGN data wrapper uxdb_fdw to public;（创建server和mapping的时候需要外部服务器权限）

```
GRANT
```

grant all on DATABASE uxdb to PUBLIC;(在创建dblink的时候需要创建本地外部表对应的schema,因此需要将对应数据库授予public,否则会报错)

```
GRANT
\c uxdb aaa
Password for user aaa:
You are now connected to database "uxdb" as user "aaa".
create public database link dblink5 connect to 'uxdb' identified by '1' USING
 (connstr='127.0.0.1:5532/uxdb', dbtype='uxdb');
CREATE DATABASE LINK
select * from "BBB"."BTESt"@dblink5;
 id
----
  1
  2
  3
  5
  5
(5 rows)
select * from t_a@dblink5;
 id | name
----+-------
  1 | a
  2 | b
  5 | hhh
  6 | kkkk
  4 | ddd
(5 rows)
```

--其他用户登录通过普通用户aaa创建的dblink5访问远程对象

```
\c uxdb uxdb
Password for user uxdb:
You are now connected to database "uxdb" as user "uxdb".
```

create user bbb password '123';
CREATE ROLE
\c uxdb bbb
Password for user bbb:
You are now connected to database "uxdb" as user "bbb".
select * from t_a@dblink5;
 id | name
----+-------
  1 | a
  2 | b
  5 | hhh
  6 | kkkk
  4 | ddd
(5 rows)

名称

CREATE DOMAIN — 定义一个新的域

大纲

```
CREATE DOMAIN name [ AS ] data_type
  [ COLLATE collation ]
  [ DEFAULT expression ]
  [ constraint [ ... ] ]
```

其中 *constraint* 是：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL | NULL | CHECK (expression) }
```

描述

CREATE DOMAIN创建一个新的域。域 本质上是一种带有可选约束（在允许的值集合上的限制）的数据类型。 定义一个域的用户将成为它的拥有者。

如果给定一个模式名（例如CREATE DOMAIN myschema.mydomain ...），那么域将被创建在该指定的模式中。 否则它会被创建在当前模式中。域的名称在其模式中的类型和域之间 必须保持唯一。

域主要被用于把字段上的常用约束抽象到一个单一的位置以便维护。例如， 几个表可能都包含电子邮件地址列，而且都要求相同的 CHECK 约束来验证 地址的语法。可以为此定义一个域，而不是在每个表上都单独设置一个约束。

要创建一个域，必须在其底层类型上拥有USAGE特权。

参数

name

要被创建的域的名称（可以被模式限定）。

data_type

域的底层数据类型。可以包括数组指示符。

collation

用于该域的可选的排序规则。如果没有指定排序规则，将使用底层 数据类型的默认排序规则。如果指定了COLLATE， 底层类型必须是可排序的。

DEFAULT *expression*

DEFAULT子句为该域数据类型的列指定一个默认值。 该值是任何没有变量的表达式（但不允许子查询）。默认值表达式 的数据类型必须匹配域的数据类型。如果没有指定默认值，那么 默认值就是空值。

默认值表达式将被用在任何没有指定列值的插入操作中。如果为一个 特定列定义了默认值，它会覆盖与域相关的默认值。继而，域默认值 会覆盖任何与底层数据类型相关的默认值。

CONSTRAINT *constraint_name*

一个约束的名称（可选）。如果没有指定，系统会生成一个名称。

NOT NULL

这个域的值通常不能为空值（但是看看下面的注释）。

NULL

这个域的值允许为空值。这是默认值。

这个子句只是为了与非标准 SQL 数据库相兼容而设计。在新的 应用中不鼓励使用它。

CHECK (*expression*)

CHECK子句指定该域的值必须满足的完整性 约束或者测试。每一个约束必须是一个产生布尔结果的表达式。 它应该使用关键词VALUE来引用要被测试的值。计算为 TRUE 或者 UNKNOWN 的表达式成功。如果该表达式产生一个 FALSE 结果，会报告一个错误并且该值不允许被转换成该域类型。

当前，CHECK表达式不能包含子查询，也不能 引用除VALUE之外的其他变量。

当一个域有多个CHECK约束，将按照其名字的 字母顺序测试它们。

注解

在把一个值转换成域类型时会检查域约束（特别是NOT NULL）。 即使有一个这样的约束，有可能一个名义上属于该域类型的列也会被读成空值。 例如，如果在一次外连接查询中，属于该域的列出现在外连接的空值端。下面 是一个更精细的例子：

INSERT INTO tab (domcol) VALUES ((SELECT domcol FROM tab WHERE false));

空的标量子-SELECT 将产生一个空值，它被认为是该域类型的值，因此不会 在其上应用任何进一步的约束检查，并且插入将会成功。

要避免这类问题很难，因为 SQL 的一般假设是空值也是每一种数据类型的合法值。 因此，最好的方法是设计一个允许空值的域约束，然后根据需要在该域类型的列上 应用列的NOT NULL约束。

UXDB假定CHECK约束的条件是不可变的，也就是说，对于相同的输入值它们总会给出相同的结果。 仅在首次将值转换为域类型时，此假设检查CHECK约束的正确性，而不是在其他时候。（这与处理表CHECK约束基本上相同，如 第 2.4.1 节 “检查约束”中所述。）

打破此假设的常见方法的一个示例是在 CHECK表达式中引用一个用户定义的函数，然后更改该函数的行为。 UXDB不会禁止这样做，但是此间如果有域类型的存储值违反CHECK约束，它也不会注意到。这将导致后续数据库转储和重新加载失败。 处理此类更改的建议方法是删除约束（使用ALTER DOMAIN），调整函数定义，重新添加约束，然后根据存储的数据重新检查约束。

示例

这个例子创建us_postal_code数据类型并且把它用在 一个表定义中。一个正则表达式测试被用来验证值是否看起来像一个 合法的 US 邮政编码：

```
CREATE DOMAIN us_postal_code AS TEXT
CHECK(
   VALUE ~ '^\d{5}$'
OR VALUE ~ '^\d{5}-\d{4}$'
);

CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

兼容性

命令**CREATE DOMAIN**符合 SQL 标准。

另见

ALTER DOMAIN(7), DROP DOMAIN(7)

名称

CREATE EVENT TRIGGER — 定义一个新的事件触发器

大纲

```
CREATE EVENT TRIGGER name
  ON event
  [ WHEN filter_variable IN (filter_value [, ... ]) [ AND ... ] ]
  EXECUTE { FUNCTION | PROCEDURE } function_name()
```

描述

CREATE EVENT TRIGGER创建一个新的事件触发器。 只要指定的事件发生并且与该触发器相关的WHEN条件（如果有）被 满足，该触发器的函数将被执行。创建事件触发器的用户会成为它的拥有者。

参数

name

给新触发器的名称。在该数据库中这个名称必须唯一。

event

会触发对给定函数调用的事件名称。

filter_variable

用来过滤事件的变量名称。这可以用来限制触发器只为它支持的那一部分 情况引发。当前唯一支持的 *filter_variable* 是TAG。

filter_value

与该触发器要为其引发的 *filter_variable*相关联 的一个值列表。对于TAG，这表示一个命令标签列表（例如 'DROP FUNCTION'）。

function_name

一个用户提供的函数，它被声明为没有参数并且返回类型 event_trigger。

在CREATE EVENT TRIGGER的语法中，关键字CREATE EVENT TRIGGER和PROCEDURE是等效的，但是被引用的函数在任何情况下都必须是函数，而不是过程。 此处关键字PROCEDURE的使用是历史性的，已弃用。

注解

只有超级用户能创建事件触发器。

在单用户模式中事件触发器被禁用。 如果一个错误的事件触发器禁用了数据库让甚至无法删除它，可以重启到 单用户模式，这样就能删除它。

示例

禁止执行任何DDL命令：

```
CREATE OR REPLACE FUNCTION abort_any_command()
  RETURNS event_trigger
 LANGUAGE pluxsql
  AS $$
BEGIN
  RAISE EXCEPTION 'command % is disabled', tg_tag;
END;
$$;

CREATE EVENT TRIGGER abort_ddl ON ddl_command_start
   EXECUTE FUNCTION abort_any_command();
```

兼容性

在 SQL 标准中没有 **CREATE EVENT TRIGGER**语句。

另见

ALTER EVENT TRIGGER(7), DROP EVENT TRIGGER(7), CREATE FUNCTION(7)

名称

CREATE EXTENSION — 安装一个扩展

大纲

```
CREATE EXTENSION [ IF NOT EXISTS ] extension_name
  [ WITH ] [ SCHEMA schema_name ]
        [ VERSION version ]
        [ FROM old_version ]
        [ CASCADE ]
```

描述

CREATE EXTENSION把一个新的扩展载入到 当前数据库中。不能有同名扩展已经被载入。

载入一个扩展本质上是运行该扩展的脚本文件。该脚本通常将创建新的 SQL对象，例如函数、数据类型、操作符以及索引支持 方法。**CREATE EXTENSION**会额外地记录 所有被创建对象的标识，这样发出 **DROP EXTENSION**时可以删除它们。

载入一个扩展要求创建其组件对象所要求的特权。对于大部分扩展这意味 这需要超级用户或者数据库拥有者的特权。为了以后特权检察的目的，运行 **CREATE EXTENSION**的用户会成为该扩展的 拥有者以及由该扩展的脚本创建的任何对象的拥有者。

参数

IF NOT EXISTS

已有同名扩展存在时不要抛出错误。这种情况下会发出一个提示。 注意，不保证现有的扩展与将要从当前可用的脚本文件创建的脚本 有任何相似。

extension_name

要安装的扩展的名称。UXDB 将使用文件 SHAREDIR/extension/*extension_name*.control 中的指令来创建该扩展。

schema_name

假定该扩展允许其内容被重定位，这是要在其中安装该扩展的对象的 模式名称。被提到的模式必须已经存在。如果没有指定并且该扩展的 控制文件也没有指定一个模式，将使用当前的默认对象创建模式。

如果该扩展在其控制文件中指定了一个schema参数， 那么不能用SCHEMA子句覆盖该模式。通常，如果 给出了一个SCHEMA子句并且它与扩展的 schema参数冲突，则会发生错误。不过，如果也给 出了CASCADE子句，则schema冲突时会忽略 *schema_name*。 给定的*schema_name* 将被用来安装任何需要的并且没有在其控制文件中指定 schema的扩展。

记住扩展本身被认为不在任何模式中：扩展具有无限定的名称，并且 要在整个数据库范围内唯一。但是属于扩展的对象可以在模式中。

version

要安装的扩展的版本。这可以写成一个标识符或者一个字符串。 默认版本在该扩展的控制文件中指定。

old_version

当且仅当尝试要安装一个扩展来替代一个“老式” 的模块（它只是一组没有被打包成扩展的对象的集合）时， 才必须指定 FROM *old_version*。 这个选项导致CREATE EXTENSION运行 另一个安装脚本把现有的对象吸收到该扩展中，而不是创建 新对象。当心SCHEMA指定的是包含已经存在 对象的模式。

用于*old_version*的值由扩展的作者决定， 且如果有多于一种版本的老式模块可以被升级到扩展，该值还可能变化。

CASCADE

自动安装这个扩展所依赖的任何还未安装的扩展。它们的依赖也会同样 被自动安装。如果给出SCHEMA子句，它会应用于这种方式 下安装的所有扩展。这个语句的其他选项不会被应用于自动安装的扩展。 特别地，这些自动安装的扩展的默认版本将被选中。

注解

在使用CREATE EXTENSION载入扩展到数据库中之前， 必须先安装好该扩展的支持文件。关于安装 UXDB提供的扩展的信息可以在 额外提供的模块中找到。

当前可以用于载入的扩展可以在系统视图ux_available_extensions或者ux_available_extension_versions中看到。

示例

安装hstore扩展到当前数据库中：

```
CREATE EXTENSION hstore;
```

升级一个hstore安装成为扩展：

```
CREATE EXTENSION hstore SCHEMA public FROM unpackaged;
```

指定安装现有hstore对象的模式。

兼容性

CREATE EXTENSION是一种 UXDB扩展。

另见

ALTER EXTENSION(7), DROP EXTENSION(7)

名称

CREATE FOREIGN DATA WRAPPER — 定义一个新的外部数据包装器

大纲

```
CREATE FOREIGN DATA WRAPPER name
  [ HANDLER handler_function | NO HANDLER ]
  [ VALIDATOR validator_function | NO VALIDATOR ]
  [ OPTIONS ( option 'value' [, ... ] ) ]
```

描述

CREATE FOREIGN DATA WRAPPER创建一个 新的外部数据包装器。定义外部数据包装器的用户将成为它的拥有者。

在数据库内外部数据包装器名称必须唯一。

只有超级用户能够创建外部数据包装器。

参数

name

要创建的外部数据包装器的名称。

HANDLER *handler_function*

*handler_function*是一个以前注册 好的函数的名称，它将被调用来为外部表检索执行函数。处理器函数必 须不能有参数，并且它的返回类型必须是fdw_handler。

可以创建一个没有处理器函数的外部数据包装器，但是使用这个包装 器的外部表只能被声明，但不能被访问。

VALIDATOR *validator_function*

validator_function 是一个之前已注册的函数的名称，它将被调用来检查给予该外部数据包装器 的选项，还有用于外部服务器、用户映射以及使用 该外部数据包装器的外部表的选项。如果没有验证器函数或者指定了 NO VALIDATOR，那么在创建时不会检查选项（ 外部数据包装器可能会在运行时忽略或者拒绝无效的选项说明，这取决于 实现）。验证器函数必须接受两个参数：一个类型是text[]， 它将包含存储在系统目录中的选项数组，另一个是类型 oid，它将是包含该选项的系统目录的 OID。返回类型 会被忽略，该函数应该使用ereport(ERROR) 函数报告无效选项。

OPTIONS (*option* '*value*' [, ...])

这个子句为新的外部数据包装器指定选项。允许的选项名称和值与每一个 外部数据包装器有关，并且它们会被该外部数据包装器的验证器函数验证。 选项名称必须唯一。

注解

UXDB的外部数据功能仍在积极的开发中。 查询的优化还很原始（也是剩下工作最多的部分）。因此，未来还有很 可观的性能提升空间。

示例

创建一个无用的外部数据包装器dummy：

```
CREATE FOREIGN DATA WRAPPER dummy;
```

用处理器函数file_fdw_handler创建一个外部数据包装器 file：

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

用一些选项创建一个外部数据包装器mywrapper：

```
CREATE FOREIGN DATA WRAPPER mywrapper
  OPTIONS (debug 'true');
```

兼容性

CREATE FOREIGN DATA WRAPPER 符合 ISO/IEC 9075-9 (SQL/MED)，不过HANDLER和VALIDATOR子句是扩展，并且标准子句 LIBRARY和LANGUAGE还没有 在UXDB中被实现。

不过要注意，整体上来说 SQL/MED 功能还没有符合。

另见

ALTER FOREIGN DATA WRAPPER(7), DROP FOREIGN DATA WRAPPER(7), CREATE SERVER(7), CREATE USER MAPPING(7), CREATE FOREIGN TABLE(7)

名称

CREATE FOREIGN TABLE — 定义一个新的外部表

大纲

```
CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ OPTIONS ( option 'value' [, ... ] ) ] [ COLLATE collation ]
 [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]

CREATE FOREIGN TABLE [ IF NOT EXISTS ] table_name
  PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] partition_bound_spec
  SERVER server_name
[ OPTIONS ( option 'value' [, ... ] ) ]
```

其中 *column_constraint* 是：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED }
```

而 *table_constraint* 是：

```
[ CONSTRAINT constraint_name ]
CHECK ( expression ) [ NO INHERIT ]
```

描述

CREATE FOREIGN TABLE在当前数据库中创建 一个新的外部表。该表将由发出这个命令的用户所拥有。

如果给定了一个模式名称（例如CREATE FOREIGN TABLE myschema.mytable ...），那么该表会被创建在指定的模式中。 否则它会被创建在当前模式中。该外部表的名称必须与同一个模式中的任何其他外部表、表、序列、索引、视图或者物化视图区分开来。

CREATE FOREIGN TABLE还将自动创建 一个数据类型来表示该外部表行相应的组合类型。因此，外部表不能和 同一个模式中任何现有的数据类型同名。

如果指定了PARTITION OF子句， 则该表被创建为具有指定边界的parent_table的分区。

要创建一个外部表，必须具有该外部服务器上的USAGE 特权，以及该表中用到的所有列类型上的USAGE特权。

参数

IF NOT EXISTS

已经存在同名关系时不要抛出错误。这种情况下会发出一个提示。注意， 并不保证已经存在的关系与将要创建的那一个相似。

table_name

要创建的表的名称（可以被模式限定）。

column_name

要在新表中创建的列名。

data_type

该列的数据类型。可以包括数组指示符。更多 UXDB支持的数据类型可见第 5 章 数据类型

COLLATE *collation*

COLLATE子句为该列（必须是一个可排序的数据类型） 赋予一个排序规则。如果没有指定，则会使用该列的数据类型的默认 排序规则。

INHERITS (*parent_table* [, ...])

可选的INHERITS子句指定了一个表的列表，新的外部表 会自动从中继承所有列。父表可以是普通表或者外部表。详见 CREATE TABLE(7)的类似形式。

PARTITION OF *parent_table* FOR VALUES *partition_bound_spec*

此语句可以用来将外部表创建为父表的一个指定区间的分区表。 详见CREATE TABLE(7)的类似形式。 注意如果父表存在UNIQUE类型的索引时，当前是不允许将外部表 创建为该父表的分区表。（另见 ALTER TABLE ATTACH PARTITION。）

CONSTRAINT *constraint_name*

一个可选的用于列或者表约束的名字。如果该约束被违背，这个约束名字会 出现在错误消息中，这样col must be positive这种约束名就 可以被用来与客户端应用交流有用的约束信息（指定包含空格的约束名需要 使用双引号）。如果没有指定约束名，系统会自动生成一个。

NOT NULL

该列不允许包含空值。

NULL

该列可以包含空值，这是默认值。

提供这个子句只是为了兼容非标准的 SQL 数据库。在新的应用中 不鼓励使用它。

CHECK (*expression*) [NO INHERIT]

CHECK子句指定一个产生布尔结果的表达式，该外部表 中的每一行都应该满足该表达式。也就是说，对于该外部表中所有的行， 这个表达式应该产生 TRUE 或者 UNKNOWN 而不能产生 FALSE。被 作为列约束指定的检查约束应该只引用该列的值，而出现在表约束中的 表达式可以引用多列。

当前，CHECK表达式不能包含子查询，也不能 引用除当前行的列之外的其他变量。可以引用系统列 tableoid，但是不能引用其他系统列。

被标记为NO INHERIT的约束将不会传播到子表上。

DEFAULT *default_expr*

DEFAULT子句为包含它的列定义赋予一个默认数据值。该 值是任意不包含变量的表达式（不允许子查询和对当前表中其他列的交叉 引用）。默认值表达式的数据类型必须匹配列的数据类型。

默认值表达式将被用在任何没有指定列值的插入操作中。如果一列没有 默认值，则默认值为空值。

GENERATED ALWAYS AS (*generation_expr*) STORED

该子句创建一个生成列。 此列不能写入，只能在读取时返回所指定的表达式的值。

关键字STORED是必须的，用来表明该列将在写入时计算。 （计算出的值将会传递给外部数据封装器负责保存，并在读取时返回。）

生成表达式可以引用表中的其他列，但不能引用其他的生成列。 所使用的函数和操作符必须是不可变。 不能引用其他表。

server_name

要用于该外部表的一个现有外部服务器的名称。有关定义一个服务器 的细节可以参考CREATE SERVER(7)。

OPTIONS (*option* '*value*' [, ...])

要与新外部表或者它的一个列相关联的选项。被允许的选项名称和值是与 每一个外部数据包装器相关的，并且它们会被该外部数据包装器的验证器 函数验证。不允许重复的选项名称（不过一个表选项和一个列选项重名是 可以的）。

注解

UXDB核心系统不会强制外部表上的约束（ 例如CHECK或NOT NULL子句），大部分外部 数据包装器也不会尝试强制它们。也就是说，这类约束会被简单地认为保持 为真。这种强制其实没什么意义，因为它只适用于通过该外部表插入或者更 新的行，而对通过其他方式修改的行（例如直接在远程服务器上修改）没有 作用。附着在外部表上的约束应该表示由外部服务器强制的一个约束。

有些特殊目的的外部数据包装器可能是它们所访问的数据的唯一一种访问 机制，在那种情况下让外部数据包装器自己来执行约束强制可能是合适的。 但是不应该假设包装器会这样做，除非它的文档说它会。

尽管UXDB不会尝试强制外部表上的约束， 但它确实假定它们对于查询优化的目的是正确的。如果在外部表中有不 满足约束的行可见，在该表上的查询可能会产生不正确的回答。确保 约束定义符合实际是用户的责任。

类似的要点也适用于生成列。生成列在本地UXDB 服务器发生插入或更新时进行计算，并将计算结果传递给外部数据封装器，由外部数据 封装器负责把结果写到外部数据存储中，但是并不强制要求查询外部表时返回的生成列 的值一定是与生成表达式一致。 总而言之，此行为可能导致查询到不正确的结果。

虽然列可以从本地分区移到外部表分区（使用的外部数据封装器支持元组路由），但是 不能将列从外部表分区移动到本地分区。

示例

创建外部表films，通过服务器film_server访问它：

```
CREATE FOREIGN TABLE films (
    code        char(5) NOT NULL,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
)
SERVER film_server;
```

创建外部表measurement_y2016m07，通过服务器 server_07访问它，作为范围分区表measurement 的分区：

```
CREATE FOREIGN TABLE measurement_y2016m07
    PARTITION OF measurement FOR VALUES FROM ('2016-07-01') TO ('2016-08-01')
    SERVER server_07;
```

兼容性

CREATE FOREIGN TABLE命令大部分符合 SQL标准。不过，与 **CREATE TABLE** 很相似，它允许NULL约束以及零列外部表。能够指定列默认值 也是一种UXDB扩展。 UXDB定义的表继承形式是非标准的。

另见

ALTER FOREIGN TABLE(7), DROP FOREIGN TABLE(7), CREATE TABLE(7), CREATE SERVER(7), IMPORT FOREIGN SCHEMA(7)

名称

CREATE FUNCTION — 定义一个新函数

大纲

```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SUPPORT support_function
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
```

描述

CREATE FUNCTION定义一个新函数。CREATE OR REPLACE FUNCTION将创建一个新函数或者替换一个现有的函数。要定义一个函数，用户必须具有该语言上的USAGE特权。

如果包括了一个模式名，那么该函数会被创建在指定的模式中。否则，它会被创建在当前模式中。新函数的名称不能匹配同一个模式中具有相同输入参数类型的任何现有函数或过程。不过，不同参数类型的函数和过程能够共享一个名字（这被称作重载）。

要替换一个现有函数的当前定义，可以使用CREATE OR REPLACE FUNCTION。但不能用这种方式更改函数的名称或者参数类型（如果尝试这样做，实际上就会创建一个新的不同的函数）。还有，CREATE OR REPLACE FUNCTION将不会让更改一个现有函数的返回类型。要这样做，必须先删除再重建该函数（在使用OUT参数时，这意味着除了删除函数之外无法更改任何OUT参数的类型）。

当CREATE OR REPLACE FUNCTION被用来替换一个现有的函数，该函数的拥有权和权限不会改变。所有其他的函数属性会按照该命令中所指定的或者隐含的来赋值。必须拥有（包括成为拥有角色的成员）该函数才能替换它。

如果删除并且重建一个函数，新函数将和旧的不一样，将必须删掉引用旧函数的现有规则、视图、触发器等。使用CREATE OR REPLACE FUNCTION更改一个函数定义不会破坏引用该函数的对象。还有，ALTER FUNCTION可以被用来更改一个现有函数的大部分辅助属性。

创建该函数的用户将成为该函数的拥有者。

要创建一个函数，必须拥有参数类型和返回类型上的USAGE特权。

参数

name

要创建的函数的名称（可以被模式限定）。

argmode

一个参数的模式：IN、OUT、INOUT或者VARIADIC。如果省略，默认为IN。只有OUT参数能跟在一个VARIADIC参数后面。还有，OUT和INOUT参数不能和RETURNS TABLE符号一起使用。

argname

一个参数的名称。一些语言（包括 SQL 和 PL/uxSQL）让在函数体中使用该名称。对于其他语言，一个输入参数的名字只是额外的文字（就该函数本身所关心的来说）。但是可以在调用一个函数时使用输入参数名来提高可读性（见第 1.3 节 “调用函数”。在任何情况下，输出参数的名称是有意义的，因为它定义了结果行类型中的列名（如果忽略一个输出参数的名称，系统将选择一个默认的列名）。

argtype

该函数参数（如果有）的数据类型（可以是模式限定的）。参数类型可以是基本类型、组合类型或者域类型，或者可以引用一个表列的类型。

根据实现语言，也可以允许指定cstring之类的“伪类型”。伪类型表示实际参数类型没有被完整指定或者不属于普通 SQL 数据类型集合。

可以写*table_name.column_name*%TYPE来引用一列的类型。使用这种特性有时可以帮助创建一个不受表定义更改影响的函数。

default_expr

如果参数没有被指定值时要用作默认值的表达式。该表达式必须能被强制为该参数的参数类型。只有输入（包括INOUT）参数可以具有默认值。所有跟随在一个具有默认值的参数之后的输入参数也必须有默认值。

rettype

返回数据类型（可能被模式限定）。返回类型可以是一种基本类型、组合类型或者域类型，也可以引用一个表列的类型。根据实现语言，也可以允许指定cstring之类的“伪类型”。如果该函数不会返回一个值，可以指定返回类型为void。

当有OUT或者INOUT参数时，可以省略RETURNS子句。如果存在，该子句必须和输出参数所表示的结果类型一致：如果有多个输出参数，则为RECORD，否则与单个输出参数的类型相同。

SETOF修饰符表示该函数将返回一个项的集合而不是一个单一项。

可以写*table_name.column_name*%TYPE来引用一列的类型。

column_name

RETURNS TABLE语法中一个输出列的名称。这实际上是另一种声明OUT参数的方法，不过RETURNS TABLE也隐含了RETURNS SETOF。

column_type

RETURNS TABLE语法中的输出列的数据类型。

lang_name

用以实现该函数的语言的名称。可以是sql、c、internal或者一个用户定义的过程语言的名称，例如pluxsql。不推荐用单引号包围该名称，并且要求区分大小写。

TRANSFORM { FOR TYPE *type_name* } [, ...] }

一个由转换构成的列表，对该函数的调用适用于它们。转换在 SQL 类型和语言相关的数据类型之间进行变换，详见CREATE TRANSFORM(7)。过程语言实现通常把有关内建类型的知识硬编码在代码中，因此那些不需要列举在这里。如果一种过程语言实现不知道如何处理一种类型并且没有转换被提供，它将回退到一种默认的行为来转换数据类型，但是这取决于具体实现。

WINDOW

WINDOW表示该函数是一个窗口函数而不是一个普通函数。当前只用于用 C 编写的函数。在替换一个现有函数定义时，不能更改WINDOW属性。

IMMUTABLE
STABLE
VOLATILE

这些属性告知查询优化器该函数的行为。最多只能指定其中一个。如果这些都不出现，则会默认为VOLATILE。

IMMUTABLE表示该函数不能修改数据库并且对于给定的参数值总是会返回相同的值。也就是说，它不会做数据库查找或者使用没有在其参数列表中直接出现的信息。如果给定合格选项，任何用全常量参数对该函数的额调用可以立刻用该函数值替换。

STABLE表示该函数不能修改数据库，并且对于相同的参数值，它在一次表扫描中将返回相同的结果。但是这种结果在不同的 SQL 语句执行期间可能会变化。对于那些结果依赖于数据库查找、参数变量（例如当前时区）等的函数来说，这是合适的（对希望查询被当前命令修改的行的AFTER触发器不适合）。还要注意current_timestamp函数族适合被标记为稳定，因为它们的值在一个事务内不会改变。

VOLATILE表示该函数的值在一次表扫描中都有可能改变，因此不能做优化。在这种意义上，相对较少的数据库函数是不稳定的，一些例子是random()、currval()、timeofday()。但是注意任何有副作用的函数都必须被分类为不稳定的，即便其结果是可以预测的，这是为了调用被优化掉。一个例子是setval()。

LEAKPROOF

LEAKPROOF表示该函数没有副作用。它不会泄露有关其参数的信息（除了通过返回值）。例如，一个只对某些参数值抛出错误消息而对另外一些却不抛出错误的函数不是防泄漏的，一个把参数值包括在任何错误消息中的函数也不是防泄漏的。这会影响系统如何执行在使用security_barrier选项创建的视图或者开启了行级安全性的表上执行查询。对于包含有非防泄漏函数的查询，系统将在任何来自查询本身的用户提供条件之前强制来自安全策略或者安全屏障的条件，防止无意中的数据暴露。被标记为防泄漏的函数和操作符被假定是可信的，并且可以在安全性策略和安全性屏障视图的条件之前被执行。此外，没有参数的函数或者不

从安全屏障视图或表传递任何参数的函数不一定要被标记为防泄漏的。详见CREATE VIEW(7)。这个选项只能由超级用户设置。

CALLED ON NULL INPUT
RETURNS NULL ON NULL INPUT
STRICT

CALLED ON NULL INPUT（默认）表示在某些参数为空值时应正常调用该函数。如果有必要，函数的作者应该负责检查空值并且做出适当的相应。

RETURNS NULL ON NULL INPUT或STRICT表示只要其任意参数为空值，该函数就会返回空值。如果指定了这个参数，当有空值参数时该函数不会被执行，而是自动返回一个空值结果。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER表示要用调用该函数的用户的特权来执行它。这是默认值。SECURITY DEFINER指定要用拥有该函数的用户的特权来执行该函数。

为了符合 SQL，允许使用关键词EXTERNAL。但是它是可选的，因为与 SQL 中不同，这个特性适用于所有函数而不仅是那些外部函数。

PARALLEL

PARALLEL UNSAFE表示该函数不能在并行模式中运行并且 SQL 语句中存在一个这样的函数会强制使用顺序执行计划。这是默认选项。PARALLEL RESTRICTED表示该函数能在并行模式中运行，但是其执行被限制在并行组的领导者中。PARALLEL SAFE表示该函数对于在并行模式中运行是安全的并且不受限制。

如果函数修改任何数据库状态、会使用子事务之类的方式改变事务、访问序列或者对设置（如setval）做出持久性的更改，它们就应该被标记为并行不安全。如果它们访问临时表、客户端连接状态、游标、预备语句或者系统无法在并行模式中同步的本地后端状态（例如setseed只能在组领导者中执行，因为另一个进程所作的更改不会在领导者中被反映出来），它们应该被标为并行受限。通常，如果一个函数是受限的或者不安全的却被标成了安全，或者它本来是不安全的却被标成了受限，在并行查询中执行时它可能会抛出错误或者产生错误的答案。如果被错误的标记， C 语言函数理论上可能展现出完全无法定义的行为，因为系统没有办法保护自己不受任意的 C 代码影响，但是在大部分情况下其结果也不会比任何其他函数差到哪里去。如果有疑问，函数应该被标为UNSAFE，这也是默认值。

COST *execution_cost*

一个给出该函数的估计执行代价的正数，单位是cpu_operator_cost。如果该函数返回一个集合，这就是每个被返回行的代价。如果没有指定代价，对 C 语言和内部函数会指定为 1 个单位，对其他语言的函数则会指定为 100 单位。更大的值会导致规划器尝试避免对该函数的不必要的过多计算。

ROWS *result_rows*

一个正数，它给出规划器期望该函数返回的行数估计。只有当该函数被声明为返回一个集合时才允许这个参数。默认假设为 1000 行。

SUPPORT *support_function*

用于此函数的`planner support function`的名称（可选的模式限定）。必须是超级用户才能使用此选项。

configuration_parameter
value

SET子句导致进入该函数时指定配置参数将被设置为指定值。并且在该函数退出时恢复到该参数之前的值。SET FROM CURRENT会把CREATE FUNCTION被执行时该参数的当前值保存为进入该函数时将被应用的值。

如果一个SET子句被附加到一个函数，那么在该函数内为同一个变量执行的SET LOCAL命令会被限制于该函数：在函数退出时该配置参数之前的值仍会被恢复。不过，一个普通的SET命令（没有LOCAL）会覆盖SET子句，更像一个之前的SET LOCAL命令所做的那样：这种命令的效果在函数退出后将会持续，除非当前事务被回滚。

更多有关允许的参数名和参数值的信息请见SET(7)。

definition

一个定义该函数的字符串常量，其含义取决于语言。它可以是一个内部函数名、一个对象文件的路径、一个 SQL 命令或者用一种过程语言编写的文本。

美元引用第 1.1.2.4 节 “美元引用的字符串常量”通常对书写函数定义字符串有所帮助，而普通单引号语法则不会有用。如果没有美元引用，函数定义中的任何单引号或者反斜线必须用双写来转义。

obj_file, *link_symbol*

当 C 语言源代码中该函数的名称与 SQL 函数的名称不同时，这种形式的AS子句被用于动态可载入 C 语言函数。字符串*obj_file*是包含编译好的C函数的动态库文件的名称，它会由LOAD(7)命令解析。字符串*link_symbol*是该函数的链接符号，也就是该函数在 C 语言源代码中的名称。如果省略链接符号，它将被假定为要定义的 SQL 函数的名称。所有函数的C名称都必须不同，因此必须为重载的C函数给出不同的C名称（例如把参数类型作为C名称的一部分）。

在重复调用引用同一对象文件的CREATE FUNCTION时，对每个会话该文件只会被载入一次。要卸载并且重新装载该文件（可能是在开发期间），需要开始一个新会话。

重载

UXDB允许函数重载，也就是说同一个名称可以被用于多个不同的函数，只要它们具有可区分的输入参数类型。不管是否使用它，在有些用户不信任另一些用户的数据库中调用函数时，这种兼容性需要安全性的预防措施，请参考第 7.3 节 “函数”。

如果两个函数具有相同的名称和输入参数类型，它们被认为相同（不考虑任何OUT参数）。因此这些声明会冲突：

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, out text) ...
```

具有不同参数类型列表的函数在创建时将不会被认为是冲突的，但是如果默认值被提供，在使用时它们有可能会冲突。例如，考虑

```
CREATE FUNCTION foo(int) ...
CREATE FUNCTION foo(int, int default 42) ...
```

调用foo(10)将会失败，因为在要决定应该调用哪个函数时会有歧义。

注解

允许把完整的SQL类型语法用于声明一个函数的参数和返回值。不过，CREATE FUNCTION会抛弃带圆括号的类型修饰符（例如类型numeric的精度域）。例如CREATE FUNCTION foo (varchar(10)) ...和CREATE FUNCTION foo (varchar) ...完全一样。

在用CREATE OR REPLACE FUNCTION替换一个现有函数时，对于更改参数名是有限制的。不能更改已经分配给任何输入参数的名称（不过可以给之前没有名称的参数增加名称）。如果有多于一个输出参数，不能更改输出参数的名称，因为可能会改变描述函数结果的匿名组合类型的列名。这些限制是为了确保函数被替换时，已有的对该函数的调用不会停止工作。

如果一个被声明为STRICT的函数带有一个VARIADIC参数，会严格检查该可变数组作为一个整体是否为非空。如果该数组有空值元素，该函数仍将被调用。

示例

这里是一些小例子，它们可以帮了解函数创建。

```
CREATE FUNCTION add(integer, integer) RETURNS integer
   AS 'select $1 + $2;'
   LANGUAGE SQL
   IMMUTABLE
   RETURNS NULL ON NULL INPUT;
```

在PL/uxSQL中，使用一个参数名称增加一个整数：

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
      BEGIN
            RETURN i + 1;
      END;
$$ LANGUAGE pluxsql;
```

返回一个包含多个输出参数的记录：

```
CREATE FUNCTION dup(in int, out f1 int, out f2 text)
   AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
   LANGUAGE SQL;

SELECT * FROM dup(42);
```

可以用更复杂的方式（用一个显式命名的组合类型）来做同样的事情：

```
CREATE TYPE dup_result AS (f1 int, f2 text);

CREATE FUNCTION dup(int) RETURNS dup_result
   AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
   LANGUAGE SQL;

SELECT * FROM dup(42);
```

另一种返回多列的方法是使用一个TABLE函数：

```
CREATE FUNCTION dup(int) RETURNS TABLE(f1 int, f2 text)
   AS $$ SELECT $1, CAST($1 AS text) || ' is text' $$
   LANGUAGE SQL;

SELECT * FROM dup(42);
```

不过，TABLE函数与之前的例子不同，因为它实际返回了一个记录集合而不只是一个记录。

安全地编写 SECURITY DEFINER函数

因为一个SECURITY DEFINER函数会被以创建它的用户的特权来执行，需要注意确保该函数不会被误用。为了安全，search_path应该被设置为排除任何不可信用户可写的模式。这可以阻止恶意用户创建对象（例如表、函数以及操作符）来掩饰该函数所要用到的对象。在这方面特别重要的是临时表模式，默认情况下它会第一个被搜索并且通常对任何用户都是可写的。可以通过强制最后搜索临时模式来得到一种安全的布局。要这样做，把ux_temp写成search_path中的最后一项。这个函数展示了安全的用法：

```
CREATE FUNCTION check_password(uname TEXT, pass TEXT)
RETURNS BOOLEAN AS $$
DECLARE passed BOOLEAN;
BEGIN
      SELECT  (pwd = $2) INTO passed
      FROM    pwds
      WHERE   username = $1;

      RETURN passed;
END;
$$  LANGUAGE pluxsql
   SECURITY DEFINER
   -- 设置一个安全的 search_path：受信的模式，然后是 'ux_temp'。
   SET search_path = admin, ux_temp;
```

这个函数的目的是为了访问表admin.pwds。但是如果没有SET子句或者带有SET子句却只提到admin，该函数会变成创建一个名为pwds的临时表。

另一点要记住的是默认情况下，会为新创建的函数给PUBLIC授予执行特权（详见第 2.7 节“权限”）。常常会希望把安全定义器函数的使用限制在某些用户中。要这样做，必须收回默认的PUBLIC特权，然后选择性地授予执行特权。为了避免出现新函数能被所有人访问的时间窗口，应在一个事务中创建它并且设置特权。例如：

```
BEGIN;
CREATE FUNCTION check_password(uname TEXT, pass TEXT) ... SECURITY DEFINER;
REVOKE ALL ON FUNCTION check_password(uname TEXT, pass TEXT) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION check_password(uname TEXT, pass TEXT) TO admins;
COMMIT;
```

兼容性

SQL标准中定义了CREATE FUNCTION命令。

对于和一些其他数据库系统的兼容性，*argmode*可以被写在*argname*之前或者之后。但只有第一种方式是兼容标准的。

对于参数默认值，SQL 标准只指定带有DEFAULT关键词的语法。带有=的语法被用在 T-SQL 和 Firebird 中。

另见

ALTER FUNCTION(7), DROP FUNCTION(7), GRANT(7), LOAD(7), REVOKE(7)

名称

CREATE GROUP — 定义一个新的数据库角色

大纲

CREATE GROUP *name* [[WITH] *option* [...]]

其中 *option* 可以是：

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | [ ENCRYPTED ] PASSWORD 'password'
  | VALID UNTIL 'timestamp'
  | IN ROLE role_name [, ...]
  | IN GROUP role_name [, ...]
  | ROLE role_name [, ...]
  | ADMIN role_name [, ...]
  | USER role_name [, ...]
  | SYSID uid
```

描述

CREATE GROUP现在是 CREATE ROLE(7)的一种别名。

兼容性

在 SQL 标准中没有**CREATE GROUP**语句。

另见

CREATE ROLE(7)

名称

CREATE INDEX — 定义一个新索引

大纲

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
 [ ONLY ] table_name [ USING method ]
   ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS
 { FIRST | LAST } ] [, ...] )
   [ INCLUDE ( column_name [, ...] ) ]
   [ WITH ( storage_parameter = value [, ... ] ) ]
   [ TABLESPACE tablespace_name ]
   [ WHERE predicate ]
```

描述

CREATE INDEX在指定关系的指定列上构建 一个索引，该关系可以是一个表或者一个物化视图。索引主要被用来提升 数据库性能（不过不当的使用会导致性能变差）。

索引的键域被指定为列名或者写在圆括号中的表达式。如果索引方法支持 多列索引，可以指定多个域。

一个索引域可以是一个从表行的一列或者更多列值进行计算的表达式。 这种特性可以被用来获得对基于基本数据某种变换的数据的快速访问。 例如，一个在upper(col)上计算的索引可以允许子句 WHERE upper(col) = 'JIM'使用索引。

UXDB提供了索引方法 B-树、哈希、GiST、SP-GiST、GIN 以及 BRIN。用户也可以定义自己的索引 方法，但是相对较复杂。

当WHERE子句存在时，会创建一个 *部分索引*。部分索引只包含表中一部分行的项， 通常索引这一部分会比表的其他部分更有用。例如，如果有一个表包含了 已付和未付订单，其中未付订单占了整个表的一小部分并且是经常被使用 的部分，可以通过只在这一部分上创建一个索引来改进性能。另一种可能 的应用是使用带有UNIQUE的 WHERE在表的一个子集上强制唯一性。更多的讨论 请见第 8.8 节 “部分索引”

WHERE子句中使用的表达式只能引用底层表的列，但 它可以引用所有列而不仅仅是被索引的列。当前， WHERE中也禁止使用子查询和聚集表达式。同样的 限制也适用于表达式索引中的表达式域。

所有在索引定义中使用的函数和操作符必须是“不可变的”， 就是说它们的结果必须仅依赖于它们的参数而不受外在因素（例如另 一个表的内容和当前的时间）的影响。这种限制确保了索引的行为是 良定的。要在一个索引表达式或者WHERE子句中 使用用户定义的函数，记住在创建函数时把它标记为不可变。

参数

UNIQUE

导致系统在索引被创建时（如果数据已经存在）或者加入数据时 检查重复值。会导致重复项的数据插入或者更新尝试将会产生一 个错误。

当唯一索引被应用在分区边上时会有额外的限制，请参考CREATE TABLE(7)。

CONCURRENTLY

当使用了这个选项时，UXDB在构建索引时　不会取得任何会阻止该表上并发插入、更新或者删除的锁。而标准的索引　构建将会把表锁住以阻止对表的写（但不阻塞读），这种锁定会持续到索　引创建完毕。在使用这个选项时有多个需要注意的地方　—　请见　“并发构建索引”一节。

对于临时表，CREATE　INDEX始终是非并发的，因为没有其他会话可以访问它们，并且创建非并发索引的成本更低。

IF NOT EXISTS

如果一个同名关系已经存在则不要抛出错误。这种情况下会发出一个提示。　注意着并不保证现有的索引与将要创建的索引有任何相似。当　IF　NOT　EXISTS被指定时，需要指定索引名。

INCLUDE

可选的INCLUDE子句指定一个列的列表，其中的列将被包括在索引中作为非键列。非键列不能作为索引扫描的条件，并且该索引所强制的任何唯一性或者排除约束都不会考虑它们。不过，只用索引的扫描可以返回非键列的内容而无需访问该索引的基表，因为在索引项中就能直接拿到它们。因此，非键列的增加允许查询使用只用索引的扫描，否则就无法使用。

保守地向索引中增加非键列是明智的，特别是很宽的列。如果一个索引元组超过索引类型允许的最大尺寸，数据插入将会失败。在任何情况下，非键列都会重复来自索引基表的数据并且让索引的尺寸膨胀，因此可能会拖慢搜索。

INCLUDE子句中列出的列不需要合适的操作符类，甚至数据类型没有为给定的访问方法定义操作符类的列都可以包括在这个子句中。

不支持把表达式作为被包括列，因为它们不能被用在只用索引的扫描中。

当前，有B-树和GiST索引访问方法支持这一特性。在B-树和GiST索引中，INCLUDE子句中列出的列的值被包括在对应于堆元组的叶子元组中，但是不包括在用于树导航的上层索引项中。

name

要创建的索引名称。这里不能包括模式名，因为索引总是被创建在其基表所在　的模式中。如果索引名称被省略，UXDB将基于基 表名称和被索引列名称选择一个合适的名称。

ONLY

如果该表是分区表，指示不要在分区上递归创建索引。默认会递归创建索引。

table_name

要被索引的表的名称（可以被模式限定）。

method

要使用的索引方法的名称。可以选择　btree、hash、　gist、suxist、　gin以及brin。　默认方法是btree。

column_name

一个表列的名称。

expression

一个基于一个或者更多个表列的表达式。如语法中所示，表达式通常必须　　被写在圆括号中。不过，如果该表达式是一个函数调用的形式，圆括号可 以被省略。

collation

要用于该索引的排序规则的名称。默认情况下，该索引使用被索引列　的排序规则或者被索引表达式的结果排序规则。当查询涉及到使用非　默认排序规则的表达式时，使用非默认排序规则的索引就能排上用场。

opclass

一个操作符类的名称。详见下文。

ASC

指定上升排序（默认）。

DESC

指定下降排序。

NULLS FIRST

指定把空值排序在非空值前面。在指定DESC时， 这是默认行为。

NULLS LAST

指定把空值排序在非空值后面。在没有指定DESC时， 这是默认行为。

storage_parameter

索引方法相关的存储参数的名称。详见 “索引存储参数”一节。

tablespace_name

在其中创建索引的表空间。如果没有指定，将会使用　default_tablespace。或者对临时表上的索引使用 temp_tablespaces。

predicate

部分索引的约束表达式。

索引存储参数

可选的WITH子句为索引指定存储参数。每一种　索引方法都有自己的存储参数集合。B-树、哈希、GiST以及SP-GiST索引方法都接受这个参数：

fillfactor

索引的填充因子是一个百分数，它决定索引方法将尝试填充索引页面的充满程度。对于B-树，在初始的索引构建过程中，叶子页面会被填充至该百分数，当在索引右端扩展索引（增加新的最大键值）时也会这样处理。如果页面后来被完全填满，它们就会被分裂，导致索引的效率逐渐退化。B-树使用了默认的填充因子 90，但是也可以选择为 10 到 100 的任何整数值。如果表是静态的，那么填充因子 100 是最好的，因为它可以让索引的物理尺寸最小化。但是对于更新负荷很重的表，较小的填充因子有利于最小化对页面分裂的需求。其他索引方法以不同但是大致类似的方式使用填充因子，不同方法的默认填充因子也不相同。

B-树索引还额外接受这个参数：

vacuum_cleanup_index_scale_factor

vacuum_cleanup_index_scale_factor针对每个索引的值。

GiST还额外接受这个参数：

buffering

决定是否用缓冲构建技术来构建索引。OFF会禁用它，ON则启用该特性，如果设置为AUTO则初始会禁用它，但是一旦索引尺寸到达effective_cache_size就会随时打开。默认值是AUTO。

GIN索引接受不同的参数：

fastupdate

这个设置控制快速更新技术的使用。它是一个布尔参数：ON启用快速更新， OFF禁用之。默认是 ON。

> **注意**
>
> 通过ALTER INDEX关闭fastupdate 会阻止未来的更新进入到待处理索引项列表中，但它不会自己处理之前的 待处理项。可以使用VACUUM或者调用gin_clean_pending_list确保处理完待处理列表的项。

gin_pending_list_limit

自定义gin_pending_list_limit参数。这个值 要以千字节来指定。

BRIN索引接受不同的参数：

pages_per_range

定义用于每一个BRIN索引项的块范围由多少个表块组成。默认是128。

autosummarize

定义是否只要在下一个页面上检测到插入就为前面的页面范围运行概要操作。

并发构建索引

创建索引可能会干扰数据库的常规操作。通常 UXDB会锁住要被索引的表，让它不能被写入， 并且用该表上的一次扫描来执行整个索引的构建。其他事务仍然可以读取表 ， 但是如果它们尝试在该表上进行插入、更新或者删除，它们会被阻塞直到索引 构建完成。如果系统是一个生产数据库，这可能会导致严重的后果。索引非常 大的表可能会需要很多个小时，而且即使是较小的表，在构建索引过程中阻塞 写入者一段时间在生产系统中也是不能接受的。

UXDB支持构建索引时不阻塞写入。这种方法通过 指定CREATE INDEX的CONCURRENTLY选项 实现。当使用这个选项时，UXDB必须执行该表的 两次扫描，此外它必须等待所有现有可能会修改或者使用该索引的事务终止。因此这种 方法比起标准索引构建过程来说要做更多工作并且需要更多时间。不过，由于它 允许在构建索引时继续普通操作，这种方式对于在生产环境中增加新索引很有用。 当然，由索引创建带来的额外 CPU 和 I/O 开销可能会拖慢其他操作。

在并发索引构建中，索引实际上在一个事务中被录入到系统目录，然后在两个 事务中发生两次表扫描。在每一次表扫描之前，索引构建必须等待已经修改了 表的现有事务终止。在第二次扫描之后，索引构建必须等待任何持有早于第二 次扫描的快照（见第 10 章 并发控制的事务终止。然后该索引最终 能被标记为准备好使用，并且CREATE INDEX命令终止。 不过即便那样，该索引也不是立刻可以用于查询：在最坏的情况下，只要早于 索引构建开始时存在的事务存在，该索引就无法使用。

如果在扫描表示出现问题，例如死锁或者唯一索引中的唯一性被违背， CREATE INDEX将会失败，但留下一个“不可用” 的索引。这个索引会被查询所忽略，因为它可能不完整。不过它仍将消耗更新 开销。uxsql的\d命令将把这类索引报告为 INVALID：

```
uxdb=# \d tab
      Table "public.tab"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 col    | integer |           |          |
Indexes:
   "idx" btree (col) INVALID
```

这种情况下推荐的恢复方法是删除该索引并且尝试再次执行CREATE INDEX CONCURRENTLY。 （另一种可能性是用REINDEX INDEX CONCURRENTLY重建该索引）。

并发构建一个唯一索引时需要注意的另一点是，当第二次表扫描开始时，唯一约束 已经被强制在其他事务上。这意味着在该索引变得可用之前，其他查询中可能就会 报告该约束被违背，或者甚至在索引构建最终失败的情况中也是这样。还有，如果在 第二次扫描时发生失败，“无效的”索引也会继续强制它的唯一性约束。

表达式索引和部分索引的并发构建也被支持。在这些表达式计算过程中发生的 错误可能导致和上述唯一约束违背类似的行为。

常规索引构建允许在同一个表上同时构建其他常规索引，但是在一个表上同时 只能有一个并发索引构建发生。在两种情况下，在索引被构建时不允许表的模式修改。另一个不同是，一个常规CREATE INDEX 命令可以在一个事务块中执行，但是 CREATE INDEX CONCURRENTLY不行。

当前不支持在分区表上并发生成索引。 然而，可以在每个分区上单独的并发构建索引，然后最终以非并发的方式创建分区索引，以减少对分区表的写入被锁定的时间。 在这种情况下，生成分区索引仅是元数据操作。

注解

关于索引何时能被使用、何时不被使用以及什么情况下它们有用的信息请 见第 8 章 索引

当前，只有 B-树、GiST、GIN 和 BRIN 索引方法支持多列索引。默认最多可以 索引 32 个域（可以在构建 UXDB修改这种限制）。当前 只有 B-树支持唯一索引。

为索引的每一列可以指定一个操作符类。该操作符 类标识要被该索引用于该列的操作符。例如，一个四字节整数上的 B-树索引 会使用int4_ops类。这个操作符类包括了用于四字节整数 的比较函数。实际上，通常列数据类型的默认操作符类就足够了。对某些数据 类型指定操作符类的主要原因是，可能会有多于一种有意义的顺序。例如， 我们可能想用绝对值或者实数部分对复数类型排序。我们可以通过为该数据 类型定义两个操作符类来做到，并且在创建索引时选择其中合适的类。更多 有关操作符类的信息请见第 8.10 节 “操作符类和操作符族”。

当在一个分区表上调用CREATE INDEX时，默认的行为是递归到所有的分区上以确保它们都具有匹配的索引。每一个分区首先会被检查是否有一个等效的索引存在，如果有则该索引将被挂接

为被创建索引的一个分区索引，而被创建的索引将成为其父索引。如果不存在匹配的索引，则会创建一个新的索引并且自动进行挂接。如果命令中没有指定索引名称，每个分区中的新索引的名称将被自动决定。如果指定了ONLY选项，则不会进行递归，并且该索引会被标记为无效（一旦所有的分区都得到该索引，ALTER INDEX ... ATTACH PARTITION可以把该索引标记为有效）。不过，要注意不管是否指定这一选项，未来使用CREATE TABLE ... PARTITION OF创建的任何分区将自动有一个匹配的索引，不管有没有指定ONLY。

对于支持有序扫描的索引方法（当前只有 B-树），可以指定可选子句ASC、 DESC、NULLS FIRST以及NULLS LAST 来修改索引的排序顺序。由于一个有序索引能前向或者反向扫描，通常创建一个 单列DESC索引没什么用处 — 一个常规索引已经提供了排序 顺序。这些选项的价值是可以创建多列索引，让它的排序顺序匹配有混合排序要求 的查询，例如SELECT ... ORDER BY x ASC, y DESC。如果想要在依靠索引避免排序步骤的查询中支持 “空值排序低”这种行为，NULLS选项就能派上用场，默认 的行为是“空值排序高”。

对于大多数索引方法，索引的创建速度取决于 maintenance_work_mem的设置。较大的值将会减少 索引创建所需的时间，当然不要把它设置得超过实际可用的内存量（那会迫使 机器进行交换）。

UXDB可以在构建索引时利用多个CPU以更快地处理表行。这种特性被称为*并行索引构建*。对于支持并行构建索引的索引方法（当前只有B-树），maintenance_work_mem指定每次索引构建操作整体可用的最大内存量，而不管启动了多少工作者进程。一般来说，一个代价模型（如果有）自动判断应该请求多少工作者进程。

增加maintenance_work_mem可以让并行索引构建受益，而等效的串行索引构建将无法受益或者得到很小的益处。注意maintenance_work_mem可能会影响请求的工作者进程的数量，因为并行工作者必须在总的maintenance_work_mem预算中占有至少32MB的份额。还必须有32MB的份额留给领袖进程。增加max_parallel_workers_maintenance可以允许使用更多的工作者，这将降低索引创建所需的时间，只要索引构建不是I/O密集型的。当然，还需要有足够的CPU计算能力，否则工作者们会闲置。

通过ALTER TABLE(7)为parallel_workers设置一个值直接控制着CREATE INDEX会对表请求多少并行工作者进程。这会完全绕过代价模型，并且防止maintenance_work_mem对请求多少并行工作者产生影响。通过ALTER TABLE将parallel_workers设置为0将禁用所有情况下的并行索引构建。

> **提示**
>
> 在把parallel_workers用于调优一次索引构建之后，可能想要重置parallel_workers。这可以避免对查询计划的无意更改，因为parallel_workers影响所有的并行表扫描。

虽然带有CONCURRENTLY选项的CREATE INDEX支持并行构建并且没有特殊的限制，但只有第一次表扫描会实际以并行方式执行。

使用DROP INDEX(7)可以移除一个索引。

以前的UXDB发行也有一种 R-树 索引方法。这种方法已经被移除，因为它比起 GiST 方法来说没有什么明显的 优势。如果指定了USING rtree，CREATE INDEX 将会把它解释为USING gist，以便把旧的数据库转换成 GiST。

示例

在表films中的列title上创建一个 B-树索引：

CREATE UNIQUE INDEX title_idx ON films (title);

要在表films的列title上创建一个唯一的B-树索引并且包括列director和rating：

CREATE UNIQUE INDEX title_idx ON films (title) INCLUDE (director, rating);

在表达式lower(title)上创建一个索引来允许高效的大小写 无关搜索：

CREATE INDEX ON films ((lower(title)));

（在这个例子中我们选择省略索引名称，这样系统会选择一个名字， 通常是films_lower_idx）。

创建一个具有非默认排序规则的索引：

CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");

创建一个具有非默认空值排序顺序的索引：

CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);

创建一个具有非默认填充因子的索引：

CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);

创建一个禁用快速更新的GIN索引：

CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH (fastupdate = off);

在表films中的列code上创建一个而索引并且 把索引放在表空间indexspace中：

CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;

在一个点属性上创建一个 GiST 索引，这样我们可以在转换函数的结果 上有效地使用 box 操作符：

```
CREATE INDEX pointloc
   ON points USING gist (box(location,location));
SELECT * FROM points
   WHERE box(location,location) && '(0,0),(1,1)'::box;
```

创建一个表而不排斥对表的写操作：

CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table (quantity);

兼容性

CREATE INDEX是一种 UXDB的语言扩展。在 SQL 标准中 没有对于索引的规定。

另见

ALTER INDEX(7), DROP INDEX(7), REINDEX(7)

名称

CREATE LANGUAGE — 定义一种新的过程语言

大纲

```
CREATE [ OR REPLACE ] [ PROCEDURAL ] LANGUAGE name
CREATE [ OR REPLACE ] [ TRUSTED ] [ PROCEDURAL ] LANGUAGE name
    HANDLER call_handler [ INLINE inline_handler ] [ VALIDATOR valfunction ]
```

描述

CREATE LANGUAGE为一个 UXDB数据库注册一种新的 过程语言。接着，可以用这种新语言定义函数和存储过程。

> **注意**
>
> 大多数过程语言已经被做成了“扩展”，并且应该用CREATE EXTENSION(7)而不是**CREATE LANGUAGE**来安装。**CREATE LANGUAGE**的直接使用现在应该被限制在扩展安装脚本中。如果在数据库中有一种“裸”语言（可能是一次升级的结果），可以用CREATE EXTENSION *langname* FROMunpackaged把它转换成一个扩展。

CREATE LANGUAGE实际上把该语言名称与 负责执行用该语言编写的函数的处理器函数关联在一起。

有两种形式的**CREATE LANGUAGE**命令。在 第一种形式中，用户只提供想要的语言的名称。 UXDB服务器会查询ux_pltemplate系统目录来决定正确的参数。在第二种形式中，用户 要提供语言参数和语言名称。第二种形式可以被用来创建一种没有定义在 ux_pltemplate中的语言，但是这种方法被认为即将 废弃。

当服务器在ux_pltemplate目录中为给定的语言名称 找到一个项时，即使命令中已经包括了语言参数，它也将使用目录中的 数据。这种行为简化了旧转储文件的载入，旧转储文件很可能包含过时的 信息。

通常，用户必须拥有 UXDB超级用户特权来注册 一种新的语言。不过，如果该语言被列举在ux_pltemplate目录中并且被标记为允许 由数据库拥有者创建（*tmpldbacreate*为真），则数据 库的拥有者可以把新语言注册在数据库中。默认是可信的语言能够由数据 库拥有者创建，但是超级用户可以通过修改 ux_pltemplate的内容来调整这种行为。 语言的创建者会成为它的拥有者，并且以后可以删除它、对它重命名或者 把它赋予给一个新的拥有者。

CREATE OR REPLACE LANGUAGE将创建 或者替换一种现有的定义。如果该语言已经存在，其参数会被根据指定的 值或者来自ux_pltemplate的值更新。但 该语言的拥有关系和权限设置不会更改，并且任何已有的用该语言编写的 函数仍然被假定有效。除了创建一种语言的普通特权需求，用户还必须是 超级用户或者已有语言的拥有者。REPLACE情况主要被用来 确保该语言存在。如果该语言有一个 ux_pltemplate项，那么 REPLACE将不会实际更改现有定义的任何东西，除非从该语 言被创建以来ux_pltemplate已经被修改 过（很少见的情况）。

参数

TRUSTED

TRUSTED指定该语言不会授予用户不该具有的 数据访问。如果在注册语言时这个关键词被省略，只有具有 UXDB超级用户特权的用户才能 使用该语言创建新函数。

PROCEDURAL

这是一个噪声词。

name

新过程语言的名称。该名称必须在该数据库的语言中唯一。

为了向后兼容，名称可以用单引号围绕。

HANDLER *call_handler*

call_handler 是一个之前注册的函数的名称，它将被调用来执行该过程语言的函数。 一种过程语言的调用处理器必须以一种编译型语言（如 C）编写并且 具有版本 1 的调用约定，它必须在 UXDB内注册为一个没有 参数并且返回language_handler类型的函数。language_handler是一种占位符类型， 它被用来 标识该函数为一个调用处理器。

INLINE *inline_handler*

inline_handler 是一个之前注册的函数的名称，它将被调用来执行一个该语言的匿名代码块（ DO(7)命令）。如果没有指定 *inline_handler*函数，则 该语言不支持匿名代码块。该处理器函数必须接受一个internal 类型的参数，该参数将是DO命令的内部表示，而且它通常 将返回void。该处理器的返回值会被忽略。

VALIDATOR *valfunction*

valfunction is the 是一个之前注册的函数的名称，当一个该语言的新函数被创建时会调用该函数来 验证新函数。如果没有指定验证器函数，那么一个新函数被创建时不会被检查。 验证器函数必须接受一个oid类型的参数，它将是要被创建的 函数的 OID，而且它通常将返回void。

一个验证器函数通常会检查函数体中的语法正确性，但是它也能查看函数的其他 属性，例如该语言能否处理特定的参数类型。为了发出一个错误，验证器函数应该 使用ereport()函数。验证器函数的返回值会被忽略。

如果指定的语言名称在ux_pltemplate中有一项，服务器会忽略 TRUSTED选项和支持函数的名称。

注解

使用DROP LANGUAGE(7)删除过程语言。

系统目录ux_language记录着有关当前已安装的语言的信息。 还有，uxsql命令\dL列出已安装的语言。

要以一种过程语言创建函数，用户必须具有对于该语言的 USAGE特权。默认情况下，对于可信语言， USAGE被授予给PUBLIC（即所有人）。 如果需要可以将它收回。

过程语言对于单个数据库来说是本地的。但是，一种语言可以被安装在 template1数据库中，这会导致它在所有后续创建的 数据库中自动变得可用。

如果对语言在服务器的ux_pltemplate中没有一项， 调用处理器函数、内联处理器函数（如果有）以及验证器函数（如果有） 必须已经存在。但是当有一个那样的项时，这些函数不必已经存在。如果 它们在数据库中不存在，将自动定义它们（如果安装中实现该语言的共享 库不可用可能会导致**CREATE LANGUAGE**失败）。

示例

创建任何标准过程语言的最好的方式是：

```
CREATE LANGUAGE plperl;
```

对于ux_pltemplate目录不知道的一种语言，需要这样的命令序列：

```
CREATE FUNCTION plsample_call_handler() RETURNS language_handler
    AS '$libdir/plsample'
    LANGUAGE C;
CREATE LANGUAGE plsample
    HANDLER plsample_call_handler;
```

兼容性

CREATE LANGUAGE是一种 UXDB扩展。

另见

ALTER LANGUAGE(7), CREATE FUNCTION(7), DROP LANGUAGE(7), GRANT(7), REVOKE(7)

名称

CREATE MATERIALIZED VIEW — 定义一个新的物化视图

大纲

```
CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name
  [ (column_name [, ...] ) ]
  [ USING method ]
  [ WITH ( storage_parameter [= value] [, ... ] ) ]
  [ TABLESPACE tablespace_name ]
  AS query
  [ WITH [ NO ] DATA ]
```

描述

CREATE MATERIALIZED VIEW定义一个查询的物化视图。 在该命令被发出时，查询会被执行并且被用来填充该视图（除非使用了 **WITH NO DATA**），并且后来可能会用 **REFRESH MATERIALIZED VIEW**进行刷新。

CREATE MATERIALIZED VIEW类似于 **CREATE TABLE AS**，不过它还会记住被用来初始化该视图的查询， 这样它可以在后来被命令刷新。一个物化视图有很多和表相同的属性，但是不支持 临时物化视图。

参数

IF NOT EXISTS

如果已经存在一个同名的物化视图时不要抛出错误。这种情况下会发出一个 提示。注意这不保证现有的物化视图与即将创建的物化视图相似。

table_name

要创建的物化视图的名称（可以被模式限定）。

column_name

新物化视图中的一个列名。如果没有提供列名，会从查询的输出列名来得到。

USING *method*

此可选子句指定用于存储新具体化视图内容的表访问方法；该方法需要是TABLE类型的访问方法。 如果未指定此选项，则为新具体化视图选择默认表访问方法。

WITH (*storage_parameter* [= *value*] [, ...])

这个子句为新的物化视图指定可选的存储参数，详见 “存储参数”一节。所有CREATE TABLE支持的参数CREATE MATERIALIZED VIEW也支持。 详见CREATE TABLE(7)。

TABLESPACE *tablespace_name*

*tablespace_name*是 要把新物化视图创建在其中的表空间的名称。如果没有指定， 将查阅default_tablespace。

query

一个SELECT(7)、TABLE 或者VALUES(7)命令。这个查询将在一个安全受限的操作中运行。 特别地，对本身会创建临时表的函数的调用将会失败。

WITH [NO] DATA

这个子句指定物化视图是否在创建时被填充。如果不是，该物化视图将被标记为 不可扫描并且在**REFRESH MATERIALIZED VIEW**被使用前不能被查询。

兼容性

CREATE MATERIALIZED VIEW是一种 UXDB扩展。

另见

ALTER MATERIALIZED VIEW(7), CREATE TABLE AS(7), CREATE VIEW(7), DROP MATERIALIZED VIEW(7), REFRESH MATERIALIZED VIEW(7)

名称

CREATE OPERATOR — 定义一个新的操作符

大纲

```
CREATE OPERATOR name (
    {FUNCTION|PROCEDURE} = function_name
    [, LEFTARG = left_type ] [, RIGHTARG = right_type ]
    [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
    [, RESTRICT = res_proc ] [, JOIN = join_proc ]
    [, HASHES ] [, MERGES ]
)
```

描述

CREATE OPERATOR定义一个新的操作符 *name*。定义操作符 的用户会成为该操作符的拥有者。如果给出一个模式名，该操作符将被创建 在指定的模式中。否则它会被创建在当前模式中。

操作符名称是最多NAMEDATALEN-1（默认为 63） 个字符的序列，这些字符可以是：

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

对名称的选择有一些限制：

- -- and /*不能出现在 操作符名称中，因为它们会被当做一段注释的开始。
- 多字符操作符名称不能以+或者- 结束，除非该名称也包含至少一个下列字符：

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例如，@-是一个被允许的操作符名称，而 *-不是。这种限制允许 UXDB解析 SQL 兼容的命令而无需记号之间的空格。
- 将=>用作一个操作符名称已经不被推荐。在未来的发行中 可能会被禁用。

在输入时!=会被映射为<>， 因此这两个名字总是等效的。

必须至少定义LEFTARG和RIGHTARG中的一个。 对于二元操作符，两者都必须被定义。对于右一元操作符，只应该定义 LEFTARG，而对于左一元操作符只应该定义 RIGHTARG。

*function_name*函数 必须在之前已经用**CREATE FUNCTION**定义好， 并且必须被定义为接受正确数量的指定类型的参数。

在CREATE OPERATOR的语法中，关键词FUNCTION和PROCEDURE是等效的，但不管哪种情况下被引用的函数都必须是一个函数而不是过程。这里对关键词PROCEDURE的是用是有历史原因的，现在已经被废弃。

其他子句指定可选的操作符优化子句。

要创建一个操作符，必须具有参数类型和返回类型上的USAGE 特权，以及底层函数上的EXECUTE特权。如果指定了一个 交换子或者求反器操作符，必须拥有这些操作符。

参数

name

要定义的操作符的名称。允许使用的字符请见上文。名称可以被模式　限定，例如CREATE OPERATOR myschema.+ (...)。如果　没有被模式限定，该操作符将被创建在当前模式中。如果两个同一模式　中的操作符在不同的数据类型上操作，它们可以具有相同的名称。这　被称为重载。

function_name

用来实现这个操作符的函数。

left_type

这个操作符的左操作数（如果有）的数据类型。忽略这个选项　　可以表示一个左一元操作符。

right_type

这个操作符的右操作数（如果有）的数据类型。忽略这个选项　　可以表示一个右一元操作符。

com_op

这个操作符的交换子。

neg_op

这个操作符的求反器。

res_proc

用于这个操作符的限制选择度估计函数。

join_proc

用于这个操作符的连接选择度估算函数。

HASHES

表示这个操作符可以支持哈希连接。

MERGES

表示这个操作符可以支持归并连接。

要在*com_op* 或者其他可选参数中给出一个模式限定的操作符名称，　请使用OPERATOR()语法，例如：

COMMUTATOR = OPERATOR(myschema.===) ,

注解

无法在**CREATE OPERATOR**中指定一个操作符的 词法优先级，因为解析器的优先级行为是硬写在代码中的。详见 第　1.1.6　节 “操作符优先级”

废弃的选项SORT1、SORT2、 LTCMP以及GTCMP以前被用来指定与支持 归并连接的操作符相关的排序操作符的名称。现在不再需要它们了，因为 相关操作符的信息可以在 B-树的操作符族中找到。如果给出了这些选项 之一，它会被忽略（除非是为了隐式设置MERGES为真）。

使用DROP OPERATOR(7)从数据库中删除用户定义的操作符。 使用ALTER OPERATOR(7)修改数据库中的操作符。

示例

下面的命令为数据类型box定义了一种新的操作符--面积相等：

```
CREATE OPERATOR === (
   LEFTARG = box,
   RIGHTARG = box,
   FUNCTION = area_equal_function,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restriction_function,
   JOIN = area_join_function,
   HASHES, MERGES
);
```

兼容性

CREATE OPERATOR是一种 UXDB扩展。在 SQL 标准中没有用户定义操作符的规定。

另见

ALTER OPERATOR(7), CREATE OPERATOR CLASS(7), DROP OPERATOR(7)

名称

CREATE OPERATOR CLASS — 定义一个新的操作符类

大纲

```
CREATE OPERATOR CLASS name [ DEFAULT ] FOR TYPE data_type
  USING index_method [ FAMILY family_name ] AS
  {  OPERATOR strategy_number operator_name [ ( op_type, op_type ) ] [ FOR SEARCH | FOR
 ORDER BY sort_family_name ]
   | FUNCTION support_number [ ( op_type [ , op_type ] ) ] function_name ( argument_type [, ...] )
   | STORAGE storage_type
  } [, ... ]
```

描述

CREATE OPERATOR CLASS创建新的操作符类。 一个操作符类定义一种特殊的数据类型如何被用于一个索引。操作符类指定为 该数据类型和索引方法扮演特殊角色或者“策略”的操作符。操作符 类还指定当该操作符类被选择用于一个索引列时，索引方法要使用的支持函数。 操作符类所使用的所有操作符和函数必须在操作符类被创建之前被定义好。

如果给出了一个模式名称，那么该操作符类会被创建在指定模式中。否则，它 会被创建在当前模式中。同一模式中的两个操作符类只有在被用于不同的索引 方法时才可以具有相同的名称。

定义操作符类的用户将成为其拥有者。当前，创建用户必须是超级用户（做出 这种限制是因为错误的操作符类定义会让服务器混淆甚至崩溃）。

CREATE OPERATOR CLASS当前不会检查操作符 类定义是否包括该索引方法所要求的所有操作符和函数，也不会检查这些操作符 和函数是否构成一个一致的集合。定义一个合法的操作符类是用户的责任。

相关的操作符类可以被组成操作符族。要把一个新的操作符类 加入到一个现有的族中，可以在CREATE OPERATOR CLASS中指定FAMILY选项。如果没有这个选项， 新的类会被放到一个同名的族中（如果族不存在会创建之）。

参数

name

要创建的操作符类的名称。该名称可以被模式限定。

DEFAULT

如果存在，该操作符类将成为其数据类型的默认操作符类。对一种 特定的数据类型和索引方法至多有一个默认操作符类。

data_type

这个操作符类所用于的列数据类型。

index_method

这个操作符类所用于的索引方法的名称。

family_name

要把这个操作符类加入其中的已有操作符族的名称。如果没有指定， 将使用一个同名操作符族（如果还不存在则创建之）。

strategy_number

用于一个与该操作符类相关联的操作符的索引方法策略号。

operator_name

一个与该操作符类相关联的操作符的名称（可以被模式限定）。

op_type

在一个OPERATOR子句中，这表示该操作符的操作数数据 类型，或者用NONE来表示一个左一元或者右一元操作符。 在操作数数据类型与该操作符的数据类型相同的一般情况下，操作数的 数据类型可以被省略。

在一个FUNCTION子句中，这表示该函数要支持的操作数 数据类型，如果它与该函数的输入数据类型（对于 B-树比较函数和哈希 函数）或者操作符类的数据类型（对于 B-树排序支持函数和所有GiST、 SP-GiST、GIN 和 BRIN 操作符类中的函数）不同。这些默认值是正确的，并且 *op_type*因此不必 在FUNCTION子句中被指定，对于 B-树排序支持函数的情 况来说，这表示跨数据类型比较。

sort_family_name

一个现有btree操作符族的名称（可以是模式限定的）， 它描述与一种排序操作符相关联的排序顺序。

如果FOR SEARCH和FOR ORDER BY都没有被 指定，那么FOR SEARCH是默认值。

support_number

用于一个与该操作符类相关联的函数的索引方法支持函数编号。

function_name

一个用于该操作符类的索引方法支持函数的函数名称（可以是 模式限定的）。

argument_type

该函数的参数数据类型。

storage_type

实际存储在索引中的数据类型。通常这和列数据类型相同，但是有些 索引方法（当前有GiST、GIN 和 BRIN）允许它们不同。 除非索引方法允许使用不同的类型，STORAGE子句必须 被省略。 如果*data_type*列被指定为anyarray， 那么*storage_type*可以被声明为anyelement以指示索引条目是属于为每个特定索引创建的实际数组类型的元素类型的成员。

OPERATOR、FUNCTION和STORAGE 子句可以以任何顺序出现。

注解

因为索引机制在使用函数之前不检查它们的权限，将一个函数或者操作符包括在 一个操作符类中相当于在其上授予公共执行权限。这对操作符类中很有用的函数 来说通常不成问题。

操作符不应该用 SQL 函数定义。SQL 函数很有可能会被内联到调用查询中，这 会妨碍优化器识别该查询匹配一个索引。

OPERATOR子句可以包括一个RECHECK选项。现在已经不再支持，因为一个索引操作符是否为“有损的”现在是在运行时实时决定的。这允许在一个操作符可能是或者可能不是有损的情况下有效地处理。

示例

下面的例子为数据类型_int4（int4数组） 定义了一个 GiST 索引操作符。

```
CREATE OPERATOR CLASS gist__int_ops
  DEFAULT FOR TYPE _int4 USING gist AS
    OPERATOR    3    &&,
    OPERATOR    6    = (anyarray, anyarray),
    OPERATOR    7    @>,
    OPERATOR    8    <@,
    OPERATOR    20   @@ (_int4, query_int),
    FUNCTION    1    g_int_consistent (internal, _int4, smallint, oid, internal),
    FUNCTION    2    g_int_union (internal, internal),
    FUNCTION    3    g_int_compress (internal),
    FUNCTION    4    g_int_decompress (internal),
    FUNCTION    5    g_int_penalty (internal, internal, internal),
    FUNCTION    6    g_int_picksplit (internal, internal),
    FUNCTION    7    g_int_same (_int4, _int4, internal);
```

兼容性

CREATE OPERATOR CLASS是一种 UXDB扩展。在 SQL 标准中没有 **CREATE OPERATOR CLASS**语句。

另见

ALTER OPERATOR CLASS(7), DROP OPERATOR CLASS(7), CREATE OPERATOR FAMILY(7), ALTER OPERATOR FAMILY(7)

名称

CREATE OPERATOR FAMILY — 定义一个新的操作符族

大纲

CREATE OPERATOR FAMILY *name* USING *index_method*

描述

CREATE OPERATOR FAMILY创建一个新的 操作符族。一个操作符族定义一个相关操作符类组成的集合，并且可能还 包含一些额外的、与这些操作符类兼容但对于任何个体索引的功能不是至关重要的操作符和支持函数（对索引至关重要的操作符和函数应该被分组在 相关的操作符类中，而不是“松散地”在操作符中。通常，单一 数据类型操作符被限制在操作符类中，而跨数据类型操作符可以松散地存在 于一个包含用于两种数据类型的操作符类的操作符族中）。

新的操作符族初始时为空。应该通过发出后续的 **CREATE OPERATOR CLASS**命令来增加包含 在其中的操作符类，还可以用可选的 **ALTER OPERATOR FAMILY**命令来增加 “松散的”操作符和它们对应的支持函数。

如果给出一个模式名称，该操作符族会被创建在指定的模式中。否则， 它会被创建在当前模式中。只有当同一个模式中的两个操作符族是用于 不同的索引方法时，它们才能拥有相同的名字。

定义一个操作符族的用户将成为它的拥有者。当前，创建用户必须是超级用户（ 做出这样的限制是因为错误的操作符族会让服务器混淆甚至崩溃）。

参数

name

要创建的操作符族的名称。该名称可以被模式限定。

index_method

这个操作符族要用于的索引方法的名称。

兼容性

CREATE OPERATOR FAMILY是一种 UXDB扩展。在 SQL 标准中没有 **CREATE OPERATOR FAMILY**语句。

另见

ALTER OPERATOR FAMILY(7), DROP OPERATOR FAMILY(7), CREATE OPERATOR CLASS(7), ALTER OPERATOR CLASS(7), DROP OPERATOR CLASS(7)

名称

CREATE POLICY — 为一个表定义一条新的行级安全性策略

大纲

```
CREATE POLICY name ON table_name
  [ AS { PERMISSIVE | RESTRICTIVE } ]
  [ FOR { ALL | SELECT | INSERT | UPDATE | DELETE } ]
  [ TO { role_name | PUBLIC | CURRENT_USER | SESSION_USER } [, ...] ]
  [ USING ( using_expression ) ]
  [ WITH CHECK ( check_expression ) ]
```

描述

CREATE POLICY为一个表定义一条行级 安全性策略。注意为了应用已被创建的策略，在表上必须启用行级安全 性（使用ALTER TABLE ... ENABLE ROW LEVEL SECURITY）。

一条策略授予权限以选择、插入、更新或者删除匹配相关策略表达式的行。 现有的表行会按照USING中指定的表达式进行检查， 而将要通过INSERT或UPDATE创建 的新行会按照WITH CHECK中指定的表达式进行检查。 当USING表达式对于一个给定行返回真时，该行对用户 可见，而返回假或空时该行不可见。当一个WITH CHECK 表达式对一行返回真时，该行会被插入或更新，而如果返回假或空时会发生 一个错误。

对于INSERT和 UPDATE语句，在BEFORE 触发器被引发后并且在任何数据修改真正发生之前，WITH CHECK表达式会被强制。因此， 一个BEFORE ROW触发器可以修改要被插入的数据，从而影响安全性策略检查的结果。WITH CHECK表达式 在任何其他约束之前被强制。

策略名称是针对每个表的。因此，一个策略名称可以被用于很多个不同的表 并且对于不同的表呈现适合于该表的定义。

策略可以被应用于特定的命令或者特定的角色。除非特别指定，新创建的策略 的默认行为是适用于所有命令和角色。多个策略可以应用于单个命令，更多细节请见下文。表 5 “按命令类型应用的策略”总结了不同类型的策略如何应用于特定的命令。

对同时具有USING和WITH CHECK 表达式（ALL和UPDATE）的策略， 如果没有定义WITH CHECK表达式，那么 USING表达式将被用于决定哪些行可见（普通 USING情况）以及允许哪些新行被增加（ WITH CHECK情况）。

如果为一个表启用了行级安全性但是没有适用的策略存在，将假定为一种 “默认否定”策略，这样任何行都不可见也不可更新。

参数

name

要创建的策略的名称。这必须和该表上已有的任何其他策略名称相区分。

table_name

该策略适用的表的名称（可以被模式限定）。

PERMISSIVE

指定策略被创建为宽容性策略。适用于一个给定查询的所有宽容性策略将被使用布尔“OR”操作符组合在一起。通过创建宽容性策略，管理员可以在能被访问的记录集合中进行增加。策略默认是宽容性的。

RESTRICTIVE

指定策略被创建为限制性策略。适用于一个给定查询的所有限制性策略将被使用布尔“AND”操作符组合在一起。通过创建限制性策略，管理员可以减少能被访问的记录集合，因为每一条记录都必须通过所有的限制性策略。

注意在限制性策略真正能发挥作用减少访问之前，需要至少一条宽容性策略来授予对记录的访问。如果只有限制性策略存在，则没有记录能被访问。当宽容性和限制性策略混合存在时，只有当一个记录能通过至少一条宽容性策略以及所有的限制性策略时，该记录才是可访问的。

command

该策略适用的命令。合法的选项是 ALL、SELECT、 INSERT、UPDATE 以及DELETE。 ALL为默认。有关这些策略如何被应用的 细节见下文。

role_name

该策略适用的角色。默认是PUBLIC，它将把策略应用 到所有的角色。

using_expression

任意的SQL条件表达式（返回 boolean）。该条件表达式不能包含任何聚集或者窗口 函数。如果行级安全性被启用，这个表达式将被增加到引用该表的查询。 让这个表达式返回真的行将可见。让这个表达式返回假或者空的任何行 将对用户不可见（在SELECT中）并且将对修改不可用（ 在UPDATE或DELETE中）。这类行 会被悄悄地禁止而不会报告错误。

check_expression

任意的SQL条件表达式（返回 boolean）。该条件表达式不能包含任何聚集或者窗口 函数。如果行级安全性被启用，这个表达式将被用在该表上的 INSERT以及 UPDATE查询中。只有让该表达式计算为真 的行才被允许。如果任何被插入的记录或者跟新后的记录导致该表达式计 算为假或者空，则会抛出一个错误。注意 *check_expression* 是根据行的新内容而不是原始内容计算的。

针对每种命令的策略

ALL

为一条策略使用ALL表示它将适用于所有命令， 不管命令的类型如何。如果存在一条ALL策略 以及更多特定的策略，则ALL策略和那些策略 会被应用。此外， ALL策略将同时适用于一个查询的选择端和修 改端，如果只定义了一个USING表达式则将 该USING表达式用于两种情况。

例如，如果发出一个UPDATE，那么 ALL策略将同时影响UPDATE 能更新哪些行（应用USING表达式）以及更新后 的行是否被允许加入到表中（如果定义了WITH CHECK 表达式，则应用之；否则使用USING表达式）。 如果一条INSERT 或者UPDATE命令尝试增加行到表中， 但行没有通过ALL策略的 WITH CHECK表达式，则整个语句将会中断。

SELECT

对一条策略使用SELECT表示它将适用于 SELECT查询，并且无论何时都要求该约束所在的关系上 的SELECT权限。其结果是在一次 SELECT查询期间，只有该关系中那些通过了SELECT策略的记录才将被返回，并且查询要求 SELECT权限，例如 UPDATE也将只能看到那些 SELECT策略允许的行。一条 SELECT策略不能具有WITH CHECK表达式，因为它只适用于正在从关系中检索记录的情况。

INSERT

为一条策略使用INSERT表示它适用于 INSERT命令。没有通过这种策略的正在被插入的行会导致策略违背错误，并且整个INSERT命令将会中止。 一条INSERT策略不能具有USING表达式，因为它只适用于正在向关系增加记录的情况。

注意在带有ON CONFLICT DO UPDATE的INSERT中，只有对通过 INSERT路径追加到关系的行才会检查 INSERT策略的WITH CHECK 表达式。

UPDATE

对策略使用UPDATE 意味着它将应用于UPDATE、SELECT FOR UPDATE和SELECT FOR SHARE 命令，还有INSERT 命令的辅助性的ON CONFLICT DO UPDATE 子句。由于UPDATE 需要提取现有的记录并且用新修改的记录代替，故UPDATE 策略接受USING 表达式和WITH CHECK 表达式。USING 表达式决定UPDATE 命令将能看到哪些要对其操作的记录，而WITH CHECK 表达式定义哪些被修改的行允许存回到关系中。

任何更新后的值无法通过WITH CHECK表达式的行 将会导致错误，并且整个命令将被中止。如果只指定了一个 USING子句，那么该子句将被用于 USING和WITH CHECK两种情况。

典型地，UPDATE命令也需要从待更新关系中的列读数据（例如在WHERE子句、RETURNING子句或在SET子句右侧的表达式中）。这种情况下，正被更新的关系上也需要SELECT权限，并且除了UPDATE策略外，也要应用适当的SELECT或者ALL策略。这样，除由UPDATE或ALL策略授权更新行之外，通过SELECT或ALL策略用也必须能访问正被更新的行。

当INSERT命令附加了ON CONFLICT DO UPDATE子句时，如果采用UPDATE路径，先以任何UPDATE策略的USING表达式检查待更新的行，然后以WITH CHECK表达式检查新修改的行。但要注意的是，不同于单独的UPDATE命令，如果现有的行不能通过USING表达式检查，则抛出错误（UPDATE路径永不会静默地避免）。

DELETE

为一条策略使用DELETE表示它适用于 DELETE命令。只有通过这条策略的行才将能被DELETE命令所看到。如果有的行不能通过该 DELETE策略的USING表达式，则 它们可以通过SELECT看到但不能被删除。

大多数情况下，DELETE命令也需要从其所删除的关系中的列读取数据（例如在WHERE子句或RETURNING子句中）。这种情况下，在该关系上也需要SELECT权限，并且除了DELETE策略，也要应用适当的SELECT或ALL策略。这样，除由DELETE或ALL策略授权删除行之外，通过SELECT或ALL策略，用户也必须能访问正被删除的行。

DELETE策略不能具有WITH CHECK表达式，因为它只适用于正在从关系中删除记录的情况， 所以没有新行需要检查。

表 5. 按命令类型应用的策略

命令	**SELECT/ALL**策略	**INSERT/ALL**策略	**UPDATE/ALL**策略		**DELETE/ALL**策略
	USING表达式	**WITH CHECK**表达式	**USING**表达式	**WITH CHECK**表达式	**USING**表达式
SELECT	现有行	—	—	—	—
SELECT FOR UPDATE/SHARE	现有行	—	现有行	—	—
INSERT	—	新行	—	—	—
INSERT ... RETURNING	新行[a]	新行	—	—	—
UPDATE	现有 & 新行 [a]	—	现有行	新行	—
DELETE	现有行[a]	—	—	—	现有行
ON CONFLICT DO UPDATE	现有 & 新行	—	现有行	新行	—

[a] 对于现有行或新行，如果需要读访问的话（例如涉及到关系内列的WHERE或RETURNING子句）。

多重策略的应用

当多种不同命令类型的策略应用于相同命令（例如SELECT和UPDATE策略应用于UPDATE命令）时，用户就必须同时具有这两种类型的权限（例如从关系中选取行和更新的权限）。这样一种策略类型的表达式就与另一种策略类型的表达式通过使用AND操作符组合在一起。

当相同命令类型的多种策略应用于同一命令时，则必须至少有一个PERMISSIVE策略授权对该关系的访问，所有的RESTRICTIVE策略必须通过。这样，所有的PERMISSIVE策略表达式都用OR来组合，所有的RESTRICTIVE策略表达式都用AND来组合，而结果用AND来组合。如果没有PERMISSIVE策略，则拒绝访问。

要注意的是，出于组合多种策略的目的，将ALL策略视为与所应用的任何其他类型的策略具有相同的类型。

例如，在UPDATE命令中，SELECT和UPDATE两种权限都需要，如果每种类型都有多个适用的策略，则将之以下面的方式组合：

expression from RESTRICTIVE SELECT/ALL policy 1
AND
expression from RESTRICTIVE SELECT/ALL policy 2
AND
...
AND
(
 expression from PERMISSIVE SELECT/ALL policy 1
 OR
 expression from PERMISSIVE SELECT/ALL policy 2
 OR
 ...
)
AND

expression from RESTRICTIVE UPDATE/ALL policy 1
AND
expression from RESTRICTIVE UPDATE/ALL policy 2
AND
...
AND
(
 expression from PERMISSIVE UPDATE/ALL policy 1
 OR
 expression from PERMISSIVE UPDATE/ALL policy 2
 OR
 ...
)

注解

要为一个表创建或者修改策略，必须是该表的拥有者。

虽然策略将被应用于针对数据库中表的显式查询上，但当系统正在执行　内部引用完整性检查或者验证约束时不会应用它们。这意味着有间接的　方法来决定一个给定的值是否存在。一个例子是向一个作为主键或者拥　有唯一约束的列中尝试插入重复值。如果插入失败则用户可以推导出该 值已经存在（这个例子假设用户被策略允许插入他们看不到的记录）。 另一个例子是一个用户被允许向一个引用了其他表的表中插入，然而另　一个表是隐藏表。通过用户向引用表中插入值可以判断存在性， 成功表示该值存在于被引用表中。为了解决这些问题，应该仔细地制作 策略以完全阻止用户插入、删除或者更新那些可能指示他们不能看到的　值的记录，或者使用生成的值（例如代理键）来代替具有外部含义的键。

通常，系统将在应用用户查询中出现的条件之前先强制由安全性策略施　加的过滤条件，这是为了防止无意中把受保护的数据暴露给可能不可信　的用户定义函数。不过，被系统（或者系统管理员）标记为 LEAKPROOF的函数和操作符可以在策略表达式之前 被计算，因为它们已经被假定为可信。

因为策略表达式会被直接加到用户查询上，它们将使用运行整个查询的用户的　　权限运行。因此，使用一条给定策略的用户必须能够访问表达式中引用的任何　表或函数，否则在尝试查询启用了行级安全性的表时，他们将简单地收到一条　没有权限的错误。不过，这不会改变视图的工作方式。就普通查询和视图来说，　权限检查和视图所引用的表的策略将使用视图拥有者的权限以及任何适用于视 图拥有者的策略。

在第　2.8　节　“行安全性策略”中可以找到额外的讨论和实际的例子。

兼容性

CREATE POLICY是一种UXDB扩展。

另见

ALTER POLICY(7), DROP POLICY(7), ALTER TABLE(7)

名称

CREATE PROCEDURE — 定义一个新的过程

大纲

```
CREATE [ OR REPLACE ] PROCEDURE
   name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  { LANGUAGE lang_name
   | TRANSFORM { FOR TYPE type_name } [, ... ]
   | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
   | SET configuration_parameter { TO value | = value | FROM CURRENT }
   | AS 'definition'
   | AS 'obj_file', 'link_symbol'
  } ...
```

简介

CREATE PROCEDURE定义一个新的过程。CREATE OR REPLACE PROCEDURE将会创建一个新过程或者替换一个已有的定义。为了能够定义过程，用户必须具有所使用的语言上的USAGE特权。

如果这个命令中包括了一个方案名称，则该过程将被创建在该方案中。否则过程将被创建在当前的方案中。新过程的名称不能匹配同一方案中具有相同输入参数类型的任何现有过程或函数。不过，具有不同参数类型的过程和函数可以共享同一个名称（这被称为重载）。

要替换一个已有过程的当前定义，请使用CREATE OR REPLACE PROCEDURE。不能用这种方式更改过程的名称或者参数类型（如果尝试这样做，实际上会创建一个新的、不同的过程）。

当CREATE OR REPLACE PROCEDURE被用来替换一个现有的过程时，该过程的拥有关系和权限保持不变。所有其他的过程属性会被赋予这个命令中指定的或者暗示的值。必须拥有（包括成为拥有角色的成员）该过程才能替换它。

创建过程的用户将成为该过程的拥有者。

为了能够创建一个过程，用户必须具有参数类型上的USAGE特权。

Parameters

name

要创建的过程的名称（可以是被方案限定的）。

argmode

参数的模式可以是：IN、INOUT或者VARIADIC。如果省略，则默认为IN（当前对过程不支持OUT参数，可使用INOUT）。

argname

参数的名称。

argtype

过程的参数（如果有）的数据类型（可以是被方案限定的）。参数类型可以是基础类型、组合类型或者域类型，或者可以引用一个表列的类型。

根据具体的实现语言，还可能可以指定“伪类型”，例如cstring。伪类型表示实际的参数类型没有完全确定，或者是位于普通SQL数据类型的集合之外。

写上*table_name.column_name*%TYPE可以引用某个列的类型。使用这种特性有时可以让过程不受表定义改变的影响。

default_expr

没有指定参数时要被用作默认值的表达式。这个表达式必须符合该参数的参数类型。跟在有默认值的参数后面的输入参数也都必须有默认值。

lang_name

用于实现该过程的语言名称。它可以是sql、c、internal或者一种用户定义的过程语言的名称，例如pluxsql。将名称包裹在单引号内的方式已经被废弃，并且要求大小写匹配。

TRANSFORM { FOR TYPE *type_name* } [, ...] }

列出对过程的调用应该应用哪些Transform。Transform负责在SQL类型和语言相关的数据类型之间进行转换，请参考CREATE TRANSFORM(7)。过程语言实现通常采用硬编码的方式保存内建类型的知识，因此它们无需在这里列出。但如果一种过程语言实现不知道如何处理一种类型并且没有提供Transform，它将回退到默认的行为来转换数据类型，但是这依赖于其实现。

[EXTERNAL] SECURITY INVOKER
[EXTERNAL] SECURITY DEFINER

SECURITY INVOKER指示过程以调用它的用户的特权来执行。这是默认方式。SECURITY DEFINER指定过程以拥有它的用户的特权来执行。

为了符合SQL标注，允许使用EXTERNAL关键词，但它是可选的，因为和SQL中不同，这个特性适用于所有的过程而不仅仅是外部过程。

SECURITY DEFINER过程不能执行事务控制语句（例如COMMIT和ROLLBACK，具体取决于实现的语言）。

configuration_parameter
value

SET子句导致在进入该过程时指定的配置参数被设置为指定的值，并且在过程退出时恢复到之前的值。SET FROM CURRENT把CREATE PROCEDURE执行时该参数的当前值保存为在进入该过程时要应用的值。

如果对过程附加一个SET子句，那么在该过程中为同一个变量执行的SET LOCAL命令的效果就被限制于该过程：在过程退出时还是会恢复到该配置参数的以前的值。不过，一个普通的SET命令（没有LOCAL）会重载这个SET子句，很像它对一个之前的SET LOCAL命令所做的事情：这样一个命令的效果将持续到过程退出之后，除非当前事务被回滚。

如果对过程附加一个SET子句，则该过程不能执行事务控制语句（例如COMMIT和ROLLBACK，具体取决于实现的语言）。

有关允许的参数名和值的更多信息请参考SET(7)。

definition

一个定义该过程的字符串常量，其含义取决于语言。它可以是一个内部的过程名、一个对象文件的路径、一个SQL命令或者以一种过程语言编写的文本。

在编写过程的定义字符串时，使用美元引用（见第 1.1.2.4 节 “美元引用的字符串常量”）而不是普通的单引号语法常常会很有帮助。如果没有美元引用，过程定义中的任何单引号或者反斜线必须以双写的方式进行转义。

obj_file, *link_symbol*

当C语言源码中的过程名与SQL过程的名称不同时，这种形式的AS子句被用于动态可装载的C语言过程。字符串*obj_file*是包含已编译好的C过程的共享库文件名，并且被按照LOAD(7)命令的方式解析。字符串*link_symbol*是该过程的链接符号，也就是该过程在C语言源代码中的名称。如果链接符号被省略，则会被假定为与正在被定义的SQL过程的名称相同。

当重复的**CREATE PROCEDURE**调用引用同一个对象文件时，只会对每一个会话装载该文件一次。要卸载或者重新载入该文件（可能是在开发期间），应该开始一个新的会话。

注解

函数创建也适用于过程，更多细节请参考CREATE FUNCTION(7)。

使用CALL(7)来执行过程。

示例

```
CREATE PROCEDURE insert_data(a integer, b integer)
LANGUAGE SQL
AS $$
INSERT INTO tbl VALUES (a);
INSERT INTO tbl VALUES (b);
$$;

CALL insert_data(1, 2);
```

兼容性

SQL标准中定义有一个**CREATE PROCEDURE**命令。详情请见CREATE FUNCTION(7)。

另见

ALTER PROCEDURE(7), DROP PROCEDURE(7), CALL(7), CREATE FUNCTION(7)

名称

CREATE PUBLICATION — 定义一个新的发布

大纲

```
CREATE PUBLICATION name
  [ FOR TABLE [ ONLY ] table_name [ * ] [, ...]
    | FOR ALL TABLES ]
  [ WITH ( publication_parameter [= value] [, ... ] ) ]
```

描述

CREATE PUBLICATION向当前数据库添加一个新的发布。 发布的名称必须与当前数据库中任何现有发布的名称不同。

发布本质上是一组表，其数据更改旨在通过逻辑复制进行复制。

参数

name

新发布的名称。

FOR TABLE

指定要添加到发布的表的列表。如果在表名之前指定了ONLY， 那么只有该表被添加到发布中。如果没有指定ONLY， 则添加表及其所有后代表（如果有的话）。可选地， 可以在表名之后指定*以明确指示包含后代表。

只有持久基表才能成为出版物的一部分。临时表、非日志表、外表、物化视图、 常规视图和分区表不能成为发布的一部分。要复制分区表， 请将各个分区添加到发布中。

FOR ALL TABLES

将发布标记为复制数据库中所有表的更改，包括在将来创建的表。

WITH (*publication_parameter* [= *value*] [, ...])

该子句指定发布的可选参数。支持下列参数：

publish （string）

这个参数决定了哪些DML操作将由新的发布发布给订阅者。 该值是用逗号分隔的操作列表。允许的操作是insert， update，delete和truncate。 默认是发布所有的动作，所以这个选项的默认值是 'insert, update, delete, truncate'。

注意

如果既没有指定FOR TABLE，也没有指定FOR ALL TABLES， 那么这个发布就是以一组空表开始的。这在稍后添加表格的情况下是有用的。

创建发布不会开始复制。它只为未来的订阅者定义一个分组和过滤逻辑。

要创建一个发布，调用者必须拥有当前数据库的CREATE权限。 （当然，超级用户不需要这个检查。）

要将表添加到发布中，调用者必须拥有该表的所有权。FOR ALL TABLES 子句要求调用者是超级用户。

添加到发布UPDATE和/或DELETE 操作的发布的表必须已经定义了REPLICA IDENTITY。 否则将在这些表上禁止这些操作。

对于INSERT ... ON CONFLICT命令， 发布将公布从命令实际产生的操作。因此，根据结果，它可以作为 INSERT或UPDATE发布，也可以根本不发布。

COPY ... FROM命令是作为INSERT操作发布的。

不发布DDL操作。

示例

创建一个发布，发布两个表中所有更改布：

```
CREATE PUBLICATION mypublication FOR TABLE users, departments;
```

创建一个发布，发布所有表中的所有更改：

```
CREATE PUBLICATION alltables FOR ALL TABLES;
```

创建一个发布，只发布一个表中的INSERT操作：

```
CREATE PUBLICATION insert_only FOR TABLE mydata
  WITH (publish = 'insert');
```

兼容性

CREATE PUBLICATION是一个UXDB 扩展。

又见

ALTER PUBLICATION(7), DROP PUBLICATION(7)

名称

CREATE ROLE — 定义一个新的数据库角色

大纲

CREATE ROLE *name* [[WITH] *option* [...]]

where *option*可以是：

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT connlimit
  | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
  | VALID UNTIL 'timestamp'
  | IN ROLE role_name [, ...]
  | IN GROUP role_name [, ...]
  | ROLE role_name [, ...]
  | ADMIN role_name [, ...]
  | USER role_name [, ...]
  | SYSID uid
```

描述

CREATE ROLE向UXDB数据库集簇增加一个新的角色。一个角色是一个实体，它可以拥有数据库对象并且拥有数据库特权。根据一个角色如何被使用，它可以被考虑成一个“用户”、一个“组”或者两者。要使用这个命令，必须具有CREATEROLE特权或者成为一个数据库超级用户。

注意角色是定义在数据库集簇层面上的，并且因此在集簇中的所有数据库中都可用。

参数

name

新角色的名称。

SUPERUSER
NOSUPERUSER

这些子句决定新角色是否是一个“超级用户”，它可以越过数据库内的所有访问限制。超级用户状态很危险并且只应该在确实需要时才用。要创建一个新超级用户，必须自己是一个超级用户。如果没有指定，默认值是NOSUPERUSER。

CREATEDB
NOCREATEDB

这些子句定义一个角色创建数据库的能力。如果指定了CREATEDB，被定义的角色将被允许创建新的数据库。指定NOCREATEDB将否定一个角色创建数据库的能力。如果没有指定，默认值是NOCREATEDB。

CREATEROLE
NOCREATEROLE

这些子句决定一个角色是否被允许创建新的角色（也就是执行CREATE ROLE）。一个带有CREATEROLE特权的角色也能修改和删除其他角色。如果没有指定，默认值是NOCREATEROLE。

INHERIT
NOINHERIT

如果新的角色是其他角色的成员，这些子句决定新角色是否从那些角色中“继承”特权，把新角色作为成员的角色称为新角色的父角色。一个带有INHERIT属性的角色能够自动使用已经被授予给其直接或间接父角色的任何数据库特权。如果没有INHERIT，在另一个角色中的成员关系只会把SET ROLE的能力授予给那个其他角色，只有在这样做后那个其他角色的特权才可用。如果没有指定，默认值是INHERIT。

LOGIN
NOLOGIN

这些子句决定一个角色是否被允许登录，也就是在客户端连接期间该角色是否能被给定为初始会话认证名称。一个具有LOGIN属性的角色可以被考虑为一个用户。没有这个属性的角色对于管理数据库特权很有用，但是却不是用户这个词的通常意义。如果没有指定，默认值是NOLOGIN，不过当CREATE ROLE被通过CREATE USER(7)调用时默认值会是LOGIN。

REPLICATION
NOREPLICATION

这些子句决定一个角色是否为复制角色。角色必须具有这个属性（或者成为一个超级用户）才能以复制模式（物理复制或者逻辑复制）连接到服务器以及创建或者删除复制槽。一个具有REPLICATION属性的角色是一个具有非常高特权的角色，并且只应被用于确实需要复制的角色上。如果没有指定，默认值是NOREPLICATION。

BYPASSRLS
NOBYPASSRLS

这些子句决定是否一个角色可以绕过每一条行级安全性（RLS）策略。默认是NOBYPASSRLS。注意 ux_dump 将默认 把row_security设置为OFF， 以确保一个表的所有内容被转储出来。如果运行 ux_dump 的用户不 具有适当的权限，将会返回一个错误。超级用户和被转储表的拥有者 总是可以绕过 RLS。

CONNECTION LIMIT *connlimit*

如果角色能登录，这指定该角色能建立多少并发连接。-1（默认值）表示无限制。注意这个限制仅针对于普通连接。预备事务和后台工作者连接都不受这一限制管辖。

[ENCRYPTED] PASSWORD '*password*'
PASSWORD NULL

设置角色的口令（口令只对具有LOGIN属性的角色有用，但是不管怎样还是可以为没有该属性的角色定义一个口令）。如果没有计划使用口令认证，可以忽略这个选项。如果没有指定

口令，口令将被设置为空并且该用户的口令认证总是会失败。也可以用PASSWORD NULL显式地写出一个空口令。

> **注意**
>
> 指定一个空字符串也将把口令设置为空。

口令总是以加密的方式存放在系统目录中。ENCRYPTED关键词没有实际效果，它只是为了向后兼容性而存在。加密的方法由配置参数password_encryption决定。如果当前的口令字符串已经是MD5加密或者SCRAM加密的格式，那么不管password_encryption的值是什么，口令字符串还是原样存储（因为系统无法解密以不同格式加密的口令字符串）。这种方式允许在转储/恢复时重载加密的口令。

VALID UNTIL '*timestamp*'

VALID UNTIL机制设置一个日期和时间，在该时间点之后角色的口令将会失效。如果这个子句被忽略，那么口令将总是有效。

IN ROLE *role_name*

IN ROLE子句列出一个或多个现有的角色，新角色将被立即作为新成员加入到这些角色中（注意没有选项可以把新角色作为一个管理员加入，需要用一个单独的GRANT命令来完成）。

IN GROUP *role_name*

IN GROUP是IN ROLE的一种已废弃的拼写方式。

ROLE *role_name*

ROLE子句列出一个或者多个现有角色，它们会被自动作为成员加入到新角色中（这实际上新角色变成了一个“组”）。

ADMIN *role_name*

ADMIN子句与ROLE相似，但是被提及的角色被使用WITH ADMIN OPTION加入到新角色中，让它们能够把这个角色中的成员关系授予给其他人。

USER *role_name*

USER子句是ROLE子句的一个已废弃的拼写方式。

SYSID *uid*

SYSID子句会被忽略，但是会为了向后兼容，还是会接受它。

注解

使用ALTER ROLE(7)来更改一个角色的属性，以及使用DROP ROLE(7)来移除一个角色。所有用CREATE ROLE指定的属性可以被后来的ALTER ROLE命令所修改。

增加和移除组角色成员的最佳方式是使用GRANT(7)和REVOKE(7)。

VALID UNTIL子句只为一个口令而不是为一个角色本身定义了一个过期时间。特别地，当使用一个非基于口令认证的方法登录时，过期时间是不会被强制的。

INHERIT属性管理可授予特权（也就是对数据库对象和角色成员关系的访问特权）的继承性。它并不适用于由CREATE ROLE和ALTER ROLE设置的特殊角色属性。例如，作为具有CREATEDB特权的角色的一个成员，并不会立刻授予创建数据库的角色，即便INHERIT被设置也是如此，在创建一个数据库之前必须通过SET ROLE(7)成为该角色。

INHERIT属性是用于向后兼容原因的默认值：在早前的UXDB发布中，用户总是能够访问其所属组的所有特权。不过，NOINHERIT更加接近于 SQL 标准中指定的语义。

要注意CREATEROLE特权。对于一个CREATEROLE角色的特权没有继承的概念。那意味着即使一个角色没有特定的特权但被允许创建其他角色，它可以轻易地创建与自身特权不同的另一个角色（除了创建具有超级用户特权的角色）。例如，如果角色“user”具有CREATEROLE特权但是没有CREATEDB特权，但是它能够创建一个带有CREATEDB特权的新角色。因此，可以把具有CREATEROLE特权的角色看成是准超级用户角色。

UXDB包括一个程序createuser，它具有和CREATE ROLE相同的功能（事实上，它会调用这个命令），但是它可以从命令 shell 中运行。

CONNECTION LIMIT只被近似地强制，如果两个新会话在几乎相同时间被开始，而此时该角色只剩下刚好一个连接“槽”，两者可能都将失败。还有，该限制从不对超级用户强制。

用这个命令指定一个非加密口令时必须加以注意。该命令将被以明文的形式传输到服务器，并且它也可能被记录在客户端命令历史或者服务器日志中。不过，命令createuser会传输加密的口令。还有，psql包含一个命令\password，它可以被用来安全地改变该口令。

例子

创建一个能登录但是没有口令的角色：

```
CREATE ROLE jonathan LOGIN;
```

创建一个有口令的角色：

```
CREATE USER davide WITH PASSWORD 'jw8s0F4';
```

（CREATE USER和CREATE ROLE完全相同，除了它带有LOGIN）。

创建一个角色，它的口令有效期截止到 2004 年底。在进入 2005 年第一秒时，该口令会失效。

```
CREATE ROLE miriam WITH LOGIN PASSWORD 'jw8s0F4' VALID UNTIL '2005-01-01';
```

创建一个能够创建数据库并且管理角色的角色：

```
CREATE ROLE admin WITH CREATEDB CREATEROLE;
```

兼容性

SQL 标准中有CREATE ROLE语句，但是标准只要求语法

```
CREATE ROLE name [ WITH ADMIN role_name ]
```

多个初始管理员以及**CREATE ROLE**的所有其他选项都是UXDB扩展。

SQL 标准定义了用户和角色的概念，但是它把它们看成是可区分的概念并且将定义用户的所有命令留给每一种数据库实现来指定。在UXDB中，我们已经选择把用户和角色统一成一种单一实体类型。因此角色比起标准中拥有更多可选的属性。

SQL 标准指定的行为可以通过给用户NOINHERIT属性来得到最大的金丝，而角色会被给予INHERIT属性。

参见

SET ROLE(7), ALTER ROLE(7), DROP ROLE(7), GRANT(7), REVOKE(7)

名称

CREATE RULE — 定义一条新的重写规则

大纲

```
CREATE [ OR REPLACE ] RULE name AS ON event
    TO table_name [ WHERE condition ]
    DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

其中 *event* 可以是以下之一：

```
    SELECT | INSERT | UPDATE | DELETE
```

描述

CREATE RULE定义一条应用于指定表或视图的 新规则。**CREATE OR REPLACE RULE**将创建一条 新规则或者替换同一个表上具有同一名称的现有规则。

UXDB规则系统允许我们定义 针对数据库表中插入、更新或者删除动作上的替代动作。大约来说，当在 一个给定表上执行给定命令时，一条规则会导致执行额外的命令。或者， INSTEAD规则可以用另一个命令替换给定的命令，或 者导致一个命令根本不被执行。规则也被用来实现SQL 视图。规则实际上 是一种命令转换机制或者命令宏。这种转换会在命令的执行开始之前进行。 如果实际上想要为每一个物理行独立地触发一个操作，可能更需要一个 触发器而不是规则。

当前，ON SELECT规则必须是无条件 INSTEAD规则并且其动作必须由一个单一 **SELECT**命令构成。因此，一条 ON SELECT规则实际上把表变成了一个视图，它的可见 内容是由该规则的**SELECT**命令返回，而不是直 接存在该表中的内容（如果有）。不过，使用 **CREATE VIEW**命令还是要比创建一个真实表并且 在其上定义一条ON SELECT规则更好。

可以通过定义ON INSERT、ON UPDATE 以及ON DELETE规则（或者这些规则的任意子集）来创建 可更新的视图，这些规则可以把视图上的更新动作替换为其他表上适当的更新 动作。如果想要支持**INSERT RETURNING**等等，那么一定要 在每一个这类规则中放上一个合适的RETURNING子句。

如果尝试为复杂视图更新使用有条件的规则，有一点是很重要的：对于 希望在该视图上允许的每一个动作，必须有一条 INSTEAD规则。如果该规则是有条件的，或者不是 INSTEAD，那么系统仍将拒绝尝试执行该更新动作， 因为它会认为在某些情况下它应该停止尝试在该视图的傀儡表上执行动作。 如果想处理有条件规则中的所有有用的情况，可以增加一条无条件的 DO INSTEAD NOTHING规则来确保系统理解它将 永远不会被调用来更新傀儡表。然后让有条件规则变成 非-INSTEAD。在它们适用的情况下，它们会加到 默认的INSTEAD NOTHING动作（不过，当前这种方法不 支持RETURNING查询）。

> **注意**
>
> 足够简单的视图自动就是可更新的（见CREATE USER(7)），它们不需要依靠用户创建的规则来变成可 更新的。不过还是可以创建一条显式规则，自动更新转换通常比显式规则效 率高。
>
> 另一种值得考虑的办法是使用INSTEAD OF触发器（见 CREATE TRIGGER(7)）代替规则。

参数

name

要创建的规则的名称。它必须与同一个表上任何其他规则的名称相区分。 同一个表上同一种事件类型的多条规则会按照其名称的字符顺序被应用。

event

时间是SELECT、 INSERT、UPDATE或者 DELETE之一。 注意包含ON CONFLICT子句的INSERT 不能被用在具有INSERT或者 UPDATE规则的表上。那种情况下请考虑使用 可更新的视图。

table_name

规则适用的表或者视图的名称（可以是模式限定的）。

condition

任意的SQL条件表达式（返回 boolean）。该条件表达式不能引用除NEW以及 OLD之外的任何表，并且不能包含聚集函数。

INSTEAD

INSTEAD指示该命令应该取代 原始命令被执行。

ALSO

ALSO指示应该在原始命令 之外执行这些命令。

如果ALSO和INSTEAD都没有被指定， 默认是ALSO。

command

组成规则动作的命令。可用的命令有SELECT、 INSERT、UPDATE、 DELETE或者NOTIFY。

在*condition*和 *command*中，名为 NEW和OLD的表可以被用来引用被 引用表中的值。在ON INSERT和 ON UPDATE规则中，NEW被用来 引用被插入或者更新的新行。在ON UPDATE和ON DELETE规则中，OLD被用来引 用被更新或者删除的现有行。

注解

要在表上创建或者修改规则，必须是表的拥有者。

在一条用于视图上INSERT、UPDATE 或者DELETE的规则中， 可以增加一个RETURNING子句来发出视图的列。如果该规则被一个INSERT RETURNING、 UPDATE RETURNING或者DELETE RETURNING命令触发，这个子句将被用来计算输出。 当规则被一个没有RETURNING的命令触发时，该规则的 RETURNING子句将被忽略。当前的实现只允许无条件 INSTEAD规则包含RETURNING。此外，用于同一事件 的所有规则中至多只能有一个RETURNING子句（这确保了只有一个 候选RETURNING子句被用来计算结果）。如果在任何可用规则中都 没有RETURNING子句，视图上的RETURNING查询将 被拒绝。

避免循环规则非常重要。例如，尽管下面的两条规则定义都被 UXDB所接受， SELECT命令将导致UXDB报告一个错误，因为会产生一条 规则的递归扩展：

```
CREATE RULE "_RETURN" AS
   ON SELECT TO t1
   DO INSTEAD
      SELECT * FROM t2;

CREATE RULE "_RETURN" AS
   ON SELECT TO t2
   DO INSTEAD
      SELECT * FROM t1;

SELECT * FROM t1;
```

当前，如果一个规则动作包含一个NOTIFY命令， 该NOTIFY命令将被无条件执行，也就是说，即使 该规则不被应用到任何行上，也会发出NOTIFY。 例如，在

```
CREATE RULE notify_me AS ON UPDATE TO mytable DO ALSO NOTIFY mytable;

UPDATE mytable SET name = 'foo' WHERE id = 42;
```

中，UPDATE期间将发出一个 NOTIFY事件，不管是否有行匹配条件 id = 42。这是一种实现限制，它可能会在未来的发行中被修复。

兼容性

CREATE RULE是一种 UXDB语言扩展， 整个查询重写系统也是这样。

另见

ALTER RULE(7), DROP RULE(7)

名称

CREATE SCHEMA — 定义一个新模式

大纲

```
CREATE SCHEMA schema_name [ AUTHORIZATION role_specification ] [ schema_element [ ... ] ]
CREATE SCHEMA AUTHORIZATION role_specification [ schema_element [ ... ] ]
CREATE SCHEMA IF NOT EXISTS schema_name [ AUTHORIZATION role_specification ]
CREATE SCHEMA IF NOT EXISTS AUTHORIZATION role_specification
```

其中 *role_specification* 可以是：

```
  user_name
| CURRENT_USER
| SESSION_USER
```

描述

CREATE SCHEMA输入一个新模式到当前数据库中。 该模式名必须与当前数据库中任何现有模式的名称不同。

一个模式本质上是一个名字空间：它包含命令对象（表、数据类型、函数以及操作符）， 对象可以与在其他模式中存在的对象重名。可以通过用模式名作为一个前缀 “限定”命名对象的名称来访问它们，或者通过把要求的模式包括 在搜索路径中来访问命名对象。一个指定非限定对象名的 CREATE命令在当前模式（搜索路径中的第一个模式，由函数 current_schema决定）中创建对象。

CREATE SCHEMA中可以选择包括子命令用以在 新模式中创建对象。这些子命令实际被当做独立的在创建该模式后被发出的命令 一样，除非使用AUTHORIZATION子句，所有被创建的对象都会由该用户拥有。

参数

schema_name

要创建的一个模式名。如果省略， *user_name*将被用作模式名。 该名称不能以ux_开始，因为这样的名称是用作系统模式的。

user_name

将拥有新模式的用户的角色名。如果省略，默认为执行该命令的用户。要 创建由另一个角色拥有的角色，必须是那个角色的一个直接或者间接成员， 或者是一个超级用户。

schema_element

要在该模式中创建的对象的定义 SQL 语句。当前，只有CREATE TABLE、CREATE VIEW、CREATE INDEX、CREATE SEQUENCE、CREATE TRIGGER以及GRANT被接受为 CREATE SCHEMA中的子句。其他类型的对象可以在模式被 创建之后用单独的命令创建。

IF NOT EXISTS

如果一个具有同名的模式已经存在，则什么也不做（不过发出一个提示）。 使用这个选项时不能包括 *schema_element*子命令。

注解

要创建一个模式，调用用户必须拥有当前数据库的CREATE　特权（当然，超级用户可以绕过这种检查）。

示例

创建一个模式：

```
CREATE SCHEMA myschema;
```

为用户joe创建一个模式，该模式也将被命名为 joe：

```
CREATE SCHEMA AUTHORIZATION joe;
```

创建一个被用户joe拥有的名为test的模式， 除非已经有一个名为test的模式（不管joe 是否拥有该已经存在的模式）。

```
CREATE SCHEMA IF NOT EXISTS test AUTHORIZATION joe;
```

创建一个模式并且在其中创建一个表和视图：

```
CREATE SCHEMA hollywood
  CREATE TABLE films (title text, release date, awards text[])
  CREATE VIEW winners AS
    SELECT title, release FROM films WHERE awards IS NOT NULL;
```

注意子命令不以分号结束。

下面是达到相同结果的等效的方法：

```
CREATE SCHEMA hollywood;
CREATE TABLE hollywood.films (title text, release date, awards text[]);
CREATE VIEW hollywood.winners AS
  SELECT title, release FROM hollywood.films WHERE awards IS NOT NULL;
```

兼容性

SQL 标准允许在**CREATE SCHEMA**中有一个 DEFAULT CHARACTER SET子句，以及当前 UXDB接受的更多子命令类型。

SQL 标准制定**CREATE SCHEMA**中的子命令 可以以任何顺序出现。当前的 UXDB实现不能处理子命令中 所有情况的向前引用。有时候可能有必要对子命令进行重排序以避免向前 引用。

根据 SQL 标准，模式的拥有者总是拥有其中的所有对象。 UXDB允许模式包含非模式 拥有者所拥有的对象。只有模式拥有者把其模式上的CREATE　特权授予给了其他人或者一个超级用户选择在该模式中创建对象时才会 发生这种事情。

IF NOT EXISTS选项是一种 UXDB扩展。

另见

ALTER SCHEMA(7), DROP SCHEMA(7)

名称

CREATE SEQUENCE — 定义一个新的序列发生器

大纲

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name [ INCREMENT
 [ BY ] increment ]
   [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
   [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
   [ OWNED BY { table_name.column_name | NONE } ]
```

描述

CREATE SEQUENCE创建一个新的序列数发生器。这涉及到用名称*name*创建并且初始化一个新的特殊的单行表。该发生器将由发出该命令的用户所拥有。

如果给出一个模式名称，则该序列将将被创建在指定的模式中。否则它会被 创建在当前模式中。临时序列存在于一个特殊的模式中，因此在创建临时序列 时不能给出模式名。序列名称必须与同一模式中任何其他序列、表、索引、 视图或者外部表的名称不同。

在序列被创建后，可以使用函数 nextval、 currval以及 setval来操作该序列。这些函数在第 6.16 节 “序列操作函数”中有介绍。

尽管无法直接更新一个序列，可以使用这样的查询：

```
SELECT * FROM name;
```

来检查一个序列的参数以及当前状态。特别地，序列的 last_value域显示被任意会话最后一次取得的值（当然， 在被打印时该值可能已经过时了，因为可能有其他会话正在执行 nextval调用）。

参数

TEMPORARY or TEMP

如果被指定，只会为这个会话创建序列对象，并且在会话退出时自动 删除它。当临时序列存在时，已有的同名永久序列（在这个会话中） 会变得不可见，不过可以用模式限定的名称来引用同名永久序列。

IF NOT EXISTS

如果已经存在一个同名的关系时不要抛出错误。这种情况下会发出一个 提示。注意这不保证现有的关系与即将创建的序列相似 -- 它甚至可能 都不是一个序列。

name

要创建的序列的名称（可以是模式限定的）。

data_type

可选的子句AS *data_type* 制定序列的数据类型。有效类型是 smallint、integer、 和bigint。默认是bigint。 数据类型决定了序列的默认最小和最大值。

increment

可选的子句INCREMENT BY *increment*指定为了 创建新值会把哪个值加到当前序列值上。一个正值将会创造一个上升 序列，负值会创造一个下降序列。默认值是 1。

minvalue
NO MINVALUE

可选的子句MINVALUE *minvalue*决定一个序列 能产生的最小值。如果没有提供这个子句或者指定了 NO MINVALUE，那么会使用默认值。 升序序列的默认值为1。降序序列的默认值为数据类型的最小值。

maxvalue
NO MAXVALUE

可选的子句MAXVALUE *maxvalue*决定该序列 的最大值。如果没有提供这个子句或者指定了NO MAXVALUE，那么将会使用默认值。 升序序列的默认值是数据类型的最大值。降序序列的默认值是-1。

start

可选的子句START WITH *start* 允许序列从任何 地方开始。对于上升序列和下降序列来说，默认的开始值分别是 *minvalue*和 *maxvalue*。

cache

可选的子句CACHE *cache*指定要预分配多少 个序列数并且把它们放在内存中以便快速访问。最小值为 1 （一次只生成 一个值，即没有缓存），默认值也是 1。

CYCLE
NO CYCLE

对于上升序列和下降序列，CYCLE选项允许序列 在分别达到*maxvalue*和*minvalue*时回卷。如果达到 该限制，下一个产生的数字将分别是*minvalue*和*maxvalue*。

如果指定了NO CYCLE，当序列到达其最大值 后任何nextval调用将返回一个错误。如果CYCLE和NO CYCLE都没有 被指定，则默认为NO CYCLE。

OWNED BY *table_name. column_name*
OWNED BY NONE

OWNED BY选项导致序列被与一个特定的表列关联 在一起，这样如果该列（或者整个表）被删除，该序列也将被自动删除。 指定的表必须和序列具有相同的拥有者并且在同一个模式中。默认选项 OWNED BY NONE指定该序列不与某个列关联。

注解

使用**DROP SEQUENCE**移除一个序列。

序列是基于bigint算法的，因此范围是不能超过一个八字节 整数的范围（-9223372036854775808 到 9223372036854775807）。

由于nextval和setval调用绝不会回滚， 如果需要序数的“无间隙”分配，则不能使用序列对象。可以 通过在一个只包含一个计数器的表上使用排他锁来构建无间隙的分配， 但是这种方案比序列对象开销更大，特别是当有很多事务并发请求序数 时。

如果对一个将由多个会话并发使用的序列对象使用了大于 1 的*cache*设置，可能会得到意想不到的结果。 每个会话会在访问该序列对象时分配并且缓存后续的序列值，并且相应地增加 该序列对象的last_value。然后，在该会话中下一次 nextval会做 *cache*-1，并且简单地 返回预分配的值而不修改序列对象。因此，任何已分配但没有在会话中使用的 数字将会在该会话结束时丢失，导致该序列中的“空洞”。

进一步，尽管多个会话能分配到不同的序列值，这些值可能会在所有会话都被考虑时生成出来。例如，*cache*的设置为 10，会话A 可能储存值 1..10 并且返回nextval=1，然后会话 B 可能储存值 11..20 并且在 A 生成nextval=2 之前返回nextval=11。因此，如果*cache*设置为 1，可以安全地假设nextval值被顺序地生成。如果*cache*设置大于 1，就只能假定nextval值都是可区分的，但不能保证它们被完全地顺序生成。还有，last_value将反映服务于任意会话的最后一个值，不管它是否已经被nextval返回过。

另一个考虑是，在这样一个序列上执行的setval将不会通知其他会话，直到它们用尽了任何已缓存的预分配值。

示例

创建一个称作serial的上升序列，从 101 开始：

```
CREATE SEQUENCE serial START 101;
```

从这个序列中选取下一个数字：

```
SELECT nextval('serial');

 nextval
---------
     101
```

再从这个序列中选取下一个数字，可使用如下方式。

```
SELECT serial.nextval;

 nextval
---------
     102
```

在一个INSERT命令中使用这个序列：

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

在一次COPY FROM后更新新列值：

```
BEGIN;
COPY distributors FROM 'input_file';
SELECT setval('serial', max(id)) FROM distributors;
END;
```

兼容性

CREATE SEQUENCE符合SQL 标准，不过下列除外：

- 使用nextval()而不是标准的**NEXT VALUE FOR** 表达式获取下一个值。
- OWNED BY子句是一种UXDB扩展。

另见

ALTER SEQUENCE(7), DROP SEQUENCE(7)

名称

CREATE SERVER — 定义一个新的外部服务器

大纲

```
CREATE SERVER [ IF NOT EXISTS ] server_name [ TYPE 'server_type' ] [ VERSION
 'server_version' ]
   FOREIGN DATA WRAPPER fdw_name
   [ OPTIONS ( option 'value' [, ... ] ) ]
```

描述

CREATE SERVER定义一个新的外部服务器。 定义该服务器的用户会成为拥有者。

外部服务器通常包装了外部数据包装器用来访问一个外部数据源所需的 连接信息。额外的用户相关的连接信息可以通过用户映射的方式来指定。

服务器名称在数据库中必须唯一。

创建服务器要求所使用的外部数据包装器上的USAGE特权。

参数

IF NOT EXISTS

如果已经存在同名的服务器，不要抛出错误。在这种情况下发出一个通知。 请注意，不能保证现有服务器与要创建的服务器类似。

server_name

要创建的外部服务器的名称。

server_type

可选的服务器类型，可能对外部数据包装器有用。

server_version

可选的服务器版本，可能对外部数据包装器有用。

fdw_name

管理该服务器的外部数据包装器的名称。

OPTIONS (*option* '*value*' [, ...])

这个子句为服务器指定选项。这些选项通常定义该服务器的连接细节， 但是实际的名称和值取决于该服务器的外部数据包装器。

注解

在使用dblink模块时，一个外部服务器的名称可以被 用作dblink_connect函数的一个参数来指示 连接参数。以这种方式使用外部服务器，需要在其上具有 USAGE特权。

示例

创建使用外部数据包装器uxdb_fdw 的服务器myserver：

```
CREATE SERVER myserver FOREIGN DATA WRAPPER uxdb_fdw OPTIONS (host 'foo', dbname
 'foodb', port '5432');
```

兼容性

CREATE SERVER符合 ISO/IEC 9075-9 (SQL/MED)。

另见

ALTER SERVER(7), DROP SERVER(7), CREATE FOREIGN DATA WRAPPER(7), CREATE FOREIGN TABLE(7), CREATE USER MAPPING(7)

名称

CREATE STATISTICS — 定义扩展统计

大纲

```
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
  [ ( statistics_kind [, ... ] ) ]
  ON column_name, column_name [, ...]
  FROM table_name
```

描述

CREATE STATISTICS将创建一个新的扩展统计对象， 追踪指定表、外部表或物化视图的数据。该统计对象将在当前数据库中创建， 被发出该命令的用户所有。

如果给定了模式名（比如，CREATE STATISTICS myschema.mystat ...）， 那么在给定的模式中创建统计对象。否则在当前模式中创建。 统计对象的名称必须与相同模式中的任何其他统计对象不同。

参数

IF NOT EXISTS

如果具有相同名称的统计对象已经存在，不会抛出一个错误，只会发出一个提示。 请注意，这里只考虑统计对象的名称，不考虑其定义细节。

statistics_name

要创建的统计对象的名称（可以有模式限定）。

statistics_kind

在此统计对象中计算的统计种类。目前支持的种类是启用n-distinct统计的 ndistinct，启用功能依赖性统计的dependencies，以及启用最常见的值列表的mcv。 如果省略该子句，则统计对象中将包含所有支持的统计类型。

column_name

被计算的统计信息包含的表格列的名称。至少必须给出两个列名，列名的顺序可以忽略。

table_name

包含计算统计信息的列的表的名称（可以是模式限定的）。

注意

必须是表的所有者才能创建读取它的统计对象。不过，一旦创建， 统计对象的所有权与基础表无关。

示例

用两个功能相关的列创建表t1， 即第一列中的值的信息足以确定另一列中的值。然后， 在这些列上构建函数依赖关系统计信息：

```
CREATE TABLE t1 (
   a   int,
   b   int
);

INSERT INTO t1 SELECT i/100, i/500
            FROM generate_series(1,1000000) s(i);

ANALYZE t1;

-- 匹配行的数量将被大大低估：
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);

CREATE STATISTICS s1 (dependencies) ON a, b FROM t1;

ANALYZE t1;

-- 现在行计数估计会更准确：
EXPLAIN ANALYZE SELECT * FROM t1 WHERE (a = 1) AND (b = 0);
```

如果没有函数依赖性统计，规划器会认为两个WHERE条件是独立的， 并且会将它们的选择性乘以一起，以致得到太小的行数估计。 通过这样的统计，规划器认识到WHERE条件是多余的，并且不会低估行数。

创建表t2与两个完全相关的列(包含相同的数据)，并且在这些列上创建一个MCV列表:

```
CREATE TABLE t2 (
   a   int,
   b   int
);

INSERT INTO t2 SELECT mod(i,100), mod(i,100)
            FROM generate_series(1,1000000) s(i);

CREATE STATISTICS s2 (mcv) ON a, b FROM t2;

ANALYZE t2;

-- valid combination (found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 1);

-- invalid combination (not found in MCV)
EXPLAIN ANALYZE SELECT * FROM t2 WHERE (a = 1) AND (b = 2);
```

MCV列表为计划器提供了关于表中普遍出现的特定值的更详细的信息，以及表中未显示的值组合的选择性上限，允许它在这两种情况下产生更好的估计值。

兼容性

SQL标准中没有**CREATE STATISTICS**命令。

又见

ALTER STATISTICS(7), DROP STATISTICS(7)

名称

CREATE SUBSCRIPTION — 定义一个新的订阅

大纲

```
CREATE SUBSCRIPTION subscription_name
  CONNECTION 'conninfo'
  PUBLICATION publication_name [, ...]
  [ WITH ( subscription_parameter [= value] [, ... ] ) ]
```

描述

CREATE SUBSCRIPTION为当前数据库添加一个新的订阅。 订阅名称必须与数据库中任何现有的订阅不同。

订阅表示到发布者的复制连接。因此，此命令不仅在本地目录中添加定义， 还会在发布者上创建复制插槽。

在运行此命令的事务提交时，将启动逻辑复制工作器以复制新订阅的数据。

参数

subscription_name

新订阅的名称。

CONNECTION '*conninfo*'

连接发布者的字符串。

PUBLICATION *publication_name*

要订阅的发布者上的发布名称。

WITH (*subscription_parameter* [= *value*] [, ...])

该子句指定订阅的可选参数。支持的参数有：

copy_data（boolean）

指定在复制启动后是否应复制正在订阅的发布中的现有数据。 默认值是true。

create_slot（boolean）

指定该命令是否要在发布者上创建复制槽。默认值是true。

enabled（boolean）

指定订阅是否应该主动复制，或者是否应该只是设置，但尚未启动。 默认值是true。

slot_name（string）

要使用的复制插槽的名称。默认行为是使用订阅名称作为插槽的名称。

当slot_name设置为NONE时， 将不会有复制槽与订阅关联。这在需要稍后手动设置复制槽的情况下会使用。 这样的订阅必须同时enabled并且 create_slot设置为false。

synchronous_commit（enum）

该参数的值会覆盖synchronous_commit设置。 默认值是off。

对于逻辑复制使用off是安全的： 如果订阅者由于缺少同步而丢失事务，数据将从发布者重新发送。

进行同步逻辑复制时，不同的设置可能是合适的。 逻辑复制工作者向发布者报告写入和刷新的位置，当使用同步复制时， 发布者将等待实际刷新。这意味着，当订阅用于同步复制时， 将订阅者的synchronous_commit设置为off 可能会增加发布服务器上COMMIT的延迟。 在这种情况下，将synchronous_commit设置为 local或更高是有利的。

connect（boolean）

指定CREATE SUBSCRIPTION是否应该连接到发布者。 将其设置为false将会改变默认值enabled、 create_slot和copy_data为false。

不允许将connect设置为false的同时将 enabled、create_slot或copy_data 设置为true。

因为该选项设置为false时不会建立连接， 因此表没有被订阅，所以当启用订阅后，不会复制任何内容。 需要运行ALTER SUBSCRIPTION ... REFRESH PUBLICATION才能订阅表。

注意

创建复制槽时(默认行为)，CREATE SUBSCRIPTION不能在事务块内部执行。

如果复制插槽不是作为同一命令的一部分创建的，则创建连接到相同数据库集群的订阅 （例如，在同一集群中的数据库之间进行复制或在同一个数据库中进行复制）只能成功 否则，CREATE SUBSCRIPTION调用将挂起。要做到这一点， 单独创建复制插槽（使用函数ux_create_logical_replication_slot 和插件名称uxoutput），并使用参数create_slot = false 创建订阅。

示例

创建一个到远程服务器的订阅，复制发布mypublication和 insert_only中的表，并在提交时立即开始复制：

```
CREATE SUBSCRIPTION mysub
    CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
    PUBLICATION mypublication, insert_only;
```

创建一个到远程服务器的订阅，复制insert_only发布中的表， 并且不开始复制直到稍后启用复制。

```
CREATE SUBSCRIPTION mysub
    CONNECTION 'host=192.168.1.50 port=5432 user=foo dbname=foodb'
    PUBLICATION insert_only
        WITH (enabled = false);
```

兼容性

CREATE SUBSCRIPTION是一个UXDB 扩展。

又见

ALTER SUBSCRIPTION(7), DROP SUBSCRIPTION(7), CREATE PUBLICATION(7), ALTER PUBLICATION(7)

名称

CREATE SYNONYM — 创建一个同义词

大纲

CREATE SYNONYM *synname* FOR *orgname*
CREATE SYNONYM *synname* FOR *funcname(arg, arg, ...)*

描述

同义词(synonym)是表、索引、视图等模式对象的一个别名。通过为模式对象创建同义词，可以隐藏对象的实际名称和所有者信息，由此为对象提提供一定的安全性保证。

参数

synname

用户自定义的同义词名称。

orgname

同义词属主的名称。

funcname(arg,arg,...)

函数对象名称及函数参数。

synname(arg,arg,...)

同义词函数名称及参数。

注解

1. 不能更新同义词名称。
2. 不能直接找出同义词与属主的映射关系。
3. 删除属主时，会报有同义词依赖的错误。

示例

表t1创建同义词，如下所示。

```
create synonym synt1on for t1;
```

索引idx1创建同义词，如下所示。

```
create synonym syni1 for idx1;
```

序列seq1创建同义词，如下所示。

```
create synonym syn1 for seq1;
```

视图vw1创建同义词，如下所示。

```
create synonym synv1 for vw1;
```

函数add创建同义词，如下所示。

```
create synonym s1 for add(integer, integer, integer);
```

名称

CREATE TABLE — 定义一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name ( [
  { column_name data_type [ column_name data_type NOT CLUSTER PRIMARY KEY ]
 [ CONSTRAINT index_name NOT CLUSTER PRIMARY KEY ( column_name ) ] [ column_name
 TIMESTAMP ON UPDATE NOW() ] [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation
 ] [ opclass ] [, ... ] ) ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
[ STORAGE ( ON tablespace_name, CLUSTERBTR ) ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name
    OF type_name [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation
 ] [ opclass ] [, ... ] ) ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name
    PARTITION OF parent_table [ (
  { column_name [ WITH OPTIONS ] [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ] { FOR VALUES partition_bound_spec | DEFAULT }
[ PARTITION BY { RANGE | LIST | HASH } ( { column_name | ( expression ) } [ COLLATE collation
 ] [ opclass ] [, ... ] ) ]
[ USING method ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

其中 *column_constraint* 是：

```
[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  GENERATED ALWAYS AS ( generation_expr ) STORED |
  GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [ ( sequence_options ) ] |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE referential_action ] [ ON UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

table_constraint 是：

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters
 [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE referential_action ] [ ON
 UPDATE referential_action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

like_option 是：

```
{ INCLUDING | EXCLUDING } { COMMENTS | CONSTRAINTS | DEFAULTS | GENERATED |
 IDENTITY | INDEXES | STATISTICS | STORAGE | ALL }
```

partition_bound_spec 是:

```
IN ( partition_bound_expr [, ...] ) |
FROM ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] )
  TO ( { partition_bound_expr | MINVALUE | MAXVALUE } [, ...] ) |
WITH ( MODULUS numeric_literal, REMAINDER numeric_literal )
```

UNIQUE、PRIMARY KEY以及EXCLUDE约束中的*index_parameters*是：

```
[ INCLUDE ( column_name [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]
```

一个EXCLUDE约束中的*exclude_element*是：

```
{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

描述

CREATE TABLE将在当前数据库中创建一个新的、初始为空的表。该表将由发出该命令的用户所拥有。

如果给定了一个模式名（例如CREATE TABLE myschema.mytable ...），那么该表被创建在指定的模式中。否则它被创建在当前模式中。临时表存在于一个特殊的模式中，因此在创建一个临时表时不能给定一个模式名。该表的名称必须与同一个模式中的任何其他表、序列、索引、视图或外部表的名称区分开。

CREATE TABLE也会自动地创建一个数据类型来表示对应于该表一行的组合类型。因此，表不能用同一个模式中任何已有数据类型的名称。

可选的约束子句指定一个插入或更新操作要成功，新的或更新过的行必须满足的约束（测试）。一个约束是一个 SQL 对象，它帮助以多种方式定义表中的合法值集合。

有两种方式来定义约束：表约束和列约束。一个列约束会作为列定义的一部分定义。一个表约束定义不与一个特定列绑定，并且它可以包含多于一个列。每一个列约束也可以被写作一个表约束，列约束只是一种当约束只影响一列时方便书写的记号习惯。

要能创建一个表，必须分别具有所有列类型或OF子句中类型的USAGE特权。

参数

TEMPORARY or TEMP

如果指定，该表被创建为一个临时表。临时表会被在会话结束时自动被删除，或者也可以选择在当前事务结束时删除（见下文的ON COMMIT）。当临时表存在时，已有的同名持久表将对于当前会话不可见，不过可以使用模式限定的名称进行引用。在一个临时表上创建的任何索引也自动地变为临时的。

自动清理守护进程不能访问并且因此也不能清理或分析临时表。由于这个原因，应该通过会话的 SQL 命令执行合适的清理和分析操作。例如，如果一个临时表将要被用于复杂的查询，最好在把它填充完毕后在其上运行ANALYZE。

可以选择将GLOBAL或LOCAL写在TEMPORARY或TEMP的前面。这当前在UXDB中没有区别并且已被废弃，见“兼容性”一节。

UNLOGGED

如果指定，该表被创建为一个不受日志记录的表。被写入到不做日志的表中的数据不会被写到预写式日志中，这让它们比普通表快非常多。不过，它们在崩溃时是不安全的：一个不做日志的表在一次崩溃或非干净关闭之后会被自动地截断。一个不做日志的表中的内容也不会被复制到后备服务器中。在一个不做日志的表上创建的任何索引也会自动地不被日志记录。

IF NOT EXISTS

如果一个同名关系已经存在，不要抛出一个错误。在这种情况下会发出一个提示。注意这不保证现有的关系是和将要被创建的表相似的东西。

table_name

要被创建的表名（可以选择用模式限定）。

OF *type_name*

创建一个类型化的表，它的结构取自于指定的组合类型（名字可以选择用模式限定）。一个类型化的表和它的类型绑定在一起，例如如果类型被删除，该表也将被删除（用DROP TYPE ... CASCADE）。

当一个类型化的表被创建时，列的数据类型由底层的组合类型决定而没有在CREATE TABLE命令中直接指定。但是CREATE TABLE命令可以对表增加默认值和约束，并且可以指定存储参数。

column_name

列的名称会在新表中被建立。

ON UPDATE NOW()

执行UPDATE语句时，当TIMESTAMP和TIMESTAMPTZ类型的列包含ON UPDATE NOW()属性时，该列会自动更新时间值，其中NOW()函数可以替换成与它同义的函数。

表 6. NOW()同义函数

时间函数
CURRENT_TIMESTAMP
CURRENT_TIMESTAMP()
CURRENT_TIMESTAMP(0-6)
NOW(0-6)
LOCALTIME
LOCALTIME()
LOCALTIME(0-6)
LOCALTIMESTAMP
LOCALTIMESTAMP()
LOCALTIMESTAMP(0-6)

在ON UPDATE NOW()自动更新为当前时间的语法中，将上面列出的函数均看做NOW()的同义函数，返回当前语句执行的时间，如果修饰的列类型为TIMESTAMP，返回不带时区的时间戳，如果修饰的列类型为TIMESTAMPTZ，返回带时区的时间戳。

表 7. 支持的sql语句

sql语句
CREATE TABLE t1 (id int,ts TIMESTAMP ON UPDATE CURRENT_TIMESTAMP)
CREATE TABLE t1 (id int,ts TIMESTAMP（p） ON UPDATE CURRENT_TIMESTAMP(p))
CREATE TABLE t1 (id int,ts TIMESTAMP ON UPDATE CURRENT_TIMESTAMP(0))
CREATE TABLE t1 (id int,ts TIMESTAMP（0） ON UPDATE CURRENT_TIMESTAMP)
alter table t1 add ts1 TIMESTAMP on update CURRENT_TIMESTAMP;
alter table t1 add ts1 TIMESTAMP（p） on update CURRENT_TIMESTAMP(p);
alter table t1 add ts1 TIMESTAMP on update CURRENT_TIMESTAMP(0);

sql语句
alter table t1 alter ts1 set on update current_timestamp ; ts1与current_timestamp的精度设置保持一致
alter table t1 alter ts1 drop on update current_timestamp ; ts1与current_timestamp的精度设置保持一致
alter table t1 alter ts2 type TIMESTAMP on update current_timestamp; ts2的目标类型精度与current_timestamp的精度设置保持一致

1. 表中p表示精度，取值范围为0-6。

2. 表中的TIMESTAMP可以改成TIMESTAMPTZ，current_timestamp均可替换成同义函数。

3. uxdb对current_timestamp函数的精度不做校验，自动更新时，与修饰的时间列精度保持一致。

4. 在未指定精度创建的表的前提下，mysql和uxdb插入或更新为当前时间的精度有差异：mysql的时间列显示的时间精度均为0，uxdb显示的时间精度在0-6之间，但是不固定。

CONSTRAINT *index_name* NOT CLUSTER PRIMARY KEY(*column_name*)
column_name data_type NOT CLUSTER PRIMARY KEY

以NOT CLUSTER PRIMARY KEY约束条件，实现建立非聚簇索引的表。创建表时候，在列名全部声明完之后使用。

CONSTRAINT约束语法建立NOT CLUSTER PRIMARY KEY约束column_name即作为主键，index_name 表示索引名。或者在需要作为主键的列名后添加语法NOT CLUSTER PRIMARY KEY。

对于非聚簇索引的表，可使用CLUSTER TABLE_NAME [USING index_name]将非聚簇索引的表转换成聚簇索引的表。index_name 是表的主键索引名。可通过\d table_name查看。

data_type

列的数据类型. 这可以包括数组 规格. 有关UXDB支持数据类型的详细信息，请参考第 5 章 数据类型。

COLLATE *collation*

COLLATE子句为该列（必须是一种可排序数据类型）赋予一个排序规则。 如果没有指定，将使用该列数据类型的默认排序规则。

INHERITS (*parent_table* [, ...])

可选的INHERITS子句指定一个表的列表， 新表将从其中自动地继承所有列。 父表可以是普通表或者外部表。

INHERITS的使用在新的子表和它的父表之间创建一种持久的关系。 对于父表的模式修改通常也会传播到子表， 并且默认情况下子表的数据会被包括在对父表的扫描中。

如果在多个父表中存在同名的列，除非父表中每一个这种列的数据类型都能匹配， 否则会报告一个错误。如果没有冲突，那么重复列会被融合来形成新表中的一个单一列。 如果新表中的列名列表包含一个也是继承而来的列名，该数据类型必须也匹配继承的列， 并且列定义会被融合成一个。如果新表显式地为列指定了任何默认值， 这个默认值将覆盖来自该

列继承声明中的默认值。 否则，任何父表都必须为该列指定相同的默认值，或者会报告一个错误。

CHECK约束本质上也采用和列相同的方式被融合： 如果多个父表或者新表定义中包含相同的命名CHECK约束， 这些约束必须全部具有相同的检查表达式，否则将报告一个错误。 具有相同名称和表达式的约束将被融合成一份拷贝。 一个父表中的被标记为NO INHERIT的约束将不会被考虑。 注意新表中一个未命名的CHECK约束将永远不会被融合， 因为那样总是会为它选择一个唯一的名字。

列的STORAGE设置也会从父表复制过来。

如果父表中的列是标识列，那么该属性不会被继承。 如果需要，可以将子表中的列声明为标识列。

PARTITION BY { RANGE | LIST | HASH } ({ *column_name* | (*expression*) } [*opclass*] [, ...])

可选的PARTITION BY子句指定了对表进行分区的策略。 这样创建的表称为分区表。 带括号的列或表达式的列表构成表的分区键。 使用范围或哈希分区时，分区键可以包含多个列或表达式（最多32个，但在构建 UXDB时可以更改此限制）， 但对于列表分区，分区键必须由单个列或表达式组成。

范围和列表分区需要 btree 运算符类，而哈希分区需要哈希运算符类。 如果没有运算符类被显式指定，将使用相应类型的默认运算符类； 如果不存在默认运算符类，则将引发错误。使用哈希分区时，所使用的运算符类必须实现支持功能 2。

分区表被分成多个子表（称为分区），它们是使用单独的CREATE TABLE命令创建的。 分区表本身是空的。插入到表中的数据行将根据分区键中的列或表达式的值路由到分区。 如果没有现有的分区与新行中的值匹配，则会报告错误。

分区表不支持EXCLUDE约束； 但是，可以在各个分区上定义这些约束。

有关表分区的更多讨论，请参阅第 2.11 节 “表分区”。

PARTITION OF *parent_table* { FOR VALUES *partition_bound_spec* | DEFAULT }

将表创建为指定父表的分区。 该表建立时，可以使用FOR VALUES创建为特定值的分区，也可以使用DEFAULT创建默认分区。

partition_bound_spec 必须对应于父表的分区方法和分区键，并且必须不能与该父表的任何现有分区重叠。 具有IN的形式用于列表分区， 具有FROM和TO的形式用于范围分区， 具有WITH的形式用于哈希分区。

*partition_bound_expr*是任何无变量表达式（不允许子查询、窗口函数、聚合函数和集返回函数）。 它的数据类型必须与相应分区键列的数据类型相匹配。 表达式在表创建时只计算一次，因此它甚至可以包含易失性表达式，如CURRENT_TIMESTAMP。

在创建列表分区时，可以指定NULL来表示分区允许分区键列为空。 但是，给定父表不能有多于一个这样的列表分区。无法为范围分区指定 NULL。

创建范围分区时，由FROM指定的下限是一个包含范围， 而用TO指定的上限是排除范围。也就是说， 在FROM列表中指定的值是该分区的相应分区键列的有效值， 而TO列表中的值不是。请注意， 必须根据按行比较的规则来理解此语句（第 6.23.5 节 “行构造器比较”）。 例如，给定PARTITION BY RANGE (x,y)，分区范围 FROM (1, 2) TO (3, 4)允许x=1与任何y>=2， x=2与任何非空y，和x=3与任何y<4。

在创建范围分区以指示列值没有下限或上限时，可以使用特殊值MINVALUE和MAXVALUE。例如，使用FROM (MINVALUE) TO (10) 定义的分区允许任何小于10的值，并且使用FROM (10) TO (MAXVALUE) 定义的分区允许任何大于或等于10的值。

创建涉及多个列的范围分区时，将MAXVALUE作为下限的一部分并将 MINVALUE作为上限的一部分也是有意义的。例如，使用 FROM (0, MAXVALUE) TO (10, MAXVALUE) 定义的分区允许第一个分区键列大于0且小于或等于10的任何行。类似地， 使用FROM ('a', MINVALUE) TO ('b', MINVALUE)定义的分区 允许第一个分区键列以″a″开头的任何行。

请注意，如果MINVALUE或MAXVALUE用于分区边界的一列， 则必须为所有后续列使用相同的值。例如，(10, MINVALUE, 0) 不是有效的边界；应该写(10, MINVALUE, MINVALUE)。

还要注意，某些元素类型，如timestamp，具有“无穷”的概念， 这只是另一个可以存储的值。这与MINVALUE和MAXVALUE不同， 它们不是可以存储的实际值，而是它们表示值无界的方式。MAXVALUE 可以被认为比任何其他值（包括“无穷”）都大的值，MINVALUE 可以被认为是比任何其他值（包括“负无穷”）都小的值。因此， 范围FROM ('infinity') TO (MAXVALUE)不是空的范围； 它只允许存储一个值— ″infinity″。

如果指定了DEFAULT，则表将创建为父表的默认分区。此选项不适用于哈希分区表。 不适合给定父级表的任何其他分区的分区键值将路由到默认分区。

当一个表已有DEFAULT 分区并且要对它添加新分区时， 必须扫描默认分区以验证它不包含可能属于新分区的任何行。 如果默认分区包含大量行，则速度可能会很慢。 如果默认分区是外表或者它具有可证明的不可能包含能放置在新分区中的行的约束，则将略过扫描

当创建哈希分区时，必须指定模数和余数。 模数必须是正整数，余数必须是小于模数的非负整数。 通常情况下，当初始设置哈希分区表时，应选择一个与分区数相等的模数，并为每个表分配相同的模数和不同的余数（请参阅下方示例）。 不过，并不要求每个分区都具有相同的模数，只要求哈希分区表里面的分区中出现的每个模数都是下一个较大模数的因数。 这允许以增量的方式增加分区数量而不需要一次移动所有数据。 例如，假设有一个包含 8 个分区的哈希分区表，每个分区有模数8，但发现有必要将分区数增加到16 个。 您可以拆分其中一个模数-8分区，然后创建两个新的模数-16分区来覆盖键空间的相同部分（一个的余数等于被拆分的分区的余数，另一个的余数等于该值加 8），而后用数据重新填充他们。 然后，可以对每一个余数-8分区重复此操作过程，直到没有剩余。 虽然这其中的每个步骤都可能会导致大量的数据移动操作，它仍然要好于建一个全新的表并一次移动全部数据。

分区必须与其所属的分区表的字段名和类型相同。 对分区表字段名或类型的修改，将自动传播到所有分区。 CHECK约束将自动被每一个分区继承，但是单独的分区可以指定额外的CHECK约束；与父表相同名称和条件的额外约束将被父表约束合并。 可以为每个分区分别指定默认值。但是请注意，在通过分区表插入元组时不会应用分区的默认值。

插入分区表中的行将自动路由到正确的分区。如果不存在合适的分区，则会发生错误。

像TRUNCATE这样的操作通常会影响一个表及其所有继承子级，这些操作将级联到所有分区，但也可能在单个分区上执行。请注意，使用DROP TABLE 删除分区需要在父表上采用ACCESS EXCLUSIVE锁。

LIKE *source_table* [*like_option* ...]

LIKE指定新表将从哪一个表自动地复制所有的列名、数据类型以及它们的非空约束。

和INHERITS不同，新表和原始表在创建完成之后是完全分离的。对原始表的更改将不会被应用到新表，并且不可能在原始表的扫描中包括新表的数据。

同样与INHERITS不同，用LIKE拷贝的列和约束不会和相似的命名列及约束融合。如果显式指定了相同的名称或者在另一个LIKE子句中指定了相同的名称，将会发出一个错误。

可选的*like_option*子句指定要复制的原始表的附加属性。 指定 INCLUDING 复制该属性，指定 EXCLUDING 忽略该属性。EXCLUDING为默认值。 如果对同一类型的对象指定了多个规范，则使用最后一个规范。可用的选项包括：

INCLUDING COMMENTS

复制的列、约束和索引的注释将被复制。默认行为是去除注释，从而导致新表中复制的列和约束没有注释。

INCLUDING CONSTRAINTS

CHECK约束将被复制。 列约束和表约束之间没有区别。非空约束始终复制到新表。

INCLUDING DEFAULTS

复制列定义的默认表达式将被复制。否则，不会复制默认表达式，从而导致新表中复制的列具有空默认值。 注意，复制调用数据库修改函数的默认值，例如nextval，可能在原始表和新表之间创建功能联系。

INCLUDING GENERATED

列定义的任何生成表达式将被复制。 默认情况下，新列将是常规基本列。

INCLUDING IDENTITY

已复制列定义的任何标识规范都将被复制。为新表的每个标识列创建一个新序列，与与旧表关联的序列分开。

INCLUDING INDEXES

原始表上的索引、PRIMARY KEY、 UNIQUE和 EXCLUDE约束将被建立在新表上。 根据默认规则选择新索引和约束的名称，而不考虑原始的命名。（此行为可避免新索引可能出现重复名称失败。）

INCLUDING STATISTICS

扩展统计信息将复制到新表。

INCLUDING STORAGE

已复制列定义的STORAGE设置将被复制。 默认行为是排除STORAGE设置，从而导致新表中已复制列具有类型规定的默认设置。

INCLUDING ALL

INCLUDING ALL 是选择所有可用的单独选项的缩写形式。 （它能被用于在INCLUDING ALL之后写单独的EXCLUDING子句，以选择部分指定选项之外的所有选项。）

LIKE子句也能被用来从视图、外部表或组合类型拷贝列定义。不适合的选项（例如来自视图的INCLUDING INDEXES）会被忽略。

CONSTRAINT *constraint_name*

一个列约束或表约束的可选名称。如果该约束被违背，约束名将会出现在错误消息中，这样类似列必须为正的约束名可以用来与客户端应用沟通有用的约束信息（指定包含空格的约束名时需要用到双引号）。如果没有指定约束名，系统将生成一个。

NOT NULL

该列不允许包含空值。

NULL

该列允许包含空值。这是默认情况。

这个子句只是提供与非标准 SQL 数据库的兼容。在新的应用中不推荐使用。

CHECK (*expression*) [NO INHERIT]

CHECK指定一个产生布尔结果的表达式，一个插入或更新操作要想成功，其中新的或被更新的行必须满足该表达式。计算出 TRUE 或 UNKNOWN 的表达式就会成功。只要任何一个插入或更新操作的行产生了 FALSE 结果，将报告一个错误异常并且插入或更新不会修改数据库。一个被作为列约束指定的检查约束只应该引用该列的值，而一个出现在表约束中的表达式可以引用多列。

当前，CHECK表达式不能包含子查询，也不能引用当前行的列之外的变量（参见第 2.4.1 节 “检查约束”。可以引用系统列tableoid，但不能引用其他系统列。

一个被标记为NO INHERIT的约束将不会传播到子表。

当一个表有多个CHECK约束时，检查完NOT NULL约束后，对于每一行会以它们名称的字母表顺序来进行检查（UXDB对于CHECK约束不遵从任何特定的引发顺序）。

DEFAULT *default_expr*

DEFAULT子句为出现在其定义中的列赋予一个默认数据。该值是可以使用变量的表达式（特别是，不允许用对其他列的交叉引用）。子查询也是不允许的。 默认值表达式的数据类型必须匹配列的数据类型。

默认值表达式将被用在任何没有为该列指定值的插入操作中。如果一列没有默认值，那么默认值为空值。

GENERATED ALWAYS AS (*generation_expr*) STORED

此子句将列创建为generated column。 列无法被写入，读取时将返回指定表达式的结果。

关键字STORED表示将在写入时计算列并将存储在磁盘上。

生成表达式可以引用表中的其他列，但不能引用其他生成的列。使用的任何函数和运算符都必须是不可改变的。不允许引用其他表。

GENERATED { ALWAYS | BY DEFAULT } AS IDENTITY [(*sequence_options*)]

该子句将列创建为标识列。 它将拥有一个隐式序列附加到它，并且新行中的列将自动从分配给它的序列中获取值。

子句ALWAYS和BY DEFAULT确定在 INSERT语句中，序列值如何优先于用户指定的值。 如果指定了ALWAYS，则只有在INSERT 语句指定OVERRIDING SYSTEM VALUE时才接受用户指定的值。 如果指定了BY DEFAULT，则用户指定的值优先。 有关详细信息，请参见INSERT(7)。 （在COPY命令中，无论此设置如何，都始终使用用户指定的值。）

可选的*sequence_options*子句可用于覆盖序列的选项。 有关详细信息，请参见CREATE SEQUENCE(7)。

UNIQUE （列约束）
UNIQUE (*column_name* [, ...]) [INCLUDE (*column_name* [, ...])] (表约束)

UNIQUE约束指定一个表中的一列或多列组成的组包含唯一的值。唯一表约束的行为与列约束的行为相同，只是表约束能够跨越多列。

对于一个唯一约束的目的来说，空值不被认为是相等的。

每一个唯一表约束必须命名一个列的集合，并且它与该表上任何其他唯一或主键约束所命名的列集合都不相同（否则它将是一个被列举了两次的约束）。

在为多级分区层次结构建立唯一约束时， 目标分区表的分区键中的所有列，以及那些由它派生的所有分区表， 必须被包含在约束定义中。

添加唯一约束将自动在使用于约束的列或列组上创建唯一的 btree索引。 可选子句INCLUDE在不强制唯一性的情况下向该索引添加一个或多个列。 请注意虽然约束在包含的列上是非强制的，但是它仍然依赖于它们。 因此，这些列上的某些操作（例如DROP COLUMN）可能会导致级联约束和索引删除。

PRIMARY KEY （列约束）
PRIMARY KEY (*column_name* [, ...]) [INCLUDE (*column_name* [, ...])] (表约束)

PRIMARY KEY约束指定表的一个或者多个列只能包含唯一（不重复）、非空的值。一个表上只能指定一个主键，可以作为列约束或表约束。

主键约束所涉及的列集合应该不同于同一个表上定义的任何唯一约束的列集合（否则，该唯一约束是多余的并且会被丢弃）。

PRIMARY KEY强制的数据约束可以看成是UNIQUE和NOT NULL的组合， 不过把一组列标识为主键也为模式设计提供了元数据，因为主键标识其他表可以依赖这一个列集合作为行的唯一标识符。

PRIMARY KEY 约束共享UNIQUE 约束放到分区表上时限制。

添加PRIMARY KEY约束将自动在用于约束的列或列组上创建唯一的 btree 索引。 可选的INCLUDE 子句允许指定将被包含在索引的非-键部分中的列的列表。 虽然包含的列的唯一性是非强制的，但约束仍依赖于它们。 因此，这些包含的列上的某些操作（例如DROP COLUMN）可能会导致级联约束和索引删除。

EXCLUDE [USING *index_method*] (*exclude_element* WITH *operator* [, ...]) *index_parameters* [WHERE (*predicate*)]

EXCLUDE子句定一个排除约束，它保证如果任意两行在指定列或表达式上使用指定操作符进行比较，不是所有的比较都将会返回TRUE。如果所有指定的操作符都测试相等，这就等价于一个UNIQUE约束，尽管一个普通的唯一约束将更快。不过，排除约束能够指定比简单相等更通用的约束。例如，可以使用&&操作符指定一个约束，要求表中没有两行包含相互覆盖的圆（见 第 5.8 节 “几何类型”）。

排除约束使用一个索引实现，这样每一个指定的操作符必须与用于索引访问方法*index_method*的一个适当的操作符类（见第 8.10 节 “操作符类和操作符族”）相关联。操作符被要求是交换的。每一个*exclude_element*可以选择性地指定一个操作符类或者顺序选项，这些在CREATE INDEX(7)中有完整描述。

访问方法必须支持amgettuple，目前这意味着GIN无法使用。尽管允许，但是在一个排除约束中使用 B-树或哈希索引没有意义，因为它无法做得比一个普通唯一索引更出色。因此在实践中访问方法将总是GiST或SP-GiST。

*predicate*允许在该表的一个子集上指定一个排除约束。在内部这会创建一个部分索引。注意在为此周围的圆括号是必须的。

REFERENCES *reftable* [(*refcolumn*)] [MATCH *matchtype*] [ON DELETE *referential_action*] [ON UPDATE *referential_action*] （column constraint）
FOREIGN KEY (*column_name* [, ...]) REFERENCES *reftable* [(*refcolumn* [, ...])] [MATCH *matchtype*] [ON DELETE *referential_action*] [ON UPDATE *referential_action*] （表约束）

这些子句指定一个外键约束，它要求新表的一列或一个列的组必须只包含能匹配被引用表的某个行在被引用列上的值。 如果*refcolumn*列表被忽略，将使用*reftable*的主键。 被引用列必须是被引用表中一个非可延迟唯一约束或主键约束的列。 用户必须在被引用的表（或整个表, 或特定的引用列）上拥有REFERENCES权限。 增加的外键约束需要SHARE ROW EXCLUSIVE 锁定引用的表。 注意外键约束不能在临时表和永久表之间定义。

被插入到引用列的一个值会使用给定的匹配类型与被引用表的值进行匹配。 有三种匹配类型：MATCH FULL、MATCH PARTIAL以及MATCH SIMPLE（这是默认值）。 MATCH FULL将不允许一个多列外键中的一列为空，除非所有外键列都是空；如果它们都是空，则不要求该行在被引用表中有一个匹配。 MATCH SIMPLE允许任意外键列为空，如果任一为空，则不要求该行在被引用表中有一个匹配。 MATCH PARTIAL现在还没有被实现（当然，NOT NULL约束能被应用在引用列上来组织这些情况发生）。

另外，当被引用列中的数据被改变时，在这个表的列中的数据上可以执行特定的动作。ON DELETE指定当被引用表中一个被引用行被删除时要执行的动作。同样，ON UPDATE指定当被引用表中一个被引用列被更新为新值时要执行的动作。如果该行被更新，但是被引用列并没有被实际改变，不会做任何动作。除了NO ACTION检查之外的引用动作不能被延迟，即便该约束被声明为可延迟的。对每一个子句可能有以下动作：

NO ACTION

产生一个错误指示删除或更新将会导致一个外键约束违背。如果该约束被延迟，并且仍存在引用行，这个错误将在约束检查时被产生。这是默认动作。

RESTRICT

产生一个错误指示删除或更新将会导致一个外键约束违背。这个动作与NO ACTION形同，不过该检查不是可延迟的。

CASCADE

删除任何引用被删除行的行，或者把引用列的值更新为被引用列的新值。

SET NULL

将引用列设置为空。

SET DEFAULT

设置引用列为它们的默认值（如果该默认值非空，在被引用表中必须有一行匹配该默认值，否则该操作将会失败）。

如果被引用列被频繁地更改，最好在引用列上加上一个索引，这样与外键约束相关的引用动作能够更高效地被执行。

DEFERRABLE
NOT DEFERRABLE

这个子句控制该约束是否能被延迟。一个不可延迟的约束将在每一次命令后立刻被检查。可延迟约束的检查将被推迟到事务结束时进行（使用SET CONSTRAINTS(7)命令）。NOT DEFERRABLE是默认值。当前，只有UNIQUE、PRIMARY KEY、EXCLUDE以及REFERENCES（外键）约束接受这个子句。NOT NULL以及CHECK约束是不可延迟的。注意在包括ON CONFLICT DO UPDATE子句的INSERT语句中，可延迟约束不能被用作冲突裁判者。

INITIALLY IMMEDIATE
INITIALLY DEFERRED

如果一个约束是可延迟的，这个子句指定检查该约束的默认时间。如果该约束是INITIALLY IMMEDIATE，它会在每一个语句之后被检查。这是默认值。如果该约束是INITIALLY DEFERRED，它只会在事务结束时被检查。约束检查时间可以用SET CONSTRAINTS(7)命令修改。

USING *method*

此可选子句指定用于存储新表内容的表访问方法；该方法需要的是类型TABLE的访问方法。如果未指定此选项，则为新表选择默认表访问方法。

WITH (*storage_parameter* [= *value*] [, ...])

这个子句为一个表或索引指定可选的存储参数，详见“存储参数”一节 。 为了向后兼容性，表的WITH子句还可以包括OIDS=FALSE以便指定新表的行不应包含 OIDs （对象标识符），OIDS=TRUE不再受支持。

WITHOUT OIDS

这是向后兼容的语法，用于声明表WITHOUT OIDS，不再支持创建表WITH OIDS。

ON COMMIT

临时表在一个事务块结束时的行为由ON COMMIT控制。三种选项是：

PRESERVE ROWS

在事务结束时不采取特殊的动作。这是默认行为。

DELETE ROWS

在每一个事务块结束时将删除临时表中的所有行。 实质上，在每一次提交时会完成一次自动的TRUNCATE(7)。 当应用于分区表上时，这不会级联到它的分区。

DROP

在当前事务块结束时将删除临时表。 当在分区表上使用时，这个操作会删除他的分区，而在具有继承子级的表上使用时，它将删除依赖的子级。

TABLESPACE *tablespace_name*

*tablespace_name*是新表要创建于其中的表空间名称。如果没有指定，将参考default_tablespace，或者如果表是临时的则参考temp_tablespaces。 对于分区表，由于表本身不需要存储，指定表空间将 default_tablespace作为默认表空间覆盖，在未显式指定其他表空间时用于任何新创建的分区。

STORAGE

STORAGE语法可以指定创建表的表空间，表空间必须已存在，此语法缺省时为该用户的默认表空间。

CLUSTERBTR

当 INI 参数 LIST_TABLE = 1 时，指定 CLUSTERBTR，则建立的表为普通 B 树表而非堆表；该参数保留。

USING INDEX TABLESPACE *tablespace_name*

这个子句允许选择与一个UNIQUE、PRIMARY KEY或者EXCLUDE约束相关的索引将被创建在哪个表空间中。如果没有指定，将参考default_tablespace，或者如果表是临时的则参考temp_tablespaces。

存储参数

WITH子句能够为表或与一个UNIQUE、PRIMARY KEY或者EXCLUDE约束相关的索引指定存储参数。 用于索引的存储参数已经在CREATE INDEX(7)中介绍过。 当前可用于表的存储参数在下文中列出。 如下文所示，对于很多这类参数，都有一个名字带有toast.前缀的附加参数，它能被用来控制该表的二级TOAST表（如果存在）的行为。 如果一个表的参数值被设置但是相应的toast.参数没有被设置，那么 TOAST 表将使用该表的参数值。 不支持为分区表指定这些参数，但可以为单个叶子分区指定它们。

fillfactor (integer)

一个表的填充因子是一个 10 到 100 之间的百分数。100（完全填满）是默认值。当一个较小的填充因子被指定时，INSERT操作会把表页面只填满到指定的百分比，每个页面上剩余的空间被保留给该页上行的更新。这就让UPDATE有机会把一行的已更新版本放在与其原始版本相同的页面上，这比把它放在一个不同的页面上效率更高。对于一个项从来不会被更新的表来说，完全填满是最好的选择，但是在更新繁重的表上则较小的填充因子更合适。这个参数不能对 TOAST 表设置。

toast_tuple_target (integer)

在我们尝试将长列值移动到TOAST表中之前，toast_tuple_target指定需要的最小元组长度， 也是在toasting开始时尝试减少长度的目标长度。 这仅影响标记为"外部"或"扩展"的列，并且仅适用于新元数 - 对现有行没有影响。 默认情况下此参数设置为允许每个块至少 4 个元组，默认块大小为 2040 字节。 有效值介于 128 字节和(块大小-标头)之间，默认大小为 8160 字节。 更改此值对于非常短或非常长的行可能没有用处。 请注意默认设置通常接近最佳状态，在某些情况下设置此参数可能会产生负面影响。 不能对TOAST表设置此参数。

parallel_workers (integer)

这个参数设置用于辅助并行扫描这个表的工作者数量。 如果没有设置这个参数，系统将基于关系的尺寸来决定一个值。 规划者或使用并行扫描的实用程序选择的工作者数量可能会比较少，例如max_worker_processes的设置较小就是一种可能的原因。

autovacuum_enabled, toast.autovacuum_enabled (boolean)

为一个特定的表启用或者禁用自动清理守护进程。如果为真，自动清理守护进程将在这个表上执行自动的VACUUM或者ANALYZE操作。如果为假，这个表不会被自动清理，不过为了阻止事务 ID 回卷时还是会对它进行自动的清理。如果autovacuum参数为假，自动清理守护进程

根本就不会运行（除非为了阻止事务 ID 回卷），设置独立的表存储参数也不会覆盖这个设置。因此显式地将这个存储参数设置为true很少有大的意义，只有设置为false才更有用。

vacuum_index_cleanup，toast.vacuum_index_cleanup（boolean）

当**VACUUM**在此表上运行时启用或禁用索引清理。默认值为true。 禁用索引清理可以显著加快**VACUUM**，但如果表修改很频繁，也可能导致索引严重膨胀。
VACUUM(7)的INDEX_CLEANUP参数，如果指定，将覆盖此选项的值。

vacuum_truncate，toast.vacuum_truncate（boolean）

启用或禁用vacuum以尝试截断此表末尾的任何空页。默认值为true。 如果true，**VACUUM**和autovacuum将执行截断，截断页的磁盘空间将返回到操作系统。 请注意，截断需要ACCESS EXCLUSIVE在表上锁定。VACUUM(7)的TRUNCATE参数，如果指定，覆盖此选项的值。

autovacuum_vacuum_threshold，toast.autovacuum_vacuum_threshold（integer）

autovacuum_vacuum_threshold参数对于每个表的值。

autovacuum_vacuum_scale_factor，toast.autovacuum_vacuum_scale_factor（float4）

autovacuum_vacuum_scale_factor参数对于每个表的值。

autovacuum_analyze_threshold（integer）

autovacuum_analyze_threshold参数对于每个表的值。

autovacuum_analyze_scale_factor（float4）

autovacuum_analyze_scale_factor参数对于每个表的值。

autovacuum_vacuum_cost_delay，toast.autovacuum_vacuum_cost_delay（floating point）

autovacuum_vacuum_cost_delay参数对于每个表的值。

autovacuum_vacuum_cost_limit，toast.autovacuum_vacuum_cost_limit（integer）

autovacuum_vacuum_cost_limit参数对于每个表的值。

autovacuum_freeze_min_age，toast.autovacuum_freeze_min_age（integer）

vacuum_freeze_min_age参数对于每个表的值。注意自动清理将忽略超过系统范围autovacuum_freeze_max_age参数一半的针对每个表的autovacuum_freeze_min_age参数。

autovacuum_freeze_max_age，toast.autovacuum_freeze_max_age（integer）

autovacuum_freeze_max_age参数对于每个表的值。注意自动清理将忽略超过系统范围参数（只能被设置得较小）一半的针对每个表的autovacuum_freeze_max_age参数。

autovacuum_freeze_table_age，toast.autovacuum_freeze_table_age（integer）

vacuum_freeze_table_age参数对于每个表的值。

autovacuum_multixact_freeze_min_age，toast.autovacuum_multixact_freeze_min_age（integer）

vacuum_multixact_freeze_min_age参数对于每个表的值。注意自动清理将忽略超过系统范围autovacuum_multixact_freeze_max_age参数一半的针对每个表的autovacuum_multixact_freeze_min_age参数。

autovacuum_multixact_freeze_max_age，toast.autovacuum_multixact_freeze_max_age（integer）

autovacuum_multixact_freeze_max_age参数对于每个表的值。注意自动清理将忽略超过系统范围参数（只能被设置得较小）一半的针对每个表的autovacuum_multixact_freeze_max_age参数。

autovacuum_multixact_freeze_table_age，toast.autovacuum_multixact_freeze_table_age（integer）

vacuum_multixact_freeze_table_age参数对于每个表的值。

log_autovacuum_min_duration，toast.log_autovacuum_min_duration（integer）

log_autovacuum_min_duration参数对于每个表的值。

user_catalog_table（boolean）

声明该表是一个用于逻辑复制目的的额外的目录表。不能对 TOAST 表设置这个参数。

注解

UXDB为每一个唯一约束和主键约束创建一个索引来强制唯一性。因此，没有必要显式地为主键列创建一个索引（详见CREATE INDEX(7)）。

在当前的实现中，唯一约束和主键不会被继承。这使得继承和唯一约束的组合相当不正常。

一个表不能有超过 1600 列（实际上，由于元组长度限制，有效的限制通常更低）。

例子

创建表films和表distributors：

```
CREATE TABLE films (
    code        char(5) CONSTRAINT firstkey PRIMARY KEY,
    title       varchar(40) NOT NULL,
    did         integer NOT NULL,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute
);

CREATE TABLE distributors (
     did    integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
     name   varchar(40) NOT NULL CHECK (name <> '')
);
```

创建有一个二维数组的表：

```
CREATE TABLE array_int (
    vector  int[][]
);
```

为表films定义一个唯一表约束。唯一表约束能够被定义在表的一列或多列上：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT production UNIQUE(date_prod)
);
```

定义一个列检查约束：

```
CREATE TABLE distributors (
    did     integer CHECK (did > 100),
    name    varchar(40)
);
```

定义一个表检查约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    CONSTRAINT con1 CHECK (did > 100 AND name <> '')
);
```

为表films定义一个主键表约束：

```
CREATE TABLE films (
    code        char(5),
    title       varchar(40),
    did         integer,
    date_prod   date,
    kind        varchar(10),
    len         interval hour to minute,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

为表distributors定义一个主键约束。下面的两个例子是等价的，第一个使用表约束语法，第二个使用列约束语法：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    PRIMARY KEY(did)
);

CREATE TABLE distributors (
    did     integer PRIMARY KEY,
    name    varchar(40)
);
```

为列name赋予一个文字常量默认值，安排列did的默认值是从一个序列对象中选择下一个值产生，并且让modtime的默认值是该行被插入的时间：

```
CREATE TABLE distributors (
    name      varchar(40) DEFAULT 'Luso Films',
    did       integer DEFAULT nextval('distributors_serial'),
    modtime   timestamp DEFAULT current_timestamp
);
```

在表distributors上定义两个NOT NULL列约束，其中之一被显式给定了一个名称：

```
CREATE TABLE distributors (
    did     integer CONSTRAINT no_null NOT NULL,
    name    varchar(40) NOT NULL
);
```

为name列定义一个唯一约束：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40) UNIQUE
);
```

同样的唯一约束用表约束指定：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name)
);
```

创建相同的表，指定表和它的唯一索引指定 70% 的填充因子：

```
CREATE TABLE distributors (
    did     integer,
    name    varchar(40),
    UNIQUE(name) WITH (fillfactor=70)
)
WITH (fillfactor=70);
```

创建表circles，带有一个排除约束阻止任意两个圆重叠：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

在表空间diskvol1中创建表cinemas：

```
CREATE TABLE cinemas (
        id serial,
        name text,
        location text
) TABLESPACE diskvol1;
```

创建一个组合类型以及一个类型化的表：

```
CREATE TYPE employee_type AS (name text, salary numeric);

CREATE TABLE employees OF employee_type (
    PRIMARY KEY (name),
    salary WITH OPTIONS DEFAULT 1000
);
```

创建一个范围分区表：

```
CREATE TABLE measurement (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (logdate);
```

创建在分区键中具有多个列的范围分区表：

```
CREATE TABLE measurement_year_month (
    logdate         date not null,
    peaktemp        int,
    unitsales       int
) PARTITION BY RANGE (EXTRACT(YEAR FROM logdate), EXTRACT(MONTH FROM
 logdate));
```

创建列表分区表：

```
CREATE TABLE cities (
    city_id      bigserial not null,
    name         text not null,
    population   bigint
) PARTITION BY LIST (left(lower(name), 1));
```

建立哈希分区表:

```
CREATE TABLE orders (
    order_id     bigint not null,
    cust_id      bigint not null,
    status       text
) PARTITION BY HASH (order_id);
```

创建范围分区表的分区：

```
CREATE TABLE measurement_y2016m07
   PARTITION OF measurement (
   unitsales DEFAULT 0
) FOR VALUES FROM ('2016-07-01') TO ('2016-08-01');
```

使用分区键中的多个列创建范围分区表的几个分区：

```
CREATE TABLE measurement_ym_older
   PARTITION OF measurement_year_month
   FOR VALUES FROM (MINVALUE, MINVALUE) TO (2016, 11);

CREATE TABLE measurement_ym_y2016m11
   PARTITION OF measurement_year_month
   FOR VALUES FROM (2016, 11) TO (2016, 12);

CREATE TABLE measurement_ym_y2016m12
   PARTITION OF measurement_year_month
   FOR VALUES FROM (2016, 12) TO (2017, 01);

CREATE TABLE measurement_ym_y2017m01
   PARTITION OF measurement_year_month
   FOR VALUES FROM (2017, 01) TO (2017, 02);
```

创建列表分区表的分区：

```
CREATE TABLE cities_ab
   PARTITION OF cities (
   CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b');
```

创建本身是分区的列表分区表的分区，然后向其添加分区：

```
CREATE TABLE cities_ab
   PARTITION OF cities (
   CONSTRAINT city_id_nonzero CHECK (city_id != 0)
) FOR VALUES IN ('a', 'b') PARTITION BY RANGE (population);

CREATE TABLE cities_ab_10000_to_100000
   PARTITION OF cities_ab FOR VALUES FROM (10000) TO (100000);
```

建立哈希分区表的分区：

```
CREATE TABLE orders_p1 PARTITION OF orders
   FOR VALUES WITH (MODULUS 4, REMAINDER 0);
CREATE TABLE orders_p2 PARTITION OF orders
   FOR VALUES WITH (MODULUS 4, REMAINDER 1);
CREATE TABLE orders_p3 PARTITION OF orders
   FOR VALUES WITH (MODULUS 4, REMAINDER 2);
CREATE TABLE orders_p4 PARTITION OF orders
   FOR VALUES WITH (MODULUS 4, REMAINDER 3);
```

建立默认分区：

```
CREATE TABLE cities_partdef
   PARTITION OF cities DEFAULT;
```

建立非聚簇索引的表，如下所示。

```
create table test(id int,name varchar(20),constraint pk_test not cluster primary key (id));

create table cluster_table_on_1(id int not cluster primary key,name varchar(20));
```

默认不加not cluster 语法所建立的同样是表示非聚簇索引的表，如下所示。

```
create table test2(id int,name varchar(20),constraint pk_test2  primary key (id));

create table have_cluster_table_on_1(id int primary key,name varchar(20));
```

storage语法使用，如下所示。

```
create table t_storage(id int) storage(on tblspc,clusterbtr);
 \d+ t_storage;
```

ON UPDATE NOW()使用，如下所示。

```
create table test(id int, ts timestamp on update now());
```

兼容性

CREATE TABLE命令遵从SQL标准，除了以下例外。

临时表

尽管CREATE TEMPORARY TABLE的语法很像 SQL 标准的语法，但事实是并不相同。在标准中，临时表只需要被定义一次并且会自动地存在（从空内容开始）于需要它们的每一个会话中。UXDB则要求每一个会话为每一个要用的临时表发出它自己的CREATE TEMPORARY TABLE命令。这允许不同的会话为不同的目的使用相同的临时表名，而标准的方法约束一个给定临时表名的所有实例都必须具有相同的表结构。

标准中对于临时表行为的定义被广泛地忽略了。UXDB在这一点上的行为和多种其他 SQL 数据库是相似的。

SQL 标准也区分全局和局部临时表，其中一个局部临时表为每一个会话中的每一个 SQL 模块具有一个独立的内容集合，但是它的定义仍然是多个会话共享的。因为UXDB不支持 SQL 模块，这种区别与UXDB无关。

为了兼容性目的，UXDB将在临时表声明中接受GLOBAL和LOCAL关键词，但是它们当前没有效果。我们不鼓励使用这些关键词，因为未来版本的UXDB可能采用一种更兼容标准的（对它们含义的）解释。

临时表的ON COMMIT子句也和 SQL 标准相似，但是有一些不同。如果忽略ON COMMIT子句，SQL 指定默认行为是ON COMMIT DELETE ROWS。但是，UXDB中的默认行为是ON COMMIT PRESERVE ROWS。SQL 中不存在ON COMMIT DROP选项。

非延迟唯一性约束

但一个UNIQUE或PRIMARY KEY约束是非可延迟的，只要一个行被插入或修改，UXDB就会立即检查唯一性。SQL 标准指出只有在语句结束时才应该强制唯一性。当一个单一命令更新多个键值时，这两者是不同的。要得到兼容标准的行为，将该约束声明为DEFERRABLE但是不延迟（即INITIALLY IMMEDIATE）。注意这可能要显著地慢于立即唯一性检查。

列检查约束

SQL 标准指出CHECK列约束只能引用它们应用到的列，只有CHECK表约束能够引用多列。UXDB并没有强制这个限制，它同样处理列检查约束和表检查约束。

EXCLUDE 约束

EXCLUDE约束类型是一种UXDB扩展。

NULL “约束”

NULL “约束”（实际上是一个非约束）是一个UXDB对 SQL 标准的扩展，它也被包括（以及对称的NOT NULL约束）在一些其他的数据库系统中以实现兼容性。因为它是任意列的默认值，它的存在就像噪声一样。

Constraint Naming

SQL标准规定在包含表或域的模式范围内表和域的约束必须具有唯一的名称。 UXDB是比较宽松的：它只需要约束名称在附加到特定表或域的约束之间是唯一的。 但是，对于基于索引的约束(UNIQUE, PRIMARY KEY, and EXCLUDEconstraints)， 这个特别的自由度并不存在， 因为关联的索引被命名为与约束相同的名称，并且索引名称在相同模式的所有关系中必须是唯一的。

当前，UXDB没有记录NOT NULL约束的名称， 因此它们不受唯一性限制的影响。这在将来的版本中可能会改变。

继承

通过INHERITS子句的多继承是一种UXDB的语言扩展。SQL:1999 以及之后的标准使用一种不同的语法和不同的语义定义了单继承。SQL:1999-风格的继承还没有被UXDB。

零列表

UXDB允许创建一个没有列的表（例如CREATE TABLE foo();）。这是一个对于 SQL 标准的扩展，它不允许零列表。零列表本身并不是很有用，但是不允许它们会为ALTER TABLE DROP COLUMN带来奇怪的特殊情况，因此忽略这种规则限制看起来更加整洁。

多个标识列

UXDB允许一个表拥有多个标识列。 该标准指定一个表最多只能有一个标识列。这主要是为了给模式更改或迁移提供更大的灵活性。 请注意，INSERT命令仅支持一个适用于整个语句的覆盖子句， 因此不支持具有不同行为的多个标识列。

Generated Columns

STORED选项不是标准，但也用于其他 SQL 实现。 SQL 标准不规定生成列的存储。

LIKE 子句

虽然 SQL 标准中有一个LIKE子句，但是UXDB接受的很多LIKE子句选项却不在标准中，并且有些标准中的选项也没有被UXDB实现。

WITH子句

WITH子句是一个UXDB扩展，存储参数不在标准中。

表空间

UXDB的表空间概念不是标准的一部分。因此，子句TABLESPACE和USING INDEX TABLESPACE是扩展。

类型化的表

类型化的表实现了 SQL 标准的一个子集。根据标准，一个类型化的表具有与底层组合类型相对应的列，以及其他的“自引用列”。UXDB不显式支持自引用列。

PARTITION BY 子句

PARTITION BY子句是UXDB的一个扩展。

PARTITION OF 子句

PARTITION OF子句UXDB的一个扩展。

参见

ALTER TABLE(7), DROP TABLE(7), CREATE TABLE AS(7), CREATE TABLESPACE(7), CREATE TYPE(7)

名称

CREATE TABLE AS — 从一个查询的结果创建一个新表

大纲

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT
 EXISTS ] table_name
    [ (column_name [, ...] ) ]
    [ USING method ]
    [ WITH ( storage_parameter [= value] [, ... ] ) | WITHOUT OIDS ]
    [ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
    [ TABLESPACE tablespace_name ]
    AS query
    [ WITH [ NO ] DATA ]
```

描述

CREATE TABLE AS创建一个表，并且用 由一个**SELECT**命令计算出来的数据填充 该表。该表的列具有和**SELECT**的输出列 相关的名称和数据类型（不过可以通过给出一个显式的新列名列表来覆 盖这些列名）。

CREATE TABLE AS和创建一个视图有些 相似，但是实际上非常不同：它会创建一个新表并且只计算该查询一次 用来初始填充新表。这个新表将不会跟踪该查询源表的后续变化。相反， 一个视图只要被查询，它的定义**SELECT** 语句就会被重新计算。

参数

GLOBAL或者LOCAL

为兼容性而忽略。不推荐使用这些关键词，详见 CREATE TABLE(7)。

TEMPORARY或者TEMP

如果被指定，该表会被创建为一个临时表。详见 CREATE TABLE(7)。

UNLOGGED

如果被指定，该表会被创建为一个不做日志的表。详见 CREATE TABLE(7)。

IF NOT EXISTS

如果已经存在一个同名的关系时不要抛出错误。这种情况下会发出一个 提示。详见CREATE TABLE(7)。

table_name

要创建的表的名称（可以被模式限定）。

column_name

新表中一列的名称。如果没有提供列名，会从查询的输出列名中得到。

USING *method*

这个可选的子句指定了用于存储新表内容的表访问方法；该方法需要是一个类型 TABLE的访问方法。如果没有指定这个选项，则选择新表的默认表访问方法。

WITH (*storage_parameter* [= *value*] [, ...])

这个子句为新表指定可选的存储参数，详见 “存储参数”一节。 为了向后兼容，表的WITH子句也能包含OIDS=FALSE来指定新表的行将不包含OID（对象标识符）。OIDS=TRUE不再支持。

WITHOUT OIDS

这是向后兼容的语法，用于声明表WITHOUT OIDS，创建表WITH OIDS不再被支持。

ON COMMIT

临时表在事务块结束时的行为可以用ON COMMIT 控制。三个选项是：

PRESERVE ROWS

在事务结束时不采取特殊的动作。这是默认行为。

DELETE ROWS

在每一个事务块结束时临时表中的所有行都将被删除。本质上， 在每次提交时会完成一次自动的TRUNCATE(7)。

DROP

在当前事务块结束时将删掉临时表。

TABLESPACE *tablespace_name*

tablespace_name 是要在其中创建新表的表空间名称。如果没有指定，将会查询 default_tablespace，临时表会查询 temp_tablespaces。

query

一个SELECT(7)、TABLE或者VALUES(7) 命令，或者是一个运行准备好的SELECT、 TABLE或者VALUES查询的EXECUTE(7)命令。

WITH [NO] DATA

这个子句指定查询产生的数据是否应该被复制到新表中。如果不是，只有 表结构会被复制。默认是复制数据。

注解

这个命令在功能上类似于SELECT INTO(7)，但是它更好， 因为不太可能被SELECT INTO语法的其他使用混淆。更 进一步，CREATE TABLE AS提供了 SELECT INTO的功能的一个超集。

示例

创建一个新表films_recent，它只由表 films中最近的项组成：

```
CREATE TABLE films_recent AS
  SELECT * FROM films WHERE date_prod >= '2002-01-01';
```

要完全地复制一个表，也可以使用TABLE命令的 简短形式：

```
CREATE TABLE films2 AS
  TABLE films;
```

用一个预备语句创建一个新的临时表films_recent， 它仅由表films中最近的项组成，使用准备好的声明。新表将在提交时被丢弃：

```
PREPARE recentfilms(date) AS
  SELECT * FROM films WHERE date_prod > $1;
CREATE TEMP TABLE films_recent ON COMMIT DROP AS
  EXECUTE recentfilms('2002-01-01');
```

兼容性

CREATE TABLE AS符合 SQL标准。下面的是非标准扩展：

- 标准要求在子查询子句周围有圆括号，在 UXDB中这些圆括号是可选的。
- 在标准中，WITH [NO] DATA子句是必要的，而 UXDB 中是可选的。
- UXDB处理临时表的方式和标准不同。 详见CREATE TABLE(7)。
- WITH子句是一种 UXDB扩展， 标准中没有存储参数。
- UXDB的表空间概念是标准的 一部分。因此，子句TABLESPACE是一种扩展。

另见

CREATE MATERIALIZED VIEW(7), CREATE TABLE(7), EXECUTE(7), SELECT(7), SELECT INTO(7), VALUES(7)

名称

CREATE TABLESPACE — 定义一个新的表空间

大纲

```
CREATE TABLESPACE tablespace_name
  [ OWNER { new_owner | CURRENT_USER | SESSION_USER } ]
  LOCATION 'directory'
  [ WITH ( tablespace_option = value [, ... ] ) ]
```

描述

CREATE TABLESPACE注册一个新的集簇范围 的表空间。表空间的名称必须与数据库集簇中现有的任何表空间不同。

表空间允许超级用户在文件系统上定义另一个位置，可以把包含数据库对象 （例如表和索引）的数据文件放在那里。

一个具有适当特权的用户可以把 *tablespace_name*传递给 CREATE DATABASE、CREATE TABLE、CREATE INDEX或者ADD CONSTRAINT 来让这些对象的数据文件存储在指定的表空间中。

> **警告**
>
> 表空间不能独立于定义它的集簇使用。

参数

tablespace_name

The name of a tablespace to be created. The name cannot begin with ux_, as such names are reserved for system tablespaces.

user_name

将拥有该表空间的用户名。如果省略，默认为执行该命令的用户。只有 超级用户能创建表空间，但是它们能把表空间的拥有权赋予给非超级 用户。

directory

要被用于表空间的目录。该目录必需存在（CREATE TABLESPACE 将不创建它），应该为空，并且必须由 UXDB系统用户拥有。该目录必须用一个绝对 路径指定。

tablespace_option

要设置或者重置的表空间参数。当前，唯一可用的参数是 seq_page_cost、random_page_cost以及effective_io_concurrency。 为一个特定表空间设定其中一个值将覆盖规划器对该表空间中表页读取的 常规代价估计，常规代价估计是由同名的配置参数所建立。如果一个表空间位于一个 比其他 I/O 子系统更慢或者更快的磁盘上，这些参数就能发挥作用。

注解

只有在支持符号链接的系统上才支持表空间。

CREATE TABLESPACE不能在一个事务块内被执行。

示例

要在文件系统位置/data/dbs创建表空间dbspace，请首先使用操作系统工具创建目录并设置正确的所有权：

```
mkdir /data/dbs
chown uxdb:uxdb /data/dbs
```

然后在内部发出表空间创建命令 UXDB:

```
CREATE TABLESPACE dbspace LOCATION '/data/dbs';
```

要创建由不同数据库用户拥有的表空间，可用类似这样的命令：

```
CREATE TABLESPACE indexspace OWNER genevieve LOCATION '/data/indexes';
```

兼容性

CREATE TABLESPACE是一种 UXDB扩展。

另见

CREATE DATABASE(7), CREATE TABLE(7), CREATE INDEX(7), DROP TABLESPACE(7), ALTER TABLESPACE(7)

名称

CREATE TEXT SEARCH CONFIGURATION — 定义一个新的文本搜索配置

大纲

```
CREATE TEXT SEARCH CONFIGURATION name (
    PARSER = parser_name |
    COPY = source_config
)
```

描述

CREATE TEXT SEARCH CONFIGURATION 创建一个新的文本搜索配置。一个文本搜索配置指定一个文本搜索解析器（ 它能把字符串解析成记号），外加一些词典（可被用来决定哪些记号是搜索 感兴趣的）。

如果只指定了解析器，那么新文本搜索配置最初没有从记号类型到词典的映射， 并且因此将忽略所有词。后续的ALTER TEXT SEARCH CONFIGURATION命令必须被用来创建映射以让该配置变得可用。 另一种方式是复制一个现有的文本搜索配置。

如果给定一个模式名称，则文本搜索配置会被创建在指定的模式中。否则它将会 被创建在当前模式中。

定义一个文本搜索配置的用户会成为其拥有者。

进一步的信息请参考第 9 章 全文搜索

参数

name

要创建的文本搜索配置的名称。该名称可以被模式限定。

parser_name

这个配置要使用的文本搜索解析器的名称。

source_config

要复制的已有文本搜索配置的名称。

注解

PARSER和COPY选项是互斥的，因为当 一个已有的配置被复制时，它的解析器选择也会被复制。

兼容性

在 SQL 标准中没有 CREATE TEXT SEARCH CONFIGURATION语句。

另见

ALTER TEXT SEARCH CONFIGURATION(7), DROP TEXT SEARCH CONFIGURATION(7)

名称

CREATE TEXT SEARCH DICTIONARY — 定义一个新的文本搜索字典

大纲

```
CREATE TEXT SEARCH DICTIONARY name (
    TEMPLATE = template
    [, option = value [, ... ]]
)
```

描述

CREATE TEXT SEARCH DICTIONARY创建一个 新的文本搜索字典。一个文本搜索字典指定一种识别搜索感兴趣或者不感兴趣 的单词的方法。 一个字典依赖于一个文本搜索模板，后者指定了实际执行该工 作的函数。通常该字典提供一些控制该模板函数细节行为的选项。

如果给出了一个模式名称，那么该文本搜索字典会被创建在指定的模式中。 否则它会被创建在当前模式中。

定义文本搜索字典的用户将成为其拥有者。

进一步的信息可参考第 9 章 全文搜索

参数

name

要创建的文本搜索字典的名称。该名称可以是模式限定的。

template

将定义这个字典基本行为的文本搜索模板的名称。

option

要为此字典设置的模板相关选项的名称。

value

用于模板相关选项的值。如果该值不是一个简单标识符或者数字，它必须 被加引用（可以按照所希望的总是对它加上引用）。

选项可以以任意顺序出现。

示例

下面的例子命令创建了一个基于 Snowball 的字典，它使用了非标准的 停用词列表。

```
CREATE TEXT SEARCH DICTIONARY my_russian (
    template = snowball,
    language = russian,
```

```
    stopwords = myrussian
);
```

兼容性

在 SQL 标准中没有 **CREATE TEXT SEARCH DICTIONARY**语句。

另见

ALTER TEXT SEARCH DICTIONARY(7), DROP TEXT SEARCH DICTIONARY(7)

名称

CREATE TEXT SEARCH PARSER — 定义一个新的文本搜索解析器

大纲

```
CREATE TEXT SEARCH PARSER name (
    START = start_function ,
    GETTOKEN = gettoken_function ,
    END = end_function ,
    LEXTYPES = lextypes_function
    [, HEADLINE = headline_function ]
)
```

描述

CREATE TEXT SEARCH PARSER创建一个 新的文本搜索解析器。一个文本搜索解析器定义把文本字符串分解成记号 并且为记号分配类型（分类）的方法。一个解析器本身并不特别有用，但 是必须与一些要用于搜索的文本搜索字典一起被绑定到一个文本搜索配置 中。

如果给出了一个模式名称，那么文本搜索解析器将被创建在指定的模式中。 否则它会被创建在当前模式中。

只有超级用户才能使用 **CREATE TEXT SEARCH PARSER**（这是因为 错误的文本搜索定义可能会让服务器混淆甚至崩溃）。

更多信息可以参考第 9 章 全文搜索

参数

name

要创建的文本搜索解析器的名称。该名称可以是模式限定的。

start_function

用于该解析器的开始函数的名称。

gettoken_function

用于该解析器的取下一个记号的函数名称。

end_function

用于该解析器的结束函数的名称。

lextypes_function

用于该解析器的词法分析器函数（一个返回其产生的记号类型集合信息的函数）的名称。

headline_function

用于该解析器的标题函数（一个对记号集合进行综述的函数）的名称。

如有必要，函数的名称可以被模式限定。参数类型没有给出， 因为函数的每个类型的参数列表无法被预先决定。除了标题函数之外， 所有函数都是必要的。

参数可以以任何顺序出现，而不是必须按照上面所展示的顺序。

兼容性

在 SQL 标准中没有 **CREATE TEXT SEARCH PARSER**语句。

另见

ALTER TEXT SEARCH PARSER(7), DROP TEXT SEARCH PARSER(7)

名称

CREATE TEXT SEARCH TEMPLATE — 定义一种新的文本搜索模板

大纲

```
CREATE TEXT SEARCH TEMPLATE name (
    [ INIT = init_function , ]
    LEXIZE = lexize_function
)
```

描述

CREATE TEXT SEARCH TEMPLATE创建一个 新的文本搜索模板。文本搜索模板定义实现文本搜索字典的函数。一个模板 本身没什么用处，但是必须被实例化为一个字典来使用。字典通常指定要给 予模板函数的参数。

如果给出了一个模式名，文本搜索模板会被创建在指定模式中。否则它会被 创建在当前模式中。

必须成为超级用户以使用 **CREATE TEXT SEARCH TEMPLATE**。做出 这种限制是因为错误的模板定义会让服务器混淆甚至崩溃。将模板与字典 分隔开来的原因是模板中封装了定义字典的“不安全”方面。在 定义字典时可以被设置的参数对非特权用户是可以安全设置的，因此创建 字典不需要拥有特权来操作。

进一步的信息可以参考第 9 章 全文搜索

参数

name

要创建的额文本搜索模板的名称。该名称可以被模式限定。

init_function

用于模板的初始化函数的名称。

lexize_function

用于模板的分词函数的名称。

如有必要，函数名称可以被模式限定。参数类型没有给出，因为每一类 函数的参数列表是预先定义好的。分词函数是必需的，但是初始化函数 是可选的。

参数可以以任何顺序出现，而不是只能按照上文的顺序。

兼容性

在 SQL 标准中没有 **CREATE TEXT SEARCH TEMPLATE**语句。

另见

ALTER TEXT SEARCH TEMPLATE(7), DROP TEXT SEARCH TEMPLATE(7)

名称

CREATE TRANSFORM — 定义一个新的转换

大纲

```
CREATE [ OR REPLACE ] TRANSFORM FOR type_name LANGUAGE lang_name (
  FROM SQL WITH FUNCTION from_sql_function_name [ (argument_type [, ...]) ],
  TO SQL WITH FUNCTION to_sql_function_name [ (argument_type [, ...]) ]
);
```

简介

CREATE TRANSFORM定义一种新的转换。 **CREATE OR REPLACE TRANSFORM**将 创建一种新的转换或者替换现有的定义。

一种转换指定了如何把一种数据类型适配到一种过程语言。例如，在用 PL/Python 编写一个使用hstore类型的函数时，PL/Python 没有关于如何在 Python 环境中表示hstore值的先验知识。语言的实现通常默认会使用文本表示，但是在一些时候这很不方便，例如 有时可能用一个联合数组或者列表更合适。

一种转换指定了两个函数：

- 一个“from SQL”函数负责将类型从 SQL 环境转换到语言。 这个函数将在该语言编写的一个函数的参数上调用。
- 一个“to SQL”函数负责将类型从语言转换到 SQL 环境。这 个函数将在该语言编写的一个函数的返回值上调用。

没有必要同时提供这些函数。如果有一种没有被指定，将在必要时使用与语言相 关的默认行为（为了完全阻止在一个方向上发生转换，也可以写一个总是报错 的转换函数）。

要创建一种转换，必须拥有该类型并且具有该类型上的 USAGE特权，拥有该语言上的USAGE特权，并且拥有 from-SQL 和 to-SQL 函数（如果 指定了）及其上的EXECUTE特权。

参数

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

from_sql_function_name[(*argument_type* [, ...])]

将该类型从 SQL 环境转换到该语言的函数名。它必须接受一个 internal类型的参数并且返回类型internal。 实参将是该转换所适用的类型，并且该函数也应该被写成认为它是那种类型（ 但是不允许声明一个返回internal但没有至少一个 internal类型参数的 SQL 层函数）。实际的返回值将与 语言的实现相关。如果没有指定参数列表，则函数名在该模式中必须唯一。

to_sql_function_name[(*argument_type* [, ...])]

将该类型从语言转换到 SQL 环境的函数名。它必须接受一个 internal类型的参数并且返回该转换所适用的类型。实参值　将与语言的实现相关。如果没有指定参数列表，则函数名在该模式中必须唯一。

注解

使用DROP TRANSFORM(7)移除转换。

示例

要为类型hstore和语言 plpythonu创建一种转换，先搞定该类型和语言：

```
CREATE TYPE hstore ...;

CREATE EXTENSION plpythonu;
```

然后创建需要的函数：

```
CREATE FUNCTION hstore_to_plpython(val internal) RETURNS internal
LANGUAGE C STRICT IMMUTABLE
AS ...;

CREATE FUNCTION plpython_to_hstore(val internal) RETURNS hstore
LANGUAGE C STRICT IMMUTABLE
AS ...;
```

最后创建转换把它们连接起来：

```
CREATE TRANSFORM FOR hstore LANGUAGE plpythonu (
   FROM SQL WITH FUNCTION hstore_to_plpython(internal),
   TO SQL WITH FUNCTION plpython_to_hstore(internal)
);
```

实际上，这些命令将被包裹在扩展中。

contrib小节包含了一些提供转换的扩展， 它们可以作为实际的例子。

兼容性

这种形式的**CREATE TRANSFORM**是一种 UXDB扩展。在 SQL标准中有一个**CREATE TRANSFORM**命令，但是它是用于把数据类型适配到 客户端语言。该用法不受 UXDB支持。

另见

CREATE FUNCTION(7), CREATE LANGUAGE(7), CREATE TYPE(7), DROP TRANSFORM(7)

名称

CREATE TRIGGER — 定义一个新触发器

大纲

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event
[ OR ... ] }
  ON table_name
  [ FROM referenced_table_name ]
  [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY
DEFERRED ] ]
  [ REFERENCING { { OLD | NEW } TABLE [ AS ] transition_relation_name } [ ... ] ]
  [ FOR [ EACH ] { ROW | STATEMENT } ]
  [ WHEN ( condition ) ]
  EXECUTE { FUNCTION | PROCEDURE } function_name ( arguments )
```

这里的*event*可以是下列之一：

```
  INSERT
  UPDATE [ OF column_name [, ... ] ]
  DELETE
  TRUNCATE
```

描述

CREATE TRIGGER创建一个新触发器。该触发器将被关联到指定的表、视图或者外部表并且在表上发生特定操作时将执行指定的函数*function_name*。

该触发器可以被指定为在一行上尝试该操作之前触发（在约束被检查并且INSERT、UPDATE或者DELETE被尝试之前）；也可以在该操作完成之后触发（在约束被检查并且INSERT、UPDATE或者DELETE完成之后）；或者取代该操作（在对一个视图插入、更新或删除的情况中）。如果该触发器在事件之前触发或者取代事件，该触发器可以跳过对当前行的操作或者改变正在被插入的行（只对INSERT以及UPDATE操作）。如果该触发器在事件之后触发，所有更改（包括其他触发器的效果）对该触发器“可见”。

一个被标记为FOR EACH ROW的触发器会对该操作修改的每一行都调用一次。例如，一个影响 10 行的DELETE将导致在目标关系上的任何ON DELETE触发器被独立调用 10 次，也就是为每一个被删除的行调用一次。与此相反，一个被标记为FOR EACH STATEMENT的触发器只会为任何给定的操作执行一次，不管该操作修改多少行（特别地，一个修改零行的操作将仍会导致任何可用的FOR EACH STATEMENT触发器被执行）。

被指定为要触发INSTEAD OF触发器事件的触发器必须被标记为FOR EACH ROW，并且只能被定义在视图上。一个视图上的BEFORE和AFTER触发器必须被标记为FOR EACH STATEMENT。

此外，触发器可以被定义成为TRUNCATE触发，但只能是FOR EACH STATEMENT。

下面的表格总结了哪些触发器类型可以被用在表、视图和外部表上：

何时	事件	行级	语句级
BEFORE	INSERT/UPDATE/DELETE	表和外部表	表、视图和外部表

何时	事件	行级	语句级
	TRUNCATE	—	表
AFTER	INSERT/UPDATE/DELETE	表和外部表	表、视图和外部表
	TRUNCATE	—	表
INSTEAD OF	INSERT/UPDATE/DELETE	视图	—
	TRUNCATE	—	—

还有，一个触发器定义可以指定一个布尔的WHEN条件，它将被测试来看看该触发器是否应该被触发。在行级触发器中，WHEN条件可以检查该行的列的新旧值。语句级触发器也可以有WHEN条件，尽管该特性对于它们不是很有用（因为条件不能引用表中的任何值）。

如果有多个同种触发器被定义为相同事件触发，它们将按照名称的字母表顺序被触发。

当CONSTRAINT选项被指定，这个命令会创建一个*约束触发器*。这和一个常规触发器相同，不过触发该触发器的时机可以使用SET CONSTRAINTS(7)调整。约束触发器必须是表上的AFTER ROW触发器。它们可以在导致触发器事件的语句末尾被引发或者在包含该语句的事务末尾被引发。在后一种情况中，它们被称作是被*延迟*。一个待处理的延迟触发器的引发也可以使用SET CONSTRAINTS立即强制发生。当约束触发器实现的约束被违背时，约束触发器应该抛出一个异常。

REFERENCING选项启用对*传递关系*的收集，传递关系是包括被当前SQL语句插入、删除或者修改的行的行集合。这个特性让触发器能看到该语句做的事情的全局视图，而不是一次只看到一行。仅对非约束触发器的AFTER触发器允许这个选项。此外，如果触发器是一个UPDATE触发器，则它不能指定*column_name*列表。OLD TABLE仅可以被指定一次，并且只能为在UPDATE或DELETE事件上引发的触发器指定，它创建的传递关系包含有该语句更新或删除的所有行的*前映像*。类似地，NEW TABLE仅可以被指定一次，并且只能为在UPDATE或INSERT事件上引发的触发器指定，它创建的传递关系包含有该语句更新或插入的所有行的*后映像*。

SELECT不修改任何行，因此无法创建SELECT触发器。规则和视图可以为需要SELECT触发器的问题提供可行的解决方案。

参数

name

给新触发器的名称。这必须与同一个表上的任何其他触发器相区别。名称不能是模式限定的 — 该触发器会继承它所在表的模式。对于一个约束触发器，这也是使用SET CONSTRAINTS修改触发器行为时要用到的名字。

BEFORE
AFTER
INSTEAD OF

决定该函数是要在事件之前、之后被调用还是会取代该事件。一个约束触发器也能被指定为AFTER。

event

INSERT、UPDATE、DELETE或者TRUNCATE之一，这指定了将要引发该触发器的事件。多个事件可以用OR指定，要求传递关系的时候除外。

对于UPDATE事件，可以使用下面的语法指定一个列的列表：

UPDATE OF *column_name1* [, *column_name2* ...]

只有当至少一个被列出的列出现在UPDATE命令的更新目标中时，或者如果列出的列之一是生成的列，而且依赖的列是UPDATE的目标，该触发器才会触发。

INSTEAD OF UPDATE事件不允许列的列表。在请求传递关系时，也不能指定列的列表。

table_name

要使用该触发器的表、视图或外部表的名称（可能是模式限定的）。

referenced_table_name

约束引用的另一个表的名称（可能是模式限定的）。这个选项被用于外键约束并且不推荐用于一般的目的。这只能为约束触发器指定。

DEFERRABLE
NOT DEFERRABLE
INITIALLY IMMEDIATE
INITIALLY DEFERRED

该触发器的默认时机。这些约束选项的细节可参考CREATE TABLE(7)文档。这只能为约束触发器指定。

REFERENCING

这个关键词紧接在一个或者两个关系名的声明之前，这些关系提供对触发语句的传递关系的访问。

OLD TABLE
NEW TABLE

这个子句指示接下来的关系名是用于前映像传递关系还是后映像传递关系。

transition_relation_name

在该触发器中这个传递关系要使用的（未限定）名称。

FOR EACH ROW
FOR EACH STATEMENT

这指定该触发器函数是应该为该触发器事件影响的每一行被引发一次，还是只为每个 SQL 语句被引发一次。如果都没有被指定，FOR EACH STATEMENT会是默认值。约束触发器只能被指定为FOR EACH ROW。

condition

一个决定该触发器函数是否将被实际执行的布尔表达式。如果指定了WHEN，只有*condition*返回true时才会调用该函数。在FOR EACH ROW触发器中，WHEN条件可以分别写OLD.*column_name*或者NEW.*column_name*来引用列的新旧行值。当然，INSERT触发器不能引用OLD并且DELETE触发器不能引用NEW。

INSTEAD OF触发器不支持WHEN条件。

当前，WHEN表达式不能包含子查询。

注意对于约束触发器，对于WHEN条件的计算不会被延迟，而是直接在行更新操作被执行之后立刻发生。如果该条件计算得不到真，那么该触发器就不会被放在延迟执行的队列中。

function_name

一个用户提供的函数，它被声明为不用参数并且返回类型trigger，当触发器引发时会执行该函数。

在CREATE TRIGGER的语法中，关键词FUNCTION和PROCEDURE是等效的，但是任何情况下被引用的函数必须是一个函数而不是过程。这里，关键词PROCEDURE的使用是有历史原因的并且已经被废弃。

arguments

一个可选的逗号分隔的参数列表，它在该触发器被执行时会被提供给该函数。参数是字符串常量。简单的名称和数字常量也可以被写在这里，但是它们将全部被转换成字符串。请检查该触发器函数的实现语言的描述来找出在函数内部如何访问这些参数，这可能与普通函数参数不同。

注解

要在一个表上创建一个触发器，用户必须具有该表上的TRIGGER特权。用户还必须具有在触发器函数上的EXECUTE特权。

使用DROP TRIGGER(7)移除一个触发器。

当一个列相关的触发器（使用UPDATE OF *column_name*语法定义的触发器）的列被列为UPDATE命令的SET列表目标时，它会被触发。即便该触发器没有被引发，一个列的值也可能改变，因为BEFORE UPDATE触发器对行内容所作的改变不会被考虑。相反，一个诸如UPDATE ... SET x = x ...的命令将引发一个位于列x上的触发器，即便该列的值没有改变。

在一个BEFORE触发器中，WHEN条件正好在函数被或者将被执行之前被计算，因此使用WHEN与在触发器函数的开始测试同一个条件没有实质上的区别。特别注意该条件看到的NEW行是当前值，虽然可能已被早前的触发器所修改。还有，一个BEFORE触发器的WHEN条件不允许检查NEW行的系统列（例如ctid），因为那些列还没有被设置。

在一个AFTER触发器中，WHEN条件正好在行更新发生之后被计算，并且它决定一个事件是否要被放入队列以便在语句的末尾引发该触发器。因此当一个AFTER触发器的WHEN条件不返回真时，没有必要把一个事件放入队列或者在语句末尾重新取得该行。如果触发器只需要为一些行被引发，就能够显著地加快修改很多行的语句的速度。

在一些情况下，单一的SQL命令可能会引发多种触发器。例如，一个带有ON CONFLICT DO UPDATE子句的INSERT可能同时导致插入和更新操作，因此它将根据需要引发这两种触发器。提供给触发器的传递关系与它们的事件类型有关，因此INSERT触发器将只看到被插入的行，而UPDATE触发器将只看到被更新的行。

由外键强制动作导致的行更新或删除（例如ON UPDATE CASCADE或ON DELETE SET NULL）被当做导致它们的SQL命令的一部分。受影响的表上的相关触发器将被引发，这样就提供了另一种方法让SQL命令引发不直接匹配其类型的触发器。在简单的情况中，请求传递关系的触发器将在一个传递关系中看到由原始SQL命令在其表中做出的所有改变。不过，有些情况中一个请求传递关系的AFTER ROW触发器的存在将导致由单个SQL命令触发的外键强制动作被分成多步，每一步都有其自己的传递关系。在这种情况下，没创建一个传递关系集合都会引发存在的所有语句级触发器，确保那些触发器能够在一个传递关系中看到每个受影响的行一次，并且只看到一次。

只有当视图上的动作被一个行级INSTEAD OF触发器处理时才会引发视图上的语句级触发器。如果动作被一个INSTEAD规则处理，那么该语句发出的任何语句都会代替提及该视图的原始语句执行，这样将被引发的触发器是替换语句中提及的表上的那些触发器。类似地，如果视图是自动可更新的，则该动作将被处理为把该语句自动重写成在视图基表上的一个动作，这样基表的语句级触发器就是要被引发的。

在分区表上创建一个行级触发器将导致在它所有的现有分区上创建相同的触发器，并且以后创建或者挂接的任何分区也将包含一个相同的触发器。分区表上的触发器只能是AFTER。

修改分区表或者带有继承子表的表会引发挂接到显式提及表的语句级触发器，但不会引发其分区或子表的语句级触发器。相反，行级触发器会在受影响的分区或子表上引发，即便它们在查询中没有被明确提及。如果一个语句级触发器用REFERENCING子句定义有传递关系，则来自所有受影响分区或子表中的行的前后映像都是可见的。在继承子表的情况中，行映像仅包括该触发器所附属的表中存在的列。当前，不能在分区或继承子表上定义带有传递关系的行级触发器。

必须要声明触发器函数为返回占位符类型opaque而不是trigger。要支持载入旧的转储文件，CREATE TRIGGER将接受一个被声明为返回opaque的函数，但是它会发出一个通知并且会把该函数的声明返回类型改为trigger。

例子

只要表accounts的一行即将要被更新时会执行函数check_account_update：

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

下面的例子与上面一个例子相同，但是只在UPDATE命令指定要更新balance列时才执行该函数：

```
CREATE TRIGGER check_update
  BEFORE UPDATE OF balance ON accounts
  FOR EACH ROW
  EXECUTE FUNCTION check_account_update();
```

这种形式只有列balance具有真正被改变的值时才执行该函数：

```
CREATE TRIGGER check_update
  BEFORE UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.balance IS DISTINCT FROM NEW.balance)
  EXECUTE FUNCTION check_account_update();
```

调用一个函数来记录accounts的更新，但是只在有东西被改变时才调用：

```
CREATE TRIGGER log_update
  AFTER UPDATE ON accounts
  FOR EACH ROW
  WHEN (OLD.* IS DISTINCT FROM NEW.*)
  EXECUTE FUNCTION log_account_update();
```

为每一个要插入到视图底层表中的行执行函数view_insert_row：

```
CREATE TRIGGER view_insert
   INSTEAD OF INSERT ON my_view
   FOR EACH ROW
   EXECUTE FUNCTION view_insert_row();
```

为每个语句执行函数check_transfer_balances_to_zero以确认transfer的行不会有净值增加：

```
CREATE TRIGGER transfer_insert
   AFTER INSERT ON transfer
   REFERENCING NEW TABLE AS inserted
   FOR EACH STATEMENT
   EXECUTE FUNCTION check_transfer_balances_to_zero();
```

为每一行执行函数check_matching_pairs以确认（同一个语句）同时对匹配对做了更改 ：

```
CREATE TRIGGER paired_items_update
   AFTER UPDATE ON paired_items
   REFERENCING NEW TABLE AS newtab OLD TABLE AS oldtab
   FOR EACH ROW
   EXECUTE FUNCTION check_matching_pairs();
```

兼容性

UXDB中的**CREATE TRIGGER**语句实现了SQL标准的一个子集。目前缺少下列功能：

- 虽然AFTER触发器的传递表名是以标准的方式用REFERENCING子句指定，但REFERENCING子句中不能指定FOR EACH ROW触发器中用到的行变量。它们以依赖于编写该触发器函数的语言的方式可用，但是对任意一种语言来说是固定的。一些语言实际上的行为就像有包含OLD ROW AS OLD NEW ROW AS NEW的REFERENCING子句存在一样。
- 标准允许把传递表与和列相关的UPDATE触发器一起使用，那么应该在传递表中可见的行集合取决于该触发器的列列表。当前UXDB没有实现这一点。
- UXDB只允许为被触发动作执行一个用户定义的函数。标准允许执行许多其他的 SQL 命令作为被触发的动作，例如**CREATE TABLE**。这种限制可以很容易地通过创建一个执行想要的命令的用户定义函数来绕过。

SQL 指定多个触发器应该以被创建时间的顺序触发。UXDB则使用名称顺序，这被认为更加方便。

SQL 指定级联删除上的BEFORE DELETE触发器在级联的DELETE完成之后引发。UXDB的行为则是BEFORE DELETE总是在删除动作之前引发，即使是一个级联删除。这被认为更加一致。 如果BEFORE触发器修改行或者在引用动作引起的更新期间阻止更新，这也是非标准行为。这能导致约束违背或者被存储的数据不遵从引用约束。

使用OR为一个单一触发器指定多个动作的能力是 SQL 标准的一个UXDB扩展。

为**TRUNCATE**引发触发器的能力是 SQL 标准的一个UXDB扩展，在视图上定义语句级触发器的能力也是一样。

CREATE CONSTRAINT TRIGGER是SQL标准的一个UXDB扩展。

参见

ALTER TRIGGER(7), DROP TRIGGER(7), CREATE FUNCTION(7), SET CONSTRAINTS(7)

名称

CREATE TYPE — 定义一种新的数据类型

大纲

```
CREATE TYPE name AS
  ( [ attribute_name data_type [ COLLATE collation ] [, ... ] ] )

CREATE TYPE name AS ENUM
  ( [ 'label' [, ... ] ] )

CREATE TYPE name AS RANGE (
  SUBTYPE = subtype
  [ , SUBTYPE_OPCLASS = subtype_operator_class ]
  [ , COLLATION = collation ]
  [ , CANONICAL = canonical_function ]
  [ , SUBTYPE_DIFF = subtype_diff_function ]
)

CREATE TYPE name (
  INPUT = input_function,
  OUTPUT = output_function
  [ , RECEIVE = receive_function ]
  [ , SEND = send_function ]
  [ , TYPMOD_IN = type_modifier_input_function ]
  [ , TYPMOD_OUT = type_modifier_output_function ]
  [ , ANALYZE = analyze_function ]
  [ , INTERNALLENGTH = { internallength | VARIABLE } ]
  [ , PASSEDBYVALUE ]
  [ , ALIGNMENT = alignment ]
  [ , STORAGE = storage ]
  [ , LIKE = like_type ]
  [ , CATEGORY = category ]
  [ , PREFERRED = preferred ]
  [ , DEFAULT = default ]
  [ , ELEMENT = element ]
  [ , DELIMITER = delimiter ]
  [ , COLLATABLE = collatable ]
)

CREATE TYPE name
```

描述

CREATE TYPE在当前数据库中注册一种新的 数据类型。定义数据类型的用户将成为它的拥有者。

如果给定一个模式名，那么该类型将被创建在指定的模式中。否则它会被 创建在当前模式中。类型名称必须与同一个模式中任何现有的类型或者域 相区别（因为表具有相关的数据类型，类型名称也必须与同一个模式中任 何现有表的名字不同）。

如上面的语法所示，有五种形式的CREATE TYPE。 它们分别创建组合类型、枚举类型、 范围类型、基础类型或者 shell 类型。下文将依次讨论前四种形式。shell 类型仅仅 是一种用于后面要定义的类型的占位符，通过发出一个不带除类型名之外其 他参数的CREATE TYPE命令可以创建这种类型。 在创建范围类型和基础类型时，需要 shell 类型作为一种向前引用。

组合类型

第一种形式的CREATE TYPE创建组合类型。 组合类型由一个属性名和数据类型的列表指定。如果属性的数据类型是可 排序的，也可以指定该属性的排序规则。组合类型本质上和表的行类型相同，但是如果只想定义一种类型，使用 CREATE TYPE避免了创建一个实际的表。单 独的组合类型也是很有用的，例如可以作为函数的参数或者返回类型。

为了能够创建组合类型，必须拥有在其所有属性类型上的 USAGE特权。

枚举类型

如第 5.7 节 “枚举类型”中所述，第二种形式的 CREATE TYPE创建枚举类型。枚举类型需要 一个带引号的标签构成的列表，每一个标签长度必须不超过 NAMEDATALEN字节（在标准的 UXDB编译中是 64 字节）。 可以创建具有零个标签的枚举类型，但是在使用ALTER TYPE(7) 添加至少一个标签之前，不能使用这种类型来保存值。

范围类型

如第 5.17 节 “范围类型”中所述，第三种形式的 CREATE TYPE创建范围类型。

范围类型的*subtype*可以 是任何带有一个相关的 B 树操作符类（用来决定该范围类型值的顺序）的类型。 通常，子类型的默认 B 树操作符类被用来决定顺序。要使用一种非默认操作符 类，可以用 *subtype_opclass*指定它 的名字。如果子类型是可排序的并且希望在该范围的顺序中使用一种非默认的 排序规则，可以用*collation*选项来指定。

可选的*canonical*函数 必须接受一个所定义的范围类型的参数，并且返回同样类型的一个值。在适 用时，它被用来把范围值转换成一种规范的形式。更多信息请见第 5.17.8 节 “定义新的范围类型”。创建一个 *canonical*函数有点棘 手，因为必须在声明范围类型之前定义它。要这样做，必须首先创建一种 shell 类型，它是一种没有属性只有名称和拥有者的占位符类型。这可以通过 发出不带额外参数的命令CREATE TYPE *name*来完成。然后可以使用该 shell 类型作为 参数和结果来声明该函数，并且最终用同样的名称来声明范围类型。这会自动 用一种合法的范围类型替换 shell 类型项。

可选的*subtype_diff*函数 必须接受两个*subtype*类型 的值作为参数，并且返回一个double precision值表示两个给定 值之间的差别。虽然这是可选的，但是提供这个函数会让该范围类型列上 GiST 索 引效率更高。详见第 5.17.8 节 “定义新的范围类型”

基础类型

第四种形式的CREATE TYPE创建一种新的 基本类型（标量类型）。为了创建一种新的基本类型，必须是一个超级 用户（做这种限制的原因是一种错误的类型定义可能让服务器混淆甚至 崩溃）。

参数可以以任意顺序出现（而不仅是按照上面所示的顺序），并且大部分 是可选的。在定义类型前，必须注册两个或者更多函数（使用 CREATE FUNCTION）。支持函数 *input_function*以及 *output_function* 是必需的，而函数 *receive_function*、 *send_function*、*type_modifier_input_function*、 *type_modifier_output_function*和 *analyze_function* 是可选的。通常来说这些函数必须是用 C 或者另外一种低层语言编写的。

*input_function*将 类型的外部文本表达转换成为该类型定义的操作符和函数所使用的内部 表达。*output_function* 执行反向的转换。输入函数可以被声明为有一个cstring 类型的参数，或者有三个类型分别为cstring、 oid、integer的参数。第一个参数是 以 C 字符串存在的输入文本，第二个参数是该类型自身的 OID（对于 数组类型则是其元素类型的 OID），第三个参数是目标列的 typmod（如果知道，不知道则将传递 -1）。输入函数必须 返回一个该数据类型本身的值。通常，一个输入函数应该被声明为 STRICT。 如果不是这样，在读到一个 NULL 输入值时，调用它时第一个参数会是 NULL。在这种情况下，该函数必须仍然返回 NULL，除非它发生了错误（ 这种情况主要是想支持域输入函数，它们可能需要拒绝 NULL 输入）。 输出函数必须被声明为有一个新数据类型的参数。输出函数必须返回类型 cstring。对于 NULL 值不会调用输出函数。

T可选的*receive_function* 会把类型的外部二进制表达转换成内部表达。如果没有提供这个函数，该类型不能参与到二进制输入中。二进制表达转换成内部形式代价更低， 然而却更容易移植（例如，标准的整数数据类型使用网络字节序作为外 部二进制表达，而内部表达是机器本地的字节序）。接收函数应该执行 足够的检查以确保该值是有效的。接收函数可以被声明为有一个internal类型的参数，或者有三个类型分别为 internal、oid、integer 的参数。第一个参数是一个指向StringInfo缓冲区的 指针，其中保存着接收到的字节串。其余可选的参数和文本输入函数的 相同。接收函数必须返回一个该数据类型本身的值。通常，一个接收函 数应该被声明为 STRICT。如果不是这样，在读到一个 NULL 输入值时， 调用它时第一个参数会是 NULL。在这种情况下，该函数必须仍然返回 NULL，除非它发生了错误（这种情况主要是想支持域接收函数，它们 可能需要拒绝 NULL 输入）。类似地，可选的 *send_function*将 内部表达转换成外部二进制表达。如果没有提供这个函数，该类型将不 能参与到二进制输出中。发送函数必须被声明为有一个新数据类型的参 数。发送函数必须返回类型bytea。对于 NULL 值不 会调用发送函数。

到这里应该在疑惑输入和输出函数是如何能被声明为具有新类型的 结果或参数的？因为必须在创建新类型之前创建这两个函数。这个问题 的答案是，新类型应该首先被定义为一种**shell type**， 它是一种占位符类型，除了名称和拥有者之外它没有其他属性。这可以 通过不带额外参数的命令CREATE TYPE *name*做到。然后用 C 写的 I/O 函数可以 被定义为引用这种 shell type。最后，用带有完整定义的 **CREATE TYPE**把该 shell type 替换为一个完全的、合 法的类型定义，之后新类型就可以正常使用了。

如果该类型支持修饰符（附加在类型声明上的可选约束，例如 char(5)或者numeric(30,2)），则需要可选的 *type_modifier_input_function* 以及*type_modifier_output_function*。 UXDB允许用户定义的类型有一个或者 多个简单常量或者标识符作为修饰符。不过，为了存储在系统目录中，该信息必须 能被打包到一个非负整数值中。所声明的修饰符会被以cstring数组的形式 传递给 *type_modifier_input_function*。 它必须检查该值的合法性（如果值错误就抛出一个错误），如果值正确，要返回 一个非负integer值，它将被存储在“typmod”列中。如果 类型没有 *type_modifier_input_function* 则类型修饰符将被拒绝。 *type_modifier_output_function* 把内部的整数 typmod 值转换回正确的形式用于用户显示。它必须返回一个 cstring值，该值就是追加到类型名称后的字符串。例如 numeric的函数可能会返回(30,2)。如果默认的显示格式 就是只把存储的 typmod 整数值放在圆括号内，则允许省略 *type_modifier_output_function*。

可选的*analyze_function* 为该数据类型的列执行与类型相关的统计信息收集。默认情况下，如果该类型有一个默认的 B-树操作符类，**ANALYZE**将尝试用 类型的“equals”和“less-than”操作符来收集统计信息。 这种行为对于非标量类型并不合适，因此可以通过指定一个自定义分析函数来 覆盖这种行为。分析函数必须被声明为有一个类型为internal的参 数，并且返回一个boolean结果。分析函数的详细 API 请见 src/include/commands/vacuum.h。

虽然只有 I/O 函数和其他为该类型创建的函数才知道新类型的内部表达的细节， 但是内部表达的一些属性必须被向 UXDB声明。其中最重要的是 *internallength*。基本数据 类型可以是定长的（这种情况下 *internallength*是一个正 整数）或者是变长的（把 *internallength*设置为VARIABLE，在内部通过把typlen设置为 -1 表示）。 所有变长类型的内部表达都必须以一个 4 字节整数开始，它给出了这个值的总 长度（注意长度域常常是被编码 过的，直接接受它是不明智的）。

可选的标志PASSEDBYVALUE表示这种数据类型的值需要 被传值而不是传引用。传值的类型必须是定长的，并且它们的内部表达不能超 过Datum类型（某些机器上是 4 字节，其他机器上是 8 字节）的 尺寸。

*alignment*参数指定数据 类型的存储对齐要求。允许的值等同于以 1、2、4 或 8 字节边界对齐。注意 变长类型的 alignment 参数必须至少为 4，因为它们需要包含一个 int4作为它们的第一个组成部分。

*storage*参数允许 为变长数据类型选择存储策略（对定长类型只允许 plain）。plain指定该类型的数 据将总是被存储在线内并且不会被压缩。extended 指定系统将首先尝试压缩一个长的数据值，并且将在数据仍然太长的情 况下把值移出主表行。external允许值被移出主表， 但是系统将不会尝试对它进行压缩。main允许压缩， 但是不鼓励把值移出主表（如果没有其他办法让行的大小变得合适，具有 这种存储策略的数据项仍将被移出主表，但比起 extended以及external项来， 这种存储策略的数据项会被优先考虑保留在主表中）。

除plain之外所有的 *storage*值都暗示 该数据类型的函数能处理被TOAST 过的值。指定的值 仅仅是决定一种可 TOAST 数据类型的列的默认 TOAST 存储策略，用户 可以使用ALTER TABLE SET STORAGE为列选取其他策略。

*like_type*参数提供 了另一种方法来指定一种数据类型的基本表达属性：从某种现有的类型中 拷贝。*internallength*、 *passedbyvalue*、 *alignment*和 *storage*的值会从指 定的类型中复制而来（也可以通过在LIKE子句中指定这些属 性的值来覆盖复制过来的值，不过通常并不这么做）。当新类型的低层 实现是以一种现有的类型为“载体”时，用这种方式指定表达特别有用。

*category*和 *preferred*参数可以被用来 帮助控制在混淆的情况下应用哪一种隐式造型。每一种数据类型都属于一个用 单个 ASCII 字符命名的分类，并且每一种类型可以是其所属分类中的 “首选”。当有助于解决重载函数或操作符时，解析器将优先 造型到首选类型（但是只能从同类的其他类型造型）。更多细节请见 第 7 章 类型转换对于没有隐式造型到任意其他类型或者 从任意其他类型造型的类型，让这些设置保持默认即可。不过，对于一组 具有隐式造型的相关类型，把它们都标记为属于同一个类别并且选择一种 或两种“最常用”的类型作为该类别的首选通常是很有用的。在 把一种用户定义的类型增加到一个现有的内建类别（例如数字或者字符串类型）中时， *category*参数特别 有用。不过，也可以创建新的全部是用户定义类型的类别。对这样的类别， 可选择除大写字母之外的任何 ASCII 字符。

如果用户希望该数据类型的列被默认为某种非空值，可以指定一个默认值。 默认值可以用DEFAULT关键词指定（这样一个默认值 可以被附加到一个特定列的显式DEFAULT子句覆盖）。

要指定一种类型是数组，用ELEMENT关键词指定该数组元素 的类型。例如，要定义一个 4 字节整数的数组（int4）， 应指定ELEMENT = int4。更多有关数组类型的细节请 见下文。

要指定在这种类型数组的外部表达中分隔值的定界符，可以把*delimiter*设置为一个特定字符。默认 的定界符是逗号（,）。注意定界符是与数组元素类型相 关的，而不是数组类型本身相关。

如果可选的布尔参数 *collatable*为真，这种 类型的列定义和表达式可能通过使用COLLATE子句携带 有排序规则信息。在该类型上操作的函数的实现负责真正利用这些信息，仅 把类型标记为可排序的并不会让它们自动地去使用这类信息。

数组类型

只要一种用户定义的类型被创建， UXDB会自动地创建一种相关的 数组类型，其名称由元素类型的名称前面加上一个下划线组成，并且如果长 度超过NAMEDATALEN字节会自动地被截断（如果这样生成的名称与一种现有类型的名称冲突，该过程将会重复直到找到一个 不冲突的名字）。这种隐式创建的数组类型是变长的并且使用内建的输入和 输出函数（array_in以及array_out）。

该数组 类型会追随其元素类型的拥有者或所在模式的任何更改，并且在元素类型被 删除时也被删除。

如果系统会自动地创建正确的数组类型，可能会很合情合理地问为什么 会有一个ELEMENT选项。使用ELEMENT 唯一有用的情况是：当在创建一种定长类型，它正好在内部是一个多个 相同东西的数组，并且除了计划给该类型提供的整体操作之外，想要 允许用下标来直接访问这些东西。例如，类型point被表示 为两个浮点数，可以使用point[0]以及point[1] 来访问它们。注意，这种功能只适用于内部形式正好是一个相同定长域 序列的定长类型。可用下标访问的变长类型必须具有 array_in以及array_out使用的一般化的内部 表达。由于历史原因（即很明显是错的，但现在改已经太晚了），定长 数组类型的下标是从零开始的，而不是像变长数组那样。

参数

name

要创建的类型的名称（可以被模式限定）。

attribute_name

组合类型的一个属性（列）的名称。

data_type

要成为组合类型的一个列的现有数据类型的名称。

collation

要关联到组合类型的一列或者范围类型的现有排序规则的名称。

label

一个字符串，它表达与枚举类型的一个值相关的文本标签。

subtype

范围类型的元素类型的名称，范围类型表示的范围属于该类型。

subtype_operator_class

用于 subtype 的 B 树操作符类的名称。

canonical_function

范围类型的规范化函数的名称。

subtype_diff_function

用于 subtype 的差函数的名称。

input_function

将数据从类型的外部文本形式转换为内部形式的函数名。

output_function

将数据从类型的内部形式转换为外部文本形式的函数名。

receive_function

将数据从类型的外部二进制形式转换成内部形式的函数名。

send_function

将数据从类型的内部形式转换为外部二进制形式的函数名。

type_modifier_input_function

将类型的修饰符数组转换为内部形式的函数名。

type_modifier_output_function

将类型的修饰符的内部形式转换为外部文本形式的函数名。

analyze_function

为该数据类型执行统计分析的函数名。

internallength

一个数字常量，它指定新类型的内部表达的字节长度。默认的假设是 它是变长的。

alignment

该数据类型的存储对齐需求。如果被指定，它必须是 char、int2、 int4或者double。默认是int4。

storage

该数据类型的存储策略。如果被指定，必须是 plain、external、 extended或者main。 默认是plain。

like_type

与新类型具有相同表达的现有数据类型的名称。会从这个类型中复制 *internallength*、*passedbyvalue*、 *alignment*以及 *storage*的值（ 除非在这个CREATE TYPE命令的其他地方用显式说 明覆盖）。

category

这种类型的分类码（一个 ASCII 字符）。 默认是 “用户定义类型”的'U'。为了创建自定义分类， 也可以选择其他 ASCII 字符。

preferred

如果这种类型是其类型分类中的优先类型则为真，否则为假。默认为假。在一个现有类型分类中创建一种新的优先类型要注意，因为这可能会导致行为上的改变。

default

数据类型的默认值。如果被省略，默认值是空。

element

被创建的类型是一个数组，这指定了数组元素的类型。

delimiter

在由这种类型组成的数组中值之间的定界符。

collatable

如果这个类型的操作可以使用排序规则信息，则为真。默认为假。

注解

由于一旦数据类型被创建，对该数据类型的使用就没有限制，创建一种基本类型 或者范围类型就等同于在类型定义中提到的函数上授予公共执行权限。对于在类 型定义中有用的函数来说这通常不是问题。但是如果设计一种类型时要求在转换 到外部形式或者从外部形式转换时使用“秘密”信息，就应该三思而 后行。

自动生成的数组类型的名称总是正好为元素类型的名称外加一个前置的下划线字符（_）。因此类型名称的长度限制比其他名称还要少一个字符。虽然现在这仍然是通常情况，但如果名称达到最大长度或者与其他下划线开头的用户类型名称冲突，数组类型的名称也可以不同于这种规则。因此依靠这种习惯编写代码现在已经不适用了。现在，可以使用ux_type.*typarray*来定位与给定类型相关的数组类型。

建议避免使用以下划线开始的类型名和表名。虽然服务器会改变生成的数组类型名称以避免与用户给定的名称冲突，仍然有混淆的风险，特别是对旧的客户端软件来说，它们可能会假定以下划线开始的类型名总是表示数组。

shell-type 的创建语法CREATE TYPE *name*不存在。创建一种新基本类型的方法是先创建它的输入函数。在这种方法中，UXDB 将首先把新数据类型的名称看做是输入函数的返回类型。在这种情况下 shell type 会被隐式地创建，并且能在剩余的 I/O 函数的定义中引用。这种方法现在仍然有效，但是已经被弃用并且可能会在未来的某个发行中被禁止。还有，为了避免由于函数定义中的打字错误导致 shell type 弄乱系统目录，当输入函数用 C 编写时，将只能用这种方法创建一种 shell type。

为了完全避免创建 shell type 而把函数对该类型名的向前引用用占位符伪类型opaque替换，cstring参数和结果也必须被声明为opaque。为了支持载入旧的转储文件，CREATE TYPE将接受使用opaque声明的I/O 函数，但是它将发出一个提示并且把函数的声明改成使用正确的类型。

示例

这个例子创建了一种组合类型并且将其用在了一个函数定义中：

```
CREATE TYPE compfoo AS (f1 int, f2 text);

CREATE FUNCTION getfoo() RETURNS SETOF compfoo AS $$
   SELECT fooid, fooname FROM foo
$$ LANGUAGE SQL;
```

这个例子创建了一个枚举类型并且将其用在一个表定义中：

```
CREATE TYPE bug_status AS ENUM ('new', 'open', 'closed');

CREATE TABLE bug (
   id serial,
   description text,
```

```
    status bug_status
);
```

这个例子创建了一个范围类型：

```
CREATE TYPE float8_range AS RANGE (subtype = float8, subtype_diff = float8mi);
```

这个例子创建了基本数据类型box然后将它用在一个表定义中：

```
CREATE TYPE box;

CREATE FUNCTION my_box_in_function(cstring) RETURNS box AS ... ;
CREATE FUNCTION my_box_out_function(box) RETURNS cstring AS ... ;

CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function
);

CREATE TABLE myboxes (
    id integer,
    description box
);
```

如果box的内部结构是四个 float4元素的一个数组，我们可能会使用：

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

这将允许用下标来访问一个 box 值的组件编号。否则该类型的行为和 前面的一样。

这个例子创建了一个大对象类型并且将它用在了一个表定义中：

```
CREATE TYPE bigobj (
    INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE
);
CREATE TABLE big_objs (
    id integer,
    obj bigobj
);
```

兼容性

创建组合类型的第一种形式的**CREATE TYPE**命令 符合SQL标准。其他的形式都是 UXDB扩展。SQL标准中的**CREATE TYPE**语句也定义了其他 UXDB中没有实现的形式。

创建一种具有零个属性的组合类型的能力是一种 UXDB对标准的背离（类似于 **CREATE TABLE**中相同的情况）。

另见

ALTER TYPE(7), CREATE DOMAIN(7), CREATE FUNCTION(7), DROP TYPE(7)

名称

CREATE USER — 定义一个新的数据库角色

大纲

CREATE USER *name* [[WITH] *option* [...]]

这里 *option* 可以是：

```
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | BYPASSRLS | NOBYPASSRLS
  | CONNECTION LIMIT connlimit
  | [ ENCRYPTED ] PASSWORD 'password' | PASSWORD NULL
  | VALID UNTIL 'timestamp'
  | IN ROLE role_name [, ...]
  | IN GROUP role_name [, ...]
  | ROLE role_name [, ...]
  | ADMIN role_name [, ...]
  | USER role_name [, ...]
  | SYSID uid
```

描述

CREATE USER现在是 CREATE ROLE(7)的一个别名。唯一的区别是 **CREATE USER**中LOGIN 被作为默认值，而NOLOGIN是 **CREATE ROLE**的默认值。

兼容性

CREATE USER语句是一种 UXDB扩展。SQL 标准 把用户的定义留给实现来解释。

另见

CREATE ROLE(7)

名称

CREATE USER MAPPING — 定义一个用户到一个外部服务器的新映射

大纲

```
CREATE USER MAPPING [IF NOT EXISTS] FOR { user_name | USER | CURRENT_USER |
 PUBLIC }
   SERVER server_name
   [ OPTIONS ( option 'value' [ , ... ] ) ]
```

描述

CREATE USER MAPPING定义一个用户 到一个外部服务器的新映射。一个用户映射通常会包含连接信息，外部数据包装器 会使用连接信息和外部服务器中包含的信息一起来访问一个外部数据源。

一个外部服务器的拥有者可以为任何服务器任何用户创建用户映射。还有， 如果一个用户被授予了服务器上的USAGE特权，该用户可以 为他们自己的用户名创建用户映射。

参数

IF NOT EXISTS

如果给定用户到给定外部服务器的映射已经存在，则不要抛出错误。 在这种情况下发出通知。请注意，不能保证现有的用户映射与要创建的映射完全相同。

user_name

要映射到外部服务器的一个现有用户的名称。 CURRENT_USER和USER匹配当前用户的名称。 当PUBLIC被指定时，一个所谓的公共映射会被创建，当没有 特定用户的映射可用时将会使用它。

server_name

将为其创建用户映射的现有服务器的名称。

OPTIONS (*option* '*value*' [, ...])

这个子句指定用户映射的选项。这些选项通常定义该映射实际的用户名和 口令。选项名必须唯一。允许的选项名和值与该服务器的外部数据包装器 有关。

示例

为用户bob、服务器foo创建一个用户映射：

```
CREATE USER MAPPING FOR bob SERVER foo OPTIONS (user 'bob', password 'secret');
```

兼容性

CREATE USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。

另见

ALTER USER MAPPING(7), DROP USER MAPPING(7), CREATE FOREIGN DATA WRAPPER(7), CREATE SERVER(7)

名称

CREATE VIEW — 定义一个新视图

大纲

CREATE [OR REPLACE] [TEMP | TEMPORARY] [RECURSIVE] VIEW *name* [(*column_name* [, ...])]
  [WITH (*view_option_name* [= *view_option_value*] [, ...])]
  AS *query*
  [WITH [CASCADED | LOCAL] CHECK OPTION]

描述

CREATE VIEW定义一个查询的视图。该视图不会被 物理上物质化。相反，在每一次有查询引用该视图时，视图的查询都会被运行。

CREATE OR REPLACE VIEW与之相似，但是如果 已经存在一个同名视图，该视图会被替换。新查询必须产生和现有试图查询相同 的列（也就是相同的列序、相同的列名、相同的数据类型），但是它可以在列表 的末尾加上额外的列。产生输出列的计算可以完全不同。

如果给定了一个模式名（例如CREATE VIEW myschema.myview ...），那么该视图会被创建在指定的模式中。否则，它会 被创建在当前模式中。临时视图存在于一个特殊模式中，因此创建临时视图时不能 给定一个模式名。视图的名称不能与同一模式中任何其他视图、表、序列、索引或 外部表同名。

参数

TEMPORARY或者TEMP

如果被指定，视图被创建为一个临时视图。在当前会话结束时会自动 删掉临时视图。当临时视图存在时，具有相同名称的已有永久视图对 当前会话不可见，除非用模式限定的名称引用它们。

如果视图引用的任何表是临时的，视图将被创建为临时视图（不管有 没有指定TEMPORARY）。

RECURSIVE

创建一个递归视图。语法

CREATE RECURSIVE VIEW [*schema* .] *view_name* (*column_names*) AS SELECT ...;

等效于

CREATE VIEW [*schema* .] *view_name* AS WITH RECURSIVE *view_name* (*column_names*) AS (SELECT ...) SELECT *column_names* FROM *view_name*;

对于一个递归视图必须指定一个视图列名列表。

name

要创建的视图的名字（可以是模式限定的）。

column_name

要用于视图列的名称列表，可选。如果没有给出，列名会根据查询 推导。

WITH (*view_option_name* [= *view_option_value*] [, ...])

这个子句为视图指定一些可选的参数，支持下列参数：

check_option (string)

这个参数可以是local或者cascaded，并且它 等效于指定 WITH [CASCADED | LOCAL] CHECK OPTION（见下文）。 可以使用ALTER VIEW(7)在一个现有视图上修改这个选项。

security_barrier (boolean)

如果希望视图提供行级安全性，应该使用这个参数。

query

提供视图的行和列的一个SELECT(7)或者 VALUES(7)命令。

WITH [CASCADED | LOCAL] CHECK OPTION

这个选项控制自动可更新视图的行为。这个选项被指定时，将检查该视图上的INSERT和UPDATE命令以确保新行满足 视图的定义条件（也就是，将检查新行来确保通过视图能看到它们）。如果新行 不满足条件，更新将被拒绝。如果没有指定CHECK OPTION，会允许该视图上的INSERT和UPDATE命令 创建通过该视图不可见的行。支持下列检查选项：

LOCAL

只根据直接定义在该视图本身的条件检查新行。任何定义在底层基视图上的 条件都不会被检查（除非它们也指定了CHECK OPTION）。

CASCADED

会根据该视图和所有底层基视图上的条件检查新行。如果 CHECK OPTION被指定，并且没有指定 LOCAL和CASCADED，则会假定为 CASCADED。

CHECK OPTION不应该和RECURSIVE视图一起使用。

注意，只有在自动可更新的、没有INSTEAD OF触发器或者 INSTEAD规则的视图上才支持CHECK OPTION。 如果一个自动可更新的视图被定义在一个具有INSTEAD OF 触发器的基视图之上，那么LOCAL CHECK OPTION可以被 用来检查该自动可更新的视图之上的条件，但具有INSTEAD OF 触发器的基视图上的条件不会被检查（一个级联检查选项将不会级联到一个 触发器可更新的视图，并且任何直接定义在一个触发器可更新视图上的检查 选项将被忽略）。如果该视图或者任何基础关系具有导致 INSERT或UPDATE命令被重写的 INSTEAD规则，那么在被重写的查询中将忽略所有检查选项， 包括任何来自于定义在带有INSTEAD规则的关系之上的自动 可更新视图的检查。

注解

使用DROP VIEW(7)语句删除视图。

要注意视图列的名称和类型将会按照想要的方式指定。例如：

CREATE VIEW vista AS SELECT 'Hello World';

是不好的形式，因为列名默认为?column?，而且列的数据类型默认为text，这可能不是用户想要的。视图结果中一个字符串更好的风格类似于这样：

CREATE VIEW vista AS SELECT text 'Hello World' AS hello;

对视图中引用的表的访问由视图拥有者的权限决定。在某些情况下，这可以 被用来提供安全但是受限的底层表访问。不过，并非所有视图都对篡改是安 全的。在视图中调用的函数会被 同样对待，就好像是直接在使用该视图的查询中调用它们一样。因此，一个 视图的用户必须具有调用视图所使用的全部函数的权限。

当CREATE OR REPLACE VIEW被用在一个现有视图上时， 只有该视图的定义 SELECT 规则被改变。其他包括拥有关系、权限和非 SELECT 规则在内的视图属性不会被更改。要替换视图，必须拥有它（包括 作为拥有角色的一个成员）。

可更新视图

简单视图是自动可更新的：系统将允许在这类视图上以在常规表上相同的方式 使用INSERT、UPDATE以及 DELETE语句。如果一个视图满足以下条件，它就是自动 可更新的：

- 在该视图的FROM列表中刚好只有一项，并且它必须是一个 表或者另一个可更新视图。
- 视图定义的顶层不能包含WITH、DISTINCT、 GROUP BY、HAVING、 LIMIT或者OFFSET子句。
- 视图定义的顶层不能包含集合操作（UNION、 INTERSECT或者EXCEPT）。
- 视图的选择列表不能包含任何聚集、窗口函数或者集合返回函数。

一个自动可更新的视图可以混合可更新列以及不可更新列。如果一个列是对底层 基本关系中一个可更新列的简单引用，则它是可更新的。否则该列是只读的，并 且在一个INSERT或者UPDATE语句尝试对 它赋值时会报出一个错误。

如果视图是自动可更新的，系统将把视图上的任何INSERT、 UPDATE或者DELETE语句转换成在底层 基本关系上的对应语句。带有ON CONFLICT UPDATE子句的 INSERT语句已经被完全支持。

如果一个自动可更新视图包含一个WHERE条件，该条件会限制 基本关系的哪些行可以被该视图上的UPDATE以及 DELETE语句修改。不过，一个允许被UPDATE 修改的行可能让该行不再满足WHERE条件，并且因此也不再能 从视图中可见。类似地，一个INSERT命令可能插入不满足WHERE条件的基本关系行，并且因此从视图中也看不到这些行 （ON CONFLICT UPDATE可能会类似地影响无法通过该视图见 到的现有行）。 CHECK OPTION可以被用来阻止INSERT和UPDATE命令创建这类从视图中无法看到的行。

如果一个自动可更新视图被标记了security_barrier属性，那么 所有该属性的WHERE条件（以及任何使用标记为 LEAKPROOF的操作符的条件）将在该视图使用者的任何条件 之前计算。注意正因为这样，不会 被最终返回的行（因为它们不会通过用户的WHERE条件）可能 仍会结束被锁定的状态。可以用EXPLAIN来查看 哪些条件被应用在关系层面（并且因此不锁定行）以及哪些不会被应用在关系 层面。

一个更加复杂的不满足所有这些条件的视图默认是只读的：系统将不允许在 该视图上的插入、更新或者删除。可以通过在该视图上创建一个 INSTEAD OF触发器来获得可更新视图的效果，

该触发器必须 把该视图上的尝试的插入等转换成其他表上合适的动作。更多信息请见CREATE TRIGGER(7)。另一种可能性是创建规则（见 CREATE RULE(7)），不过实际中触发器更容易理解和正确使用。

注意在视图上执行插入、更新或删除的用户必须具有该视图上相应的插入、 更新或删除特权。此外，视图的拥有者必须拥有底层基本关系上的相关特权， 但是执行更新的用户并不需要底层基本关系上的任何权限。

示例

创建一个由所有喜剧电影组成的视图：

```
CREATE VIEW comedies AS
   SELECT *
   FROM films
   WHERE kind = 'Comedy';
```

创建的视图包含创建时film表中的列。尽管* 被用来创建该视图，后来被加入到该表中的列不会成为该视图的组成部分。

创建带有LOCAL CHECK OPTION的视图：

```
CREATE VIEW universal_comedies AS
   SELECT *
   FROM comedies
   WHERE classification = 'U'
   WITH LOCAL CHECK OPTION;
```

这将创建一个基于comedies视图的视图，只显示 kind = 'Comedy'和classification = 'U'的电影。 如果新行没有classification = 'U'，在该视图中的任何 INSERT或UPDATE尝试将被拒绝， 但是电影的kind将不会被检查。

用CASCADED CHECK OPTION创建一个视图：

```
CREATE VIEW ux_comedies AS
   SELECT *
   FROM comedies
   WHERE classification = 'ux'
   WITH CASCADED CHECK OPTION;
```

这将创建一个检查新行的kind和classification 的视图。

创建一个由可更新列和不可更新列混合而成的视图：

```
CREATE VIEW comedies AS
   SELECT f.*,
          country_code_to_name(f.country_code) AS country,
          (SELECT avg(r.rating)
           FROM user_ratings r
           WHERE r.film_id = f.id) AS avg_rating
   FROM films f
```

```
  WHERE f.kind = 'Comedy';
```

这个视图将支持INSERT、UPDATE 以及DELETE。所有来自于films表的列都 将是可更新的，而计算列country和avg_rating 将是只读的。

创建一个由数字 1 到 100 组成的递归视图：

```
CREATE RECURSIVE VIEW public.nums_1_100 (n) AS
  VALUES (1)
UNION ALL
  SELECT n+1 FROM nums_1_100 WHERE n < 100;
```

注意在这个CREATE中尽管递归的视图名称是方案限定的，但它内部的自引用不是方案限定的。这是因为隐式创建的CTE的名称不能是方案限定的。

兼容性

CREATE OR REPLACE VIEW是一种 UXDB的语言扩展。临时 视图的概念也是这样。WITH (...)子句也是一种扩展。

另见

ALTER VIEW(7), DROP VIEW(7), CREATE MATERIALIZED VIEW(7)

名称

DEALLOCATE — 释放一个预备语句

大纲

```
DEALLOCATE [ PREPARE ] { name | ALL }
```

描述

DEALLOCATE被用来释放一个之前 准备好的 SQL 语句。如果不显式地释放一个预备语句，会话结束 时会释放它。

更多关于预备语句的信息请见PREPARE(7)。

参数

PREPARE

这个关键词会被忽略。

name

要释放的预备语句的名称。

ALL

释放所有预备语句。

兼容性

SQL 标准包括一个**DEALLOCATE**语句， 但是只用于嵌入式 SQL。

另见

EXECUTE(7), PREPARE(7)

名称

DECLARE — 定义一个游标

大纲

```
DECLARE name [ BINARY ] [ INSENSITIVE ] [ [ NO ] SCROLL ]
    CURSOR [ { WITH | WITHOUT } HOLD ] FOR query
```

描述

DECLARE允许用户创建游标， 游标可以被用来在大型查询暂停时检索少量的行。游标被创建后，可以用FETCH(7)从中取得行。

> **注意**
>
> 这个页面描述在 SQL 命令层面上游标的用法。如果想要在 PL/uxSQL函数中使用游标，规则是不同的。

参数

name

要创建的游标的名称。

BINARY

让游标返回二进制数据而不是返回文本格式数据。

INSENSITIVE

指示从游标中检索数据的过程不受游标创建之后在其底层表上发生的更新的 影响。在UXDB中，这是默认的 行为。因此这个关键词没有实际效果，仅仅被用于兼容 SQL 标准。

SCROLL
NO SCROLL

SCROLL指定游标可以用非顺序（例如，反向） 的方式从中检索行。根据查询的执行计划的复杂度，指定 SCROLL可能导致查询执行时间上的性能损失。 NO SCROLL指定游标不能以非顺序的方式从中检索 行。默认是允许在某些情况下滚动，但这和指定 SCROLL不完全相同。详情请见 “注解”一节。

WITH HOLD
WITHOUT HOLD

WITH HOLD指定该游标在创建它的事务提交 之后还能被继续使用。WITHOUT HOLD指定该游标 不能在创建它的事务之外使用。如果两者都没有指定，则默认为 WITHOUT HOLD。

query

用于提供该游标返回的行的SELECT(7)或者 VALUES(7)命令。

关键词BINARY、 INSENSITIVE和SCROLL 可以以任意顺序出现。

注解

普通游标以文本格式返回数据，这和SELECT产生的数据一样。 BINARY选项指定游标应该以二进制格式返回数据。这减少了服务 器和客户端的转换负担，但程序员需要付出更多工作来处理与平台相关的二进制 数据格式。例如，如果一个查询从一个整数列中返回一个值一，用一个默认游标 将得到一个字符串1，而使用一个二进制游标将得到该值的四字节 内部表示（big-endian 大端字节顺序）。

使用二进制游标时应该注意。很多应用（包括uxsql）还没有准备好处理二进制游标，它们仍然期待数据以文本格式到来。

> **注意**
>
> 当客户端应用使用“扩展查询”协议发出一个 FETCH命令，绑定协议消息会指定使用文本还是 二进制格式检索数据。这种选择会覆盖定义游标时指定的方式。因此 在使用扩展查询协议时，这样一个二进制游标的概念实际是被废弃的 — 任何游标都可以被视作文本或者二进制。

除非指定了WITH HOLD，这个命令创建的游标 只能在当前事务中使用。因此，没有WITH HOLD 的DECLARE在事务块外是没有用的：游标只会生存 到该语句结束。因此如果这种命令在事务块之外被使用， UXDB会报告一个错误。 定义事务块需要使用BEGIN(7)和 COMMIT(7)（或者ROLLBACK(7)）。

如果指定了WITH HOLD并且创建游标的事务 成功提交，在同一个会话中的后续事务中还能够继续访问该游标（ 但是如果创建事务被中止，游标会被移除）。一个用 WITH HOLD创建的游标可以用一个显式的 CLOSE命令关闭，或者会话结束时它 也会被关闭。在当前的实现中，由一个被保持游标表示的行会被复 制到一个临时文件或者内存区域中，这样它们才会在后续事务中保持可用。

当查询包括FOR UPDATE或FOR SHARE时， 不能指定WITH HOLD。

在定义一个将被反向取元组的游标时，应该指定SCROLL 选项。这是 SQL 标准所要求的。不过，为了和早期版本兼容， 如果游标的查询计划足够简单到支持它不需要额外的开销，UXDB会允许在没有 SCROLL的情况下反向取元组。不过，建议应用开发者 不要依赖于从没有用SCROLL创建的游标中反向取 元组。如果指定了NO SCROLL，那么任何情况下都不 允许反向取元组。

当查询包括FOR UPDATE或FOR SHARE时， 也不允许反向取元组。因此在这种情况下不能指定SCROLL。

> **注意**
>
> 如果可滚动和WITH HOLD游标调用了任何不稳定的函数，它们可能给出预期之外的结果。当重新取得一个之前取得过的行时，那些函数会被重新执行，这可能导致得到与第一次不同的结果。对这类情况的一种变通方法是，声明游标为WITH HOLD并且在从其中读取任何行之前提交事务。这将强制该游标的整个输出被物化在临时存储中，这样针对每一行只会执行一次不稳定函数。

如果游标的查询包括FOR UPDATE或者FOR SHARE，那么被返回的行会在它们第一次被取得时被锁定，这和带有 这些选项的常规SELECT(7)命令一样。此外，被返回的 行将是最新的版本，

因此这些选项提供了被 SQL 标准称为 “敏感游标”的等效体（把INSENSITIVE与 FOR UPDATE或者FOR SHARE一起指定是错误）。

> **注意**
>
> 如果游标要和**UPDATE ... WHERE CURRENT OF**或者 **DELETE ... WHERE CURRENT OF**一起使用，通常推荐 使用FOR UPDATE。使用FOR UPDATE可以 阻止其他会话在行被取得和被更新之间修改行。如果没有 FOR UPDATE，当行在游标创建后被更改后，一个后续的 WHERE CURRENT OF命令将不会产生效果。
>
> 另一个使用FOR UPDATE的原因是，如果没有它，当游标查询不符合 SQL 标准的“简单可更新”规则时，后续的 WHERE CURRENT OF可能会失败（特别地，该游标必须只引用一个 表并且没有使用分组或者ORDER BY）。不是简单可更新的游标可能 成功也可能不成功，这取决于计划选择的细节。因此在最坏的情况下，应用可能会 在测试时成功但是在生产中失败。如果指定了FOR UPDATE， 则保证游标是可更新的。
>
> 不把FOR UPDATE和WHERE CURRENT OF一起用的 主要原因是，需要游标时可滚动的或者对于后续更新不敏感（也就是说，继续显示 旧的数据）。如果这是需求，应密切关注安上述警示。

SQL 标准只对嵌入式SQL中的游标做出了规定。 UXDB服务器没有为游标实现 **OPEN**语句。当游标被声明时就被认为已经 被打开。不过，ECUX（ UXDB的嵌入式 SQL 预处理器） 支持标准 SQL 游标习惯，包括那些**DECLARE** 和**OPEN**语句。

可以通过查询ux_cursors 系统视图可以看到所有可用的游标。

示例

声明一个游标：

```
DECLARE liahona CURSOR FOR SELECT * FROM films;
```

更多游标的例子请见FETCH(7)。

兼容性

SQL 标准认为游标是否默认对底层数据的并发更新敏感是与实现相关的。在 UXDB中，默认游标对此是不敏 感的，并且可以通过指定FOR UPDATE让它变得对此敏感。其他 产品的行为可能有所不同。

SQL 标准只允许在嵌入式SQL和模块中使用游标。 UXDB允许以交互的方式使用游标。

二进制游标是一种UXDB 扩展。

另见

CLOSE(7), FETCH(7), MOVE(7)

名称

DELETE — 删除一个表的行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]

DELETE table_name WHERE [ condition ]
```

描述

DELETE从指定表中删除满足WHERE子句的行。如果WHERE子句没有出现，效果将会是删除表中的所有行。结果是一个合法的空表。

DELETE不加FROM功能与delete from功能相同。

> **提示**
>
> TRUNCATE(7)提供移除表中所有行的快速机制。

有两种方式可以使用数据库中其他表中包含的信息来删除一个表的行： 使用子选择或者在USING子句中指定额外的表。 哪种技术更合适取决于特定的环境。

可选的RETURNING子句导致DELETE 基于实际被删除的每一行计算并且返回值。任何使用被删除表列或者 USING中提到的其他表的列的表达式都可以被计算。 RETURNING列表的语法和SELECT的 输出列表语法相同。

要从表中删除行，必须具有其上的DELETE特权，以及USING子句中任何表以及其值在*condition*中被读取的表上的SELECT特权。

一般需要在WHERE子句中指定条件来删除对应的记录，条件语句可以使用AND或OR运算符来指定一个或多个。

参数

with_query

WITH子句允许指定一个或者多个子查询，在 DELETE查询中可以用子查询的名字来引用它们。 详见第 4.8 节 “WITH查询（公共表表达式）”和SELECT(7)。

table_name

要从其中删除行的表名（可以是模式限定的）。如果在表名前指定 ONLY，只会从提到的表中删除匹配的行。如果没有指定 ONLY，还会删除该表的任何继承表中的匹配行。可选地，可以在表名后面指定*来显式指定要包括继承表。

alias

目标表的一个别名。提供别名时，它会完全隐藏该表的真实名称。例如， 对于DELETE FROM foo AS f， DELETE语句的剩余部分都会用 f而不是 foo来引用该表。

using_list

一个表表达式的列表，它允许在WHERE条件中出现 来自其他表的列。这和SELECT语句的“FROM 子句”一节中指定 的表列表相似。例如，可以指定表的别名。除非想要进行自连接， 否则不要在*using_list* 再写上目标表。

condition

一个返回boolean类型值的表达式。只有让这个 表达式返回true的行才将被删除。

cursor_name

要在WHERE CURRENT OF情况中使用的游标 的名称。最近一次从这个游标中取出的行将被删除。该游标 必须是DELETE的目标表上的非分组查询。 注意不能在使用WHERE CURRENT OF的同时 指定一个布尔条件。有关将游标用于 WHERE CURRENT OF的更多信息请见 DECLARE(7)。

output_expression

在每一行被删除后，会被DELETE计算并且返回的表达式。 该表达式可以使用*table_name* 以及USING中的表的任何列。写成*可以返回所有列。

output_name

被返回列的名称。

输出

在成功完成时，一个DELETE命令会返回以下形式 的命令标签：

DELETE *count*

*count*是被删除行的数目。 注意如果有一个BEFORE DELETE触发器抑制删除，那么该数目 可能小于匹配*condition* 的行数。如果*count*为 0， 表示查询没有删除行（这并非一种错误）。

如果DELETE命令包含RETURNING子句， 则结果会与包含有RETURNING列表中定义的列和值的SELECT语句结果相似，这些结果是在被该命令删除的 行上计算得来。

注解

通过在USING子句中指定其他的表， UXDB允许在 WHERE条件中引用其他表的列。例如，要 删除有一个给定制片人制作的所有电影，可以这样做：

```
DELETE FROM films USING producers
  WHERE producer_id = producers.id AND producers.name = 'foo';
```

这里实际发生的事情是在films和 producers之间进行连接，然后删除 所有成功连接的films行。这种语法不 属于标准。更标准的方式是：

```
DELETE FROM films
  WHERE producer_id IN (SELECT id FROM producers WHERE name = 'foo');
```

在一些情况下，连接形式比子查询形式更容易书写或者执行更快。

示例

删除所有电影，但音乐剧除外：

```
--方式1
DELETE FROM films WHERE kind <> 'Musical';

--方式2
DELETE films WHERE kind <> 'Musical';
```

清空表films：

```
DELETE FROM films;
```

删除已完成的任务，返回被删除行的明细：

```
DELETE FROM tasks WHERE status = 'DONE' RETURNING *;
```

删除tasks中游标c_tasks 当前位于其上的行：

```
DELETE FROM tasks WHERE CURRENT OF c_tasks;
```

兼容性

这个命令符合SQL标准，不过 USING和RETURNING子句是 UXDB扩展，在 DELETE中使用WITH也是扩展。

又见

TRUNCATE(7)

名称

DISCARD — 抛弃会话状态

大纲

DISCARD { ALL | PLANS | SEQUENCES | TEMPORARY | TEMP }

描述

DISCARD释放与一个数据库会话相关的内部资源。 这个命令有助于部分或者完全重置该会话的状态。有几个子命令来 释放不同类型的资源。DISCARD ALL变体把所有 其他形式都包含在内，并且还会重置额外的状态。

参数

PLANS

释放所有已缓存的查询计划，强制在下一次使用相关预备语句时 重新做计划。

SEQUENCES

丢弃所有已缓存的序列相关的状态，包括 currval()/lastval()信息 以及任何还未被nextval()返回的预分配的 序列值（预分配序列值的描述请见 CREATE SEQUENCE(7)）；

TEMPORARY or TEMP

删除当前会话中创建的所有临时表。

ALL

释放与当前会话相关的所有临时资源并且把会话重置为初始状态。 当前这和执行以下语句序列的效果相同：

```
SET SESSION AUTHORIZATION DEFAULT;
RESET ALL;
DEALLOCATE ALL;
CLOSE ALL;
UNLISTEN *;
SELECT ux_advisory_unlock_all();
DISCARD PLANS;
DISCARD SEQUENCES;
DISCARD TEMP;
```

注解

DISCARD ALL不能在事务块内执行。

兼容性

DISCARD是一种 UXDB扩展。

名称

DO — 执行一个匿名代码块

大纲

DO [LANGUAGE *lang_name*] *code*

[DECLARE [*declare_section*]]
BEGIN
execution_section
END;
/

描述

DO执行一个匿名代码块，或者换句话说执行一个以一种过程语言编写的瞬时匿名函数。
DECLARE与DO执行匿名块语法的区别：语法上相当于DO执行匿名块语法只保留了code部分，功能上一致。

代码块就好像是一个没有参数并且返回void的函数的函数体。它会被在一次时间内解析并且执行。

可选的LANGUAGE子句可以卸载代码块之前或者之后。

参数

code

要被执行的过程语言代码。就像在 CREATE FUNCTION中一样，必须把它指定为一个 字符串。推荐使用一个美元引用的文本。

lang_name

编写该代码的过程语言的名称。如果省略，默认为pluxsql。

declare_section

声明变量，包括变量名和类型，例如“var int”。

execution_section

匿名块中要执行的语句。

注解

要使用的过程语言必须已经用CREATE EXTENSION安装在 当前数据库中。默认已经安装了pluxsql，但是其他语言没有被 安装。

用户必须拥有该过程语言的USAGE特权，如果该语言 是不可信的则必须是一个超级用户。这和创建一个该语言的函数对 特权的要求相同。

如果在事务块中执行DO，过程代码则无法执行事务控制语句。只有在自己的事务中执行DO时，才允许使用事务控制语句。

例子

把模式public中所有视图上的所有特权授予 给角色webuser：

```
DO $$DECLARE r record;
BEGIN
   FOR r IN SELECT table_schema, table_name FROM information_schema.tables
         WHERE table_type = 'VIEW' AND table_schema = 'public'
   LOOP
      EXECUTE 'GRANT ALL ON ' || quote_ident(r.table_schema) || '.' || quote_ident(r.table_name) || '
 TO webuser';
   END LOOP;
END$$;

DECLARE VAR INT;
BEGIN
SELECT COUNT(*) INTO VAR FROM UX_CLASS;
RAISE NOTICE '%', VAR;
END;
/
```

兼容性

SQL 标准中没有DO语句。

另见

CREATE LANGUAGE(7)

名称

DROP ACCESS METHOD — 移除一种访问方法

大纲

DROP ACCESS METHOD [IF EXISTS] *name* [CASCADE | RESTRICT]

简介

DROP ACCESS METHOD移除一种现有的访问方法。只有超级用户能够删除访问方法。

参数

IF EXISTS

如果该访问方法不存在，则不会抛出错误。这种情况下会发出一个提示。

name

一种现有的访问方法的名称。

CASCADE

自动删除依赖于该访问方法的对象（例如操作符类、操作符族以及索引），并且接着删除所有依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该访问方法，则拒绝删除它。这是默认设置。

示例

删除访问方法heptree：

```
DROP ACCESS METHOD heptree;
```

兼容性

DROP ACCESS METHOD是一种UXDB扩展。

另见

CREATE ACCESS METHOD(7)

名称

DROP AGGREGATE — 移除一个聚集函数

大纲

DROP AGGREGATE [IF EXISTS] *name* (*aggregate_signature*) [, ...] [CASCADE | RESTRICT]

这里*aggregate_signature*是：

* |
[*argmode*] [*argname*] *argtype* [, ...] |
[[*argmode*] [*argname*] *argtype* [, ...]] ORDER BY [*argmode*] [*argname*] *argtype* [, ...]

描述

DROP AGGREGATE移除一个现有的 聚集函数。要执行这个命令，当前用户必须是该聚集函数的拥有者。

参数

IF EXISTS

如果该聚集不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有聚集函数的名称（可以是模式限定的）。

argmode

一个参数的模式：IN或VARIADIC。 如果被忽略，默认值是IN。

argname

一个参数的名称。注意**DROP AGGREGATE** 并不真正关心参数名称，因为决定聚集函数的身份时只需要参数数据类型。

argtype

该聚集函数所操作的一个输入数据类型。要引用一个零参数的聚集函数，写 *来替代参数说明列表。要引用一个有序集聚集函数，在直接和 聚集参数说明之间写上ORDER BY。

CASCADE

自动删除依赖于该聚集函数的对象（例如使用它的视图），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该聚集函数，则拒绝删除它。这是默认值。

注解

ALTER AGGREGATE(7)下描述了另一种引用有序集聚集的语法。

示例

要为类型integer移除聚集函数myavg：

```
DROP AGGREGATE myavg(integer);
```

要移除假想集聚集函数myrank，该函数接收一个排序列的 任意列表和直接参数的一个匹配的列表：

```
DROP AGGREGATE myrank(VARIADIC "any" ORDER BY VARIADIC "any");
```

要在一个命令中删除多个聚合函数：

```
DROP AGGREGATE myavg(integer), myavg(bigint);
```

兼容性

在 SQL 标准中没有**DROP AGGREGATE**语句。

另见

ALTER AGGREGATE(7), CREATE AGGREGATE(7)

名称

DROP CAST — 移除一个造型

大纲

DROP CAST [IF EXISTS] (*source_type* AS *target_type*) [CASCADE | RESTRICT]

描述

DROP CAST移除一个之前定义好的造型。

要能删除一个造型，必须拥有源数据类型或目标数据类型。这也是　　创建一个造型所要求的特权。

参数

IF EXISTS

　　如果该造型不存在则不要抛出一个错误，而是发出一个提示。

source_type

　　该造型的源数据类型的名称。

target_type

　　该造型的目标数据类型的名称。

CASCADE
RESTRICT

　　这些关键词没有任何效果，因为在造型上没有依赖性。

示例

要移除从类型text到类型int的造型：

```
DROP CAST (text AS int);
```

兼容性

DROP CAST命令符合 SQL 标准。

另见

CREATE CAST(7)

名称

DROP COLLATION — 移除一个排序规则

大纲

DROP COLLATION [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP COLLATION移除一个之前 定义好的排序规则。要能删除一个排序规则，必须拥有它。

参数

IF EXISTS

如果该排序规则不存在则不要抛出一个错误，而是发出一个提示。

name

排序规则的名称。排序规则名称可以是模式限定的。

CASCADE

自动删除依赖于该排序规则的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节“依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该排序规则，则拒绝删除它。这是默认值。

示例

要删除名为german的排序规则：

```
DROP COLLATION german;
```

兼容性

除了IF EXISTS选项之外， **DROP COLLATION**命令符合 SQL标准。该选项是一个 UXDB扩展。

另见

ALTER COLLATION(7), CREATE COLLATION(7)

名称

DROP CONVERSION — 移除一个转换

大纲

DROP CONVERSION [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP CONVERSION移除一个之前定义好的 转换。要能删除一个转换，必须拥有该转换。

参数

IF EXISTS

如果该转换不存在则不要抛出一个错误，而是发出一个提示。

name

转换的名称。转换名称可以是模式限定的。

CASCADE
RESTRICT

这些关键词没有任何效果，因为在转换上没有依赖性。

示例

要删除名为myname的转换：

```
DROP CONVERSION myname;
```

兼容性

在 SQL 标准中没有**DROP CONVERSION**语句， 但是有一个**DROP TRANSLATION**语句。还有 对应的**CREATE TRANSLATION**语句，它与 UXDB 中的**CREATE CONVERSION** 语句相似。

另见

ALTER CONVERSION(7), CREATE CONVERSION(7)

名称

DROP DATABASE — 移除一个数据库

大纲

DROP DATABASE [IF EXISTS] *name*

描述

DROP DATABASE移除一个数据库。它会 移除该数据库的系统目录项并且删除包含数据的文件目录。它只能由数据库 拥有者执行。还有，当或者任何其他人已经连接到目标数据库时，它不能被执行（连接到uxdb或者任何其他数据库来发出这 个命令）。

DROP DATABASE不能被撤销。请注意使用!

参数

IF EXISTS

如果该数据库不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的数据库的名称。

注解

DROP DATABASE不能在一个事务块内执行。

在连接到目标数据库时，这个命令不能被执行。因此，使用程序dropdb会更方便，它是这个命令的一个包装器。

兼容性

SQL 标准中没有**DROP DATABASE**语句。

另见

CREATE DATABASE(7)

名称

DROP DATABASE LINK — 删除数据库链接

大纲

DROP DATABASE LINK [IF EXISTS] [*schema.*]*linkname*

描述

删除数据库链接

UXDB数据库实现CREATE/DROP DATABASE LINK相关语法及功能支持场景如下表所示。

参数

[*schema.*]*linkname*

dblink　　名称，对于私有dblink，可以切换到对于的schema下去删除，也可以指定schema.linkname来删除。

名称

DROP DOMAIN — 移除一个域

大纲

DROP DOMAIN [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP DOMAIN移除一个域。只有域的拥有者 才能移除它。

参数

IF EXISTS

如果该域不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有域的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该域的对象（例如表列），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该域，则拒绝删除它。这是默认值。

示例

要移除域box：

```
DROP DOMAIN box;
```

兼容性

除了IF EXISTS选项，这个命令符合 SQL 标准。该选项 是一个UXDB扩展。

另见

CREATE DOMAIN(7), ALTER DOMAIN(7)

名称

DROP EVENT TRIGGER — 移除一个事件触发器

大纲

```
DROP EVENT TRIGGER [ IF EXISTS ] name [ CASCADE | RESTRICT ]
```

描述

DROP EVENT TRIGGER移除一个已有的 事件触发器。要执行这个命令，当前用户必须是事件触发器的拥有者。

参数

IF EXISTS

如果该事件触发器不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的事件触发器的名称。

CASCADE

自动删除依赖于该触发器的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节“依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

示例

销毁触发器snitch：

```
DROP EVENT TRIGGER snitch;
```

兼容性

在 SQL 标准中没有**DROP EVENT TRIGGER**语句。

另见

CREATE EVENT TRIGGER(7), ALTER EVENT TRIGGER(7)

名称

DROP EXTENSION — 移除一个扩展

大纲

DROP EXTENSION [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP EXTENSION从数据库移除扩展。 删除一个扩展会导致其组成对象也被删除。

必须拥有该扩展才能使用**DROP EXTENSION**。

参数

IF EXISTS

如果该扩展不存在则不要抛出一个错误，而是发出一个提示。

name

一个已安装扩展的名称。

CASCADE

自动删除依赖于该扩展的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该扩展（而不是它自己拥有的成员 对象和其他被列在同一个**DROP**命令中 的扩展），则拒绝删除它。这是默认值。

示例

要从当前数据库移除扩展hstore：

```
DROP EXTENSION hstore;
```

如果hstore的任何对象在该数据库库中 正在使用，例如有一个表的列是hstore类型，这个 命令都将会失败。加上CASCADE选项可以强制 把这些依赖对象也移除。

兼容性

DROP EXTENSION是一个 UXDB扩展。

另见

CREATE EXTENSION(7), ALTER EXTENSION(7)

名称

DROP FOREIGN DATA WRAPPER — 移除一个外部数据包装器

大纲

DROP FOREIGN DATA WRAPPER [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP FOREIGN DATA WRAPPER 移除一个已有的外部数据包装器。要执行这个命令，当前用户 必须是该外部数据包装器的拥有者。

参数

IF EXISTS

如果该外部数据包装器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有外部数据包装器的名称。

CASCADE

自动删除依赖于该外部数据包装器的对象（例如服务器），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该外部数据包装器，则拒绝删除它。这是默认值。

示例

删除外部数据包装器dbi：

```
DROP FOREIGN DATA WRAPPER dbi;
```

兼容性

DROP FOREIGN DATA WRAPPER 符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句 是一个UXDB扩展。

另见

CREATE FOREIGN DATA WRAPPER(7), ALTER FOREIGN DATA WRAPPER(7)

名称

DROP FOREIGN TABLE — 移除一个外部表

大纲

DROP FOREIGN TABLE [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP FOREIGN TABLE移除一个 外部表。只有一个外部表的拥有者才能移除它。

参数

IF EXISTS

如果该外部表不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的外部表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该外部表的对象（例如视图），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该外部表，则拒绝删除它。这是默认值。

示例

要销毁两个外部表films 和distributors：

```
DROP FOREIGN TABLE films, distributors;
```

兼容性

这个命令符合 ISO/IEC 9075-9 (SQL/MED)，不过该标准只允许每个命令 中删除一个外部表并且没有IF EXISTS选项。该选项是一个 UXDB扩展。

另见

ALTER FOREIGN TABLE(7), CREATE FOREIGN TABLE(7)

名称

DROP FUNCTION — 移除一个函数

大纲

```
DROP FUNCTION [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

描述

DROP FUNCTION移除一个已有函数 的定义。要执行这个命令用户必须是该函数的拥有者。该函数的参数 类型必须被指定，因为多个不同的函数可能会具有相同的函数名和不 同的参数列表。

参数

IF EXISTS

如果该函数不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有函数的名称（可以是模式限定的）。 如果未指定参数列表，则该名称在其模式中必须是唯一的。

argmode

一个参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略， 则默认为IN。注意**DROP FUNCTION**并不真正关心 OUT参数，因为决定函数的身份时只需要输入参数。 因此列出IN、INOUT和 VARIADIC参数足以。

argname

一个参数的名称。注意 **DROP FUNCTION**并不真正关心 参数名称，因为决定函数的身份时只需要参数的数据类型。

argtype

如果函数有参数，这是函数参数的数据类型（可以是模式限定的）。

CASCADE

自动删除依赖于该函数的对象（例如操作符和触发器），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该函数，则拒绝删除它。这是默认值。

示例

这个命令移除平方根函数：

```
DROP FUNCTION sqrt(integer);
```

在一个命令中删除多个函数：

```
DROP FUNCTION sqrt(integer), sqrt(bigint);
```

如果函数名称在其模式中是唯一的，则可以在不带参数列表的情况下引用它：

```
DROP FUNCTION update_employee_salaries;
```

请注意，这与

```
DROP FUNCTION update_employee_salaries();
```

不同，后者引用一个零个参数的函数，而第一个变体才可以引用具有任意数量参数的函数， 包括零，只要该名称是唯一的。

兼容性

该命令符合SQL标准，使用这些UXDB扩展：

- 该标准只允许每个命令删除一个函数。
- IF EXISTS选项
- 能够指定参数模式和名称

另见

CREATE FUNCTION(7), ALTER FUNCTION(7), DROP PROCEDURE(7), DROP ROUTINE(7)

名称

DROP GROUP — 移除一个数据库角色

大纲

DROP GROUP [IF EXISTS] *name* [, ...]

描述

DROP GROUP现在是 DROP ROLE(7)的一个别名。

兼容性

在 SQL 标准中没有**DROP GROUP**语句。

另见

DROP ROLE(7)

名称

DROP INDEX — 移除一个索引

大纲

DROP INDEX [CONCURRENTLY] [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP INDEX从数据库系统中 移除一个已有的索引。要执行这个命令必须是该索引的拥 有者。

参数

CONCURRENTLY

删除索引并且不阻塞在索引基表上的并发选择、插入、更新和删除操作。一个 普通的DROP INDEX会要求该表上的排他锁，这样会阻塞 其他访问直至索引删除完成。通过这个选项，该命令会等待直至冲突事务完成。

在使用这个选项时有一些需要注意的事情。只能指定一个索引名称，并且不支持CASCADE选项（因此，一个支持UNIQUE或者 PRIMARY KEY约束的索引不能以这种方式删除）。还有，常规 的DROP INDEX命令可以在一个事务块内执行，而 DROP INDEX CONCURRENTLY不能。

对于临时表，DROP INDEX始终是非并发的，因为没有其他会话可以访问它们，而且丢弃非并发索引更加便宜。

IF EXISTS

如果该索引不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的索引的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该索引的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该索引，则拒绝删除它。这是默认值。

示例

这个命令将移除索引title_idx：

```
DROP INDEX title_idx;
```

兼容性

DROP INDEX是一个 UXDB语言扩展。在 SQL 标准中没有提供索引。

另见

CREATE INDEX(7)

名称

DROP LANGUAGE — 移除一个过程语言

大纲

DROP [PROCEDURAL] LANGUAGE [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP LANGUAGE移除一个之前注册的过程语言的定义。必须是一个超级用户或者该语言的拥有者才能使用**DROP LANGUAGE**。

> **注意**
>
> 大部分过程语言已经被做成了“扩展”，因此应该用DROP EXTENSION(7)而不是**DROP LANGUAGE**删除。

参数

IF EXISTS

如果该语言不存在则不要抛出一个错误，而是发出一个提示。

name

一个已有过程语言的名称。为了向前兼容，这个名称可以用单引号包围。

CASCADE

自动删除依赖于该语言的对象（例如该语言中的函数），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该语言，则拒绝删除它。这是默认值。

示例

这个命令移除过程语言plsample：

```
DROP LANGUAGE plsample;
```

兼容性

在 SQL 标准中没有**DROP LANGUAGE**语句。

另见

ALTER LANGUAGE(7), CREATE LANGUAGE(7)

名称

DROP MATERIALIZED VIEW — 移除一个物化视图

大纲

DROP MATERIALIZED VIEW [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP MATERIALIZED VIEW删除一个 现有的物化视图。要执行这个命令必须是该物化视图的拥有者。

参数

IF EXISTS

如果该物化视图不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的物化视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该物化视图的对象（例如其他物化视图或常规视图），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该物化视图，则拒绝删除它。这是默认值。

示例

这个命令将移除名为order_summary的物化视图：

```
DROP MATERIALIZED VIEW order_summary;
```

兼容性

DROP MATERIALIZED VIEW是一个 UXDB扩展。

另见

CREATE MATERIALIZED VIEW(7), ALTER MATERIALIZED VIEW(7), REFRESH MATERIALIZED VIEW(7)

名称

DROP OPERATOR — 移除一个操作符

大纲

DROP OPERATOR [IF EXISTS] *name* ({ *left_type* | NONE } , { *right_type* | NONE }) [, ...]
[CASCADE | RESTRICT]

描述

DROP OPERATOR从数据库系统中 删除一个现有的操作符。要执行这个命令，必须是该操作符的拥有者。

参数

IF EXISTS

如果该操作符不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有的操作符的名称（可以是模式限定的）。

left_type

该操作符左操作数的数据类型，如果没有左操作数就写 NONE。

right_type

该操作符右操作数的数据类型，如果没有右操作数就写 NONE。

CASCADE

自动删除依赖于该操作符的对象（例如使用它的视图），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该操作符，则拒绝删除它。这是默认值。

示例

为类型integer移除幂操作符 a^b：

DROP OPERATOR ^ (integer, integer);

为类型bit移除左一元按位补操作符 ~b：

DROP OPERATOR ~ (none, bit);

为类型bigint移除右一元阶乘操作符 x!：

DROP OPERATOR ! (bigint, none);

在一条命令中删除多个操作符：

DROP OPERATOR ~ (none, bit), ! (bigint, none);

兼容性

SQL 标准中没有**DROP OPERATOR**语句。

另见

CREATE OPERATOR(7), ALTER OPERATOR(7)

名称

DROP OPERATOR CLASS — 移除一个操作符类

大纲

DROP OPERATOR CLASS [IF EXISTS] *name* USING *index_method* [CASCADE | RESTRICT]

描述

DROP OPERATOR CLASS删除一个现有的 操作符类。要执行这个命令，必须是该操作符类的拥有者。

DROP OPERATOR CLASS不会删除任何被 该类引用的操作符或者函数。如果有索引依赖于该操作符类，将需要指 定CASCADE来完成删除。

参数

IF EXISTS

如果该操作符类不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有的操作符类的名称（可以是模式限定的）。

index_method

该操作符类适用的索引访问方法的名称。

CASCADE

自动删除依赖于该操作符类的对象（例如索引），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该操作符类，则拒绝删除它。这是默认值。

注解

DROP OPERATOR CLASS将不会删除包含该类的 操作符族，即使该族中已经没有任何成员（特别是由 **CREATE OPERATOR CLASS**隐式创建的族）。一个 空操作符族是无害的，但是为了整洁可能希望用 **DROP OPERATOR FAMILY**移除该操作符族，或者 一开始就使用**DROP OPERATOR FAMILY**会更好。

示例

移除 B-树操作符类widget_ops：

```
DROP OPERATOR CLASS widget_ops USING btree;
```

如果有任何使用该操作符类的索引存在，这个命令都不会成功。增加 CASCADE可以把这类索引与该操作符类一起删除。

兼容性

SQL 标准中没有**DROP OPERATOR CLASS**语句。

另见

ALTER OPERATOR CLASS(7), CREATE OPERATOR CLASS(7), DROP OPERATOR FAMILY(7)

名称

DROP OPERATOR FAMILY — 移除一个操作符族

大纲

DROP OPERATOR FAMILY [IF EXISTS] *name* USING *index_method* [CASCADE | RESTRICT]

描述

DROP OPERATOR FAMILY删除一个 现有的操作符族。要执行这个命令，必须是该操作符族的拥有者。

DROP OPERATOR FAMILY包括删除 该族所包含的任何操作符类，但是它不会删除该族所引用的任何操作 符或函数。如果有任何依赖于该族中操作符类的索引存在，将需要 指定CASCADE来完成删除。

参数

IF EXISTS

如果该操作符族不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有操作符族的名称（可以是模式限定的）。

index_method

该操作符族适用的索引访问方法的名称。

CASCADE

自动删除依赖于该操作符族的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节“依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该操作符族，则拒绝删除它。这是默认值。

示例

移除 B-树操作符族float_ops：

```
DROP OPERATOR FAMILY float_ops USING btree;
```

如果有任何使用该族中操作符类的索引存在，这个命令都不会成功。增加 CASCADE可以把这类索引与该操作符族一起删除。

兼容性

SQL 标准中没有**DROP OPERATOR FAMILY** 语句。

另见

ALTER OPERATOR FAMILY(7), CREATE OPERATOR FAMILY(7), ALTER OPERATOR CLASS(7), CREATE OPERATOR CLASS(7), DROP OPERATOR CLASS(7)

名称

DROP OWNED — 移除一个数据库角色拥有的数据库对象

大纲

DROP OWNED BY { *name* | CURRENT_USER | SESSION_USER } [, ...] [CASCADE | RESTRICT]

描述

DROP OWNED删除当前数据库中被指定 角色之一拥有的所有对象。任何已被授予给给定角色在当前数据库中对 象上或者在共享对象（数据库、表空间）上的特权也将会被收回。

参数

name

其对象将被删除并且其特权将被收回的角色的名称。

CASCADE

自动删除依赖于受影响对象的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节“依赖跟踪”）。

RESTRICT

如果有任何其他数据库对象依赖于一个受影响的对象， 则拒绝删除一个角色所拥有的对象。这是默认值。

注解

DROP OWNED常常被用来为移除一个 或者多个角色做准备。因为**DROP OWNED** 只影响当前数据库中的对象，通常需要在包含将被移除角色所拥有的对象 的每一个数据库中都执行这个命令。

使用CASCADE选项可能导致这个命令递归去删除由其他 用户所拥有的对象。

REASSIGN OWNED(7)命令是另一种选择，它可以把一个 或多个角色所拥有的所有数据库对象重新授予给其他角色。不过， **REASSIGN OWNED**不处理其他对象的特权。

角色所拥有的数据库、表空间将不会被移除。

兼容性

DROP OWNED命令是一个 UXDB扩展。

另见

REASSIGN OWNED(7), DROP ROLE(7)

名称

DROP POLICY — 从一个表移除一条行级安全性策略

大纲

DROP POLICY [IF EXISTS] *name* ON *table_name* [CASCADE | RESTRICT]

描述

DROP POLICY从该表移除指定的策略。注意如果从 一个移除了最后一条策略并且该表的行级安全性仍被 **ALTER TABLE**启用，则默认的否定策略将被使用。 不管该表的策略存在与否，ALTER TABLE ... DISABLE ROW LEVEL SECURITY都可以被用来禁用一个表的行级安全性。

参数

IF EXISTS

如果该策略不存在也不抛出一个错误。这种情况下会发出一个提示。

name

要删除的策略的名称。

table_name

该策略所在的表的名称（可以被模式限定）。

CASCADE
RESTRICT

这些关键词不会产生效果，因为它们不依赖于策略。

例子

要在名为my_table上删除策略p1：

```
DROP POLICY p1 ON my_table;
```

兼容性

DROP POLICY是一种UXDB扩展。

另见

CREATE POLICY(7), ALTER POLICY(7)

名称

DROP PROCEDURE — 移除一个过程

大纲

```
DROP PROCEDURE [ IF EXISTS ] name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] [, ...]
    [ CASCADE | RESTRICT ]
```

简介

DROP PROCEDURE移除一个现有过程的定义。为了执行这个命令，用户必须是该过程的拥有者。该过程的参数类型必须指定，因为可能存在多个不同的过程具有相同名称和不同参数列表。

参数

IF EXISTS

如果该过程不存在也不抛出一个错误。这种情况下会发出一个提示。

name

现有过程的名称（可以是被方案限定的）。如果没有指定参数列表，则该名称在其所属的方案中必须是唯一的。

argmode

参数的模式：IN或者VARIADIC。如果省略，默认为IN。

argname

参数的名称。注意，其实**DROP PROCEDURE**并不在意参数名称，因为只需要参数的数据类型来确定过程的身份。

argtype

该过程如果有参数，参数的数据类型（可以是被方案限定的）。

CASCADE

自动删除依赖于该过程的对象，然后接着删除依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该过程，则拒绝删除它。这是默认选项。

示例

```
DROP PROCEDURE do_db_maintenance();
```

兼容性

这个命令符合SQL标准，不过UXDB做了这些扩展：

- 标准仅允许每个命令删除一个过程。
- IF EXISTS选项
- 指定参数模式和名称的能力

另见

CREATE PROCEDURE(7), ALTER PROCEDURE(7), DROP FUNCTION(7), DROP ROUTINE(7)

名称

DROP PUBLICATION — 删除一个发布

大纲

DROP PUBLICATION [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP PUBLICATION从数据库中删除一个现有的发布。

发布只能被它自己的所有者或超级用户删除。

参数

IF EXISTS

如果发布不存在，不要抛出一个错误。在这种情况下发出一个提示。

name

现有发布的名称。

CASCADE
RESTRICT

这些关键词没有任何作用，因为发布没有依赖关系。

示例

删除一个发布：

```
DROP PUBLICATION mypublication;
```

兼容性

DROP PUBLICATION是一个UXDB 扩展。

又见

CREATE PUBLICATION(7), ALTER PUBLICATION(7)

名称

DROP ROLE — 移除一个数据库角色

大纲

DROP ROLE [IF EXISTS] *name* [, ...]

描述

DROP ROLE移除指定的角色。要删除一个 超级用户角色，必须自己就是一个超级用户。要删除一个非超级用户角 色，必须具有CREATEROLE特权。

如果一个角色仍然被集簇中任何数据库中引用，它就不能被移除。如果尝试 移除将会抛出一个错误。在删除该角色前，必须删除（或者重新授予所有 权）它所拥有的所有对象并且收回该已经授予给该角色的在其他对象上的特 权。REASSIGN OWNED(7)和DROP OWNED(7) 命令可以用于这个目的。

不过，没有必要移除涉及该角色的角色成员关系。 **DROP ROLE**会自动收回目标角色在其他角色中的成员 关系，以及其他角色在目标角色中的成员关系。其他角色不会被删除也不 会被影响。

参数

IF EXISTS

如果该角色不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的角色的名称。

注解

UXDB包括一个程序dropuser具有和这个命令完全相同的功能（事实 上它会调用这个命令），但是该程序可以从 shell 运行。

示例

要删除一个角色：

```
DROP ROLE jonathan;
```

兼容性

SQL 标准定义了**DROP ROLE**， 但是它只允许一次删除一个角色并且它指定了和 UXDB不同的特权需求。

另见

CREATE ROLE(7), ALTER ROLE(7), SET ROLE(7)

名称

DROP ROUTINE — 删除一个例程

大纲

DROP ROUTINE [IF EXISTS] *name* [([[*argmode*] [*argname*] *argtype* [, ...]])] [, ...]
  [CASCADE | RESTRICT]

简介

DROP ROUTINE删除一个现有例程的定义，它可以是一个聚集函数、一个普通函数或者过程。有关参数的描述、更多的示例以及进一步的细节请参考DROP AGGREGATE(7)、DROP FUNCTION(7)以及DROP PROCEDURE(7)。

示例

删除类型integer的例程foo：

```
DROP ROUTINE foo(integer);
```

不管foo是一个聚集、函数或是一个过程，这个命令都能起作用。

兼容性

这个命令符合SQL标准，不过UXDB做了下面这些扩展：

- 标准仅允许每个命令删除一个例程。
- IF EXISTS选项
- 指定参数模式和名称的能力
- 聚集函数是一种扩展。

另见

DROP AGGREGATE(7), DROP FUNCTION(7), DROP PROCEDURE(7), ALTER ROUTINE(7)

注意CREATE ROUTINE命令不存在。

名称

DROP RULE — 移除一个重写规则

大纲

DROP RULE [IF EXISTS] *name* ON *table_name* [CASCADE | RESTRICT]

描述

DROP RULE删除一个重写规则。

参数

IF EXISTS

如果该规则不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的规则的名称。

table_name

该规则适用的表或视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该规则的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该规则，则拒绝删除它。这是默认值。

示例

要删除重写规则newrule：

```
DROP RULE newrule ON mytable;
```

兼容性

DROP RULE是一个 UXDB语言扩展，整个 查询重写系统也是这样。

另见

CREATE RULE(7), ALTER RULE(7)

名称

DROP SCHEMA — 移除一个模式

大纲

DROP SCHEMA [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP SCHEMA从数据库中移除模式。

一个模式只能由其拥有者或一个超级用户删除。注意即使拥有者不拥有 该模式中的某些对象，也能删除该模式（以及所有含有的对象）。

参数

IF EXISTS

如果该模式不存在则不要抛出一个错误，而是发出一个提示。

name

一个模式的名称。

CASCADE

自动删除包含在该模式中的对象（表、函数等），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果该模式含有任何对象，则拒绝删除它。这是默认值。

注解

使用CASCADE选项可能会使这条命令移除除 指定模式之外其他模式中的对象。

示例

要从数据库中移除模式mystuff及其中 所包含的对象：

```
DROP SCHEMA mystuff CASCADE;
```

兼容性

DROP SCHEMA完全符合 SQL 标准， 不过该标准只允许在每个命令中删除一个模式并且没有 IF EXISTS选项。该选项是一个 UXDB扩展。

另见

ALTER SCHEMA(7), CREATE SCHEMA(7)

名称

DROP SEQUENCE — 移除一个序列

大纲

DROP SEQUENCE [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP SEQUENCE移除序数生成器。 一个序列只能被其拥有者或超级用户删除。

参数

IF EXISTS

如果该序列不存在则不要抛出一个错误，而是发出一个提示。

name

一个序列的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该序列的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该序列，则拒绝删除它。这是默认值。

示例

要移除序列serial：

```
DROP SEQUENCE serial;
```

兼容性

DROP SEQUENCE符合 SQL标准，不过该标准只允许每个命令中删除一个 序列并且没有IF EXISTS选项。该选项是一个 UXDB扩展。

另见

CREATE SEQUENCE(7), ALTER SEQUENCE(7)

名称

DROP SERVER — 移除一个外部服务器描述符

大纲

DROP SERVER [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP SERVER移除一个现有的外部服务器 描述符。要执行这个命令，当前用户必须是该服务器的拥有者。

参数

IF EXISTS

如果该服务器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有服务器的名称。

CASCADE

自动删除依赖于该服务器的对象（例如用户映射），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该服务器，则拒绝删除它。这是默认值。

示例

如果一个服务器foo存在则删除它：

```
DROP SERVER IF EXISTS foo;
```

兼容性

DROP SERVER符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句是一个 UXDB扩展。

另见

CREATE SERVER(7), ALTER SERVER(7)

名称

DROP STATISTICS — 删除扩展统计

大纲

DROP STATISTICS [IF EXISTS] *name* [, ...]

描述

DROP STATISTICS删除数据库中的统计对象。 只有统计对象的所有者、模式的所有者或超级用户可以删除统计对象。

参数

IF EXISTS

name

要删除的统计对象的名称（可以有模式修饰）。

示例

删除不同模式中的两个统计对象，如果不存在时不会失败：

```
DROP STATISTICS IF EXISTS
  accounting.users_uid_creation,
  public.grants_user_role;
```

兼容性

SQL标准中没有**DROP STATISTICS**命令。

又见

ALTER STATISTICS(7), CREATE STATISTICS(7)

名称

DROP SUBSCRIPTION — 删除一个订阅

大纲

DROP SUBSCRIPTION [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP SUBSCRIPTION删除数据库集群中的一个订阅。

只有超级用户才可以删除订阅。

如果订阅与复制槽相关联，就不能在事务块内部执行DROP SUBSCRIPTION。 （可以使用ALTER SUBSCRIPTION取消关联复制槽。）

参数

name

要删除的订阅的名称。

CASCADE
RESTRICT

这些关键词没有任何作用，因为订阅没有依赖关系。

注意

当删除与远程主机（正常状态）上的复制槽相关联的订阅时，DROP SUBSCRIPTION 将连接到远程主机，并尝试删除该复制槽，作为其操作的一部分。这是必要的， 以便释放远程主机上为订阅分配的资源。如果失败，因为远程主机不可访问， 或者因为远程复制槽不能被删除，或者复制槽不存在或从不存在， 则DROP SUBSCRIPTION命令将失败。要在这种情况下继续， 请执行ALTER SUBSCRIPTION ... SET (slot_name = NONE) 来解除复制槽与订阅的关联。之后，DROP SUBSCRIPTION 将不再尝试对远程主机执行任何操作。请注意，如果远程复制槽仍然存在， 则应手动删除该插槽；否则将继续保留WAL，最终可能导致磁盘空间不足。

如果订阅与复制槽相关联，那么不能在事务块内部执行DROP SUBSCRIPTION。

示例

删除一个订阅：

```
DROP SUBSCRIPTION mysub;
```

兼容性

DROP SUBSCRIPTION是一个UXDB 扩展。

又见

CREATE SUBSCRIPTION(7), ALTER SUBSCRIPTION(7)

名称

DROP SYNONYM — 删除一个同义词

大纲

DROP SYNONYM *synname*
DROP SYNONYM *synname(arg, arg, ...)*

描述

删除现有同义词。

参数

synname

用户自定义的同义词名称。

synname(arg,arg,...)

同义词函数名称及参数。

示例

删除索引idx1同义词，如下所示。

```
drop synonym syni1;
```

删除序列seq1同义词，如下所示。

```
drop synonym syn1;
```

删除视图vw1同义词，如下所示。

```
drop synonym synv1;
```

另见

CREATE SYNONYM

名称

DROP TABLE — 移除一个表

大纲

DROP TABLE [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP TABLE从数据库移除表。只有表拥有者、 模式拥有者和超级用户能删除一个表。要清空一个表中的行但是不销毁该表， 可以使用DELETE(7)或者TRUNCATE(7)。

DROP TABLE总是移除目标表的任何索引、规则、 触发器和约束。不过，要删除一个被视图或者另一个表的外键约束所引用的表， 必须指定CASCADE（CASCADE将会把依赖的视图 也完全移除，但是对于外键它将只移除外键约束，而完全不会移除其他表）。

参数

IF EXISTS

如果该表不存在则不要抛出一个错误，而是发出一个提示。

name

要删除的表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该表的对象（例如视图），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该表，则拒绝删除它。这是默认值。

示例

要销毁两个表films和 distributors：

```
DROP TABLE films, distributors;
```

兼容性

这个命令符合 SQL 标准，不过该标准只允许每个命令删除一个表并且没有 IF EXISTS选项。该选项是一个 UXDB扩展。

另见

ALTER TABLE(7), CREATE TABLE(7)

名称

DROP TABLESPACE — 移除一个表空间

大纲

DROP TABLESPACE [IF EXISTS] *name*

描述

DROP TABLESPACE从系统中移除一个表空间。

一个表空间只能被其拥有者或超级用户删除。在一个表空间能被删除前，其中 必须没有任何数据库对象。即使当前数据库中没有对象正在使用该表空间，也 可能有其他数据库的对象存在于其中。还有，如果该表空间被列在任何活动会 话的temp_tablespaces设置中， **DROP**也可能会失败，因为可能有临时文件存在其中。

参数

IF EXISTS

如果该表空间不存在则不要抛出一个错误，而是发出一个提示。

name

一个表空间的名称。

注解

DROP TABLESPACE不能在一个事务块内执行。

示例

要从系统移除表空间mystuff：

```
DROP TABLESPACE mystuff;
```

兼容性

DROP TABLESPACE是一个 UXDB扩展。

另见

CREATE TABLESPACE(7), ALTER TABLESPACE(7)

名称

DROP TEXT SEARCH CONFIGURATION — 移除一个文本搜索配置

大纲

DROP TEXT SEARCH CONFIGURATION [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP TEXT SEARCH CONFIGURATION 删除一个现有的文本搜索配置。要执行这个命令，必须是该配置的拥有者。

参数

IF EXISTS

如果该文本搜索配置不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索配置的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索配置的对象，然后删除所有　　依赖于那些对象的对象（见第　2.14　节　“依赖跟踪”。

RESTRICT

如果有任何对象依赖于该文本搜索配置，则拒绝删除它。这是默认值。

示例

移除文本搜索配置my_english：

DROP TEXT SEARCH CONFIGURATION my_english;

如果有任何在to_tsvector调用中引用该配置的　索引存在，这个命令都不会成功。增加CASCADE可以把这类索引与该文本搜索配置一起删除。

兼容性

SQL 标准中没有**DROP TEXT SEARCH CONFIGURATION** 语句。

另见

ALTER TEXT SEARCH CONFIGURATION(7), CREATE TEXT SEARCH CONFIGURATION(7)

名称

DROP TEXT SEARCH DICTIONARY — 移除一个文本搜索字典

大纲

DROP TEXT SEARCH DICTIONARY [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP TEXT SEARCH DICTIONARY删除一个 现有的文本搜索字典。要执行这个命令，必须是该字典的拥有者。

参数

IF EXISTS

如果该文本搜索字典不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索字典的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索字典的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该文本搜索字典，则拒绝删除它。这是默认值。

示例

移除文本搜索字典english：

```
DROP TEXT SEARCH DICTIONARY english;
```

如果有任何使用该字典的文本搜索配置存在，这个命令都不会成功。增加 CASCADE可以把这类配置与字典一起删除。

兼容性

SQL 标准中没有**DROP TEXT SEARCH DICTIONARY** 语句。

另见

ALTER TEXT SEARCH DICTIONARY(7), CREATE TEXT SEARCH DICTIONARY(7)

名称

DROP TEXT SEARCH PARSER — 移除一个文本搜索解析器

大纲

DROP TEXT SEARCH PARSER [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP TEXT SEARCH PARSER删除一个 现有的文本搜索解析器。必须作为一个超级用户来使用这个命令。

参数

IF EXISTS

如果该文本搜索解析器不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索解析器的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索解析器的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该文本搜索解析器，则拒绝删除它。这是默认值。

示例

移除文本搜索解析器my_parser：

```
DROP TEXT SEARCH PARSER my_parser;
```

如果有任何使用该解析器的文本搜索配置存在，这个命令都不会成功。增加 CASCADE可以把这类配置与解析器一起删除。

兼容性

SQL 标准中没有**DROP TEXT SEARCH PARSER** 语句。

另见

ALTER TEXT SEARCH PARSER(7), CREATE TEXT SEARCH PARSER(7)

名称

DROP TEXT SEARCH TEMPLATE — 移除一个文本搜索模板

大纲

DROP TEXT SEARCH TEMPLATE [IF EXISTS] *name* [CASCADE | RESTRICT]

描述

DROP TEXT SEARCH TEMPLATE删除一个 现有的文本搜索模板。必须作为一个超级用户来使用这个命令。

参数

IF EXISTS

如果该文本搜索模板不存在则不要抛出一个错误，而是发出一个提示。

name

一个现有文本搜索模板的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该文本搜索模板的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该文本搜索模板，则拒绝删除它。这是默认值。

示例

移除文本搜索模板thesaurus：

```
DROP TEXT SEARCH TEMPLATE thesaurus;
```

如果有任何使用该模板的文本搜索字典存在，这个命令都不会成功。增加 CASCADE可以把这类字典与模板一起删除。

兼容性

SQL 标准中没有**DROP TEXT SEARCH TEMPLATE** 语句。

另见

ALTER TEXT SEARCH TEMPLATE(7), CREATE TEXT SEARCH TEMPLATE(7)

名称

DROP TRANSFORM — 移除转换

大纲

DROP TRANSFORM [IF EXISTS] FOR *type_name* LANGUAGE *lang_name* [CASCADE | RESTRICT]

简介

DROP TRANSFORM移除一个之前定义的转换。

为了删除一种转换，必须拥有该类型和语言。这些同样也是创建转换所需要的 特权。

参数

IF EXISTS

如果该转换不存在也不要抛出一个错误。这种情况下会发出一个提示。

type_name

该转换的数据类型的名称。

lang_name

该转换的语言的名称。

CASCADE

自动删除依赖于该转换的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该转换，则拒绝删除它。这是默认行为。

示例

要删除用于类型hstore和语言plpythonu 的转换：

```
DROP TRANSFORM FOR hstore LANGUAGE plpythonu;
```

兼容性

这种形式的**DROP TRANSFORM**是一种 UXDB扩展。详见 CREATE TRANSFORM(7)。

另见

CREATE TRANSFORM(7)

名称

DROP TRIGGER — 移除一个触发器

大纲

DROP TRIGGER [IF EXISTS] *name* ON *table_name* [CASCADE | RESTRICT]

描述

DROP TRIGGER移除一个现有的触发器定义。 要执行这个命令，当前用户必须是触发器基表的拥有者。

参数

IF EXISTS

如果该触发器不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的触发器的名称。

table_name

定义了该触发器的表的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该触发器的对象，然后删除所有 依赖于那些对象的对象（见第 2.14 节“依赖跟踪”）。

RESTRICT

如果有任何对象依赖于该触发器，则拒绝删除它。这是默认值。

示例

销毁表films上的触发器 if_dist_exists：

```
DROP TRIGGER if_dist_exists ON films;
```

兼容性

UXDB中的 **DROP TRIGGER**语句与 SQL 标准不 兼容。在 SQL 标准中，不同表上也不能有同名的触发器，因此其 命令是简单的DROP TRIGGER *name*.

另见

CREATE TRIGGER(7)

名称

DROP TYPE — 移除一个数据类型

大纲

DROP TYPE [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP TYPE移除一种用户定义的数据类型。 只有一个类型的拥有者才能移除它。

参数

IF EXISTS

如果该类型不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的数据类型的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该类型的对象（例如表列、函数、操作符），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该类型，则拒绝删除它。这是默认值。

示例

要移除数据类型box：

```
DROP TYPE box;
```

兼容性

这个命令类似于 SQL 标准中的对应命令，但IF EXISTS子句 是一个UXDB扩展。但要注意 UXDB中**CREATE TYPE**命令的很大部分以及数据类 型扩展机制都与 SQL 标准不同。

另见

ALTER TYPE(7), CREATE TYPE(7)

名称

DROP USER — 移除一个数据库角色

大纲

DROP USER [IF EXISTS] *name* [, ...]

描述

DROP USER现在是 DROP ROLE(7)另一种写法。

兼容性

DROP USER语句是一个 UXDB扩展。SQL 标准把 用户的定义留给具体实现自行解释。

另见

DROP ROLE(7)

名称

DROP USER MAPPING — 移除一个用于外部服务器的用户映射

大纲

DROP USER MAPPING [IF EXISTS] FOR { *user_name* | USER | CURRENT_USER | PUBLIC }
SERVER *server_name*

描述

DROP USER MAPPING从外部服务器移除 一个已有的用户映射。

一个外部服务器的拥有者可以为该服务器的任何用户删除用户映射。如果 该服务器上的USAGE特权被授予了一个用户，它也能删除 用于它们自己的用户名的用户映射。

参数

IF EXISTS

如果该用户映射不存在则不要抛出一个错误，而是发出一个提示。

user_name

该映射的用户名。CURRENT_USER 和USER匹配当前用户的名称。PUBLIC 被用来匹配系统中所有现存和未来的用户名。

server_name

用户映射的服务器名。

示例

删除一个用户映射bob（服务器foo），如果它存在：

```
DROP USER MAPPING IF EXISTS FOR bob SERVER foo;
```

兼容性

DROP USER MAPPING符合 ISO/IEC 9075-9 (SQL/MED)。IF EXISTS子句是一个 UXDB扩展。

另见

CREATE USER MAPPING(7), ALTER USER MAPPING(7)

名称

DROP VIEW — 移除一个视图

大纲

DROP VIEW [IF EXISTS] *name* [, ...] [CASCADE | RESTRICT]

描述

DROP VIEW删除一个现有的视图。要执行 这个命令必须是该视图的拥有者。

参数

IF EXISTS

如果该视图不存在则不要抛出一个错误，而是发出一个提示。

name

要移除的视图的名称（可以是模式限定的）。

CASCADE

自动删除依赖于该视图的对象（例如其他视图），然后删除所有 依赖于那些对象的对象（见第 2.14 节 “依赖跟踪”。

RESTRICT

如果有任何对象依赖于该视图，则拒绝删除它。这是默认值。

示例

这个命令将移除名为kinds的视图：

```
DROP VIEW kinds;
```

兼容性

这个命令符合 SQL 标准，不过该标准只允许在每个命令中删除一个视图 并且没有IF EXISTS选项。该选项是一个 UXDB扩展。

另见

ALTER VIEW(7), CREATE VIEW(7)

名称

END — 提交当前事务

大纲

```
END [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

描述

END提交当前事务。所有该事务做的更改便得 对他人可见并且被保证发生崩溃时仍然是持久的。这个命令是一种 UXDB扩展，它等效于 COMMIT(7)。

参数

WORK
TRANSACTION

可选关键词，它们没有效果。

AND CHAIN

如果规定了AND CHAIN，则立即启动与刚完成事务具有相同事务特征(参见 SET TRANSACTION(7))的新事务。 否则，没有新事务被启动。

注解

使用ROLLBACK(7)可以中止一个事务。

当不在一个事务中时发出**END**没有危害，但是会 产生一个警告消息。

示例

要提交当前事务并且让所有更改持久化：

```
END;
```

兼容性

END是一种 UXDB扩展，它提供和 COMMIT(7)等效的功能，后者在 SQL 标准中指定。

另见

BEGIN(7), COMMIT(7), ROLLBACK(7)

名称

EXECUTE — 执行一个预备语句

大纲

EXECUTE *name* [(*parameter* [, ...])]

描述

EXECUTE被用来执行一个之前准备好的语句。 由于预备语句只在会话期间存在，该预备语句必须在当前会话中由一个更早 执行的**PREPARE**语句所创建。

如果创建预备语句的**PREPARE**语句指定了一些参数， 必须向**EXECUTE**语句传递一组兼容的参数，否则会 发生错误。注意（与函数不同）预备语句无法基于其参数的类型或者数量重载。 在一个数据库会话中，预备语句的名称必须唯一。

更多创建和使用预备语句的信息请见PREPARE(7)。

参数

name

要执行的预备语句的名称。

parameter

给预备语句的参数的实际值。这必须是一个能得到与该参数数据类型（ 在预备语句创建时决定）兼容的值的表达式。

输出

EXECUTE返回的命令标签是预备语句的命令标签而不是 EXECUTE。

例子

在PREPARE(7)文档的“例子”一节小节给出了例子。

兼容性

SQL 标准包括了一个**EXECUTE**语句， 但是只被用于嵌入式 SQL。这个版本的 **EXECUTE**语句也用了一种有点不同的语法。

另见

DEALLOCATE(7), PREPARE(7)

名称

EXPLAIN — 显示一个语句的执行计划

大纲

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

这里 *option*可以是：

```
ANALYZE [ boolean ]
VERBOSE [ boolean ]
COSTS [ boolean ]
SETTINGS [ boolean ]
BUFFERS [ boolean ]
TIMING [ boolean ]
SUMMARY [ boolean ]
FORMAT { TEXT | XML | JSON | YAML }
```

描述

这个命令显示UXDB计划器为提供的语句所生成的执行计划。该执行计划会显示将怎样扫描语句中引用的表 — 普通的顺序扫描、索引扫描等等 — 以及在引用多个表时使用何种连接算法来把来自每个输入表的行连接在一起。

显示中最重要的部分是估计出的语句执行代价，它是计划器对于该语句要运行多久的猜测（以任意的代价单位度量，但是习惯上表示取磁盘页面的次数）。事实上会显示两个数字：在第一行能被返回前的启动代价，以及返回所有行的总代价。对于大部分查询来说总代价是最重要的，但是在一些情景中（如EXISTS中的子查询），计划器将选择更小的启动代价来代替最小的总代价（因为执行器将在得到一行后停止）。此外，如果用一个LIMIT子句限制返回行的数量，计划器会在终端代价之间做出适当的插值来估计到底哪个计划是真正代价最低的。

ANALYZE选项导致该语句被实际执行，而不仅仅是被计划。那么实际的运行时间统计会被显示出来，包括在每个计划结点上花费的总时间（以毫秒计）以及它实际返回的行数。这对观察计划器的估计是否与实际相近很有用。

> **重要**
>
> 记住当使用了ANALYZE选项时语句会被实际执行。尽管EXPLAIN将丢弃SELECT所返回的任何输出，照例该语句的其他副作用还是会发生。如果希望在INSERT、UPDATE、DELETE、CREATE TABLE AS或者EXECUTE上使用EXPLAIN ANALYZE而不希望它们影响数据，可以使用下面的方法：
>
> ```
> BEGIN;
> EXPLAIN ANALYZE ...;
> ROLLBACK;
> ```

只有ANALYZE和VERBOSE选项能被指定，并且必须按照上述的顺序，不要把选项列表放在圆括号内。只支持没有圆括号的语法。我们期望所有新的选项将只在圆括号语法中支持。

参数

ANALYZE

执行命令并且显示实际的运行时间和其他统计信息。这个参数默认被设置为FALSE。

VERBOSE

显示关于计划的额外信息。特别是：计划树中每个结点的输出列列表、模式限定的表和函数名、总是把表达式中的变量标上它们的范围表别名，以及总是打印统计信息被显示的每个触发器的名称。这个参数默认被设置为FALSE。

COSTS

包括每一个计划结点的估计启动和总代价，以及估计的行数和每行的宽度。这个参数默认被设置为TRUE。

SETTINGS

包括有关配置参数的信息。具体来说，包括影响查询计划的选项，其值与内置默认值不同。此参数默认为FALSE。

BUFFERS

包括缓冲区使用的信息。特别是：共享块命中、读取、标记为脏和写入的次数、本地块命中、读取、标记为脏和写入的次数、以及临时块读取和写入的次数。一次命中表示避免了一次读取，因为需要的块已经在缓存中找到了。共享块包含着来自于常规表和索引的数据，本地块包含着来自于临时表和索引的数据，而临时块包含着在排序、哈希、物化计划结点和类似情况中使用的短期工作数据。脏块的数量表示被这个查询改变的之前未被修改块的数量，而写入块的数量表示这个后台在查询处理期间从缓存中替换出去的脏块的数量。为一个较高层结点显示的块数包括它的所有子结点所用到的块数。在文本格式中，只会打印非零值。只有当ANALYZE也被启用时，这个参数才能使用。它的默认被设置为FALSE。

TIMING

在输出中包括实际启动时间以及在每个结点中花掉的时间。反复读取系统时钟的负荷在某些系统上会显著地拖慢查询，因此在只需要实际的行计数而不是实际时间时，把这个参数设置为FALSE可能会有用。即便用这个选项关闭结点层的计时，整个语句的运行时间也总是会被度量。只有当ANALYZE也被启用时，这个参数才能使用。它的默认被设置为TRUE。

SUMMARY

在查询计划之后包含摘要信息（例如，总计的时间信息）。当使用ANALYZE 时默认包含摘要信息，但默认情况下不包含摘要信息，但可以使用此选项启用摘要信息。 使用EXPLAIN EXECUTE中的计划时间包括从缓存中获取计划所需的时间 以及重新计划所需的时间（如有必要）。

FORMAT

指定输出格式，可以是 TEXT、XML、JSON 或者 YAML。非文本输出包含和文本输出格式相同的信息，但是更容易被程序解析。这个参数默认被设置为TEXT。

boolean

指定被选中的选项是否应该被打开或关闭。可以写TRUE、ON或1来启用选项，写FALSE、OFF或0禁用它。*boolean*值也能被忽略，在这种情况下会假定值为TRUE。

statement

想查看其执行计划的任何SELECT、INSERT、UPDATE、DELETE、VALUES、EXECUTE、DECLARE、CREATE TABLE AS或者CREATE MATERIALIZED VIEW AS语句。

输出

这个命令的结果是为*statement*选中的计划的文本描述，可能还标注了执行统计信息。第 11.1 节“使用EXPLAIN”描述了所提供的信息。

注解

为了允许UXDB查询计划器在优化查询时能做出合理的知情决策，查询中用到的所有表的ux_statistic数据应该能保持为最新。通常这个工作会由autovacuum daemon负责自动完成。但是如果一个表最近在内容上有大量的改变，我们可能需要做一次手动的ANALYZE(7)而不是等待autovacuum 捕捉这些改变。

为了度量执行计划中每个节点的运行时成本，当前的EXPLAIN ANALYZE的当前实现为查询执行增加了性能分析开销。 这样，在一个查询上运行EXPLAIN ANALYZE有时候比正常执行该查询要慢很多。 开销的量取决于该查询的性质，以及使用的平台。最坏的情况会发生在那些自身执行时间很短的结点上，以及在那些具有相对较慢的有关时间的操作系统调用的机器上。

例子

有一个具有单个integer列和 10000 行的表，要显示在其上的一个简单查询的计划：

```
EXPLAIN SELECT * FROM foo;

                       QUERY PLAN
-------------------------------------------------------------
 Seq Scan on foo  (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

这里有同样一个查询的 JSON 输出格式：

```
EXPLAIN (FORMAT JSON) SELECT * FROM foo;
        QUERY PLAN
--------------------------------
[                          +
  {                        +
    "Plan": {              +
      "Node Type": "Seq Scan",+
      "Relation Name": "foo", +
      "Alias": "foo",        +
      "Startup Cost": 0.00,  +
      "Total Cost": 155.00,  +
      "Plan Rows": 10000,    +
      "Plan Width": 4        +
    }                        +
  }                          +
]
```

```
(1 row)
```

如果有一个索引，并且我们使用了一个带有可索引WHERE条件的查询，EXPLAIN可能会显示一个不同的计划：

```
EXPLAIN SELECT * FROM foo WHERE i = 4;

                        QUERY PLAN
-----------------------------------------------------------------
 Index Scan using fi on foo  (cost=0.00..5.98 rows=1 width=4)
   Index Cond: (i = 4)
(2 rows)
```

这里是同一查询的 YAML 格式：

```
EXPLAIN (FORMAT YAML) SELECT * FROM foo WHERE i='4';
          QUERY PLAN
--------------------------------
 - Plan:                   +
     Node Type: "Index Scan"  +
     Scan Direction: "Forward"+
     Index Name: "fi"         +
     Relation Name: "foo"     +
     Alias: "foo"             +
     Startup Cost: 0.00       +
     Total Cost: 5.98         +
     Plan Rows: 1             +
     Plan Width: 4            +
     Index Cond: "(i = 4)"
(1 row)
```

XML 格式我们留给读者做练习。

这里是去掉了代价估计的同样一个计划：

```
EXPLAIN (COSTS FALSE) SELECT * FROM foo WHERE i = 4;

         QUERY PLAN
----------------------------
 Index Scan using fi on foo
   Index Cond: (i = 4)
(2 rows)
```

这里是一个使用聚集函数的查询的查询计划例子：

```
EXPLAIN SELECT sum(i) FROM foo WHERE i < 10;

                           QUERY PLAN
---------------------------------------------------------------------------
 Aggregate  (cost=23.93..23.93 rows=1 width=4)
   ->  Index Scan using fi on foo  (cost=0.00..23.92 rows=6 width=4)
```

```
      Index Cond: (i < 10)
(3 rows)
```

这里是一个使用EXPLAIN EXECUTE显示预备查询的执行计划的例子：

```
PREPARE query(int, int) AS SELECT sum(bar) FROM test
   WHERE id > $1 AND id < $2
   GROUP BY foo;

EXPLAIN ANALYZE EXECUTE query(100, 200);

                                                       QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=9.54..9.54 rows=1 width=8) (actual time=0.156..0.161 rows=11 loops=1)
   Group Key: foo
   ->  Index Scan using test_pkey on test  (cost=0.29..9.29 rows=50 width=8) (actual time=0.039..0.091
 rows=99 loops=1)
         Index Cond: ((id > $1) AND (id < $2))
 Planning time: 0.197 ms
 Execution time: 0.225 ms
(6 rows)
```

当然，这里显示的有关数字取决于表涉及到的实际内容。此外，ANALYZE命令使用随机采样来估计数据统计。因此，在一次新的ANALYZE运行之后，代价估计可能会改变，即便是表中数据的实际分布没有改变也是如此。

兼容性

在 SQL 标准中没有定义EXPLAIN语句。

参见

ANALYZE(7)

名称

FETCH — 使用游标从查询中检索行

大纲

FETCH [*direction* [FROM | IN]] *cursor_name*

其中 *direction* 可以为空或者以下之一：

NEXT
PRIOR
FIRST
LAST
ABSOLUTE *count*
RELATIVE *count*
count
ALL
FORWARD
FORWARD *count*
FORWARD ALL
BACKWARD
BACKWARD *count*
BACKWARD ALL

描述

FETCH从之前创建的一个游标中检索行。

游标具有一个相关联的位置，FETCH会用到该位置。 游标位置可能会位于查询结果的第一行之前、结果中任意行之上或者 结果的最后一行之后。在被创建时，游标被定位在第一行之前。在取出 一些行后，该游标被定位在最近被取出的行上。如果 FETCH运行超过了可用行的末尾，则该游标会被定位 在最后一行之后（如果向后取，则是第一行之前）。 FETCH ALL或者FETCH BACKWARD ALL将总是让游标被定位于最后一行之后或者第一行之前。

NEXT、PRIOR、FIRST、 LAST、ABSOLUTE、RELATIVE 形式会在适当移动游标后取出一行。如果没有这样一行，将返回一个空 结果，并且视情况将游标定位在第一行之前或者最后一行之后。

使用FORWARD和BACKWARD的形式会在 向前移动或者向后移动的方向上检索指定数量的行，然后将游标定位在 最后返回的行上（如果*count*超过可用的行数，则定位 在所有行之后或者之前）。

RELATIVE 0、FORWARD 0以及 BACKWARD 0都会请求检索当前行但不移动游标，也就是 重新取最近被取出的行。 只要游标没有被定位在第一行之前或者最后一行 之后，这种操作都会成功，否则不会返回任何行。

> **注意**
>
> 这个页面描述在 SQL 命令层面上对游标的使用。如果想要在 PL/uxSQL函数中使用游标，规则会有所不同。

参数

direction

direction 定义获取方向以及要取得的行数。它可以是下列之一：

NEXT

取出下一行。如果省略*direction*，这将是默认值。

PRIOR

取出当前位置之前的一行。

FIRST

取出该查询的第一行（和ABSOLUTE 1相同）。

LAST

取出该查询的最后一行（和ABSOLUTE -1相同）。

ABSOLUTE *count*

取出该查询的第*count*个行，如果*count*为负则是从尾部开始取出 第abs(*count*)个行。如果*count*超出范围，将定位在第一行 之前或者最后一行之后。特别地，ABSOLUTE 0 会定位在第一行之前。

RELATIVE *count*

取出第*count*个后继行，如果 *count*为负 则是取出前面的第abs(*count*)个行。 RELATIVE 0重新取出当前行（如果有）。

count

取出接下来的*count*行（和 FORWARD *count*相同）。

ALL

取出所有剩余的行（和FORWARD ALL相同）。

FORWARD

取出下一行（和NEXT相同）。

FORWARD *count*

取出接下来的*count*行。 FORWARD 0重新取出当前行。

FORWARD ALL

取出所有剩下的行。

BACKWARD

取出当前行前面的一行（和PRIOR相同）。

BACKWARD *count*

取出前面的*count*行（反向扫描）。 BACKWARD 0会重新取出当前行。

BACKWARD ALL

取出所有当前位置之前的行（反向扫描）。

count

count 是一个可能带有符号的整数常量，它决定要取得的位置或者行数。对于FORWARD和BACKWARD情况，指定一个负的 *count*等效于改变 FORWARDhe BACKWARD的意义。

cursor_name

一个已打开游标的名称。

输出

如果成功完成，FETCH命令返回一个下面形式的命令标签：

FETCH *count*

*count*是取得的行数（可能 为零）。注意在uxsql中，命令标签将不会实际显示， 因为uxsql会显示被取得的行。

注解

如果想要使用FETCH的任意变体而不使用带有正计数的 FETCH NEXT或者FETCH FORWARD， 应该用SCROLL声明游标。对于简单查询， UXDB将允许从不带 SCROLL的游标中反向取得行，但最好不要依赖这种行为。 如果游标被声明为带有SCROLL，则不允许反向取得。

用ABSOLUTE取行并不比用相对移动快多少：不管则样， 底层实现都必须遍历所有的中间行。用负绝对值获取的情况更糟：必须读到 查询尾部来找到最后一行，并且接着从那里反向开始遍历。不过，回卷到查询 的开始（正如FETCH ABSOLUTE 0）是很快的。

DECLARE(7)被用来定义游标。使用 MOVE(7)可改变游标位置而不检索数据。

示例

下面的例子用一个游标遍历一个表：

```
BEGIN WORK;

-- 建立一个游标：
DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- 在游标 liahona 中取出前 5 行：
FETCH FORWARD 5 FROM liahona;

 code |          title          | did | date_prod  |   kind   | len
```

```
-------+-------------------------+-----+------------+----------+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    | 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic | 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic | 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   | 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    | 02:28

-- 取出前面一行：
FETCH PRIOR FROM liahona;

 code  |  title  | did | date_prod  |  kind  |  len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- 关闭游标并且结束事务：
CLOSE liahona;
COMMIT WORK;
```

兼容性

SQL 标准只定义**FETCH**在嵌入式 SQL 中使用。 这里描述的**FETCH**变体返回数据时就好像数据是 一个**SELECT**结果，而不是被放在主变量中。 除这一点之外，**FETCH**完全向上兼容于 SQL 标准。

涉及FORWARD和BACKWARD的 **FETCH**形式，以及形式FETCH *count*和FETCH ALL（其中FORWARD是隐式的）都是 UXDB扩展。

SQL 标准只允许FROM在游标名之前。使用 IN的选项或者完全省去它们是一种扩展。

另见

CLOSE(7)，DECLARE(7)，MOVE(7)

名称

GRANT — 定义访问特权

大纲

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } routine_name [ ( [ [ argmode ] [ arg_name
 ] arg_type [, ...] ] ) ] [, ...]
         | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    TO role_specification [, ...] [ WITH GRANT OPTION ]
```

```
GRANT { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
   ON SCHEMA schema_name [, ...]
   TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { CREATE | ALL [ PRIVILEGES ] }
   ON TABLESPACE tablespace_name [, ...]
   TO role_specification [, ...] [ WITH GRANT OPTION ]

GRANT { USAGE | ALL [ PRIVILEGES ] }
   ON TYPE type_name [, ...]
   TO role_specification [, ...] [ WITH GRANT OPTION ]
```

其中*role_specification*可以是：

```
   [ GROUP ] role_name
 | PUBLIC
 | CURRENT_USER
 | SESSION_USER
```

```
GRANT role_name [, ...] TO role_name [, ...] [ WITH ADMIN OPTION ]
```

描述

GRANT命令由两种基本的变体：一种授予在一个数据库对象（表、列、视图、外部表、序列、数据库、外部数据包装器、外部服务器、函数、过程、过程语言、模式或表空间）上的特权，另一个授予一个角色中的成员关系。这些变体在很多方面都相似，但是也有很多不同，所以还是得分别描述它们。

在数据库对象上 GRANT

这种**GRANT**命令的变体将一个数据库对象上的指定特权交给一个或多个角色。如果有一些已经被授予，这些特权会被加入到它们之中。

关键词PUBLIC指示特权要被授予给所有角色，包括那些可能稍后会被创建的角色。PUBLIC可以被认为是一个被隐式定义的总是包含所有角色的组。任何特定角色都将具有直接授予给它的特权、授予给它作为成员所在的任何角色的特权以及被授予给PUBLIC的特权。

如果指定了WITH GRANT OPTION，特权的接收者可以接着把它授予给其他人。如果没有授权选项，接收者就不能这样做。授权选项不能被授予给PUBLIC。

没有必要把权限授予给一个对象的拥有者（通常就是创建该对象的用户）， 因为拥有者默认具有所有的特权（不过拥有者可能为了安全选择撤回一些 它们自己的特权）。

删除一个对象或者以任何方式修改其定义的权力是不被当作一个可授予特权的，它被固化在拥有者中，并且不能被授予和撤回（不过，相似地效果可以通过授予或者撤回在拥有该对象的角色中的成员关系来实现，见下文）。拥有者也隐式地拥有该对象的所有授权选项。

The possible privileges are:

SELECT
INSERT
UPDATE
DELETE
TRUNCATE
REFERENCES
TRIGGER
CREATE
CONNECT
TEMPORARY
EXECUTE
USAGE

特定类型的权限，如 第 2.7 节 “权限”中所定义。

TEMP

TEMPORARY的替代拼写。

ALL PRIVILEGES

授予对象的类型可用的所有权限。PRIVILEGES关键词在UXDB中是可选的，尽管它是严格的 SQL 所需要的。

FUNCTION语法适用于简单函数、聚合函数和窗口函数，但不适用于过程；对过程使用PROCEDURE。 或者，使用ROUTINE来引用函数、聚合函数、窗口函数或过程而不管其精确类型。

还有一个选项，可以在一个或多个模式中对所有相同类型的对象授予特权。此功能当前仅支持表、序列、函数和过程。 ALL TABLES也会影响视图和外表，就像特定对象 GRANT命令。ALL FUNCTIONS也会影响聚合和窗口函数，但不影响过程，就像特定对象GRANT命令一样。 使用 ALL ROUTINES 来包括过程。

角色上的 GRANT

GRANT命令的这种变体把一个角色中的成员关系授予一个或者多个其他角色。一个角色中的成员关系是有意义的，因为它会把授予给一个角色的特权带给该角色的每一个成员。

如果指定了WITH ADMIN OPTION，成员接着可以把该角色中的成员关系授予给其他用户，也可以撤回该角色中的成员关系。如果没有管理选项，普通用户就不能做这些工作。一个角色不被认为持有自身的WITH ADMIN OPTION，但是它可以从一个会话用户匹配该角色的数据库会话中授予或撤回自身中的成员关系。数据库超级用户能够授予或撤回任何角色中任何人的成员关系。具有CREATEROLE特权的角色能够授予或者撤回任何非超级用户角色中的成员关系。

和特权的情况不同，一个角色中的成员关系不能被授予PUBLIC。还要注意这种形式的命令不允许噪声词GROUP。

注解

REVOKE(7)命令被用来撤回访问特权。

用户和组的概念已经被统一到一种单一类型的实体（被称为一个角色）。因此不再需要使用关键词GROUP来标识一个被授权者是一个用户或者一个组。在该命令中仍然允许GROUP，但是它只是一个噪音词而已。

如果一个用户持有特定列或者其所在的整个表的特权， 该用户可以在该列上执行SELECT、INSERT等命令。在表层面上授予特权 然后对一列撤回该特权将不会按照希望的运作： 表级别的授权不会受到列级别操作的影响。

当一个对象的非拥有者尝试GRANT该对象上的特权，如果该用户在该对象上什么特权都不拥有，该命令将立刻失败。只要有一些特权可用，该命令将继续，但是它将只授予那些用户具有授权选项的特权。如果不持有授权选项，GRANT ALL PRIVILEGES形式将发出一个警告消息。而如果不持有命令中特别提到的任何特权的授权选项，其他形式将会发出一个警告（原则上这些语句也适用于对象拥有者，但是由于拥有者总是被视为持有所有授权选项，因此这种情况不会发生）。

需要注意的是，数据库超级用户可以访问所有对象而不管对象特权的设置。这可与 Unix 系统中的root权力相提并论。对于root来说，除非绝对必要，使用一个超级用户来操作是不明智的。

如果一个超级用户选择发出一个GRANT或者REVOKE命令，该命令将被执行，好像它是由被影响对象的拥有者发出的一样。特别地，通过这样一个命令授予的特权将好像是由对象拥有者授予的一样（对于角色成员关系，该成员关系好像是由该角色本身授予的一样）。

GRANT以及REVOKE也可以由一个不是受影响对象拥有者的角色完成，不过该角色是拥有该对象的角色的一个成员，或者是在该对象上持有特权的WITH GRANT OPTION的角色的一个成员。在这种情况下，特权将被记录为由实际拥有该对象的角色授予或者是由持有特权的WITH GRANT OPTION的角色授予。例如，如果表t1被角色g1拥有，u1是它的一个成员，那么u1可以把t1上的特权授予给u2，但是那些特权将好像是直接由g1授予的。角色g1的任何其他成员可以稍后撤回它们。

如果执行GRANT的角色间接地通过多于一条角色成员关系路径持有所需的特权，将不会指定哪一个包含它的角色将被记录为完成了该授权。在这样的情况中，最好使用SET ROLE来成为想用其做GRANT的特定角色。

授予一个表上的权限不会自动地扩展权限给该表使用的任何序列，包括绑定在SERIAL列上的序列。序列上的权限必须被独立设置。

有关特定的特权类型以及如何检查对象特权的更多信息，请参见第 2.7 节 “权限”。

例子

把表films上的插入特权授予给所有用户：

```
GRANT INSERT ON films TO PUBLIC;
```

把视图kinds上的所有可用特权授予给用户manuel：

```
GRANT ALL PRIVILEGES ON kinds TO manuel;
```

注意虽然上述语句被一个超级用户或者kinds的拥有者执行时确实会授予所有特权，但是当由其他人执行时将只会授予那些执行者拥有授权选项的权限。

把角色admins中的成员关系授予给用户joe：

```
GRANT admins TO joe;
```

兼容性

根据 SQL 标准，ALL PRIVILEGES中的PRIVILEGES关键词是必须的。SQL 标准不支持在每个命令中设置超过一个对象上的特权。

UXDB允许一个对象拥有者 撤回它们拥有的普通特权：例如，一个表拥有者可以通过撤回其自身拥有 的INSERT、UPDATE、DELETE 和TRUNCATE特权让该表对它们自己只读。根据 SQL 标准 这是不可能发生的。原因在于UXDB 认为拥有者的特权是由拥有者授予给它们自己的，因此它们也能够撤回它们。 在 SQL 标准中，拥有者的特权是有一个假设的实体“_SYSTEM”所授予。由于不是“_SYSTEM”，拥有者就不能撤回这些权力。

根据 SQL 标准，授权选项可以被授予给PUBLIC， UXDB 只支持把授权选项授予给角色。

SQL 标准提供了其他对象类型上的USAGE特权：字符集、排序规则、翻译。

在 SQL 标准中，序列只有一个USAGE特权，它控制NEXT VALUE FOR表达式的使用，该表达式等效于 UXDB 中的函数nextval。序列的特权SELECT和UPDATE是 UXDB 扩展。应用序列的USAGE特权到currval函数也是一个 UXDB 扩展（该函数本身也是）。

数据库、表空间、模式和语言上的特权都是UXDB扩展。

参见

REVOKE(7), ALTER DEFAULT PRIVILEGES(7)

名称

IMPORT FOREIGN SCHEMA — 从一个外部服务器导入表定义

大纲

```
IMPORT FOREIGN SCHEMA remote_schema
  [ { LIMIT TO | EXCEPT } ( table_name [, ...] ) ]
  FROM SERVER server_name
  INTO local_schema
  [ OPTIONS ( option 'value' [, ... ] ) ]
```

简介

IMPORT FOREIGN SCHEMA创建表示存在于 外部服务器上的表的外部表。新外部表将由发出该命令的用户所拥有并且用 匹配远程表的正确的列定义和选项创建。

默认情况下，存在于外部服务器上一个特定模式中的所有表和视图都会被导入。 根据需要，表的列表可以被限制到一个指定的子集，或者可以排除特定的表。 新外部表都被创建在一个必须已经存在的目标模式中。

要使用**IMPORT FOREIGN SCHEMA**，用户必 须具有外部服务器上的USAGE特权以及在目标模式上的CREATE特权。

参数

remote_schema

要从哪个远程模式导入。一个远程模式的特定含义依赖于所使用的外部数据包装器。

LIMIT TO (*table_name* [, ...])

只导入匹配给定表名之一的外部表。外部模式中其他的表将被忽略。

EXCEPT (*table_name* [, ...])

把指定的外部表排除在导入之外。除了列在这里的表之外，外部模式 中存在的所有表都将被导入。

server_name

要从哪个外部服务器导入。

local_schema

被导入的外部表将创建在其中的模式。

OPTIONS (*option* '*value*' [, ...])

要在导入期间使用的选项。允许使用的选项名称和值与每一个外部数据包装器 有关。

示例

从服务器film_server上的远程模式foreign_films 中导入表定义，把外部表创建在本地模式films中：

```
IMPORT FOREIGN SCHEMA foreign_films
   FROM SERVER film_server INTO films;
```

同上，但是只导入两个表actors和 directors（如果存在）：

```
IMPORT FOREIGN SCHEMA foreign_films LIMIT TO (actors, directors)
   FROM SERVER film_server INTO films;
```

兼容性

IMPORT FOREIGN SCHEMA命令符合 SQL标准，不过OPTIONS 子句是一种UXDB扩展。

另见

CREATE FOREIGN TABLE(7), CREATE SERVER(7)

名称

INSERT — 在一个表中创建新行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ AS alias ] [ ( column_name [, ...] ) ]
    [ OVERRIDING { SYSTEM | USER } VALUE ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ ON CONFLICT [ conflict_target ] conflict_action ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

其中 *conflict_target* 可以是以下之一：

```
    ( { index_column_name | ( index_expression ) } [ COLLATE collation ] [ opclass ] [, ...] )
  [ WHERE index_predicate ]
    ON CONSTRAINT constraint_name
```

并且 *conflict_action* 是以下之一：

```
    DO NOTHING
    DO UPDATE SET { column_name = { expression | DEFAULT } |
                    ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
                    ( column_name [, ...] ) = ( sub-SELECT )
                  } [, ...]
              [ WHERE condition ]
```

描述

INSERT将新行插入到一个表中。我们可以 插入一个或者更多由值表达式指定的行，或者插入来自一个查询的零行 或者更多行。

目标列的名称可以以任意顺序列出。如果没有给出列名列表，则有两种确定 目标列的可能性。第一种是以被声明的顺序列出该表的所有列。另一种可能 性是，如果VALUES 子句或者*query*只提 供*N*个列，则以被声明的顺序列出该表的前 *N*列。VALUES 子句或者 *query*提供的值会被从左至右关联到这些显式或者隐式 给出的目标列。

每一个没有出现在显式或者隐式列列表中的列都将被默认填充，如果为该列 声明过默认值则用默认值填充，否则用空值填充。

如果任意列的表达式不是正确的数据类型，将会尝试自动类型转换。

ON CONFLICT可以用来指定发生唯一约束或者排除约束 违背错误时的替换动作（见下文的“ON CONFLICT 子句”一节）。

可选的RETURNING子句让INSERT根据 实际被插入（如果使用了ON CONFLICT DO UPDATE子句， 可能是被更新）的每一行来计算和返回值。这主要用来获取由默认值提供 的值，例如一个序列号。不过，允许在其中包括使用该表列的任何表达式。 RETURNING列表的语法与SELECT的输出 列表的相同。只有被成功地插入或者更新的行才将被返回。例如，如果一 行被锁定但由于不满足ON CONFLICT DO UPDATE ... WHERE clause *condition*没有被更新，该行将不被返回。

为了向表中插入，必须具有其上的INSERT特权。 如果存在ON CONFLICT DO UPDATE子句，还要求该表上 的UPDATE特权。

如果一个列列表被指定，只需要其中的列上的INSERT 特权。类似地，在指定了ON CONFLICT DO UPDATE时，只 需要被列出要更新的列上的UPDATE特权。不过， ON CONFLICT DO UPDATE还要求其值被 ON CONFLICT DO UPDATE表达式或者 *condition*使用的列上的SELECT特权。

使用RETURNING子句需要RETURNING中提到的所有列的 SELECT权限。 如果使用*query*子句从查询中插入行， 则当然需要对查询中使用的任何表或列具有SELECT权限。

参数

插入

这个小节介绍了在只插入新行时可以使用的参数。 专门用于ON CONFLICT子句的 参数会单独介绍。

with_query

WITH子句允许指定一个或者更多子查询，在 INSERT查询中可以用名称引用这些子查询。详见 第 4.8 节 “WITH查询（公共表表达式）”以及SELECT(7)。

query （SELECT语句）也可以包含一个 WITH子句。在这种情况下 *query*中可以引用 两组*with_query*，但是第二个优先级更 高（因为它被嵌套更近）。

table_name

一个已有表的名称（可以被模式限定）。

alias

table_name 的替补名称。当提供了一个别名时，它会完全隐藏掉表的实际名称。 当ON CONFLICT DO UPDATE的目标是一个被排除的表时这特别有用，因为那将被当作表示要被插入行的特殊表的名称。

column_name

名为*table_name*的表中的一个列 的名称。如有必要，列名可以用一个子域名或者数组下标限定（指向 一个组合列的某些列中插入会让其他域为空）。当用 ON CONFLICT DO UPDATE引用一列时，不要在一个 目标列的说明中国包括表名。例如， INSERT INTO table_name ... ON CONFLICT DO UPDATE SET table_name.col = 1是非法的（这遵循UPDATE 的一般行为）。

OVERRIDING SYSTEM VALUE

如果没有这个子句，为定义为GENERATED ALWAYS的标识列指定一个明确的值（不是DEFAULT）就是一种错误。这个子句推翻了这种限制。

OVERRIDING USER VALUE

如果指定这个子句，则会忽略提供给定义为GENERATED BY DEFAULT的标识列的值，并且应用默认的由序列生成的值。

例如，当在表之间拷贝值时，这个子句有能派上用场。INSERT INTO tbl2 OVERRIDING USER VALUE SELECT * FROM tbl1将从tbl1中拷贝所有在tbl2中不是标识列的列，而tbl2中标识列的值将由与tbl2关联的序列产生。

DEFAULT VALUES

所有列都将被其默认值填充（例如这种形式下不允许OVERRIDING子句）。

expression

要赋予给相应列的表达式或者值。

DEFAULT

相应的列将被其默认值填充。

query

提供要被插入行的查询（SELECT语句）。 其语法描述请参考SELECT(7)语句。

output_expression

在每一行被插入或更新后由INSERT命令计算并且返回的 表达式。该表达式可以使用*table_name* 指定的表中的任何列。写成*可返回被插入或更新行的所有列。

output_name

要用于被返回列的名称。

ON CONFLICT 子句

可选的ON CONFLICT子句为出现唯一性违背或排除 约束违背错误时提供另一种可供选择的动作。对于每一个要插入的行， 不管是插入进行下去还是由*conflict_target* 指定的一个仲裁者约束或者索引被违背，都会 采取可供选择的*conflict_action*。 ON CONFLICT DO NOTHING简单地把避免插入行。 ON CONFLICT DO UPDATE则会 更新与要插入的行冲突的已有行。

*conflict_target*可以执行 唯一索引推断。在执行推断时，它由一个或者多个 *index_column_name* 列或者 *index_expression* 表达式以及一个可选的 *index_predicate*构成。所有刚好包含*conflict_target*指定的列/表达式的*table_name*唯一索引（不管顺序）都 会被推断为（选择为）仲裁者索引。如果指定了 *index_predicate*，它 必须满足仲裁者索引（也是推断过程的一个进一步的要求）。注意这意味着如果 有一个满足其他条件的非部分唯一索引（没有谓词的唯一索引）可用，它将被 推断为仲裁者（并且会被ON CONFLICT使用）。如果推断 尝试不成功，则会发生一个错误。

ON CONFLICT DO UPDATE保证一个原子的 INSERT或者 UPDATE结果。在没有无关错误的前提下，这两种 结果之一可以得到保证，即使在很高的并发度也能保证。这也可以被称作 UPSERT — “UPDATE 或 INSERT”。

conflict_target

通过选择仲裁者索引来指定哪些行与 ON CONFLICT在其上采取可替代动作的行相冲突。 要么执行唯一索引推断，要么显式命名一个 约束。对于ON CONFLICT DO NOTHING来说，它对于指定一个*conflict_target*是可选的。 在被省略时，与所有有效约束（以及唯一索引）的冲突都会被处理。对于 ON CONFLICT DO UPDATE，必须 提供一个*conflict_target*。

conflict_action

*conflict_action*指定一个可替换的 ON CONFLICT动作。它可以是 DO NOTHING，也可以是一个指定在冲突情况下 要被执行的UPDATE动作细节的DO UPDATE子句。ON CONFLICT DO UPDATE中的SET和 WHERE子句能够使用该表的名称（或者别名） 访问现有的行，并且可以用特殊的被排除 表访问要插入的行。这个动作要求被排除 列所在目标表的任何列上的SELECT特权。

注意所有行级BEFORE INSERT触发器的效果都会 反映在被排除值中，因为那些效果可能会让该行避免被插入。

index_column_name

一个*table_name*列 的名称。它被用来推断仲裁者索引。它遵循CREATE INDEX格式。这要求 *index_column_name* 上的SELECT特权。

index_expression

和*index_column_name*类似，但是 被用来推断出现在索引定义中的*table_name*列（非简单列）上的 表达式。遵循CREATE INDEX格式。这要求 任何出现在*index_expression*中的列上的SELECT特权。

collation

指定时，强制相应的*index_column_name*或 *index_expression* 使用一种特定的排序规则以便在推断期间能被匹配上。通常 会被省略，因为排序规则通常不会影响约束违背的发生。遵循CREATE INDEX格式。

opclass

指定时，强制相应的*index_column_name*或 *index_expression* 使用特定的操作符类以便在推断期间能被匹配上。通常会被省略， 因为相等语义在一种类型的操作符类 之间都是等价的，或者因为足以信任已定义的唯一索引具有适当的 相等定义。遵循CREATE INDEX格式。

index_predicate

用于允许推断部分唯一索引。任何满足该谓词（不一定需要真的是 部分索引）的索引都能被推断。遵循CREATE INDEX格式。这要求任何出现在*index_predicate*中的列上 的SELECT特权。

constraint_name

用名称显式地指定一个仲裁者约束， 而不是推断一个约束或者索引。

condition

一个能返回boolean值的表达式。只有让这个表达式返回 true的行才将被更新，不过在采用ON CONFLICT DO UPDATE动作时所有的行都会被锁定。 注意*condition*会被最后计算，即一个冲突 被标识为要更新的候选对象之后。

注意不支持把排除约束作为ON CONFLICT DO UPDATE的 仲裁者。在所有的情况中，只支持NOT DEFERRABLE约束和 唯一索引作为仲裁者。

带有ON CONFLICT DO UPDATE子句的 INSERT是一种“确定性的” 语句。这表明不允许该命令影响任何单个现有行超过一次，如果发生则会 发生一个基数违背错误。要插入的行不应该在仲裁者索引或约束所限制的 属性上相重复。

注意，当前不支持用分区表上的INSERT的ON CONFLICT DO UPDATE子句更新冲突行的分区键，因为那样会让行移动到新的分区中。

> **提示**
>
> 使用唯一索引推断通常比使用ON CONFLICT ON CONSTRAINT *constraint_name*直接提名一个约束更好。当底层索引被以重叠方式替换成另一个或多或少等效的索引时，推断将能继续正确地工作，例如在删除要被替换的索引之前使用CREATE UNIQUE INDEX ... CONCURRENTLY。

输出

成功完成时，INSERT命令会返回以下形式的命令标签：

INSERT *oid count*

*count*是被插入或更新的行数。 *oid*总是0（过去，如果*count*恰好为1， 并且目标表被声明为WITH OIDS，则它是分配给插入行的OID， 否则为0， 但现在已不再支持创建WITH OIDS表）。

如果INSERT命令包含RETURNING子句， 其结果会类似于包含RETURNING列表中定义的列和值的 SELECT语句，这些结果是由该命令在被插入或更新行上 计算得到。

注解

如果指定的表是一个分区表，每一行都会被路由到合适的分区并且插入其中。如果指定的表是一个分区，如果输入行之一违背该分区的约束则将发生错误。

示例

向films中插入一行：

```
INSERT INTO films VALUES
  ('UA502', 'Bananas', 105, '1971-07-13', 'Comedy', '82 minutes');
```

在这个例子中，len列被省略并且因此会具有默认值：

```
INSERT INTO films (code, title, did, date_prod, kind)
  VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

这个例子为日期列使用DEFAULT子句而不是指定一个值：

```
INSERT INTO films VALUES
  ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes');
INSERT INTO films (code, title, did, date_prod, kind)
  VALUES ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama');
```

插入一个完全由默认值构成的行：

```
INSERT INTO films DEFAULT VALUES;
```

用多行VALUES语法插入多个行：

```
INSERT INTO films (code, title, did, date_prod, kind) VALUES
  ('B6717', 'Tampopo', 110, '1985-02-10', 'Comedy'),
  ('HG120', 'The Dinner Game', 140, DEFAULT, 'Comedy');
```

这个例子从表tmp_films中获得一些行插入到表 films中，两个表具有相同的列布局：

```
INSERT INTO films SELECT * FROM tmp_films WHERE date_prod < '2004-05-07';
```

这个例子插入数组列：

```
-- 为 noughts-and-crosses 游戏创建一个空的 3x3 棋盘
INSERT INTO tictactoe (game, board[1:3][1:3])
  VALUES (1, '{{" "," "," "},{" "," "," "},{" "," "," "}}');
-- 实际上可以不用上面例子中的下标
INSERT INTO tictactoe (game, board)
  VALUES (2, '{{X," "," "},{" ",O," "},{" ",X," "}}');
```

向表distributors中插入一行，返回由 DEFAULT子句生成的序号：

```
INSERT INTO distributors (did, dname) VALUES (DEFAULT, 'XYZ Widgets')
  RETURNING did;
```

增加为 Acme Corporation 管理账户的销售人员的销量，并且把整个被 更新的行以及当前时间记录到一个日志表中：

```
WITH upd AS (
  UPDATE employees SET sales_count = sales_count + 1 WHERE id =
    (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation')
    RETURNING *
)
INSERT INTO employees_log SELECT *, current_timestamp FROM upd;
```

酌情插入或者更新新的 distributor。假设已经定义了一个唯一索引来约束 出现在did列中的值。注意，特殊的 excluded表被用来引用原来要插入的值：

```
INSERT INTO distributors (did, dname)
  VALUES (5, 'Gizmo Transglobal'), (6, 'Associated Computing, Inc')
  ON CONFLICT (did) DO UPDATE SET dname = EXCLUDED.dname;
```

插入一个 distributor，或者在一个被排除的行（具有一个匹配约束的列或者 会让行级前（或者后）插入触发器引发的列的行）存在时不处理要插入的行。 例子假设已经定义了一个唯一触发器来约束出现在did列 中的值：

```
INSERT INTO distributors (did, dname) VALUES (7, 'Redline GmbH')
  ON CONFLICT (did) DO NOTHING;
```

酌情插入或者更新新的 distributor。例子假设已经定义了一个唯一触发器来 约束出现在did列中的值。WHERE子句被用 来限制实际被更新的行（不过，任何没有被更新的已有行仍将被锁定）：

```
-- 根据一个特定的 ZIP 编码更新 distributors
INSERT INTO distributors AS d (did, dname) VALUES (8, 'Anvil Distribution')
   ON CONFLICT (did) DO UPDATE
   SET dname = EXCLUDED.dname || ' (formerly ' || d.dname || ')'
   WHERE d.zipcode <> '21201';

-- 直接在语句中命名一个约束（使用相关的索引来判断是否做
-- DO NOTHING 动作）
INSERT INTO distributors (did, dname) VALUES (9, 'Antwerp Design')
   ON CONFLICT ON CONSTRAINT distributors_pkey DO NOTHING;
```

如果可能就插入新的 distributor，否则DO NOTHING。 例子假设已经定义了一个唯一索引，它约束让is_active 布尔列为true的行子集上did列中的值：

```
-- 这个语句可能推断出一个在 "did" 上带有谓词 "WHERE is_active"
-- 的部分唯一索引，但是它可能也只是使用了 "did" 上的一个常规唯一约束
INSERT INTO distributors (did, dname) VALUES (10, 'Conrad International')
   ON CONFLICT (did) WHERE is_active DO NOTHING;
```

兼容性

INSERT符合 SQL 标准，不过 RETURNING子句是一种 UXDB扩展， 在 INSERT中使用WITH也是， 用ON CONFLICT指定一个替代动作也是扩展。 还有，标准不允许省略列名列表但不通过VALUES子句或者*query*填充 所有列的情况。

SQL标准指定只有存在一个总是会生成值的标识列时才能指定OVERRIDING SYSTEM VALUE。而UXDB在任何情况下都允许这个子句，并且在不适用时会忽略它。

*query*子句可能的限制在 SELECT(7)有介绍。

名称

INSERT ALL — 按照条件在多表中创建新行

大纲

```
INSERT ALL
  WHEN condition
  THEN INTO insert_into_clause
  [ VALUES values_clause  ]
  [ INTO insert_into_clause [ VALUES values_clause  ] ]...
  [ WHEN condition
 THEN INTO insert_into_clause
 [ VALUES values_clause  ]
 [ INTO insert_into_clause [ VALUES values_clause  ] ]...
  ]...
  [ ELSE INTO insert_into_clause
  [ VALUES values_clause
  [ INTO insert_into_clause [ VALUES values_clause ] ]...
SELECTSTMT
```

描述

从一张表里面读取数据插入到多张表，最普通的方法是两个insert into select语句，这种方法会存在着两次插入的数据不一致的情况。如果要解决这个问题，必须通过人为加锁的方式，会影响并发性，会使读两次的数据不一样，使用insert all语法可以解决这个问题。

使用多个insert into select语句意味着执行力多次select查询。insert all 语法只需要读取一次原表，就可以完成多次插入，没有并发性问题，不需要额外的存储空间，只需要一条SQL语句就可以搞定。所以在性能上有提升。

INSERT ALL（无WHEN条件）：最基本的多表插入功能，实现单条SQL，单次select，对多张表插入数据。

INSERT ALL（带WHEN条件）：在insert all（无when条件）的基础上，加上when条件判断。只有当when条件为真时，才执行后面的into子句，否则不做插入操作。后面的else子句否决之前所有的when条件。

参数

这个小节介绍了可以使用的参数。

condition

when条件表达式，是任一计算得到布尔类型结果的表达式。

insert_into_clause

into子句，例如into table1。

values_clause

values子句，例如values(id)。

SELECTSTMT

select查询子句，例如select * from table1。

输出

成功完成时，INSERT命令会返回以下形式的命令标签：

INSERT *oid count*

*count*是被插入或更新的行数。*oid*总是0。

示例

insert all（无when条件），如下所示。

```
--方式1
insert all
into sample_1 (id,data,no) values(id,data,no)
into sample_2 (id,data,no) values(id,data,no)
into sample_3 (id,data,no) values(id,data,no);

--方式2
insert all
into sample_1  values(id,data,no)
into sample_2  values(id,data,no)
into sample_3  values(id,data,no);
```

insert all（带when条件），如下所示。

```
--方式1
insert all
when no < 5 then
into sample_1 (id,data,no) values(id,data,no)
when data < 70 then
into sample_2 (id,data,no) values(id,data,no)
else
into sample_3 (id,data,no) values(id,data,no);

--方式2
insert all
when no < 5 then
into sample_1  values(id,data,no)
when data < 70 then
into sample_2  values(id,data,no)
else
into sample_3  values(id,data,no);
```

又见

INSERT FIRST

名称

INSERT FIRST — 按照条件在单表中创建新行

大纲

```
INSERT FIRST
  WHEN condition
  THEN INTO insert_into_clause
  [ VALUES values_clause  ]
  [ INTO insert_into_clause [ VALUES values_clause  ] ]...
  [ WHEN condition
 THEN INTO insert_into_clause
 [ VALUES values_clause  ]
 [ INTO insert_into_clause [ VALUES values_clause  ] ]...
  ]...
  [ ELSE INTO insert_into_clause
  [ VALUES values_clause
  [ INTO insert_into_clause [ VALUES values_clause ] ]...
SELECTSTMT
```

描述

INSERT FIRST在INSERT ALL（带when条件）的基础上，实现每条数据只会插入第一次。即，每条数据只有在第一次满足WHEN条件时执行插入操作。后面即便再次满足了某条WHEN条件，也不会执行插入。

参数

这个小节介绍了可以使用的参数。

condition

when条件表达式，是任一计算得到布尔类型结果的表达式。

insert_into_clause

into子句，例如into table1。

values_clause

values子句，例如values(id)。

SELECTSTMT

select查询子句，例如select * from table1。

输出

成功完成时，INSERT命令会返回以下形式的命令标签：

INSERT *oid count*

*count*是被插入或更新的行数。*oid*总是0。

示例

insert first ，如下所示。

```
--方式1
insert first
when no < 5 then
into sample_1 (id,data,no) values(id,data,no)
when data < 70 then
into sample_2 (id,data,no) values(id,data,no)
else
into sample_3 (id,data,no) values(id,data,no);

--方式2
insert first
when no < 5 then
into sample_1  values(id,data,no)
when data < 70 then
into sample_2  values(id,data,no)
else
into sample_3  values(id,data,no);
```

又见

INSERT ALL

名称

LISTEN — 监听一个通知

大纲

LISTEN *channel*

描述

LISTEN在名为*channel*的通知频道上将当前会话注册为一个监听者。如果当前会话已经被注册为这个通知频道的一个监听者，则什么也不会发生。

只要命令NOTIFY *channel*被调用（不管是在这个会话还是在另一个连接到同一数据库的会话中），所有当前正在该通知频道上监听的会话都会被通知，并且每一个会话将会接着通知连接到它的客户端应用。

可以使用UNLISTEN命令在一个给定通知频道上反注册一个会话。当会话结束时，它的监听注册会被自动清除。

一个客户端应用检测通知事件的必用方法取决于它使用的UXDB应用编程接口。如果使用libuxsql库，应用会将LISTEN作为一个普通 SQL 命令发出，并且接着必须周期性地调用函数UXSQLnotifies来查看是否接收到通知事件。其他诸如libuxtcl的接口提供了更高层次上的处理通知事件的方法。事实上，通过使用libuxtcl应用程序员甚至不必直接发出LISTEN或UNLISTEN。更多细节可参阅所使用的接口的文档。

NOTIFY(7)包含了使用LISTEN和NOTIFY的更广泛的讨论。

参数

channel

一个通知频道的名称（任意标识符）。

注解

LISTEN在事务提交时生效。如果在一个后来被回滚的事务中执行了LISTEN或UNLISTEN，被监听的通知频道集合不会变化。

一个已经执行了LISTEN的事务不能为两阶段提交做准备。

例子

从uxsql中配置并执行一个监听/通知序列

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

兼容性

在 SQL 标准中没有LISTEN语句。

参见

NOTIFY(7), UNLISTEN(7)

名称

LOAD — 载入一个共享库文件

大纲

LOAD '*filename*'

描述

这个命令把一个共享库文件载入到UXDB服务器 的地址空间中。如果该文件已经被载入，这个命令什么也不会做。只要调用 包含 C 函数的共享库文件中的一个函数，这些共享库文件就会被自动载入。 因此，一次显式的**LOAD**通常只在载入一个通过 “钩子”修改服务器行为而不是提供一组函数的库时需要。

库文件名通常只是一个裸文件名，在服务器的库搜索路径（由 dynamic_library_path设置）中寻找。或者， 它可以作为完整的路径名称给出。无论哪种情况，平台的标准共享库文件扩展名都可以省略。

非超级用户只能把**LOAD**应用在位于 $libdir/plugins/中的库文件 — 指定的 *filename* 必须正好以该字符串开始（确保在那里只安装了 “安全的”库是数据库管理员的责任）。

兼容性

LOAD是一种 UXDB扩展。

另见

CREATE FUNCTION(7)

名称

LOCK — 锁定一个表

大纲

LOCK [TABLE] [ONLY] *name* [*] [, ...] [IN *lockmode* MODE] [NOWAIT]

其中 *lockmode* 可以是以下之一：

ACCESS SHARE | ROW SHARE | ROW EXCLUSIVE | SHARE UPDATE EXCLUSIVE
| SHARE | SHARE ROW EXCLUSIVE | EXCLUSIVE | ACCESS EXCLUSIVE

描述

LOCK TABLE获得一个表级锁，必要时会等待任何冲突锁被释放。如果指定了NOWAIT，LOCK TABLE不会等待以获得想要的锁：如果它不能立刻得到，该命令会被中止并且发出一个错误。一旦获取到，该锁会被在当前事务中一直持有（没有UNLOCKTABLE命令，锁总是在事务结束时被释放）。

当一个视图被锁定时，出现在该视图定义查询中的所有关系也将被使用同样的锁模式递归地锁住。

在为引用表的命令自动获取锁时，UXDB总是尽可能使用最不严格的锁模式。提供LOCK TABLE是用于想要更严格的锁定的情况。例如，假设一个应用运行一个READ COMMITTED隔离级别的事务，并且需要确保一个表中的数据在该事务的期间保持稳定。要实现这个目的，必须在查询之前在表上获得SHARE锁模式。这将阻止并发的数据更改并且确保该表的后续读操作会看到已提交数据的一个稳定视图，因为SHARE锁模式与写入者所要求的ROW EXCLUSIVE锁有冲突，并且LOCK TABLE *name* IN SHARE MODE语句将等待，直到任何并发持有ROWEXCLUSIVE模式锁的持有者提交或者回滚。因此，一旦得到锁，就不会有未提交的写入还没有解决。更进一步，在释放该锁之前，任何人都不能开始。

要在运行在REPEATABLE READ或SERIALIZABLE 隔离级别的事务中得到类似的效果，必须在执行任何 SELECT或者数据修改语句之前执行 LOCK TABLE语句。一个 REPEATABLE READ或者SERIALIZABLE事务的 数据视图将在它的第一个SELECT或者数据修改语句开始 时被冻结。在该事务中稍后的一个LOCK TABLE仍将阻止并发写 — 但它不会确保该事务读到的东西对应于最新的已提交值。

如果一个此类事务正要修改表中的数据，那么它应该使用 SHARE ROW EXCLUSIVE锁模式来取代 SHARE模式。这会保证一次只有一个此类事务运行。如果 不用这种模式，死锁就可能出现：两个事务可能都要求 SHARE模式，并且都不能获得 ROW EXCLUSIVE模式来真正地执行它们的更新（注意一个 事务所拥有的锁不会冲突，因此一个事务可以在它持有SHARE 模式时获得ROW EXCLUSIVE模式 — 但是如果有其他 人持有SHARE模式时则不能）。为了避免死锁，确保所有的 事务在同样的对象上以相同的顺序获得锁，并且如果在一个对象上涉及多 种锁模式，事务应该总是首先获得最严格的那种模式。

更多关于锁模式和锁策略的信息可见第 10.3 节 “显式锁定”。

参数

name

要锁定的一个现有表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，只有该表会被锁定。如果没有指定了 ONLY，该表和它所有的后代表（如果有）都会被锁定。可选地，在表名后指定*来显式地表示把后代表包括在内。

命令LOCK TABLE a, b;等效于 LOCK TABLE a; LOCK TABLE b;。这些表会被按照在 LOCK TABLE中指定的顺序一个一个 被锁定。

lockmode

锁模式指定这个锁和哪些锁冲突。锁模式在第 10.3 节 “显式锁定”中描述。

如果没有指定锁模式，那儿将使用最严格的模式ACCESS EXCLUSIVE。

NOWAIT

指定LOCK TABLE不等待任何冲突锁被释放： 如果所指定的锁不能立即获得，那么事务就会中止。

注解

LOCK TABLE ... IN ACCESS SHARE MODE要求目标表上的SELECT特权。LOCK TABLE ... IN ROW EXCLUSIVE MODE要求目标表上的INSERT、UPDATE、DELETE或TRUNCATE特权。所有其他形式的LOCK要求表级UPDATE、DELETE或TRUNCATE特权。

在该视图上执行锁定的用户必须具有该视图上相应的特权。此外视图的拥有者必须拥有底层基关系上的相关特权，但是执行锁定的用户不需要底层基关系上的任何权限。

LOCK TABLE在一个事务块外部没有用处：锁将只保持到语句完成。因此如果在一个事务块外部使用了LOCK，UXDB会报告一个错误。使用BEGIN(7)和COMMIT(7)（或者ROLLBACK(7)）定义一个事务块。

LOCK TABLE只处理表级锁，因此涉及到 ROW的模式名称在这里都是不当的。这些模式名称应该通常 被解读为用户在被锁定表中获取行级锁的意向。还有， ROW EXCLUSIVE模式是一个可共享的表锁。记住就 LOCK TABLE而言，所有的锁模式都具有相同的语义， 只有模式的冲突规则有所不同。关于如何获取一个真正的行级锁的信息， 请见SELECT参考文档中的 第 10.3.2 节 “行级锁”和“锁定子句”一节。

示例

在将要向一个外键表中执行插入时在主键表上获得一个 SHARE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
   WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- 如果记录没有被返回就做 ROLLBACK
INSERT INTO films_user_comments VALUES
   (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

在将要执行一次删除操作前在主键表上取一个 SHARE ROW EXCLUSIVE锁：

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
   (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

兼容性

在 SQL 标准中没有**LOCK TABLE**，SQL 标准中使用 **SET TRANSACTION**指定事务上的并发层次。 UXDB也支持这样做，详见 SET TRANSACTION(7)。

除ACCESS SHARE、ACCESS EXCLUSIVE和 SHARE UPDATE EXCLUSIVE锁模式之外， UXDB锁模式和 **LOCK TABLE**语法与 Oracle中的兼容。

名称

MERGE INTO — 对需要更新的表同时进行更新、插入操作

大纲

[WITH]MERGE INTO { *table_name* } [alias]
 USING {{ *table_name* | *view*} | *subquery* } [alias]
 ON *condition*
 { [*merge_update_clause*]
 [*merge_insert_clause*]
 [...]}

其中*merge_update_clause*是：

WHEN MATCHED
 [AND *condition*] THEN UPDATE SET *column* = { *expr* | DEFAULT }
[, *column* = { *expr* | DEFAULT }] [...]
[DELETE]

其中*merge_insert_clause*是：

WHEN NOT MATCHED
 [AND *condition*] THEN INSERT [(*column* [, *column*]...)]
 { VALUES ({ *expr* } [, { *expr* }]...) | DEFAULT VALUES | DO NOTHING }

描述

merge into通过两表相互关联，对需要更新的表同时进行更新、插入操作。

使用MERGE语句从一个或多个源中选择行以进行更新或插入到表或视图中。可以指定条件以确定是要更新还是插入到目标表或视图中。

该语句是组合多个操作的便捷方法，可以避免使用多个INSERT，UPDATE和DELETE DML语句。MERGE是确定性声明。也就是说，不能在同一MERGE语句中多次更新目标表的同一行。

参数

这个小节介绍了可以使用的参数。

table_name

表名称，表必须已经存在。

alias

别名，对源表、目标表操作。

view

视图。

subquery

select子句。

condition

条件语句。

column

表属性。

expr

获取表属性的值。

merge_update_clause

该merge_update_clause指定目标表的新列值。如果ON子句的条件为true，则执行此更新。如果执行update子句，则将激活目标表上定义的所有更新触发器。

如果仅在指定条件为true时才希望数据库执行更新操作，请指定where_clause。条件可以是数据源表或目标表。如果条件不成立，则在将行合并到表中时，数据库将跳过更新操作。

指定DELETE where_clause以在填充或更新表时清理表中的数据。受此子句影响的唯一行是目标表中由合并操作更新的行。也就是说，DELETE WHERE条件评估更新的值，而不是UPDATE SET ... WHERE条件评估的原始值。如果目标表的行满足DELETE条件，但未包含在ON子句定义的联接中，则不会删除该行。对于每行删除，将激活目标表上定义的所有删除触发器。

可以单独或通过merge_insert_clause指定此子句。如果两者都指定，则它们可以采用任何顺序。 对merge_update_clause的 限制此子句受以下限制： 不能更新ON条件子句中引用的列，更新视图时，不能指定DEFAULT。

merge_insert_clause

如果ON子句的条件为false ，则merge_insert_clause指定要插入到目标表的列中的值。如果执行了insert子句，则将激活目标表上定义的所有插入触发器。如果在INSERT关键字之后省略列列表，则目标表中的列数必须与VALUES子句中的值数匹配。 要将所有源行插入到表中，可以在ON子句条件中使用常量过滤谓词。常量过滤器谓词的一个示例为ON（0 = 1）。Oracle数据库会识别这样的谓词，并将所有源行无条件地插入到表中。此方法不同于省略merge_update_clause。在这种情况下，数据库仍必须执行。一个联接。使用常量过滤器谓词时，不执行连接。

仅当指定条件为真时，才希望Oracle数据库执行插入操作时，请指定where_clause。该条件只能引用数据源表。 Oracle Database对条件不成立的所有行都跳过插入操作。

可以单独或通过merge_update_clause指定此子句。如果两者都指定，则它们可以采用任何顺序。合并到视图的限制更新视图时，不能指定DEFAULT。

示例

- update和insert同时使用

```
insert into A_MERGE values(1,'liuwei',20);
insert into A_MERGE values(2,'zhangbin',21);
insert into A_MERGE values(3,'fuguo',20);
```

```
insert into B_MERGE values(1,2,'zhangbin',30,'吉林');
insert into B_MERGE values(2,4,'yihe',33,'黑龙江');
insert into B_MERGE values(3,3,'fuguo',44,'山东')
```

使用merge into用B_MERGE来更新A_MERGE中的数据，如下所示。

```
MERGE INTO A_MERGE A USING (select AID,NAME,YEAR from B_MERGE B) C ON
 (A.id=C.AID)
WHEN MATCHED THEN
UPDATE SET A.YEAR=C.YEAR
WHEN NOT MATCHED THEN
INSERT(ID,NAME,YEAR) VALUES(C.AID,C.NAME,C.YEAR);
```

- 只update

首先向B_MERGE中插入两个数据，来为了体现出只update没有insert，必须有一个数据--是A中已经存在的另一个数据是A中不存在的，插入数据语句如下，如下所示。

```
insert into B_MERGE values(4,1,'liuwei',80,'江西');
insert into B_MERGE values(5,5,'tiantian',23,'河南');
```

用B_MERGE来更新A_MERGE，如下所示。

```
merge into A_MERGE A USING (select B.AID,B.NAME,B.YEAR from B_MERGE B) C
 ON(A.ID=C.AID)
WHEN MATCHED THEN
UPDATE SET A.YEAR=C.YEAR;
```

- 只insert

B表必须有一个数据--是A中已经存在的另一个数据是A中不存在的，更新B表数据，如下所示。

```
update B_MERGE set year=70 where AID=2;
```

用B_MERGE来更新A_MERGE中的数据，如下所示。

```
merge into A_MERGE A USING (select B.AID,B.NAME,B.YEAR from B_MERGE B) C
 ON(A.ID=C.AID)
WHEN NOT MATCHED THEN
insert(ID,NAME,YEAR) VALUES(C.AID,C.NAME,C.YEAR);
```

- 带where条件的update和insert

B表必须有一个数据--是A中已经存在的另一个数据是A中不存在的，如下所示。

```
 update B_MERGE set year=100 where AID=2;
 insert into B_MERGE values(6,6,'youxuan',23,'北京');
```

带where条件的update，如下所示。

```
merge into A_MERGE A USING (select B.AID,B.NAME,B.YEAR from B_MERGE B) C
 ON(A.ID=C.AID)
WHEN MATCHED THEN
UPDATE SET A.YEAR=C.YEAR where AID=2;
```

带where条件的insert，如下所示。

```
merge into A_MERGE A USING (select B.AID,B.NAME,B.YEAR from B_MERGE B) C
 ON(A.ID=C.AID)
WHEN NOT MATCHED THEN
insert(ID,NAME,YEAR) VALUES(C.AID,C.NAME,C.YEAR)  where AID=6;
```

- 带delete的update

  merge into通过and可实现where条件判断。

```
MERGE INTO A_MERGE A USING (select B.AID,B.NAME,B.YEAR from B_MERGE B) C
 ON(A.ID=C.AID)
WHEN MATCHED AND (A.id = 1) THEN DELETE
WHEN NOT MATCHED THEN INSERT(ID,NAME,YEAR)
 VALUES(C.AID,C.NAME,C.YEAR);
```

全插入insert。

有时需要将一张表中所有的数据插入到另外一张表，此时就可以添加常量过滤谓词来实现，让其只满足匹配和不匹配，这样就只有update或者只有insert。这里要无条件全插入，则只需将on中条件设置为永假。

用B_MERGE来更新A_MERGE，如下所示。

```
MERGE INTO A_MERGE A USING (select AID,NAME,YEAR from B_MERGE B) C ON
 (1=0)
WHEN MATCHED THEN
UPDATE SET A.YEAR=C.YEAR
WHEN NOT MATCHED THEN
INSERT(ID,NAME,YEAR) VALUES(C.AID,C.NAME,C.YEAR);
```

名称

MOVE — 定位一个游标

大纲

MOVE [*direction* [FROM | IN]] *cursor_name*

其中*direction*可以为空或者以下之一：

NEXT
PRIOR
FIRST
LAST
ABSOLUTE *count*
RELATIVE *count*
count
ALL
FORWARD
FORWARD *count*
FORWARD ALL
BACKWARD
BACKWARD *count*
BACKWARD ALL

描述

MOVE重新定位一个游标而不检索任何数据。 **MOVE**的工作完全像 **FETCH**命令，但是它只定位游标并且不返回行。

用于**MOVE**命令的参数和 **FETCH**命令的一样，可参考 FETCH(7)。

输出

成功完成时，**MOVE**命令返回的命令标签形式是

MOVE *count*

*count*是一个 具有同样参数的**FETCH**命令会返回的 行数（可能为零）。

示例

```
BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- 跳过前 5 行：
MOVE FORWARD 5 IN liahona;
MOVE 5
```

```
-- 从游标 liahona 中取第 6 行：
FETCH 1 FROM liahona;
 code  | title  | did | date_prod  |  kind  |  len
-------+--------+-----+------------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22 | Action | 01:37
(1 row)

-- 关闭游标 liahona 并且结束事务：
CLOSE liahona;
COMMIT WORK;
```

兼容性

在 SQL 标准中没有**MOVE**语句。

另见

CLOSE(7)， DECLARE(7)， FETCH(7)

名称

NOTIFY — 生成一个通知

大纲

NOTIFY *channel* [, *payload*]

描述

NOTIFY命令发送一个通知事件以及一个可选的“载荷”字符串给每个正在监听的客户端应用，这些应用之前都在当前数据库中为指定的频道名执行过LISTEN *channel*。通知对所有用户都可见。

NOTIFY为访问同一个UXDB数据库的进程集合提供了一种简单的进程间通讯机制。伴随着通知可以发送一个载荷字符串，通过使用数据库中的表从通知者向监听者传递额外的数据，也可以构建用于传输结构化数据的高层机制。

由一个通知事件传递给客户端的信息包括通知频道名称、发出通知的会话的服务器进程PID以及载荷字符串，如果载荷字符串没有被指定则它为空字符串。

将在一个给定数据库以及其他数据库中使用的频道名称由数据库设计者定义。通常，频道名称与数据库中某个表的名称相同，并且通知事件其实就意味着：“我改变了这个表，来看看改了什么吧”。但是NOTIFY和LISTEN命令并未强制这样的关联。例如，一个数据库设计者可以使用几个不同的频道名称来标志一个表上的不同种类的改变。另外，载荷字符串可以被用于多种不同的情况。

当NOTIFY被用来标志对一个特定表的改变时， 一种有用的编程技巧是把NOTIFY 放在一个由表更新触发的语句触发器中。在这种方式中，每当表被改变时 都将自动发生通知，并且应用程序员不可能会忘记发出通知。

NOTIFY以一些重要的方式与 SQL 事务互动。首先，如果一个NOTIFY在一个事务内执行，在事务被提交之前，该通知事件都不会被递送。这是合适的，因为如果该事务被中止，所有其中的命令都将不会产生效果，包括NOTIFY在内。但如果期望通知事件被立即递送，那这种行为就会令人不安。其次，如果一个监听会话收到了一个通知信号而它正在一个事务中，在该事务完成（提交或者中止）之前，该通知事件将不会被递送给它连接的客户端。同样，原因在于如果一个通知在一个事务内被递送且该事务后来被中止，我们会希望该通知能以某种方式被撤销 — 但是服务器一旦把通知发送给客户端就无法“收回它”。因此通知事件只能在事务之间递送。其中的要点是把NOTIFY用作实时信号的应用应该让它们事务尽可能短小。

如果从同一个事务多次用相同的载荷字符串对同一个频道名称发送通知，数据库服务器能决定只递送一个单一的通知。另一方面，带有不同载荷字符串的通知将总是作为不同的通知被递送。类似地，来自不同事务的通知将不会被折叠成一个通知。除了丢弃后来生成的重复通知实例之外，NOTIFY保证来自同一个事务的通知按照它们被发送的顺序被递送。还可以保证的是，来自不同事务的消息会按照其事务被提交的顺序递送。

一个执行NOTIFY的客户端自己也同时在监听同一个通知频道是很常见的事。在这种情况下，和所有其他监听会话一样，它会取回一个通知事件。根据应用的逻辑，这可能导致无用的工作，例如从自己刚刚写入的一个表中读出相同的更新。可以通过关注发出通知的服务器进程PID（在通知事件消息中提供）与自己的会话PID（可以从libuxsql得到）是否相同来避免这种额外的工作。当两者相同时，该通知事件就是当前会话自己发出的，所以可以忽略。

参数

channel

要对其发信号的通知频道的名称（任意标识符）。

payload

要通过通知进行沟通的“载荷”字符串。这必须是一个简单的字符串。在默认配置下，该字符串不能超过 8000 字节（如果需要发送二进制数据或者更多信息，最好是把它放在一个数据库表中并且发送该记录的键）。

注解

有一个队列保持着已经发送但是还没有被所有监听会话处理的通知。如果该队列被占满，调用NOTIFY的事务将在提交时失败。该队列非常大（标准安装中是 8GB）并且应该足以应付几乎每一种用例。不过，如果一个会话执行了NOTIFY并且接着长时间进入一个事务，不会发生清理操作。一旦该队列使用过半，将在日志文件中看到警告，它指出哪个会话阻止了清理。在这种情况中，应该确保这个会话结束它的当前事务，这样清理才能够进行下去。

函数ux_notification_queue_usage返回队列中当前被 待处理通知所占据的比例。详见第 6.25 节“系统信息函数和运算符”。

一个已经执行了NOTIFY的事务不能为两阶段提交做准备。

ux_notify

要发送一个通知，也能使用函数ux_notify(text, text)。该函数采用频道名称作为第一个参数，而载荷则作为第二个参数。如果需要使用非常量的频道名称和载荷，这个函数比NOTIFY命令更容易使用。

例子

从uxsql配置和执行一个监听/通知序列：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
NOTIFY virtual, 'This is the payload';
Asynchronous notification "virtual" with payload "This is the payload" received from server process
 with PID 8448.

LISTEN foo;
SELECT ux_notify('fo' || 'o', 'pay' || 'load');
Asynchronous notification "foo" with payload "payload" received from server process with PID 14728.
```

兼容性

在 SQL 标准中没有NOTIFY语句。

参见

LISTEN(7), UNLISTEN(7)

名称

PREPARE — 为执行准备一个语句

大纲

PREPARE *name* [(*data_type* [, ...])] AS *statement*

描述

PREPARE创建一个预备语句。预备语句是一种服务器端对象，它可以被用来优化性能。当PREPARE语句被执行时，指定的语句会被解析、分析并且重写。当后续发出一个EXECUTE命令时，该预备语句会被规划并且执行。这种工作的划分避免了重复性的解析分析工作，不过允许执行计划依赖所提供的特定参数值。

预备语句可以接受参数：在执行时会被替换到语句中的值。在创建预备语句时，可以用位置引用参数，如$1、$2等。也可以选择性地指定参数数据类型的一个列表。当一个参数的数据类型没有被指定或者被声明为unknown时，其类型会从该参数第一次被引用的环境中推知（如果可能）。在执行该语句时，在EXECUTE语句中为这些参数指定实际值。更多有关于此的信息可参考EXECUTE(7)。

预备语句只在当前数据库会话期间存在。当会话结束时，预备语句会消失，因此在重新使用之前必须重新建立它。这也意味着一个预备语句不能被多个数据库客户端同时使用。不过，每一个客户端可以创建它们自己的预备语句来使用。预备语句可以用DEALLOCATE(7)命令手工清除。

当一个会话要执行大量类似语句时，预备语句可能会有最大性能优势。如果该语句很复杂（难于规划或重写），例如，如果查询涉及很多表的连接或者要求应用多个规则，性能差异将会特别明显。如果语句相对比较容易规划和重写，但是执行起来开销相对较大，那么预备语句的性能优势就不那么显著了。

参数

name

给这个特定预备语句的任意名称。它在一个会话中必须唯一并且后续将被用来执行或者清除一个之前准备好的语句。

data_type

预备语句一个参数的数据类型。如果一个特定参数的数据类型没有被指定或者被指定为unknown，将从该参数第一次被引用的环境中推得。要在预备语句本身中引用参数，可以使用 $1、$2等。

statement

任何SELECT、INSERT、UPDATE、DELETE或者VALUES语句。

注解

可以使用generic plan或custom plan执行已准备好的语句。 通用计划在所有执行中都是相同的，而自定义计划是为特定执行所生成的，使用调用中给出的参数值。 使用通用计划可以避免计划开销，但在某些情况下，自定义计划的执行效率要高得多，因为规划器可以利用参数值的知识。 （当然，如果准备好的语句没有参数，则这是没有意义的，并且始终应使用通用计划。）

默认情况下（也就是当plan_cache_mode设定为 auto时），对已经准备好的具有参数的语句，服务器将自动选择使用通用或自定义计划。 当前的规则是，前五次执行都是使用自定义计划完成的，并且计算这些计划的平均估计成本。然后创建通用计划，并将其估计成本与自定义计划的平均成本相比较。 如果通用计划的成本没有比平均自定义计划成本高太多，那么后续执行将使用通用计划，以使重复的再看起来更可取。

这种探索式方法可以重写，强迫服务器使用通用或自定义计划，通过将plan_cache_mode分别设置为force_generic_plan或 force_custom_plan 。 如果通用计划的成本估计由于某种原因很不理想，则此设置主要很有用，尽管其实际成本远高于自定义计划的实际成本，可以选择此设置。

要检查UXDB为一个预备语句使用的查询计划，可以使用EXPLAIN(7)，例如：

EXPLAIN EXECUTE *name*(*parameter_values*);

如果使用的是一个通用计划，它将包含参数符号$*n*，而一个定制计划则会把提供的参数值替换进去。

更多关于查询规划以及UXDB为此所收集的统计信息的内容，请见ANALYZE(7)文档。

尽管预备语句主要是为了避免重复对语句进行解析分析以及规划，但是只要上一次使用该预备语句后该语句中用到的数据库对象发生了定义性（DDL）改变，UXDB将会对该语句强制进行重新分析和重新规划。还有，如果search_path的值发生变化，也将使用新的search_path重新解析该语句。这些规则让预备语句的使用在语义上几乎等效于反复提交相同的查询文本，但是能在性能上获利（如果没有对象定义被改变，特别是如果最优计划保持不变时）。该语义等价性不完美的一个例子是：如果语句用一个未限定的名称引用表，并且之后在search_path中更靠前的模式中创建了一个新的同名表，则不会发生自动的重解析，因为该语句使用的对象没有被改变。不过，如果某些其他更改造成了重解析，后续使用中都会引用新表。

可以通过查询ux_prepared_statements系统视图来看到会话中所有可用的预备语句。

例子

为一个INSERT语句创建一个预备语句，然后执行它：

```
PREPARE fooplan (int, text, bool, numeric) AS
   INSERT INTO foo VALUES($1, $2, $3, $4);
EXECUTE fooplan(1, 'Hunter Valley', 't', 200.00);
```

为一个SELECT语句创建一个预备语句，然后执行它：

```
PREPARE usrrptplan (int) AS
   SELECT * FROM users u, logs l WHERE u.usrid=$1 AND u.usrid=l.usrid
   AND l.date = $2;
EXECUTE usrrptplan(1, current_date);
```

在这个示例中，第二个参数的数据类型没有被指定，因此会从使用$2的环境中推知。

兼容性

SQL 标准包括一个PREPARE语句，但是它只用于嵌入式 SQL。这个版本的PREPARE语句也使用了一种有些不同的语法。

另见

DEALLOCATE(7), EXECUTE(7)

名称

PREPARE TRANSACTION — 为两阶段提交准备当前事务

大纲

PREPARE TRANSACTION *transaction_id*

描述

PREPARE TRANSACTION为两阶段提交准备 当前事务。在这个命令之后，该事务不再与当前会话关联。相反，它的状态 被完全存储在磁盘上，并且有很高的可能性它会被提交成功（即便在请求提 交前发生数据库崩溃）。

一旦被准备好，事务稍后就可以分别用 COMMIT PREPARED(7) 或者ROLLBACK PREPARED(7)提交或者回滚。可以从任何 会话而不仅仅是执行原始事务的会话中发出这些命令。

从发出命令的会话的角度来看，**PREPARE TRANSACTION**不像**ROLLBACK**命令： 在执行它之后，就没有活跃的当前事务，并且该预备事务的效果也不再可见（ 如果该事务被提交，效果将重新变得可见）。

如果由于任何原因**PREPARE TRANSACTION** 命令失败，它会变成一个**ROLLBACK**：当前事务会被取消。

参数

transaction_id

一个任意的事务标识符， **COMMIT PREPARED**或者**ROLLBACK PREPARED** 以后将用这个标识符来标识这个事务。该标识符必须写成一个字符串， 并且长度必须小于 200 字节。它也不能与任何当前已经准备好的事务的标识符相同。

注解

PREPARE TRANSACTION并不是设计为在应用或者交互式 会话中使用。它的目的是允许一个外部事务管理器在多个数据库或者其他事务性 来源之间执行原子的全局事务。除非在编写一个事务管理器，否则可能不会 用到**PREPARE TRANSACTION**。

这个命令必须在一个事务块中使用。事务块用BEGIN(7)开始。

当前在已经执行过任何涉及到临时表或者会话的临时命名空间、创建带 WITH HOLD的游标或者执行 **LISTEN**、**UNLISTEN**或**NOTIFY**的 事务中，不允许**PREPARE**该事务。这些特性与当前会话 绑定得太过紧密，所以对一个要被准备的事务来说没有什么用处。

如果用**SET**（不带LOCAL选项）修改过事务的 任何运行时参数，这些效果会持续到 **PREPARE TRANSACTION**之后，并且将不会被后续的任何 **COMMIT PREPARED**或 **ROLLBACK PREPARED**所影响。因此，在这一 方面**PREPARE TRANSACTION**的行为更像 **COMMIT**而不是**ROLLBACK**。

所有当前可用的准备好事务被列在ux_prepared_xacts系统视图中。

> **注意**
>
> 让一个事务处于准备好状态太久是不明智的。这将会干扰**VACUUM**回收存储的能力，并且在极限情况下可能导致数据库关闭以阻止事务 ID 回卷。还

> 要记住，该事务会继续持有它已经持有的锁。该特性的设计用法是，只要一个外部事务管理器已经验证其他数据库也准备好了要提交，一个准备好的事务将被正常地提交或者回滚。
>
> 如果没有建立一个外部事务管理器来跟踪准备好的事务并且确保它们被迅速地结束，最好禁用准备好事务特性（设置max_prepared_transactions为零）。这将防止意外地创建准备好事务，不然该事务有可能被忘记并且最终导致问题。

例子

为两阶段提交准备当前事务，使用foobar作为事务标识符：

```
PREPARE TRANSACTION 'foobar';
```

兼容性

PREPARE TRANSACTION是一种 UXDB扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA）， 但是那些系统的 SQL 方面未被标准化。

另见

COMMIT PREPARED(7), ROLLBACK PREPARED(7)

名称

REASSIGN OWNED — 更改一个数据库角色拥有的数据库对象的拥有关系

大纲

```
REASSIGN OWNED BY { old_role | CURRENT_USER | SESSION_USER } [, ...]
        TO { new_role | CURRENT_USER | SESSION_USER }
```

描述

REASSIGN OWNED指示系统把 *old_role*拥有 的任何数据库对象的拥有关系更改为 *new_role*。

参数

old_role

一个角色的名称。这个角色在当前数据库中所拥有的 所有对象以及所有共享对象（数据库、表空间）的 所有权都将被重新赋予给 *new_role*。

new_role

将作为受影响对象的新拥有者的角色名称。

注解

REASSIGN OWNED经常被用来为移除一个 或者多个角色做准备。因为**REASSIGN OWNED**不影响其他数据库中的对象，通常需要在包含有 被删除的角色所拥有的对象的每一个数据库中都执行这个命令。

REASSIGN OWNED同时要求源角色和目标 角色上的成员资格。

DROP OWNED(7)命令可以简单地删掉一个或者多个角色所拥有 的所有数据库对象。

REASSIGN OWNED命令不会影响授予给*old_roles*的在它们不拥有的对象上的任何特权。 同样，它不会影响**ALTER DEFAULT PRIVILEGES**创建的默认特权。**DROP OWNED**可以回收那些特权。

兼容性

REASSIGN OWNED命令是一种 UXDB扩展。

另见

DROP OWNED(7), DROP ROLE(7), ALTER DATABASE(7)

名称

REFRESH MATERIALIZED VIEW — 替换一个物化视图的内容

大纲

```
REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name
  [ WITH [ NO ] DATA ]
```

描述

REFRESH MATERIALIZED VIEW完全替换一个 物化视图的内容。必须是该物化视图的属主才能执行这个命令.旧的内容会被抛弃。如果指定了 WITH DATA（或者作为默认值），支持查询将被执行以 提供新的数据，并且会让物化视图将处于可扫描的状态。如果指定了 WITH NO DATA，则不会生成新数据并且会让物化视图 处于一种不可扫描的状态。

CONCURRENTLY和WITH NO DATA 不能被一起指定。

参数

CONCURRENTLY

对物化视图的刷新不阻塞在该物化视图上的并发选择。如果没有这个选项， 一次影响很多行的刷新将使用更少的资源并且更快结束，但是可能会阻塞 其他尝试从物化视图中读取的连接。这个选项在只有少量行被影响的情况 下可能会更快。

只有当物化视图上有至少一个UNIQUE索引（只用列名 并且包括所有行）时，才允许这个选项。也就是说，它不能是表达式索引或者包括WHERE子句。

当物化视图还未被填充时，这个选项不能被使用。

即使带有这个选项，对于任意一个物化视图一次也只能运行一个 REFRESH。

name

要刷新的物化视图的名称（可以被模式限定）。

注解

虽然用于未来的CLUSTER(7)操作的默认索引会被保持， **REFRESH MATERIALIZED VIEW**不会基于这个属性排序产生 的行。如果希望数据在产生时排序，必须在支持查询中使用 ORDER BY子句。

示例

这个命令将使用物化视图order_summary定义中的查询 来替换该物化视图的内容，并且让它处于一种可扫描的状态：

```
REFRESH MATERIALIZED VIEW order_summary;
```

这个命令将释放与物化视图annual_statistics_basis相关 的存储并且让它变成一种不可扫描的状态：

```
REFRESH MATERIALIZED VIEW annual_statistics_basis WITH NO DATA;
```

兼容性

REFRESH MATERIALIZED VIEW是一种 UXDB扩展。

另见

CREATE MATERIALIZED VIEW(7), ALTER MATERIALIZED VIEW(7), DROP MATERIALIZED VIEW(7)

名称

REINDEX — 重建索引

大纲

```
REINDEX [ ( VERBOSE ) ] { INDEX | TABLE | SCHEMA | DATABASE | SYSTEM }
 [ CONCURRENTLY ] name
```

描述

REINDEX使用索引的表里存储的数据重建一个索引， 并且替换该索引的旧拷贝。有一些场景需要使用**REINDEX**：

- 一个索引已经损坏，并且不再包含合法数据。尽管理论上这不会发生， 实际上索引会因为软件缺陷或硬件失效损坏。 **REINDEX**提供了一种恢复方法。
- 一个索引变得“臃肿”，其中包含很多空的或者近乎为空的页面。 UXDB中的 B-树索引在特定的非 常规访问模式下可能会发生这种情况。**REINDEX** 提供了一种方法来减少索引的空间消耗，即制造一个新版本的索引，其中没有 死亡页面。
- 修改了一个索引的存储参数（例如填充因子），并且希望确保这种修改完全 生效。
- 如果索引在用CONCURRENTLY选项创建失败，该索引保留为一个“invalid”。 这类索引是无用的，但是可以方便的用**REINDEX**来重建它们。注意，只有**REINDEX INDEX**可以在无效的索引上执行并发创建。

参数

INDEX

重新创建指定的索引。

TABLE

重新创建指定表的所有索引。如果该表有一个二级 “TOAST”表，它也会被重索引。

SCHEMA

重建指定方案的所有索引。如果这个方案中的一个表有次级的“TOAST”表，它也会被重建索引。共享系统目录上的索引也会被处理。这种形式的**REINDEX**不能在事务块内执行。

DATABASE

重新创建当前数据库内的所有索引。共享的系统目录上的索引也会被 处理。这种形式的**REINDEX**不能在一个 事务块内执行。

SYSTEM

重新创建当前数据库中在系统目录上的所有索引。共享系统目录上的 索引也被包括在内。用户表上的索引则不会被处理。这种形式的 **REINDEX**不能在一个事务块内执行。

name

要被重索引的特定索引、表或者数据库的名字。索引和表名可以被 模式限定。当前，REINDEX DATABASE和 REINDEX SYSTEM只能重索引当前数据库，因此 它们的参数必须匹配当前数据库的名称。

CONCURRENTLY

使用此选项时，UXDB 将重建索引，而不在表上采取任何阻止并发插入、更新或删除的锁；标准的索引重建将会锁定表上的写操作(而不是读操作)，直到它完成。 使用此选项一时，有几个事项需要注意；请参阅“立即重建索引”一节。

对于临时表，REINDEX始终是非并发的，因为没有其他会话可以访问它们，并且非并发重新索引更便宜。

VERBOSE

在每个索引被重建时打印进度报告。

注解

如果怀疑一个用户表上的索引损坏，可以使用 REINDEX INDEX或者 REINDEX TABLE简单地重建该索引 或者表上的所有索引。

如果需要从一个系统表上的索引损坏中恢复，就更困难一些。在 这种情况下，对系统来说重要的是没有使用过任何可疑的索引本身（ 实际上，这种场景中，可能会发现服务器进程会在启动时立刻崩溃， 这是因为对于损坏的索引的依赖）。要安全地恢复，服务器必须用 -P选项启动，这将阻止它使用索引来进行系统 目录查找。

这样做的一种方法是关闭服务器，并且启动一个单用户的 UXDB服务器，在其命令行 中包括-P选项。然后，可以发出 REINDEX DATABASE、REINDEX SYSTEM、 REINDEX TABLE或者REINDEX INDEX， 具体使用哪个命令取决于想要重构多少东西。如果有疑问，可以使用REINDEX SYSTEM来选择重建数据库中的所有系统索引。 然后退出单用户服务器会话并且重启常规的服务器。

在另一种方法中，可以开始一个常规的服务器会话，在其命令行选项 中包括-P。这样做的方法与客户端有关，但是在 所有基于libuxsql的客户端中都可以在开始客户端 之前设置UXOPTIONS环境变量为-P。 注意虽然这种方法不要求用锁排斥其他客户端，在修复完成之前避免 其他用户连接到受损的数据库才是更加明智的。

REINDEX类似于删除索引并且重建索引，在其中 索引内容会被从头开始建立。不过，锁定方面的考虑却相当不同。 REINDEX会用锁排斥写，但不会排斥在索引的父表上的读。 它也会在被处理的索引上取得一个排他锁，该锁将会阻塞对该索引的使用尝试。 相反，DROP INDEX 会暂时在附表上取得一个排他锁，阻塞 写和读。后续的CREATE INDEX会排斥写但不排斥读，由于 该索引不存在，所以不会有读取它的尝试，这意味着不会有阻塞但是读操作可能 被强制成昂贵的顺序扫描。

重索引单独一个索引或者表要求用户是该索引或表的拥有者。对方案或数据库重建索引要求是该方案或者数据库的拥有者。注意因此非超级用户有时无法重建其他用户拥有的表上的索引。不过，作为一种特例，当一个非超级用户发出REINDEX DATABASE、REINDEX SCHEMA或者REINDEX SYSTEM时，共享目录上的索引将被跳过，除非该用户拥有该目录（通常不会是这样）。当然，超级用户总是可以重建所有的索引。

不支持重建分区表的索引或者分区索引。不过可以单独为每个分区重建索引。

立即重建索引

重建索引可能会影响数据库的常规操作。通常UXDB会锁定重建的表以防止写操作，并通过单次扫描表来执行整个索引构建。　　其他事务仍可以读取表，但如果它们尝试在表中插入、更新或删除行，它们将被阻止，直到索引重建完成。　　如果系统是实时生产数据库，这可能会产生严重影响。非常大的表可能需要几个小时才能编制索引，即使对于较小的表，索引重建也会锁定编写器，这些时间段对于生产系统来说是不可接受的。

UXDB支持以最少的写入锁定来重建索引。此方法通过指定**REINDEX**的CONCURRENTLY选项来调用。　使用此选项时，UXDB必须对需要重新生成的每个索引执行两次表扫描，并等待可能使用索引的所有现有事务的终止。　此方法需要比标准索引重建更大的工作量，并且需要相当长的时间才能完成，因为它需要等待可能修改索引的未完成的事务。　但是，由于它允许在重建索引时继续正常操作，此方法可用于在生产环境中重建索引。当然，重建索引所需的额外　CPU、内存和 I/O 负载可能会减慢其他操作的速度。

以下步骤发生在并发重建索引中。 每个步骤在单独的事务中运行。 如果要重建多个索引，则每个步骤在进入到下一步之前都要循环遍历所有索引。

1. 新的临时索引定义将添加到目录ux_index中。　　此定义将用于替换旧索引。　　一个SHARE UPDATE　EXCLUSIVE会话级别的锁将放在要重建的索引以及其关联的表上，以防止处理时的任何模式修改。

2. 为每个新索引完成生成索引的首个操作。　　生成索引后，其标志ux_index.indisready切换到“true”使其准备好插入，使其在执行生成的事务完成后对其他会话可见。　此步骤在每个索引的单独事务中完成。

3. 然后执行第二个操作以添加在第一个操作运行时添加的元组。此步骤也在每个索引的单独事务中完成。

4. 引用索引的所有约束都已更改以引用新的索引定义，并且索引名称也已经更改。　　此时，ux_index.indisvalid会为新索引切换到“true”，以及为旧索引切换到“false”，并且缓存无效会导致引用旧索引的所有会话失效。

5. 旧索引有ux_index.indisready切换到“false”以防止任何新的元组插入，在等待可能引用旧索引的查询之后完成。

6. 旧索引被丢弃。索引和表的SHARE UPDATE EXCLUSIVE会话锁被释放。

如果在重建索引时出现问题，例如唯一索引中的唯一性冲突， **REINDEX**命令将失败，但会留下一个　“invalid”新索引，在已经存在的索引之外。　出于查询目的此索引将被忽略，因为它可能不完整;但是它仍将消耗更新开销。uxsql \d命令将此类索引报告为 INVALID:

```
uxdb=# \d tab
      Table "public.tab"
 Column |  Type   | Modifiers
--------+---------+-----------
 col    | integer |
Indexes:
   "idx" btree (col)
   "idx_ccnew" btree (col) INVALID
```

在这种情况下，推荐的恢复方法是删除无效索引，然后尝试重新执行**REINDEX　CONCURRENTLY**。在处理期间创建的并发索引的名称以ccnew为后缀，如果它是一个我们未能删除的旧索引定义，后缀是ccold。　可以使用DROP INDEX删除无效索引，包括无效的toast 索引。

常规索引创建允许在同一表上的其他常规索引创建同时发生，但在一个表上一次只能发生一个并发索引创建。在这两种情况下，不允许同时对表上其他类型的模式进行修改。 另一个区别是，常规REINDEX TABLE或REINDEX INDEX命令可以在事务块中执行，但REINDEX CONCURRENTLY不能执行。

REINDEX SYSTEM 不支持 CONCURRENTLY 因为系统目录不能并发重新索引。

此外，排除约束的索引不能并发重新编制索引。 如果此命令中直接命名了这样的索引，则会引发错误。 如果并发重新编制具有排除约束索引的表或数据库，将跳过这些索引。 （它可以不使用CONCURRENTLY选项来重新编制这样的索引）。

示例

重建单个索引：

```
REINDEX INDEX my_index;
```

重建表my_table上的所有索引：

```
REINDEX TABLE my_table;
```

重建一个特定数据库中的所有索引，且不假设系统索引已经可用：

```
$ export UXOPTIONS="-P"
$ uxsql broken_db
...
broken_db=> REINDEX DATABASE broken_db;
broken_db=> \q
```

重建表的索引，在重建索引过程中不阻止对相关关系进行读写操作：

```
REINDEX TABLE CONCURRENTLY my_broken_table;
```

兼容性

在 SQL 标准中没有REINDEX命令。

See Also

CREATE INDEX(7), DROP INDEX(7)

名称

RELEASE SAVEPOINT — 销毁一个之前定义的保存点

大纲

RELEASE [SAVEPOINT] *savepoint_name*

描述

RELEASE SAVEPOINT销毁在当前事务 中之前定义的一个保存点。

销毁一个保存点会使得它不能再作为一个回滚点，但是它没有其他用户 可见的行为。它不会撤销在该保存点被建立之后执行的命令的效果（要 这样做，可见ROLLBACK TO SAVEPOINT(7)）。当不再需要一个 保存点时销毁它允许系统在事务结束之前回收一些资源。

RELEASE SAVEPOINT也会销毁所有 在该保存点建立之后建立的保存点。

参数

savepoint_name

要销毁的保存点的名称。

注解

指定一个不是之前定义的保存点名称是错误。

当事务处于中止状态时不能释放保存点。

如果多个保存点具有相同的名称，只有最近被定义的那个会被释放。

示例

建立并且销毁一个保存点：

```
BEGIN;
   INSERT INTO table1 VALUES (3);
   SAVEPOINT my_savepoint;
   INSERT INTO table1 VALUES (4);
   RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上述事务将插入 3 和 4。

兼容性

这个命令符合SQL标准。该标准指定关键词 SAVEPOINT是强制需要的，但 UXDB允许省略。

另见

BEGIN(7), COMMIT(7), ROLLBACK(7), ROLLBACK TO SAVEPOINT(7), SAVEPOINT(7)

名称

RESET — 把一个运行时参数的值恢复到默认值

大纲

RESET *configuration_parameter*
RESET ALL

描述

RESET把运行时参数恢复到它们的默认值。 RESET是

SET *configuration_parameter* TO DEFAULT

的另一种写法。 详见SET(7)。

默认值被定义为如果在当前会话中没有发出过SET， 参数必须具有的值。这个值的实际来源可能是一个编译在内部的默认值、 配置文件、命令行选项、或者针对每个数据库或者每个用户的默认设置。 这和把它定义成“在会话开始时该参数得到的值”有细微的差 别，因为如果该值来自于配置文件，它将被重置为现在配置文件所指定 的任何东西。

RESET的事务行为和SET相同： 它的效果会被事务回滚撤销。

参数

configuration_parameter

一个可设置的运行时参数名称。

ALL

把所有可设置的运行时参数重置为默认值。

示例

把timezone配置变量设置为默认值：

RESET timezone;

兼容性

RESET是一种 UXDB扩展。

另见

SET(7), SHOW(7)

名称

REVOKE — 移除访问特权

大纲

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | REFERENCES } ( column_name [, ...] )
    [, ...] | ALL [ PRIVILEGES ] ( column_name [, ...] ) }
    ON [ TABLE ] table_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { USAGE | SELECT | UPDATE }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { SEQUENCE sequence_name [, ...]
         | ALL SEQUENCES IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [ PRIVILEGES ] }
    ON DATABASE database_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON DOMAIN domain_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN DATA WRAPPER fdw_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON FOREIGN SERVER server_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]
```

```
REVOKE [ GRANT OPTION FOR ]
    { EXECUTE | ALL [ PRIVILEGES ] }
    ON { { FUNCTION | PROCEDURE | ROUTINE } function_name [ ( [ [ argmode ] [ arg_name
] arg_type [, ...] ] ) ] [, ...]
        | ALL { FUNCTIONS | PROCEDURES | ROUTINES } IN SCHEMA schema_name [, ...] }
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON LANGUAGE lang_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { SELECT | UPDATE } [, ...] | ALL [ PRIVILEGES ] }
    ON LARGE OBJECT loid [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { { CREATE | USAGE } [, ...] | ALL [ PRIVILEGES ] }
    ON SCHEMA schema_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { CREATE | ALL [ PRIVILEGES ] }
    ON TABLESPACE tablespace_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ GRANT OPTION FOR ]
    { USAGE | ALL [ PRIVILEGES ] }
    ON TYPE type_name [, ...]
    FROM { [ GROUP ] role_name | PUBLIC } [, ...]
    [ CASCADE | RESTRICT ]

REVOKE [ ADMIN OPTION FOR ]
    role_name [, ...] FROM role_name [, ...]
    [ CASCADE | RESTRICT ]
```

描述

REVOKE命令收回之前从一个或者更多角色　　授予的特权。关键词PUBLIC隐式定义的全部角色的组。

特权类型的含义见GRANT(7)命令的描述。

注意任何特定角色拥有的特权包括直接授予给它的特权、从它作为其成员的　角色中得到的特权以及授予给PUBLIC的特权。因此，　　从PUBLIC收回SELECT特权并不一定会意味　着所有角色都会失去在该对象上的SELECT特权：那些直接被授　　予的或者通过另一个角色被授予的角色仍

然会拥有它。类似地，从一个用户 收回SELECT后，如果PUBLIC或者另一个 成员关系角色仍有SELECT权利，该用户还是可以使用 SELECT。

如果指定了GRANT OPTION FOR，只会回收该特权 的授予选项，特权本身不被回收。否则，特权及其授予选项都会被回收。

如果一个用户持有一个带有授予选项的特权并且把它授予给了其他用户， 那么被那些其他用户持有的该特权被称为依赖特权。如果第一个用户持有 的该特权或者授予选项正在被收回且存在依赖特权，指定 CASCADE可以连带回收那些依赖特权，不指定则会 导致回收动作失败。这种递归回收只影响通过可追溯到该 REVOKE命令的主体的用户链授予的特权。因此， 如果该特权经由其他用户授予给受影响用户，受影响用户可能实际上还 保留有该特权。

在回收一个表上的特权时，也会在该表的每一个列上自动回收对应的列 特权（如果有）。在另一方面，如果一个角色已经被授予一个表上的 特权，那么从个别的列上回收同一个特权将不会生效。

在回收一个角色中的成员关系时，GRANT OPTION被改 称为ADMIN OPTION，但行为是类似的。也要注意这种 形式的命令不允许噪声词GROUP。

注解

用户只能回收由它直接授出的特权。例如，如果用户 A 已经把一个带有 授予选项的特权授予给了用户 B，并且用户 B 接着把它授予给了用户 C， 那么用户 A 无法直接从 C 收回该特权。反而，用户 A 可以从用户 B 收回 该授予选项并且使用CASCADE选项，这样该特权会被 依次从用户 C 回收。对于另一个例子，如果 A 和 B 都把同一个特权授予 给了 C，A 能够收回它们自己的授权但不能收回 B 的授权，因此 C 实际上 仍将拥有该特权。

当一个对象的非拥有者尝试REVOKE该对象上的特权时， 如果该用户在该对象上什么特权都不拥有，该命令会立刻失败。只要有某个 特权可用，该命令将继续，但是它只会收回那些它具有授予选项的特权。 如果没有持有授予选项，REVOKE ALL PRIVILEGES形式 将发出一个警告，而其他形式在没有持有该命令中特别提到的任何特权的 授予选项时就会发出警告（原则上，这些语句也适用于对象拥有者，但是 由于拥有者总是被认为持有所有授予选项，这些情况永远不会发生）。

如果一个超级用户选择发出一个GRANT或者 REVOKE命令，该命令就好像被受影响对象的拥有者发出 的一样被执行。因为所有特权最终来自于对象拥有者（可能是间接地通过 授予选项链），可以由超级用户收回所有特权，但是这可能需要前述的 CASCADE。

REVOKE也可以由一个并非受影响对象的拥有者的角色 完成，但是该角色应该是一个拥有该对象的角色的成员或者是一个在该对象 上拥有特权的WITH GRANT OPTION的角色的成员。 在这种情况中，该命令就好像被实际拥有该对象或者特权的 WITH GRANT OPTION的包含角色发出的一样被执行。 例如，如果表t1被角色g1拥有，而u1 是g1的一个成员，那么u1能收回t1 上被记录为由g1授出的特权。这会包括由u1 以及由角色g1的其他成员完成的授予。

如果执行REVOKE的角色持有通过多于一条角色成员 关系路径间接得到的特权，其中哪一条包含将被用于执行该命令的 角色是没有被指明的。在这种情况中，最好使用 SET ROLE成为想作为其身份执行 REVOKE的特定角色。如果无法做到这一点 可能会导致回收超过预期的特权，或者根本回收不了任何东西。

关于特定权限类型的更多信息请参加 第 2.7 节 “权限”以及如何检查对象的权限。

示例

从 public 收回表films上的插入特权：

REVOKE INSERT ON films FROM PUBLIC;

从用户manuel收回视图 kinds上的所有特权：

REVOKE ALL PRIVILEGES ON kinds FROM manuel;

注意着实际意味着“收回所有我授出的特权”。

从用户joe收回角色admins中的成员关系：

REVOKE admins FROM joe;

兼容性

GRANT(7)命令的兼容性注解同样适用于 **REVOKE**。根据标准，关键词 RESTRICT或CASCADE 是必要的，但是UXDB默认假定为 RESTRICT。

另见

GRANT(7), ALTER DEFAULT PRIVILEGES(7)

名称

ROLLBACK — 中止当前事务

大纲

```
ROLLBACK [ WORK | TRANSACTION ] [ AND [ NO ] CHAIN ]
```

描述

ROLLBACK回滚当前事务并且导致 该事务所作的所有更新都被抛弃。

参数

WORK
TRANSACTION

可选关键词，没有效果。

AND CHAIN

如果指定了AND CHAIN，则立即启动与刚刚完成事务具有相同事务特征（参见 SET TRANSACTION(7)）的新事务。 否则，不会启动任何新事务。

注解

使用COMMIT(7)可成功地终止一个事务。

在一个事务块之外发出**ROLLBACK**会发出一个警告并且不会有效果。事务块之外的**ROLLBACK AND CHAIN** 是一个错误。

示例

要中止所有更改：

```
ROLLBACK;
```

兼容性

命令**ROLLBACK**符合 SQL 标准。窗体ROLLBACK TRANSACTION是一个UXDB 扩展。

另见

BEGIN(7), COMMIT(7), ROLLBACK TO SAVEPOINT(7)

名称

ROLLBACK PREPARED — 取消一个之前为两阶段提交准备好的事务

大纲

ROLLBACK PREPARED *transaction_id*

描述

ROLLBACK PREPARED回滚 一个处于准备好状态的事务。

参数

transaction_id

要被回滚的事务的事务标识符。

注解

要回滚一个准备好的事务，必须是原先执行该事务的同一个用户或者 是一个超级用户。但是必须处在执行该事务的同一个会话中。

这个命令不能在一个事务块内被执行。准备好的事务会被立刻回滚。

ux_prepared_xacts 系统视图中列出了当前可用的所有准备好的事务。

例子

用事务标识符foobar回滚对应的事务：

```
ROLLBACK PREPARED 'foobar';
```

兼容性

ROLLBACK PREPARED是一种 UXDB扩展。其意图是用于 外部事务管理系统，其中有些已经被标准涵盖（例如 X/Open XA）， 但是那些系统的 SQL 方面未被标准化。

另见

PREPARE TRANSACTION(7), COMMIT PREPARED(7)

名称

ROLLBACK TO SAVEPOINT — 回滚到一个保存点

大纲

ROLLBACK [WORK | TRANSACTION] TO [SAVEPOINT] *savepoint_name*

描述

回滚在该保存点被建立之后执行的所有命令。该保存点保持有效并且可以在　以后再次回滚到它（如果需要）。

ROLLBACK TO SAVEPOINT隐式地销毁在所提及的保存点 之后建立的所有保存点。

参数

savepoint_name

要回滚到的保存点。

注解

使用RELEASE SAVEPOINT(7)销毁一个保存点而 不抛弃在它建立之后被执行的命令的效果。

指定一个没有被建立的保存点是一种错误。

相对于保存点，游标有一点非事务的行为。在保存点被回滚时，任何在该保存点　内被打开的游标将会被关闭。如果一个先前打开的游标在一个保存点内被 **FETCH**或**MOVE**命令所影响，而该该保存点 后来又被回滚，那么该游标将保持**FETCH**使它指向的位置（也 就是说由**FETCH**导致的游标动作不会被回滚）。回滚也不能　撤销关闭一个游标。不过，其他由游标查询导致的副作用（例如被该查询所调用的易变函数的副作用）　可以被回滚，只要它们发生在一个后来被回滚的保存点期间。 如果一个游标的执行导致事务中止，它会被置于一种不能被执行的状态，这样当 事务被用**ROLLBACK TO SAVEPOINT**恢复后，该游标也不再能 被使用。

示例

要撤销在my_savepoint建立后执行的命令的效果：

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

游标位置不会受保存点回滚的影响：

```
BEGIN;

DECLARE foo CURSOR FOR SELECT 1 UNION SELECT 2;

SAVEPOINT foo;
```

```
FETCH 1 FROM foo;
 ?column?
----------
        1

ROLLBACK TO SAVEPOINT foo;

FETCH 1 FROM foo;
 ?column?
----------
        2

COMMIT;
```

兼容性

SQL标准指定关键词 SAVEPOINT是强制的，但是UXDB 和Oracle允许省略它。SQL 只允许WORK而 不是TRANSACTION作为ROLLBACK之后的噪声词。 还有，SQL 有一个可选的子句 AND [NO] CHAIN，当前 UXDB并不支持。在其他方面，这个命令符合 SQL 标准。

另见

BEGIN(7), COMMIT(7), RELEASE SAVEPOINT(7), ROLLBACK(7), SAVEPOINT(7)

名称

SAVEPOINT — 在当前事务中定义一个新的保存点

大纲

SAVEPOINT *savepoint_name*

描述

SAVEPOINT在当前事务中建立一个新保存点。

保存点是事务内的一种特殊标记，它允许所有在它被建立之后执行的命令被 回滚，把该事务的状态恢复到它处于保存点时的样子。

参数

savepoint_name

给新保存点的名字。

注解

使用ROLLBACK TO SAVEPOINT(7)回滚到一个保存点。 使用RELEASE SAVEPOINT(7)销毁一个保存点， 但保持在它被建立之后执行的命令的效果。

保存点只能在一个事务块内建立。可以在一个事务内定义多个保存点。

示例

要建立一个保存点并且后来撤销在它建立之后执行的所有命令的效果：

```
BEGIN;
   INSERT INTO table1 VALUES (1);
   SAVEPOINT my_savepoint;
   INSERT INTO table1 VALUES (2);
   ROLLBACK TO SAVEPOINT my_savepoint;
   INSERT INTO table1 VALUES (3);
COMMIT;
```

上面的事务将插入值 1 和 3，但不会插入 2。

要建立并且稍后销毁一个保存点：

```
BEGIN;
   INSERT INTO table1 VALUES (3);
   SAVEPOINT my_savepoint;
   INSERT INTO table1 VALUES (4);
   RELEASE SAVEPOINT my_savepoint;
COMMIT;
```

上面的事务将插入 3 和 4。

兼容性

当建立另一个同名保存点时，SQL 要求之前的那个保存点自动被销毁。在 UXDB中，旧的保存点会被保留，不过在进行 回滚或释放时只能使用最近的那一个（用 RELEASE SAVEPOINT释放较新的保存点将会 导致较旧的保存点再次变得可以被 ROLLBACK TO SAVEPOINT和 RELEASE SAVEPOINT访问）。在其他方面， SAVEPOINT完全符合 SQL。

另见

BEGIN(7), COMMIT(7), RELEASE SAVEPOINT(7), ROLLBACK(7), ROLLBACK TO SAVEPOINT(7)

名称

SECURITY LABEL — 定义或更改应用到一个对象的安全标签

大纲

```
SECURITY LABEL [ FOR provider ] ON
{
  TABLE object_name |
  COLUMN table_name.column_name |
  AGGREGATE aggregate_name ( aggregate_signature ) |
  DATABASE object_name |
  DOMAIN object_name |
  EVENT TRIGGER object_name |
  FOREIGN TABLE object_name
  FUNCTION function_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  LARGE OBJECT large_object_oid |
  MATERIALIZED VIEW object_name |
  [ PROCEDURAL ] LANGUAGE object_name |
  PROCEDURE procedure_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  PUBLICATION object_name |
  ROLE object_name |
  ROUTINE routine_name [ ( [ [ argmode ] [ argname ] argtype [, ...] ] ) ] |
  SCHEMA object_name |
  SEQUENCE object_name |
  SUBSCRIPTION object_name |
  TABLESPACE object_name |
  TYPE object_name |
  VIEW object_name
} IS 'label'
```

其中 *aggregate_signature* 是：

```
* |
[ argmode ] [ argname ] argtype [ , ... ] |
[ [ argmode ] [ argname ] argtype [ , ... ] ] ORDER BY [ argmode ] [ argname ] argtype [ , ... ]
```

描述

SECURITY LABEL对一个数据库对象应用一个安全 标签。可以把任意数量的安全标签（每个标签提供者对应一个）关联到一个给定 的数据库对象。标签提供者是使用函数register_label_provider注册自己的可装载模块。

> **注意**
>
> register_label_provider不是一个 SQL 函数，它只能在被载入 到后端的 C 代码中调用。

标签提供者决定一个给定标签是否合法并且它是否可以被分配该标签给一个给定 对象。一个给定标签的含义也同样由标签提供者判断。 UXDB没有对一个标签提供者是否必须或者如何解释 安全标签做出限定，它仅仅只是提供了一种机制来存储它们。实际上，这个功能是 为了允许与基

于标签的强制访问控制（MAC）系统集成（例如 SELinux）。这类系统会基于对象标签而不是传统的自主 访问控制（DAC）概念（例如用户和组）做出所有访问控制决定。

参数

object_name
table_name.column_name
aggregate_name
function_name
procedure_name
routine_name

要被贴上标签的对象的名称。表、聚集、域、外部表、函数、存储过程、例程、序列、类型和视图 的名称可以是模式限定的。

provider

这个标签相关联的提供者的名称。所提到的提供者必须已被载入并且必须赞同所提出 的标签操作。如果正好只载入了一个提供者，可以出于简洁的需要忽略提供者的名称。

argmode

一个函数，存储过程或者聚集函数参数的模式：IN、OUT、 INOUT或者VARIADIC。如果被忽略，默认值会是 IN。注意SECURITY LABEL并不真正 关心OUT参数，因为判断函数的身份时只需要输入参数。因此列出 IN、INOUT和VARIADIC参数足矣。

argname

一个函数，存储过程或者聚集函数参数的名称。注意SECURITY LABEL 并不真正关心参数的名称，因为判断函数的身份时只需要参数的数据类型。

argtype

一个函数，存储过程或聚集函数参数的数据类型。

large_object_oid

大对象的 OID。

PROCEDURAL

这是一个噪声词。

label

写成一个字符串文本的新安全标签。如果写成NULL表示删除 原有的安全标签。

示例

下面的例子展示了如何更改一个表的安全标签。

```
SECURITY LABEL FOR selinux ON TABLE mytable IS 'system_u:object_r:seuxsql_table_t:s0';
```

兼容性

在 SQL 标准中没有**SECURITY LABEL**命令。

另见

src/test/modules/dummy_seclabel

名称

SELECT, TABLE, WITH — 从一个表或视图检索行

大纲

[WITH [RECURSIVE] *with_query* [, ...]]
SELECT [ALL | TOP *top_item* | DISTINCT [ON (*expression* [, ...])]]
  [* | *expression* [[AS] *output_name*] [, ...]]
  [FROM *from_item* [, ...]]
  [WHERE *condition*]
  [START WITH *condition* CONNECT BY [NOCYCLE] *condition* | CONNECT BY
[NOCYCLE] *condition* START WITH *condition* | CONNECT BY [NOCYCLE] *condition*]
  [GROUP BY *grouping_element* [, ...]]
  [HAVING *condition* [, ...]]
  [WINDOW *window_name* AS (*window_definition*) [, ...]]
  [{ UNION | INTERSECT | EXCEPT | MINUS } [ALL | DISTINCT] *select*]
  [ORDER BY *expression* [ASC | DESC | USING *operator*] [NULLS { FIRST | LAST }] [, ...]]
  [LIMIT { *count* | ALL }]
  [OFFSET *start* [ROW | ROWS]]
  [FETCH { FIRST | NEXT } [*count*] { ROW | ROWS } ONLY]
  [FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [OF *table_name* [, ...]]
[NOWAIT | SKIP LOCKED | WAIT N] [...]]
  [q/Q] ['] [*quote_delimiter*][*c*...] [*quote_delimiter*][']

其中 *from_item* 可以是以下之一：

  [ONLY] *table_name* [*] [[AS] *alias* [(*column_alias* [, ...])]]
        [TABLESAMPLE | SAMPLE *sampling_method* (*argument* [, ...]) [REPEATABLE (*seed*
)]]
  [LATERAL] (*select*) [AS] [*alias*] [(*column_alias* [, ...])]
  with_query_name [[AS] *alias* [(*column_alias* [, ...])]]
  [LATERAL] *function_name* ([*argument* [, ...]])
        [WITH ORDINALITY] [[AS] *alias* [(*column_alias* [, ...])]]
  [LATERAL] *function_name* ([*argument* [, ...]]) [AS] *alias* (*column_definition* [, ...])
  [LATERAL] *function_name* ([*argument* [, ...]]) AS (*column_definition* [, ...])
  [LATERAL] ROWS FROM(*function_name* ([*argument* [, ...]]) [AS (*column_definition* [, ...])]
[, ...])
        [WITH ORDINALITY] [[AS] *alias* [(*column_alias* [, ...])]]
  from_item [NATURAL] *join_type from_item* [ON *join_condition* | USING (*join_column* [, ...])]

并且 *grouping_element* 可以是以下之一：

  ( )
  expression
  (*expression* [, ...])
  ROLLUP ({ *expression* | (*expression* [, ...]) } [, ...])
  CUBE ({ *expression* | (*expression* [, ...]) } [, ...])
  GROUPING SETS (*grouping_element* [, ...])

并且 *with_query* 是：

with_query_name [(*column_name* [, ...])] AS [[NOT] MATERIALIZED] (*select* | *values* | *insert* | *update* | *delete*)

TABLE [ONLY] *table_name* [*]

描述

SELECT从零或更多表中检索行。 SELECT的通常处理如下：

1. WITH列表中的所有查询都会被计算。这些查询实际 充当了在FROM列表中可以引用的临时表。在 FROM中被引用多次的WITH查 询只会被计算一次，除非另有说明，否则NOT MATERIALIZED。 （见下文的“WITH 子句”一节）。

2. FROM列表中的所有元素都会被计算（ FROM中的每一个元素都是一个真实表或者虚拟表）。 如果在FROM列表中指定了多于一个元素，它们会被 交叉连接在一起（见下文的“FROM 子句”一节）。

3. 如果指定了WHERE子句，所有不满足该条件的行都会 被从输出中消除（见下文的“WHERE 子句”一节）。

4. 如果指定了GROUP BY子句或者如果有聚集函数，输出 会被组合成由在一个或者多个值上匹配的行构成的分组，并且在其上计算聚 集函数的结果。如果出现了HAVING子句，它会消除不 满足给定条件的分组（见下文的 “GROUP BY 子句”一节以及 “HAVING 子句”一节）。

5. 对于每一个被选中的行或者行组，会使用SELECT 输出表达式计算实际的输出行（见下文的“SELECT 列表”一节）。

6. SELECT DISTINCT从结果中消除重复的行。 SELECT DISTINCT ON消除在所有指定表达式上匹 配的行。SELECT ALL（默认）将返回所有候选行， 包括重复的行（见下文的“DISTINCT 子句”一节）。

7. 通过使用操作符UNION、 INTERSECT和EXCEPT，多于 一个SELECT语句的输出可以被整合形成 一个结果集。UNION操作符返回位于一个或者两 个结果集中的全部行。INTERSECT操作符返回同时 位于两个结果集中的所有行。EXCEPT操作符返回 位于第一个结果集但不在第二个结果集中的行。在所有三种情况下， 重复行都会被消除（除非指定ALL）。可以增加噪 声词DISTINCT来显式地消除重复行。注意虽然 ALL是SELECT自身的默认行为， 但这里DISTINCT是默认行为（见下文的 “UNION 子句”一节、“INTERSECT 子句”一节以及 “EXCEPT 子句”一节）。

8. 如果指定了ORDER BY子句，被返回的行会以指定的 顺序排序。如果没有给定ORDER BY，系统会以能最 快产生行的顺序返回它们（见下文的 “ORDER BY 子句”一节）。

9. 如果指定了LIMIT（或FETCH FIRST） 或者OFFSET子句，SELECT 语句只返回结果行的一个子集（见下文的“LIMIT 子句”一节）。

10. 如果指定了FOR UPDATE、 FOR NO KEY UPDATE、 FOR SHARE 或者FOR KEY SHARE， SELECT语句会把被选中的行锁定而不让并发 更新访问它们（见下文的“锁定子句”一节）。

必须拥有在一个SELECT命令中使用的每一列上的 SELECT特权。FOR NO KEY UPDATE、 FOR UPDATE、 FOR SHARE或者FOR KEY SHARE 还要求（对这样选中的每一个表至少一列的）UPDATE 特权。

参数

WITH 子句

WITH子句允许指定一个或者多个在主查询中可以 其名称引用的子查询。在主查询期间子查询实际扮演了临时表或者视图 的角色。每一个子查询都可以是一个SELECT、 TABLE、VALUES、INSERT、 UPDATE或者 DELETE语句。在WITH中书写 一个数据修改语句（INSERT、UPDATE或者 DELETE）时，通常要包括一个 RETURNING子句。构成被主查询读取的临时表的是 RETURNING的输出，而不是该语句修改的 底层表。如果省略RETURNING，该语句仍会被执行，但是它 不会产生输出，因此它不能作为一个表从主查询引用。

对于每一个WITH查询，都必须指定一个名称（无需模 式限定）。可选地，可以指定一个列名列表。如果省略该列表，会从该子查 询中推导列名。

如果指定了RECURSIVE，则允许一个 SELECT子查询使用名称引用自身。 这样一个子查询的形式必须是

non_recursive_term UNION [ALL | DISTINCT] *recursive_term*

其中递归自引用必须出现在UNION的右手边。每个 查询中只允许一个递归自引用。不支持递归数据修改语句，但是 可以在一个数据查询语句中使用一个递归 SELECT查询的结果。例子可见第 4.8 节 “WITH查询（公共表表达式）”。

RECURSIVE的另一个效果是 WITH查询不需要被排序：一个查询可以引用另一个 在列表中比它靠后的查询（不过，循环引用或者互递归没有实现）。 如果没有RECURSIVE，WITH 查询只能引用在WITH列表中位置更前面的兄弟 WITH查询。

当WITH子句中有多个查询时，RECURSIVE应只编写一次，紧跟在WITH之后。 它适用于WITH子句中的所有查询，尽管它对不使用递归或前向引用的查询没有影响。

主查询以及WITH查询全部（理论上）在同一时间 被执行。这意味着从该查询的任何部分都无法看到 WITH中的一个数据修改语句的效果，不过可以读 取其RETURNING输出。如果两个这样的数据修改语句 尝试修改相同的行，结果将无法确定。

WITH查询的一个关键属性是，即使主查询多次引用它们，它们通常每次执行主查询只计算一次。 特别是，数据修改语句确保执行一次而且只执行一次，而与主查询是否读取它们的全部或任何输出无关。

但是，WITH查询可以标记为NOT MATERIALIZED以移除此保证。 在这种情况下，WITH查询可以折叠到主查询中，就好像它是主查询的FROM子句中的简单的sub-SELECT。 如果主查询多次引用WITH查询，则会导致重复计算;但是，如果每次此类使用只需要WITH查询的总输出中的几行，NOT MATERIALIZED可以通过允许查询联合优化来节省开销。 NOT MATERIALIZED被忽略，如果它被附加到一个递归的WITH查询，或者不是边际效应无关的（也就是说，不是包含非易失性函数的普通的SELECT）。

默认情况下，如果查询在主查询的FROM子句中仅一次使用，则边际效应无关的WITH查询将折叠到主查询中。 这允许在语义不可见的情况下两个查询级别的联合优化。 但是，通过将WITH查询标记为MATERIALIZED，可以防止此类折叠。 这可能很有用，例如，如果WITH查询被用作优化围栏，以防止规划者选择错误计划。

更多信息请见第 4.8 节 “WITH查询（公共表表达式）”。

TOP 子句

在使用select语句时，可以使用top子句来进行筛选结果。

top_item的语法如下所示。

([(]n[)])[,[(]m[)]]

n和m的类型是int，仅支持正整数。

1. TOP n 表示选择结果集的前n条记录。
2. TOP n,m 表示选择第n条记录之后的m条记录。

> **注意**
>
> TOP子句不能与LIMIT同时出现在查询语句中。

FROM 子句

FROM子句为SELECT 指定一个或者更多源表。如果指定了多个源表，结果将是所有源表的 笛卡尔积（交叉连接）。但是通常会增加限定条件（通过 WHERE）来把返回的行限制为该笛卡尔积的一个小子集。

FROM子句可以包含下列元素：

table_name

一个现有表或视图的名称（可以是模式限定的）。如果在表名前指定了 ONLY，则只会扫描该表。如果没有指定 ONLY，该表及其所有后代表（如果有）都会被扫描。可 选地，可以在表名后指定*来显式地指示包括后代表。

alias

一个包含别名的FROM项的替代名称。别名被用于让书写简洁或者消除自连接中的混淆（其中同一个表会被扫描多次）。当提供一个别名时，表或者函数的实际名称会被隐藏。例如，给定FROM foo AS f，SELECT的剩余部分就必须以f而不是foo来引用这个FROM项。如果写了一个别名，还可以写一个列别名列表来为该表的一个或者多个列提供替代名称。

TABLESAMPLE | SAMPLE *sampling_method* (*argument* [, ...]) [REPEATABLE (*seed*)]

*table_name*之后的TABLESAMPLE(同义词SAMPLE)子句表示应该用指定的*sampling_method*来检索表中行的子集。这种采样优先于任何其他过滤器（例如WHERE子句）。标准UXDB发布包括两种采样方法：BERNOULLI和SYSTEM，其他采样方法可以通过扩展安装在数据库中。

BERNOULLI以及SYSTEM采样方法都接受一个参数，它表示要采样的表的分数，表示为一个 0 到 100 之间的百分数。这个参数可以是任意的实数值表达式（其他的采样方法可能接受更多或者不同的参数）。这两种方法都返回一个随机选取的该表采样，其中包含了指定百分数的表行。BERNOULLI方法扫描整个表并且用指定的几率选择或者忽略行。SYSTEM方法会做块层的采样，每个块都有指定的机会能被选中，被选中块中的所有行都会被返回。在指定较小的采样百分数时，SYSTEM方法要比BERNOULLI方法快很多，但是前者可能由于聚簇效应返回随机性较差的表采样。

可选的REPEATABLE子句指定一个用于产生采样方法中随机数的种子数或表达式。种子值可以是任何非空浮点值。如果查询时表没有被更改，指定相同种子和*argument*值的两个查询将会选择该表相同的采样。但是不同的种子值通常将会产生不同的采样。如果没有给出REPEATABLE，则会基于一个系统产生的种子为每一个查询选择一个新的随机采样。注意有些扩展采样方法不接受REPEATABLE，并且将总是为每一次使用产生新的采样。

select

一个子-SELECT可以出现在FROM子句中。这就好像把它的输出创建为一个存在于该SELECT命令期间的临时表。注意子-SELECT必须用圆括号包围，并且必须为它提供一个别名。也可以在这里使用一个VALUES(7)命令。

with_query_name

可以通过写一个WITH查询的名称来引用它，就好像该查询的名称是一个表名（实际上，该WITH查询会为主查询隐藏任何具有相同名称的真实表。如果必要，可以使用带模式限定的方式以相同的名称来引用真实表）。可以像表一样，以同样的方式提供一个别名。

function_name

函数调用可以出现在FROM子句中（对于返回结果 集合的函数特别有用，但是可以使用任何函数）。这就好像把该函数的 输出创建为一个存在于该SELECT命令期 间的临时表。当为该函数调用增加可选的 WITH ORDINALITY子句时，会在该函数 的输出列之后追加一个新的列来为每一行编号。

可以用和表一样的方式提供一个别名。如果写了一个别名，还可以写一个 列别名列表来为该函数的组合返回类型的一个或者多个属性提供替代名称， 包括由ORDINALITY（如果有）增加的新列。

通过把多个函数调用包围在ROWS FROM(...)中可以把它们 整合在单个FROM-子句项中。这样一个项的输出是把每一个 函数的第一行串接起来，然后是每个函数的第二行，以此类推。如果有些 函数产生的行比其他函数少，则在缺失数据的地方放上空值，这样被返回 的总行数总是和产生最多行的函数一样。

如果函数被定义为返回record数据类型，那么必须出现一个 别名或者关键词AS，后面跟上形为 (*column_name data_type* [, ...])的列定义列表。列定义列表必须匹配该函数返回的列的实际 数量和类型。

在使用ROWS FROM(...)语法时，如果函数之一要求一个列 定义列表，最好把该列定义列表放在ROWS FROM(...)中该 函数的调用之后。当且仅当正好只有一个函数并且没有 WITH ORDINALITY子句时，才能把列定义列表放在 ROWS FROM(...)结构后面。

要把ORDINALITY和列定义列表一起使用，必须使用 ROWS FROM(...)语法，并且把列定义列表放在 ROWS FROM(...)里面。

join_type

One of

- [INNER] JOIN
- LEFT [OUTER] JOIN
- RIGHT [OUTER] JOIN

- FULL [OUTER] JOIN

- CROSS JOIN

对于INNER和OUTER连接类型，必须指定 一个连接条件，即 NATURAL、ON *join_condition*或者 USING (*join_column* [, ...]) 之一（只能有一种）。其含义见下文。对于CROSS JOIN，上述子句不能出现。

一个JOIN子句联合两个FROM项（ 为了方便我们称之为“表”，尽管实际上它们可以是任何类型 的FROM项）。如有必要可以使用圆括号确定嵌套的顺序。 在没有圆括号时，JOIN会从左至右嵌套。在任何情 况下，JOIN的联合比分隔FROM-列表 项的逗号更强。

CROSS JOIN和INNER JOIN 会产生简单的笛卡尔积，也就是与在FROM的顶层列出两个 表得到的结果相同，但是要用连接条件（如果有）约束该结果。 CROSS JOIN与INNER JOIN ON (TRUE)等效，也就是说条件不会移除任何行。这些连接类型只是一种 记号上的方便，因为没有什么是用纯粹的FROM和 WHERE能做而它们不能做的。

LEFT OUTER JOIN返回被限制过的笛卡尔积 中的所有行（即所有通过了其连接条件的组合行），外加左手表中 没有相应的通过了连接条件的右手行的每一行的拷贝。通过在右手列中插入空值，这种左手行会被扩展为连接表的完整行。注意在决 定哪些行匹配时，只考虑JOIN子句自身的条件。之后 才应用外条件。

相反，RIGHT OUTER JOIN返回所有连接行，外加每 一个没有匹配上的右手行（在左端用空值扩展）。这只是为了记号 上的方便，因为可以通过交换左右表把它转换成一个LEFT OUTER JOIN。

FULL OUTER JOIN返回所有连接行，外加每 一个没有匹配上的左手行（在右端用空值扩展），再外加每一个没有 匹配上的右手行（在左端用空值扩展）。

ON *join_condition*

join_condition 是一个会得到boolean类型值的表达式（类似于一个 WHERE子句），它说明一次连接中哪些行被认为 相匹配。

USING (*join_column* [, ...])

形式USING (a, b, ...)的子句是 ON left_table.a = right_table.a AND left_table.b = right_table.b ...的简写。还有， USING表示每一对相等列中只有一个会被 包括在连接输出中。

NATURAL

NATURAL是一个USING列表的速记，该列表中提到两个表中具有匹配名称的所有的列。如果没有公共列名，则NATURAL等效于ON TRUE。

LATERAL

LATERAL关键词可以放在一个 子-SELECT FROM项前面。这允许该 子-SELECT引用FROM列表中在它之 前的FROM项的列（如果没有LATERAL，每一 个子-SELECT会被独立计算并且因此不能交叉引用 任何其他的FROM项）。

LATERAL也可以放在一个函数调用 FROM项前面，但是在这种情况下它只是一个噪声 词，因为在任何情况下函数表达式都可以引用在它之前的 FROM项。

LATERAL项可以出现在FROM列表 顶层，或者一个JOIN中。在后一种情况中，它也可以引用其作为右手端的JOIN左手端上的任何项。

当一个FROM项包含LATERAL交叉引用 时，计算会如此进行：对提供被交叉引用列的FROM项的每 一行或者提供那些列的多个FROM项的每一个行集，使用该 行或者行集的那些列值计算LATERAL项。结果行会与 计算得到它们的行进行通常的连接。对来自哪些列的源表的每一行或者行 集都会重复这样的步骤。

列的源表必须以INNER或者LEFT的方式连接到 LATERAL项，否则就没有用于为LATERAL项计算每一个行集的良定行集。尽管 *X* RIGHT JOIN LATERAL *Y*这样的结构在语法上是合法的， 但实际上不允许用于在*Y*中引用 *X*。

WHERE 子句

可选的WHERE子句的形式

WHERE *condition*

其中*condition*是任一计算得到布尔类型结果的表达式。任何不满足这个条件的行都会从输出中被消除。如果用一行的实际值替换其中的变量引用后，该表达式返回真，则该行符合条件。

*condition*新增了“+”语法。“+”语法和left join，right join在执行计划和执行效率上并没有差异，在UXDB数据库中“+”语法就是对left join，right join的一个简写。

在查询分析过程中，将“+”重新转化为left join，right join的表达式去处理。

> **注意**
>
> “+”语法不支持全连接。“+”在左边表示右连接，“+”在右边表示左连接。

“+”语法左连接查询，如下所示。

select * from t_A a,t_B b where a.id=b.id(+);

“+”语法右连接查询，如下所示。

select * from t_A a,t_B b where a.id(+)=b.id;

“+”语法带子句的查询，如下所示。

select * from t_A a where exists (select 1 from t_B b where a.id(+) = b.id);

select * from t_A a where exists (select 1 from t_B b where a.id = b.id(+));

START WITH子句

层次查询（Hierarchical Query）是一种具有特殊功能的查询语句，通过它能够将分层数据按照层次关系展示出来。分层数据是指关系表中的数据之间具有层次关系。

表 8. 层次查询参数说明

参数	描述
START WITH *condition*	*condition*确定了首层数据。

参数	描述
CONNECT BY*condition*	*condition*确定了上下层次数据之间的关系。其中 PRIOR 作为 CONNECT BY 短语专用的表达式修饰符，语义为规定表达式范围内的列引用指向父层的列，未被 PRIOR 修饰的列引用指向待查询的关系对象的列；NOCYCLE 关键词存在时查询将舍弃层次关系死循环行，NOCYCLE 不存在时查询遇到死循环行时将报告异常，导致查询失败。
ORDER SIBLINGS BY*expression* ...	确定每一层数据的排序条件。

层次查询的目标表达式中可以包含如下的伪列和函数。

表 9. 层次查询的伪列和函数

参数	描述
prior expr	表示父行的 expr 表达式的值。
level	表示本行的层次顺序数，顶层行的level为整数1，其余层行的level为父行level+1。
connect_by_root c_expr	表示本行的顶层祖先行对应的 c_expr 表达式的值。
connect_by_iscycle	本行存在循环的子行时为整数1，否则为0。
connect_by_isleaf	本行不存在非循环的子行时为整数1，否则为0。
sys_connect_by_path(a_expr_l,a_expr_r)	顶层行表达为字符串并 (a_expr_r || a_expr_l)，其余行表达为字符串并 (父行本表达式 || a_expr_r || a_expr_l)。

注意

order by短语会破坏数据的层次关系，不建议和层次查询搭配使用，应使用order siblings by短语。

GROUP BY 子句

可选的GROUP BY子句的形式

GROUP BY *grouping_element* [, ...]

GROUP BY语句用于结合聚合函数，根据一个或多个列对结果集进行分组。

GROUP BY语句有两种使用方式，如下所示。

- 语义明确的列是指在select中出现的字段必须在group by子句中或者聚合函数中出现。
- 语义不明确的列是指在select中出现的字段不用在group by子句中或者聚合函数中出现。

GROUP BY将会把所有被选择的行中共享相同分组表达式值的那些行压缩成一个行。一个被用在*grouping_element*中的*expression*可以是输入列名、输出列（SELECT列表项）的名称或序号或者

由输入列值构成的任意表达式。在出现歧义时，GROUP BY名称将被解释为输入列名而不是输出列名。

如果任何GROUPING SETS、ROLLUP或者CUBE作为分组元素存在，则GROUP BY子句整体上定义了数个独立的分组集。其效果等效于在子查询间构建一个UNION ALL，子查询带有分组集作为它们的GROUP BY子句。处理分组集的进一步细节请见第 4.2.4 节 GROUPING SETS、CUBE和ROLLUP”。

聚集函数（如果使用）会在组成每一个分组的所有行上进行计算，从而为每一个分组产生一个单独的值（如果有聚集函数但是没有GROUP BY子句，则查询会被当成是由所有选中行构成的一个单一分组）。传递给每一个聚集函数的行集合可以通过在聚集函数调用附加一个FILTER子句来进一步过滤，详见第 1.2.7 节 “聚集表达式”。当存在一个FILTER子句时，只有那些匹配它的行才会被包括在该聚集函数的输入中。

当存在GROUP BY子句或者任何聚集函数时， SELECT列表表达式不能引用非分组列（除非它 出现在聚集函数中或者它函数依赖于分组列），因为这样做会导致返回 非分组列的值时会有多种可能的值。如果分组列是包含非分组列的表的主键（ 或者主键的子集），则存在函数依赖。

记住所有的聚集函数都是在HAVING子句或者 SELECT列表中的任何“标量”表达式之前被计算。 这意味着一个CASE表达式不能被用来跳过一个聚集表达式的 计算，见第 1.2.14 节 “表达式计算规则”。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能和 GROUP BY一起指定。

grouping sets和having有限制，group by grouping sets(())不能和having子句一起使用。

在distinct子句和order by子句一起使用时，支持order by子句中的字段，可以不在distinct target list中出现。

若需要出现语义不明确的列时，设置sql_mode值为’’；需要语义明确的列时，设置sql_mode值为ONLY_FULL_GROUP_BY。标准模式和兼容模式配置sql_mode参数有两种方法，如下所示。

1. 修改配置文件uxsinodb.conf中sql_mode的值，默认值为ONLY_FULL_GROUP_BY。

2. 在客户端uxsql中，设置sql_mode参数。

GROUP BY示例，如下所示。

```
--同时使用group by和order by子句进行查询
select a, b from tab group by a, b order by c;

--同时使用group by和having子句进行查询
select a, b from tab group by a, b having d < 5;

--同时使用group by、order by和having子句进行查询
select a, e from tab group by a, e having e > 5 order by a;

--使用distinct子句进行查询
select distinct a, b from tab having c > 4;
select distinct a, e from tab group by a having d > 5 order by c;
select distinct a, e from tab group by a, b, e, d having d > 5 order by e;

--在group by子句中使用grouping sets函数
```

```
select a, b, avg(c) from tab group by grouping sets (a, b, ());
select a, b, avg(c) from tab group by grouping sets (a, b, ()) having a  < 2;

--使用grouping sets(()) having进行查询
select 1 from tab group by grouping sets (a,()) having a < 2;
```

HAVING 子句

可选的HAVING子句的形式

HAVING *condition*

其中*condition*与 WHERE子句中指定的条件相同。

HAVING消除不满足该条件的分组行。 HAVING与WHERE不同： WHERE会在应用GROUP BY之前过滤个体行，而HAVING过滤由 GROUP BY创建的分组行。 *condition*中引用 的每一个列必须无歧义地引用一个分组列（除非该引用出现在一个聚集 函数中或者该非分组列函数依赖于分组列。

即使没有GROUP BY子句，HAVING 的存在也会把一个查询转变成一个分组查询。这和查询中包含聚集函数但没有 GROUP BY子句时的情况相同。所有被选择的行都被认为是一个 单一分组，并且SELECT列表和 HAVING子句只能引用聚集函数中的表列。如果该 HAVING条件为真，这样一个查询将会发出一个单一行； 否则不返回行。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能与HAVING一起指定。

WINDOW 子句

可选的WINDOW子句的形式

WINDOW *window_name* AS (*window_definition*) [, ...]

其中*window_name* 是一个可以从OVER子句或者后续窗口定义中引用的名称。*window_definition*是

[*existing_window_name*]
[PARTITION BY *expression* [, ...]]
[ORDER BY *expression* [ASC | DESC | USING *operator*] [NULLS { FIRST | LAST }] [, ...]]
[*frame_clause*]

如果指定了一个*existing_window_name*， 它必须引用WINDOW列表中一个更早出现的项。新窗口将从 该项中复制它的划分子句以及排序子句（如果有）。在这种情况下，新窗口 不能指定它自己的PARTITION BY子句，并且它只能在被复制 窗口没有ORDER BY的情况下指定该子句。新窗口总是使用它 自己的帧子句，被复制的窗口不必指定一个帧子句。

PARTITION BY列表元素的解释以 “GROUP BY 子句”一节元素的方式 进行，不过它们总是简单表达式并且绝不能是输出列的名称或编号。另一个区 别是这些表达式可以包含聚集函数调用，而这在常规GROUP BY 子句中是不被允许的。它们被允许的原因是窗口是出现在分组和聚集之后的。

类似地，ORDER BY列表元素的解释也以 “ORDER BY 子句”一节元素的方式进行， 不过该表达式总是被当做简单表达式并且绝不会是输出列的名称或编号。

可选的*frame_clause*为依赖帧的窗口函数 定义窗口帧（并非所有窗口函数都依赖于帧）。窗口帧是查询中 每一样（称为当前行）的相关行的集合。 *frame_clause*可以是

{ RANGE | ROWS | GROUPS } *frame_start* [*frame_exclusion*]
{ RANGE | ROWS | GROUPS } BETWEEN *frame_start* AND *frame_end* [*frame_exclusion*]

之一，其中*frame_start*和*frame_end*可以是

UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING

之一，并且*frame_exclusion*可以是

EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS

之一。如果省略*frame_end*，它会被默认为CURRENT ROW。限制是： *frame_start*不能是UNBOUNDED FOLLOWING， *frame_end*不能是UNBOUNDED PRECEDING， 并且*frame_end*的选择在上面of *frame_start*以及*frame_end* 选项的列表中不能早于 *frame_start*的选择— 例如 RANGE BETWEEN CURRENT ROW AND *offset* PRECEDING是不被允许的。

默认的帧选项是RANGE UNBOUNDED PRECEDING，它和 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW相同。它把帧设置为从分区开始直到当前行的最后一个平级行（被该窗口的ORDER BY子句认为等价于当前行的行，如果没有ORDER BY则所有的行都是平级的）。通常， UNBOUNDED PRECEDING表示从分区第一行开始的帧，类似地 UNBOUNDED FOLLOWING表示以分区最后一行结束的帧，不论是处于RANGE、ROWS或者GROUPS模式中。在ROWS模式中， CURRENT ROW表示以当前行开始或者结束的帧。而在 RANGE或者GROUPS模式中它表示当前行在ORDER BY排序中的第一个 或者最后一个平级行开始或者结束的帧。 *offset* PRECEDING和*offset* FOLLOWING选项的含义会随着帧模式而变化。在ROWS模式中，*offset*是一个整数，表示帧开始或者结束于当前行之前或者之后的那么多行处。在GROUPS模式中，*offset*是一个整数，表示真开始或者结束于当前行的平级组之前或者之后那么多个平级组处，其中平级组是一组根据窗口的ORDER BY子句等效的行。在RANGE模式中，*offset*选项的使用要求在窗口定义中正好有一个ORDER BY列。那么该帧包含的行的排序列值不超过*offset*且小于（对于PRECEDING）或者大于（对于FOLLOWING）当前行的排序列值。在这些情况中，*offset*表达式的数据类型取决于排序列的数据类型。对于数字排序列，它通常与排序列是相同类型，但对于datetime类型的排序列它是interval。在所有这些情况中，*offset*的值必须是非空和非负。此外，虽然*offset*并非必须是简单常量，但它不能包含变量、聚集函数或者窗口函数。

*frame_exclusion*选项允许从帧中排除当前行周围的行，即便根据帧的起始选项来说它们应该被包含在帧中。EXCLUDE CURRENT ROW把当前行从帧中排除。EXCLUDE GROUP把当前行和它

在排序上的平级行从帧中排除。EXCLUDE TIES从帧中排除当前行的任何平级行，但是不排除当前行本身。EXCLUDE NO OTHERS只是明确地指定不排除当前行或其平级行的默认行为。

注意，如果ORDER BY排序无法把行唯一地排序，则ROWS模式可能产生不可预测的结果。RANGE以及GROUPS模式的目的是确保在ORDER BY顺序中平等的行被同样对待：一个给定平级组中的所有行将在一个帧中或者被从帧中排除。

WINDOW子句的目的是指定出现在查询的“SELECT 列表”一节或“ORDER BY 子句”一节中的窗口函数的行为。这些函数可以在它们的OVER子句中用名称引用WINDOW子句项。不过，WINDOW子句项不是必须被引用。如果在查询中没有用到它，它会被简单地忽略。可以使用根本没有任何WINDOW子句的窗口函数，因为窗口函数调用可以直接在其OVER子句中指定它的窗口定义。不过，当多个窗口函数都需要相同的窗口定义时，WINDOW子句能够减少输入。

当前，FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE和FOR KEY SHARE不能和WINDOW一起被指定。

窗口函数的详细描述在第 1.2.8 节“窗口函数调用”以及第 4.2.5 节“窗口函数处理”中。

q/Q子句

在字符串中输入一些特殊符号（比如单引号、双引号等）时，需要在每个特殊符号前加一个转义符号。当符号较多时，会导致输入非常不直观。

使用q’转义字符语法可以在输入特殊符号时不需要增加转义符号，使长文本、多符号的字符串输入更直观。

q/Q ['] [*quote_delimiter*][*c*...] [*quote_delimiter*][']

- q和Q只有一个，两个单引号、两个定界符字段必须有。
- q或Q表示将使用替代转义机制。这种机制允许为文本字符串使用多种定界符(quote_delimiter)。
- 最外面’’是两个单引号，分别在开口定界符之前和闭合定界符之后。
- c是用户字符集的任何成员，可以是引号（“），可以是定界符（没有紧随其后的单引号）。
- 定界符是除单引号、制表符和返回值之外的任何单字节。
- 如果开口定界符是[，{，<，或(，那么闭合定界符必须是相应的]，}，>，或)。在所有其他情况下，开口和闭合定界符必须使用相同的字符。

SELECT 列表

SELECT列表（位于关键词SELECT和FROM之间）指定构成SELECT语句输出行的表达式。这些表达式可以（并且通常确实会）引用FROM子句中计算得到的列。

正如在表中一样，SELECT的每一个输出列都有一个名称。在一个简单的SELECT中，这个名称只是被用来标记要显示的列，但是当SELECT是一个大型查询的一个子查询时，大型查询会把该名称看做子查询产生的虚表的列名。要指定用于输出列的名称，在该列的表达式后面写上AS *output_name*（可以省略AS，但只能在期望的输出名称不匹配任何UXDB关键词（见附录 A,）时省略。为了避免和未来增加的关键词冲突，推荐总是写上AS或者用双引号引用输出名称）。如果不指定列名，UXDB会自动选择一个名称。如果列的表达式是一个简单的列引用，那么被选

择的名称就和该列的名称相同。在使用函数或者类型名称 的更复杂的情况中，系统可能会生成诸如 ?column?之类的名称。

一个输出列的名称可以被用来在ORDER BY以及 GROUP BY子句中引用该列的值，但是不能用于 WHERE和HAVING子句（在其中 必须写出表达式）。

可以在输出列表中写*来取代表达式，它是被选中 行的所有列的一种简写方式。还可以写 *table_name*.*，它 是只来自那个表的所有列的简写形式。在这些情况中无法用 AS指定新的名称，输出行的名称将和表列的名称相同。

根据 SQL 标准，输出列表中的表达式应该在应用DISTINCT、ORDER BY或者LIMIT之前计算。在使用DISTINCT时显然必须这样做，否则就无法搞清到底在区分什么值。不过，在很多情况下如果先计算ORDER BY和LIMIT再计算输出表达式会很方便，特别是如果输出列表中包含任何 volatile 函数或者代价昂贵的函数时尤其如此。通过这种行为，函数计算的顺序更加直观并且对于从未出现在输出中的行将不会进行计算。只要输出表达式没有被DISTINCT、ORDER BY或者GROUP BY引用，UXDB实际将在排序和限制行数之后计算输出表达式（作为一个反例，SELECT f(x) FROM tab ORDER BY 1显然必须在排序之前计算f(x)）。包含有集合返回函数的输出表达式实际是在排序之后和限制行数之前被计算，这样LIMIT才能切断来自集合返回函数的输出。

DISTINCT 子句

如果指定了SELECT DISTINCT，所有重复的行会被从结果集中移除（为每一组重复的行保留一行）。SELECT ALL则指定相反的行为：所有行都会被保留，这也是默认情况。

SELECT DISTINCT ON (*expression* [, ...])只保留在给定表达式上计算相等的行集合中的第一行。DISTINCT ON表达式使用和ORDER BY相同的规则（见上文）解释。注意，除非用ORDER BY来确保所期望的行出现在第一位，每一个集合的“第一行”是不可预测的。例如：

```
SELECT DISTINCT ON (location) location, time, report
  FROM weather_reports
  ORDER BY location, time DESC;
```

为每个地点检索最近的天气报告。但是如果我们不使用ORDER BY来强制对每个地点的时间值进行降序排序，我们为每个地点得到的报告的时间可能是无法预测的。

DISTINCT ON表达式必须匹配最左边的ORDER BY表达式。ORDER BY子句通常将包含额外的表达式，这些额外的表达式用于决定在每一个DISTINCT ON分组内行的优先级。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE不能和DISTINCT一起使用。

兼容模式下支持distinct与order by子查询排序。在UXDB标准模式，仅支持distinct与order by子查询顺序排序，不支持逆序排序。

使用该功能需要初始化兼容模式的数据库，即初始化阶段增加参数--running-mode compatible。

表 10. 子查询排序的处理区别表

功能	UXDB标准模式	UXDB兼容模式	备注
子查询正序	正序排列，null为最大值	正序排列，null为最大值	select distinct on(id) id

功能	UXDB标准模式	UXDB兼容模式	备注
			from (select id from t_distinct order by id asc);
子查询逆序	乱序	逆序排列，null为最大值	select distinct on(id) id from (select id from t_distinct order by id desc);
是否null值去重	是	是	

UNION 子句

UNION子句具有下面的形式：

select_statement UNION [ALL | DISTINCT] *select_statement*

select_statement 是任何没有ORDER BY、LIMIT、 FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和FOR KEY SHARE子句的 SELECT语句（如果子表达式被包围在圆括号内， ORDER BY和LIMIT可以被附着到其上。如果没有 圆括号，这些子句将被应用到UNION的结果而不是右手边 的表达式上）。

UNION操作符计算所涉及的 SELECT语句所返回的行的并集。如果一行 至少出现在两个结果集中的一个内，它就会在并集中。作为 UNION两个操作数的 SELECT语句必须产生相同数量的列并且对应位置上的列必须具有兼容的数据类型。

UNION的结果不会包含重复行，除非指定了 ALL选项。ALL会阻止消除重复（因此， UNION ALL通常显著地快于UNION， 尽量使用ALL）。可以写DISTINCT来 显式地指定消除重复行的行为。

除非用圆括号指定计算顺序， 同一个SELECT语句中的多个 UNION操作符会从左至右计算。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和 FOR KEY SHARE不能用于UNION结果或者 UNION的任何输入。

INTERSECT 子句

INTERSECT子句具有下面的形式：

select_statement INTERSECT [ALL | DISTINCT] *select_statement*

select_statement 是任何没有ORDER BY，LIMIT、FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE以及FOR KEY SHARE子句的 SELECT语句。

INTERSECT操作符计算所涉及的 SELECT语句返回的行的交集。如果 一行同时出现在两个结果集中，它就在交集中。

INTERSECT的结果不会包含重复行，除非指定了 ALL选项。如果有ALL，一个在左表中有 *m*次重复并且在右表中有*n* 次重复的行将会在结果中出现 min(*m*,*n*) 次。 DISTINCT可以写DISTINCT来 显式地指定消除重复行的行为。

除非用圆括号指定计算顺序， 同一个SELECT语句中的多个 INTERSECT操作符会从左至右计算。 INTERSECT的优先级比 UNION更高。也就是说， A UNION B INTERSECT C将被读成A UNION (B INTERSECT C)。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和 FOR KEY SHARE不能用于INTERSECT结果或者 INTERSECT的任何输入。

EXCEPT 子句

EXCEPT子句具有下面的形式：

select_statement EXCEPT [ALL | DISTINCT] *select_statement*

select_statement 是任何没有ORDER BY、LIMIT、FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE以及FOR KEY SHARE子句的 SELECT语句。

EXCEPT操作符计算位于左 SELECT语句的结果中但不在右 SELECT语句结果中的行集合。

EXCEPT的结果不会包含重复行，除非指定了 ALL选项。如果有ALL，一个在左表中有 *m*次重复并且在右表中有 *n*次重复的行将会在结果集中出现 max(*m*-*n*,0) 次。 DISTINCT可以写DISTINCT来 显式地指定消除重复行的行为。

除非用圆括号指定计算顺序， 同一个SELECT语句中的多个 EXCEPT操作符会从左至右计算。EXCEPT的优先级与 UNION相同。

当前，FOR NO KEY UPDATE、FOR UPDATE、 FOR SHARE和 FOR KEY SHARE不能用于EXCEPT结果或者 EXCEPT的任何输入。

MINUS 子句

可选的ORDER BY子句的形式如下所示。

select_statement1 MINUS [ALL | DISTINCT]*select_statement2*

MINUS关键字使两个同类型结果集之间做差运算，返回位于第一个结果集但不在第二个结果集中的行。

MINUS返回所有在select_statement1的结果中但是不在select_statement2的结果中的行（也叫做两个查询的差）。为了计算两个查询的差，这两个查询必须是“操作兼容的”，意味着它们都返回同样数量的列，并且对应的列有兼容的数据类型。

ORDER BY 子句

可选的ORDER BY子句的形式如下：

ORDER BY *expression* [ASC | DESC | USING *operator*] [NULLS { FIRST | LAST }] [, ...]

ORDER BY子句导致结果行被按照指定的表达式排序。 如果两行按照最左边的表达式是相等的，则会根据下一个表达式比较它们， 依次类推。如果按照所有指定的表达式它们都是相等的，则它们被返回的 顺序取决于实现。

每一个*expression* 可以是输出列（SELECT列表项）的名称或 者序号，它也可以是由输入列值构成的任意表达式。

序号指的是输出列的顺序（从左至右）位置。这种特性可以为不具有唯一 名称的列定义一个顺序。这不是绝对必要的，因为总是可以使用 AS子句为输出列赋予一个名称。

也可以在ORDER BY子句中使用任意表达式，包括没 有出现在SELECT输出列表中的列。因此，下面的语句是合法的：

```
SELECT name FROM distributors ORDER BY code;
```

这种特性的一个限制是一个应用在UNION、 INTERSECT或EXCEPT子句结果上的 ORDER BY只能指定输出列名称或序号，但不能指定表达式。

如果一个ORDER BY表达式是一个既匹配输出列名称又匹配 输入列名称的简单名称，ORDER BY将把它解读成输出列名 称。这与在同样情况下GROUP BY会做出的选择相反。这种 不一致是为了与 SQL 标准兼容。

可以为ORDER BY子句中的任何表达式之后增加关键词 ASC（上升）DESC（下降）。如果没有指定， ASC被假定为默认值。或者，可以在USING 子句中指定一个特定的排序操作符名称。一个排序操作符必须是某个 B-树操作符族的小于或者大于成员。ASC通常等价于 USING <而DESC通常等价于 USING >（但是一种用户定义数据类型的创建者可以 准确地定义默认排序顺序是什么，并且它可能会对应于其他名称的操作符）。

如果指定NULLS LAST，空值会排在非空值之后；如果指定 NULLS FIRST，空值会排在非空值之前。如果都没有指定， 在指定或者隐含ASC时的默认行为是NULLS LAST， 而指定或者隐含DESC时的默认行为是 NULLS FIRST（因此，默认行为是空值大于非空值）。 当指定USING时，默认的空值顺序取决于该操作符是否为 小于或者大于操作符。

注意顺序选项只应用到它们所跟随的表达式上。例如 ORDER BY x, y DESC和 ORDER BY x DESC, y DESC是不同的。

字符串数据会被根据引用到被排序列上的排序规则排序。根据需要可以通过在 *expression*中包括一个 COLLATE子句来覆盖，例如 ORDER BY mycolumn COLLATE "en_US"。更多信息请见第 1.2.10 节 “排序规则表达式”

LIMIT 子句

LIMIT子句由两个独立的子句构成：

```
LIMIT { count | ALL }
OFFSET start
```

*count*指定要返回 的最大行数，而*start* 指定在返回行之前要跳过的行数。在两者都被指定时，在开始计算要返回的 *count*行之前会跳过 *start*行。

如果*count*表达式计算 为 NULL，它会被当成LIMIT ALL，即没有限制。如果 *start*计算为 NULL，它会被当作OFFSET 0。

SQL:2008 引入了一种不同的语法来达到相同的结果， UXDB也支持它：

OFFSET *start* { ROW | ROWS }
FETCH { FIRST | NEXT } [*count*] { ROW | ROWS } ONLY

在这种语法中，标准要求*start*或*count*是一个文本常量、一个参数或者一个变量名。而作为一种UXDB的扩展，还允许其他的表达式，但通常需要被封闭在圆括号中以避免歧义。如果在一个FETCH子句中省略 *count*，它的默认值为 1。 ROW和ROWS以及 FIRST和NEXT是噪声，它们不影响 这些子句的效果。根据标准，如果都存在，OFFSET子句 必须出现在FETCH子句之前。但是 UXDB更宽松，它允许两种顺序。

在使用LIMIT时，用一个ORDER BY子句把 结果行约束到一个唯一顺序是个好办法。否则讲得到该查询结果行的 一个不可预测的子集 — 可能要求从第 10 到第 20 行，但是在 什么顺序下的第 10 到第 20 呢？除非指定ORDER BY， 是不知道顺序的。

查询规划器在生成一个查询计划时会考虑LIMIT，因此 根据使用的LIMIT和OFFSET，很可能 得到不同的计划（得到不同的行序）。所以，使用不同的 LIMIT/OFFSET值来选择一个查询结果的 不同子集将会给出不一致的结果，除非 用ORDER BY强制一种可预测的结果顺序。这不是一个 缺陷，它是 SQL 不承诺以任何特定顺序（除非使用 ORDER BY来约束顺序）给出一个查询结果这一事实造 成的必然后果。

如果没有一个ORDER BY来强制选择一个确定的子集， 重复执行同样的LIMIT查询甚至可能会返回一个表中行 的不同子集。同样，这也不是一种缺陷，再这样一种情况下也无法 保证结果的确定性。

锁定子句

FOR UPDATE、FOR NO KEY UPDATE、 FOR SHARE和FOR KEY SHARE 是锁定子句，它们影响SELECT 把行从表中取得时如何对它们加锁。

锁定子句的一般形式：

FOR *lock_strength* [OF *table_name* [, ...]] [NOWAIT | SKIP LOCKED | WAIT N]

其中*lock_strength*可以是

UPDATE
NO KEY UPDATE
SHARE
KEY SHARE

更多关于每一种行级锁模式的信息可见 第 10.3.2 节 “行级锁”。

为了防止该操作等待其他事务提交，可使用NOWAIT或者SKIP LOCKED选项。使用NOWAIT时，如果选中的行不能被立即锁定，该语句会报告错误而不是等待。使用SKIP LOCKED时，无法被立即锁定的任何选中行都会被跳过。跳过已锁定行会提供数据的一个不一致的视图，因此这不适合于一般目的的工作，但是可以被用来避免多个用户访问一个类似队列的表时出现锁竞争。注意NOWAIT和SKIP LOCKED只适合行级锁 — 所要求的ROW SHARE表级锁仍然会以常规的方式（见第 10 章 并发控制取得。如果想要不等待的表级锁，可以先使用带NOWAIT的LOCK(7)。

防止FOR UPDATE语句长时间等待其他事务提交。SELECT FOR UPDATE的锁定子句中增加语法WAIT N（N表示秒数），如果选中的行在N秒不能被立即锁定，则该语句会在N秒后结束事务并报告错误。

WAIT N说明：N表示等待获取对应表或者行锁的时间，单位：秒，类型：integer，范围 between 0 and 2147483。当超过Ns未获取到对应的锁时，报错。在表加上排它锁的时候（lock table），那么返回结果直到表被解锁。当在Ns内获取对应表或者行锁的时候，则正常返回结果。

select * from test for update wait 10;

如果在一个锁定子句中提到了特定的表，则只有来自于那些表的行会被锁定，任何SELECT中用到的其他表还是被简单地照常读取。一个没有表列表的锁定子句会影响该语句中用到的所有表。如果一个锁定子句被应用到一个视图或者子查询，它会影响在该视图或子查询中用到的所有表。不过，这些子句不适用于主查询引用的WITH查询。如果希望在一个WITH查询中发生行锁定，应该在该WITH查询内指定一个锁定子句。

如果有必要对不同的表指定不同的锁定行为，可以写多个锁定子句。 如果同一个表在多于一个锁定子句中被提到（或者被隐式的影响到）， 那么会按照所指定的最强的锁定行为来处理它。类似地，如果在任何 影响一个表的子句中指定了NOWAIT，就会按照 NOWAIT的行为来处理该表。否则如果 SKIP LOCKED在任何影响该表的子句中被指定， 该表就会被按照SKIP LOCKED来处理。

如果被返回的行无法清晰地与表中的行保持一致，则不能使用锁定子句。 例如锁定子句不能与聚集一起使用。

当一个锁定子句出现在一个SELECT查询的顶层时， 被锁定的行正好就是该查询返回的行。在连接查询的情况下，被锁定 的行是那些对返回的连接行有贡献的行。此外，自该查询的快照起满足 查询条件的行将被锁定，如果它们在该快照后被更新并且不再满足 查询条件，它们将不会被返回。如果使用了LIMIT，只要 已经返回的行数满足了限制，锁定就会停止（但注意被OFFSET跳过的行将被锁定）。类似地，如果在一个游标 的查询中使用锁定子句，只有被该游标实际取出或者跳过的行才将被 锁定。

当一个锁定子句出现在一个子-SELECT中时，被锁定 行是那些该子查询返回给外层查询的行。这些被锁定的行的数量可能比 从子查询自身的角度看到的要少，因为来自外层查询的条件可能会被用 来优化子查询的执行。例如：

SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss WHERE col1 = 5;

将只锁定具有col1 = 5的行（虽然在子查询中并没有写上 该条件）。

早前的发行无法维持一个被之后的保存点升级的锁。例如，这段代码：

BEGIN;
SELECT * FROM mytable WHERE key = 1 FOR UPDATE;
SAVEPOINT s;
UPDATE mytable SET ... WHERE key = 1;
ROLLBACK TO s;

在ROLLBACK TO之后将无法维持FOR UPDATE锁。已经修复这个问题。

> **注意**
>
> 一个运行在READ COMMITTED事务隔离级别并且使用ORDER BY和锁定子句的SELECT命令有可能返回无序的行。 这是因为ORDER BY会被首先应用。该命令对结果排序，但是可能 接着在尝试获得一个或者多个行上的锁

时阻塞。一旦SELECT解除 阻塞，某些排序列值可能已经被修改，从而导致那些行变成无序的（尽管它们根 据原始列值是有序的）。根据需要，可以通过在子查询中放置 FOR UPDATE/SHARE来解决之一问题，例如

SELECT * FROM (SELECT * FROM mytable FOR UPDATE) ss ORDER BY column1;

注意这将导致锁定mytable的所有行，而顶层的 FOR UPDATE只会锁定实际被返回的行。这可能会导致显著的 性能差异，特别是把ORDER BY与LIMIT或者其他 限制组合使用时。因此只有在并发更新排序列并且要求严格的排序结果时才推 荐使用这种技术。

在REPEATABLE READ或者SERIALIZABLE 事务隔离级别上这可能导致一个序列化失败（SQLSTATE 是'40001'），因此在这些隔离级别下不可能收到无序行。

TABLE 命令

命令

TABLE *name*

等价于

SELECT * FROM *name*

它可以被用作一个顶层命令，或者用在复杂查询中以节省空间。只有 WITH、UNION、INTERSECT、EXCEPT、 ORDER BY、LIMIT、OFFSET、 FETCH以及FOR锁定子句可以用于 TABLE。不能使用WHERE子句和任何形式 的聚集。

示例

把表films与表 distributors连接：

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

      title       | did |     name     | date_prod |   kind
------------------+-----+--------------+------------+----------
 The Third Man    | 101 | British Lion | 1949-12-23 | Drama
 The African Queen | 101 | British Lion | 1951-08-11 | Romantic
 ...
```

要对所有电影的len列求和并且用 kind对结果分组：

```
SELECT kind, sum(len) AS total FROM films GROUP BY kind;
```

```
  kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
```

要对所有电影的len列求和、对结果按照 kind分组并且显示总长小于 5 小时的分组：

```
SELECT kind, sum(len) AS total
   FROM films
   GROUP BY kind
   HAVING sum(len) < interval '5 hours';
```

```
  kind   | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
```

下面两个例子都是根据第二列（name）的内容来排序结果：

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;
```

```
 did |       name
-----+------------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
```

接下来的例子展示了如何得到表distributors和 actors的并集，把结果限制为那些在每个表中以 字母 W 开始的行。只想要可区分的行，因此省略了关键词 ALL。

```
distributors:              actors:
 did |     name            id |     name
-----+--------------       ----+----------------
 108 | Westward                1 | Woody Allen
 111 | Walt Disney             2 | Warren Beatty
 112 | Warner Bros.            3 | Walter Matthau
 ...                       ...
```

```
SELECT distributors.name
   FROM distributors
   WHERE distributors.name LIKE 'W%'
UNION
SELECT actors.name
   FROM actors
   WHERE actors.name LIKE 'W%';

     name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

这个例子展示了如何在FROM子句中使用函数， 分别使用和不使用列定义列表：

```
CREATE FUNCTION distributors(int) RETURNS SETOF distributors AS $$
   SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors(111);
 did |    name
-----+-------------
 111 | Walt Disney

CREATE FUNCTION distributors_2(int) RETURNS SETOF record AS $$
   SELECT * FROM distributors WHERE did = $1;
$$ LANGUAGE SQL;

SELECT * FROM distributors_2(111) AS (f1 int, f2 text);
 f1  |     f2
-----+-------------
 111 | Walt Disney
```

这里是带有增加的序数列的函数的例子：

```
SELECT * FROM unnest(ARRAY['a','b','c','d','e','f']) WITH ORDINALITY;
 unnest | ordinality
--------+----------
 a      |        1
 b      |        2
 c      |        3
 d      |        4
 e      |        5
 f      |        6
(6 rows)
```

这个例子展示了如何使用简单的WITH子句：

```
WITH t AS (
    SELECT random() as x FROM generate_series(1, 3)
  )
SELECT * FROM t
UNION ALL
SELECT * FROM t

        x
--------------------
 0.534150459803641
 0.520092216785997
0.0735620250925422
 0.534150459803641
 0.520092216785997
0.0735620250925422
```

注意该WITH查询只被计算一次，这样我们得到的两个 集合具有相同的三个随机值。

这个例子使用WITH RECURSIVE从一个只显示 直接下属的表中寻找雇员 Mary 的所有下属（直接的或者间接的）以及他们的间接层数：

```
WITH RECURSIVE employee_recursive(distance, employee_name, manager_name) AS (
    SELECT 1, employee_name, manager_name
    FROM employee
    WHERE manager_name = 'Mary'
  UNION ALL
    SELECT er.distance + 1, e.employee_name, e.manager_name
    FROM employee_recursive er, employee e
    WHERE er.employee_name = e.manager_name
  )
SELECT distance, employee_name FROM employee_recursive;
```

注意这种递归查询的典型形式：一个初始条件，后面跟着 UNION，然后是查询的递归部分。要确保 查询的递归部分最终将不返回任何行，否则该查询将无限循环（ 更多例子见第 4.8 节“WITH查询（公共表表达式）”）。

这个例子使用LATERAL为manufacturers 表的每一行应用一个集合返回函数get_product_names()：

```
SELECT m.name AS mname, pname
FROM manufacturers m, LATERAL get_product_names(m.id) pname;
```

当前没有任何产品的制造商不会出现在结果中，因为这是一个内连接。 如果我们希望把这类制造商的名称包括在结果中，我们可以：

```
SELECT m.name AS mname, pname
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true;
```

子查询支持无别名示例，如下所示。

```
select * from (select id from test_a where id < 3);
```

```
select * from (select id from test_a where id < 3) as a;
select * from (select id from test_a where id < 3) a;
```

TOP子句示例如下所示。

```
select top 1,3 * from test;
select top 3 id,name from test;
select top (1) (id) from test01;
select top (1),(4) * from test01;
```

MINUS子句示例如下所示。

```
select * from tbtest_101 minus select * from tbtest_102;
```

层次查询示例如下所示。

```
--创建测试表
create table player(keyid int,parent_keyid int,name varchar(16),salary int,sex varchar(4));
insert into player values(1,0,'zhangsan','1000000','f');
insert into player values(2,1,'lisi','50500','m');
insert into player values(3,1,'wangwu','60000','m');
insert into player values(4,1,'houzi','65000','m');
insert into player values(5,2,'maliu','30000','f');
insert into player values(6,2,'liuqi','25000','m');
insert into player values(7,4,'gouba','23000','m');
insert into player values(8,4,'dujiu','21000','f');

--自上而下遍历，查询以keyid为n开始的节点的所有子孙节点
select keyid,parent_keyid,name,prior name,salary
from player
start with keyid=1
connect by prior keyid = parent_keyid;

select keyid,parent_keyid,name,prior name,salary
  from player
  start with keyid=2
  connect by prior keyid = parent_keyid;

--自下而上遍历，查询以keyid为6的所有祖先节点
select keyid,parent_keyid,name,salary
from player
start with keyid = 6
connect by prior parent_keyid = keyid;
```

q’转义字符示例如下所示。

```
select q'!name LIKE '%DBMS_%%'!';
select q'*!名称类似’%DBMS_%%'!*';
select q'"name like '['";
select q'<'So,' she said, 'It's finished.'>';
select q'{SELECT * FROM employees WHERE last_name = 'Smith';}';
select q'[SELECT * FROM employees WHERE last_name = 'Smith';]';
```

select q'(SELECT * FROM employees WHERE last_name = 'Smith';)';

兼容性

当然，SELECT语句是兼容 SQL 标准的。 但是也有一些扩展和缺失的特性。

省略的**FROM**子句

UXDB允许我们省略 FROM子句。一种简单的使用是计算简单表达式 的结果：

```
SELECT 2+2;

 ?column?
----------
        4
```

某些其他SQL数据库需要引入一个假的 单行表放在该SELECT的 FROM子句中才能做到这一点。

注意，如果没有指定一个FROM子句，该查询 就不能引用任何数据库表。例如，下面的查询是非法的：

```
SELECT distributors.* WHERE distributors.name = 'Westward';
```

UXDB之前的发行会接受这种形式的查询，并且为该查询引用的每一个表在FROM子句中隐式增加一个项。现在已经不再允许这样做。

空**SELECT**列表

SELECT之后的输出表达式列表可以为空， 这会产生一个零列的结果表。对 SQL 标准来说这不是合法的 语法。UXDB允许 它是为了与允许零列表保持一致。不过在使用 DISTINCT时不允许空列表。

省略**AS**关键词

在 SQL 标准中，只要新列名是一个合法的列名（就是说与任何保留关键词不同）， 就可以省略输出列名之前的可选关键词AS。 UXDB要稍微严格些：只要新列名匹配 任何关键词（保留或者非保留）就需要AS。推荐的习惯是使用 AS或者带双引号的输出列名来防止与未来增加的关键词可能的冲突。

在FROM项中，标准和 UXDB都允许省略非保留 关键词别名之前的AS。但是由于语法的歧义，这无法 用于输出列名。

ONLY和继承

在书写ONLY时，SQL 标准要求在表名周围加上圆括号，例如 SELECT * FROM ONLY (tab1), ONLY (tab2) WHERE ...。UXDB 认为这些圆括号是可选的。

UXDB允许写一个拖尾的*来 显式指定包括子表的非-ONLY行为。而标准则不允许 这样。

（这些点同等地适用于所有支持ONLY选项的 SQL 命令）。

TABLESAMPLE子句限制

当前只在常规表和物化视图上接受TABLESAMPLE子句。 根据 SQL 标准，应该可以把它应用于任何FROM项。

FROM中的函数调用

UXDB允许一个函数调用被直接写作 FROM列表的一个成员。在 SQL 标准中，有必要把这样一个函数 调用包裹在一个子-SELECT中。也就是说，语法 FROM *func*(...) *alias* 近似等价于 FROM LATERAL (SELECT *func*(...)) *alias*。 注意该LATERAL被认为是隐式的，这是因为标准对于 FROM中的一个UNNEST()项要求 LATERAL语义。UXDB会把 UNNEST()和其他集合返回函数同样对待。

GROUP BY和ORDER BY可用的名字空间

在 SQL-92 标准中，一个ORDER BY子句只能使用输出 列名或者序号，而一个GROUP BY子句只能使用基于输 入列名的表达式。UXDB扩展了 这两种子句以允许它们使用其他的选择（但如果有歧义时还是使用标准的 解释）。UXDB也允许两种子句 指定任意表达式。注意出现在一个表达式中的名称将总是被当做输入列名而 不是输出列名。

SQL:1999 及其后的标准使用了一种略微不同的定义，它并不完全向后兼容 SQL-92。不过，在大部分的情况下， UXDB会以与 SQL:1999 相同的 方式解释ORDER BY或GROUP BY表达式。

函数依赖

只有当一个表的主键被包括在GROUP BY列表中时， UXDB才识别函数依赖（允许 从GROUP BY中省略列）。SQL 标准指定了应该要识别 的额外情况。

LIMIT和OFFSET

LIMIT和OFFSET子句是UXDB-特有的语法，在MySQL也被使用。SQL:2008 标准已经引入了具有相同功能的子句OFFSET ... FETCH {FIRST|NEXT}...（如上文“LIMIT 子句”一节中所示）。这种语法也被IBM DB2使用（Oracle编写的应用常常使用自动生成的rownum列来实现这些子句的效果）。

UXDB数据库的rownum利用limit语法进行改造；如：rownum<3等价于limit 2。

```
SELECT ROWNUM,* FROM t1_rn WHERE ROWNUM = 1;
SELECT ROWNUM,* FROM t1_rn WHERE ROWNUM > 1;
SELECT ROWNUM,* FROM t1_rn WHERE ROWNUM = ID;
SELECT ROWNUM,* FROM t1_rn WHERE ROWNUM < 3 and ROWNUM < 4;
SELECT rm FROM (SELECT *,rownum rm FROM t1_rn) t;
SELECT rm FROM (SELECT rownum rm,* FROM t1_rn) t WHERE rm < 3;
SELECT rm  FROM (SELECT rownum AS rm,Id FROM t1_rn ORDER BY Id DESC) t;
SELECT ROWNUM,* from t1_rn,t2_rn;
```

FOR NO KEY UPDATE、FOR UPDATE、FOR SHARE、FOR KEY SHARE

尽管 SQL 标准中出现了FOR UPDATE，但标准只允许它作为 DECLARE CURSOR的一个选项。 UXDB允许它出现在任何 SELECT查询以及子-SELECT中，但这是 一种扩展。FOR NO KEY

UPDATE、FOR SHARE 以及FOR KEY SHARE变体以及NOWAIT 和SKIP LOCKED选项没有在标准中出现。

WITH中的数据修改语句

UXDB允许把**INSERT**、 **UPDATE**以及**DELETE**用作WITH 查询。这在 SQL 标准中是找不到的。

非标准子句

DISTINCT ON (...)是 SQL 标准的扩展。

ROWS FROM(...)是 SQL 标准的扩展。

WITH的MATERIALIZED 和 NOT MATERIALIZED 选项是SQL标准的扩展。

名称

SELECT INTO — 从一个查询的结果定义一个新表

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    INTO [ TEMPORARY | TEMP | UNLOGGED ] [ TABLE ] new_table
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]
```

描述

SELECT INTO创建一个新表并且用一个查询 计算得到的数据填充它。这些数据不会像普通的SELECT那样被返回给客户端。新表的列具有 和SELECT的输出列相关的名称和数据类型。

参数

TEMPORARY or TEMP

如果被指定，该表被创建为一个临时表。详见 CREATE TABLE(7)。

UNLOGGED

如果被指定，该表被创建为一个不做日志的表。详见 CREATE TABLE(7)。

new_table

要创建的表的名字（可以是模式限定的）。

所有其他参数在SELECT(7)中有详细描述。

注解

CREATE TABLE AS(7)在功能上与 SELECT INTO相似。CREATE TABLE AS 是被推荐的语法，因为这种形式的SELECT INTO在ECUX 或PL/uxSQL中不可用，因为它们对 INTO子句的解释不同。此外，CREATE TABLE AS提供的功能是 SELECT INTO的超集。

与CREATE TABLE AS相比，SELECT INTO 不允许指定属性，就像带有USING *method*的表访问方法或者带有TABLESPACE *tablespace_name*的表的表空间。 如果必要，使用CREATE TABLE AS(7)。因此，为新表选择默认表访问方法。

示例

创建一个只由来自films的最近项构成的 新表films_recent：

```
SELECT * INTO films_recent FROM films WHERE date_prod >= '2002-01-01';
```

兼容性

SQL 标准使用**SELECT INTO**表示把值选择 到一个宿主程序的标量变量中，而不是创建一个新表。这实际上就是 ECUX和 PL/uxSQL 中的用法。 UXDB使用 **SELECT INTO**的来表示表创建是有历史原因的。 最好在新代码中使用**CREATE TABLE AS**。

另见

CREATE TABLE AS(7)

名称

SET — 更改一个运行时参数

大纲

SET [SESSION | LOCAL] *configuration_parameter* { TO | = } { *value* | '*value*' | DEFAULT }
SET [SESSION | LOCAL] TIME ZONE { *timezone* | LOCAL | DEFAULT }

描述

SET命令更改运行时配置参数。很多服务器配置参数可以用 SET即时更改（但是有些需要超级用户 特权才能更改，并且还有一些在服务器或者会话启动之后不能被更改）。 SET只影响当前会话所使用的值。

如果在一个事务内发出SET （或者等效的SET SESSION）而该事务后来 中止，在该事务被回滚时SET命令的效果会 消失。一旦所在的事务被提交，这些效果将会持续到会话结束（除非被另一个SET所覆盖）。

SET LOCAL的效果只持续到当前事务结束， 不管事务是否被提交。一种特殊情况是在一个事务内SET后面跟着 SET LOCAL： SET LOCAL值将会在该事务结束前一直可见， 但是之后（如果该事务被提交）SET值将会生效。

SET或SET LOCAL 的效果也会因为回滚到早于它们的保存点而消失。

如果在一个函数内使用SET LOCAL并且该函数 还有对同一变量的SET选项（见 CREATE FUNCTION(7)），在函数退出时 SET LOCAL命令的效果会消失。也就是说，该 函数被调用时的值会被恢复。这允许用 SET LOCAL在函数内动态地或者重复地更改 一个参数，同时仍然能便利地使用SET选项来保存以及恢复调用 者的值。不过，一个常规的SET命令会覆盖它所在的任何函 数的SET选项，除非回滚，它的效果将一直保持。

参数

SESSION

指定该命令对当前会话有效（这是默认值）。

LOCAL

指定该命令只对当前事务有效。在COMMIT或者 ROLLBACK之后，会话级别的设置会再次生效。 在事务块外部发出这个参数会发出一个警告并且不会有效果。

configuration_parameter

一个可设置运行时参数的名称。

value

参数的新值。根据特定的参数，值可以被指定为字符串常量、标识符、 数字或者以上构成的逗号分隔列表。写DEFAULT 可以指定把该参数重置成它的默认值（也就是说在当前会话中还没有 执行SET命令时它具有的值）。

除了服务器配置参数， 还有一些参数只能用SET命令设置 或者具有特殊的语法：

SCHEMA

SET SCHEMA '*value*'是 SET search_path TO *value*的一个别名。 使用这种语法只能指定一个模式。

NAMES

SET NAMES *value*是 SET client_encoding TO *value*的一个别名。

SEED

为随机数生成器（函数random）设置 一个内部种子。允许的值是 -1 和 1 之间的浮点数，它会被乘上 $2^{31}-1$。

也可以通过调用函数setseed来设置种子：

SELECT setseed(*value*);

TIME ZONE

SET TIME ZONE *value*是 SET timezone TO *value*的一个别 名。语法SET TIME ZONE允许用于时区指定的特 殊语法。这里是合法值的例子：

'PST8PDT'

加州伯克利的时区。

'Europe/Rome'

意大利的时区。

-7

UTC 以西 7 小时的时区（等效于 PDT）。正值则是 UTC 以东。

INTERVAL '-08:00' HOUR TO MINUTE

UTC 以西 8 小时的时区（等效于 PST）。

LOCAL
DEFAULT

把时区设置为本地时区（也就是说服务器的timezone默认值）。

以数字或区间给出的时区设置在内部被翻译成 POSIX 时区语法。 例如，在SET TIME ZONE -7之后， SHOW TIME ZONE将会报告 <-07>+07。

有关时区的更多信息可见第 5.5.3 节 “ 时区”。

注解

函数set_config提供了等效的功能，见 第 6.26 节 “系统管理函数”。此外，可以更新ux_settings 系统视图来执行与SET等效的工作。

示例

设置模式搜索路径：

```
SET search_path TO my_schema, public;
```

把日期风格设置为UXDB的“日在月之前”的输入习惯：

```
SET datestyle TO uxdb, dmy;
```

设置时区为加州伯克利：

```
SET TIME ZONE 'PST8PDT';
```

设置时区为意大利：

```
SET TIME ZONE 'Europe/Rome';
```

兼容性

SET TIME ZONE扩展了 SQL 标准定义的语法。标准 只允许数字的时区偏移量而 UXDB允许更灵活的时区说明。 所有其他SET特性都是 UXDB扩展。

另见

RESET(7), SHOW(7)

名称

SET CONSTRAINTS — 为当前事务设置约束检查时机

大纲

SET CONSTRAINTS { ALL | *name* [, ...] } { DEFERRED | IMMEDIATE }

描述

SET CONSTRAINTS设置当前事务内约束检查 的行为。IMMEDIATE约束在每个语句结束时被检查。 DEFERRED约束直到事务提交时才被检查。每个约束都有 自己的IMMEDIATE或DEFERRED模式。

在创建时，一个约束会被给定三种特性之一： DEFERRABLE INITIALLY DEFERRED、DEFERRABLE INITIALLY IMMEDIATE或者 NOT DEFERRABLE。第三类总是 IMMEDIATE并且不会受到 SET CONSTRAINTS命令的影响。前两类在每个 事务开始时都处于指定的模式，但是它们的行为可以在一个事务内用 SET CONSTRAINTS更改。

带有一个约束名称列表的SET CONSTRAINTS 只更改那些约束（都必须是可延迟的）的模式。每一个约束名称都可以是 模式限定的。如果没有指定模式名称，则当前的模式搜索路径将被用来寻找 第一个匹配的名称。SET CONSTRAINTS ALL 更改所有可延迟约束的模式。

当SET CONSTRAINTS把一个约束的模式从 DEFERRED改成IMMEDIATE时， 新模式会有追溯效果：任何还没有解决的数据修改（本来会在事务结束时 被检查）会转而在SET CONSTRAINTS命令的执行期间被检查。如果任何这种约束被违背， SET CONSTRAINTS将会失败（并且不会改 变该约束模式）。这样，SET CONSTRAINTS可以被用来在一个事务中的特定点强制进 行约束检查。

当前，只有UNIQUE、PRIMARY KEY、 REFERENCES（外键）以及EXCLUDE 约束受到这个设置的影响。 NOT NULL以及CHECK约束总是在一行 被插入或修改时立即检查（不是在语句结束时）。 没有被声明为DEFERRABLE的唯一和排除约束也会被 立刻检查。

被声明为“约束触发器”的触发器的引发也受到这个设置 的控制 — 它们会在相关约束被检查的同时被引发。

注解

因为UXDB并不要求约束名称在模式内 唯一（但是在表内要求唯一），可能有多于一个约束匹配指定的约束名称。在这种 情况下SET CONSTRAINTS将会在所有的匹配上操作。 对于一个非模式限定的名称，一旦在搜索路径中的某个模式中发现一个或者多个匹 配，路径中后面的模式将不会被搜索。

这个命令只修改当前事务内约束的行为。在事务块外部发出这个命令会产生一个 警告并且也不会有任何效果。

兼容性

这个命令符合 SQL 标准中定义的行为，但有一点限制：在 UXDB中，它不会应用在 NOT NULL和CHECK约束上。还有， UXDB会立刻检查非可延迟的 唯一约束，而不是按照标准建议的在语句结束时检查。

名称

SET ROLE — 设置当前会话的当前用户标识符

大纲

```
SET [ SESSION | LOCAL ] ROLE role_name
SET [ SESSION | LOCAL ] ROLE NONE
RESET ROLE
```

描述

这个命令把当前 SQL 会话的当前用户标识符设置为 *role_name*。 角色名可以写成一个标识符或者一个字符串。在 SET ROLE之后，对 SQL 命令的权限检查时就 好像该角色就是原先登录的角色一样。

当前会话用户必须是指定的 角色*role_name* 的一个成员（如果会话用户是一个超级用户，则可以选择任何角色）。

SESSION和LOCAL修饰符发挥的作用和 常规的SET(7)命令一样。

NONE和RESET形式把当前用户标识符 重置为当前会话用户标识符。这些形式可以由任何用户执行。

注解

使用这个命令可以增加特权或者限制特权。如果会话用户角色具有 INHERIT属性，则它会自动具有它能 SET ROLE到的所有角色的全部特权。在这种情况下 SET ROLE实际会删除所有直接分配给会话用户的特 权以及分配给会话用户作为其成员的其他角色的特权，只留下所提及 角色可用的特权。换句话说，如果会话用户没有 NOINHERIT属性，SET ROLE会删除 直接分配给会话用户的特权而得到所提及角色可用的特权。

特别地，当一个超级用户选择SET ROLE到一个非 超级用户角色时，它们会丢失其超级用户特权。

SET ROLE的效果堪比 SET SESSION AUTHORIZATION(7)，但是涉及的特权检查 完全不同。还有，SET SESSION AUTHORIZATION决定 后来的SET ROLE命令可以使用哪些角色， 不过用 SET ROLE更改角色并不会改变后续 SET ROLE能够使用的角色集。

SET ROLE不会处理角色的ALTER ROLE(7) 设置指定的会话变量。这只在登录期间发生。

SET ROLE不能在一个 SECURITY DEFINER函数中使用。

示例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter
```

```
SET ROLE 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | paul
```

兼容性

UXDB允许标识符 语法（"*rolename*"），而 SQL 标准要求 角色名被写成字符串。SQL 不允许在事务中使用这个命令，而 UXDB并不做此限 制，因为并没有原因需要这样做。和RESET语法 一样，SESSION和 LOCAL修饰符是一种 UXDB扩展。

另见

SET SESSION AUTHORIZATION(7)

名称

SET SESSION AUTHORIZATION — 设置当前会话的会话用户标识符和当前用户标识符

大纲

SET [SESSION | LOCAL] SESSION AUTHORIZATION *user_name*
SET [SESSION | LOCAL] SESSION AUTHORIZATION DEFAULT
RESET SESSION AUTHORIZATION

描述

这个命令把当前 SQL 会话的会话用户标识符和当前用户标识符设置为 *user_name*。 用户名可以被写成一个标识符或者一个字符串。例如，可以使用这个 命令临时成为一个无特权用户并且稍后切换回来成为一个超级用户。

会话用户标识符初始时被设置为客户端提供的（可能已认证的）用户名。 当前用户标识符通常等于会话用户标识符，但是可能在 SECURITY DEFINER函数和类似机制的环境中临时更改。 也可以用SET ROLE(7)更改它。当前用户标识符与 权限检查相关。

会话用户标识符只能在初始会话用户 已认证用户具有超级用户特权时被更改。 否则，只有该命令指定已认证用户名时才会被接受。

SESSION和LOCAL修饰符发挥的作用和常规 SET(7)命令一样。

DEFAULT和RESET形式把会话用户标识符和 当前用户标识符重置为初始的已认证用户名。这些形式可以由任何用户执行。

注解

SET SESSION AUTHORIZATION不能在一个 SECURITY DEFINER函数中使用。

示例

```
SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 peter        | peter

SET SESSION AUTHORIZATION 'paul';

SELECT SESSION_USER, CURRENT_USER;

 session_user | current_user
--------------+--------------
 paul         | paul
```

兼容性

SQL 标准允许一些其他表达式出现在文本 *user_name*的位置上，但是实际上这些 选项并不重要。UXDB允许标 识符语法（"*username*"），而 SQL 标准不允许。 SQL 不允许在事务中使用这个命令，而 UXDB并不做此限 制，因为并没有原因需要这样做。和RESET语法 一样，SESSION和LOCAL修饰符是一种 UXDB扩展。

标准把执行这个命令所需的特权留给实现定义。

另见

SET ROLE(7)

名称

SET TRANSACTION — 设置当前事务的特性

大纲

```
SET TRANSACTION transaction_mode [, ...]
SET TRANSACTION SNAPSHOT snapshot_id
SET SESSION CHARACTERISTICS AS TRANSACTION transaction_mode [, ...]
```

其中 *transaction_mode* 是下列之一：

```
  ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
  READ WRITE | READ ONLY
  [ NOT ] DEFERRABLE
```

描述

SET TRANSACTION命令设置当前 会话的特性。SET SESSION CHARACTERISTICS设置一个会话后续事务的默认 事务特性。在个体事务中可以用 SET TRANSACTION覆盖这些默认值。

可用的事务特性是事务隔离级别、事务访问模式（读/写或只读）以及 可延迟模式。此外，可以选择一个快照，不过只能用于当前事务而不能 作为会话默认值。

一个事务的隔离级别决定当其他事务并行运行时该事务能看见什么数据：

READ COMMITTED

一个语句只能看到在它开始前提交的行。这是默认值。

REPEATABLE READ

当前事务的所有语句只能看到这个事务中执行的第一个查询或者 数据修改语句之前提交的行。

SERIALIZABLE

当前事务的所有语句只能看到这个事务中执行的第一个查询或者 数据修改语句之前提交的行。如果并发的可序列化事务间的读写 模式可能导致一种那些事务串行（一次一个）执行时不可能出现 的情况，其中之一将会被回滚并且得到一个 serialization_failure错误。

SQL 标准定义了一种额外的级别：READ UNCOMMITTED。在 UXDB中READ UNCOMMITTED被视作 READ COMMITTED。

一个事务执行了第一个查询或者数据修改语句（ SELECT、 INSERT、DELETE、 UPDATE、FETCH或COPY）之后就无法更改事务隔离级别。 更多有关事务隔离级别和并发控制的信息可见第 10 章 并发控制。

事务的访问模式决定该事务是否为读/写或者只读。读/写是默认值。 当一个事务为只读时，如果 SQL 命令 INSERT、UPDATE、 DELETE和COPY FROM 要写的表不是一个临时表，则它们不被允许。不允许 CREATE、ALTER以及 DROP命令。不允许COMMENT、GRANT、REVOKE、 TRUNCATE。如果EXPLAIN ANALYZE 和EXECUTE要执行的命令是上述命令之一， 则它们也不被允许。这是一种高层的只读概念，它不能阻止所有对 磁盘的写入。

只有事务也是SERIALIZABLE以及 READ ONLY时，DEFERRABLE 事务属性才会有效。当一个事务的所有这三个属性都被选择时，该事务在 第一次获取其快照时可能会阻塞，在那之后它运行时就不会有 SERIALIZABLE事务的开销并且不会有任何牺牲或者 被一次序列化失败取消的风险。这种模式很适合于长时间运行的报表或者 备份。

SET TRANSACTION SNAPSHOT命令允许新的事务 使用与一个现有事务相同的快照运行。已经存在的事务 必须已经把它的快照用ux_export_snapshot函数（ 见第 6.26.5 节 “快照同步函数”）导出。 该函数会返回一个快照标识符，SET TRANSACTION SNAPSHOT需要被给定一个快照标识符来指定要导入的快照。 在这个命令中该标识符必须被写成一个字符串，例如'000003A1-1'。 SET TRANSACTION SNAPSHOT只能在一个事务的 开始执行，并且要在该事务的第一个查询或者数据修改语句（ SELECT、 INSERT、DELETE、 UPDATE、FETCH或 COPY）之前执行。此外，该事务必须已经被设置 为SERIALIZABLE或者 REPEATABLE READ隔离级别（否则，该快照将被立刻抛弃， 因为READ COMMITTED模式会为每一个命令取一个新快照）。 如果导入事务使用了SERIALIZABLE隔离级别，那么导入快照 的事务必须也使用该隔离级别。还有，一个非只读可序列化事务不能导入来自只读 事务的快照。

注解

如果执行SET TRANSACTION之前没有 START TRANSACTION或者 BEGIN，它会发出一个警告并且不会有任何效果。

可以通过在BEGIN或者 START TRANSACTION中指定想要的*transaction_modes*来省掉 SET TRANSACTION。但是在 SET TRANSACTION SNAPSHOT中该选项不可用。

会话默认的事务模式也可以通过设置配置参数 default_transaction_isolation、default_transaction_read_only和 default_transaction_deferrable来设置（实际上 SET SESSION CHARACTERISTICS只是用 SET设置这些变量的等效体）。这意味着可以通过配置文件、ALTER DATABASE等方式设置默认值。

示例

要用一个已经存在的事务的同一快照开始一个新事务，首先要从该现有 事务导出快照。这将会返回快照标识符，例如：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT ux_export_snapshot();
 ux_export_snapshot
---------------------
 00000003-0000001B-1
(1 row)
```

然后在一个新开始的事务的开头把该快照标识符用在一个 SET TRANSACTION SNAPSHOT命令中：

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET TRANSACTION SNAPSHOT '00000003-0000001B-1';
```

兼容性

SQL标准中定义了这些命令，不过 DEFERRABLE事务模式和 SET TRANSACTION SNAPSHOT形式除外，这两者是 UXDB扩展。

SERIALIZABLE是标准中默认的事务隔离级别。在 UXDB中默认值是普通的 READ COMMITTED，但是可以按上述的方式更改。

在 SQL 标准中，可以用这些命令设置一个其他的事务特性：诊断区域 的尺寸。这个概念与嵌入式 SQL 有关，并且因此没有在 UXDB服务器中实现。

SQL 标准要求连续的*transaction_modes*之间有逗号， 但是出于历史原因 UXDB允许省略逗号。

名称

SHOW — 显示一个运行时参数的值

大纲

SHOW *name*
SHOW ALL

描述

SHOW将显示运行时参数的当前设置。这些变量可以使用SET语句、编辑uxsinodb.conf配置参数、通过UXOPTIONS环境变量（使用libuxsql或者基于libuxsql的应用时）或者启动uxdb服务器时通过命令行标志设置。

参数

name

一个运行时参数的名称。此外，有一些可以显示但不能设置的参数：

SERVER_VERSION

显示服务器的版本号。

SERVER_ENCODING

显示服务器端的字符集编码。当前，这个参数可以被显示 但不能被设置，因为该设置是在数据库创建时决定的。

LC_COLLATE

显示数据库的排序规则（文本序）的区域设置。当前， 这个参数可以被显示但不能被设置，因为该设置是在 数据库创建时决定的。

LC_CTYPE

显示数据库的字符分类的区域设置。当前， 这个参数可以被显示但不能被设置，因为该设置 是在数据库创建时决定的。

IS_SUPERUSER

如果当前角色具有超级用户特权则为真。

ALL

显示所有配置参数的值，并带有描述。

注解

函数current_setting产生等效的输出，见 第 6.26 节 “系统管理函数”。还有， ux_settings 系统事务产生同样的信息。

示例

显示参数DateStyle的当前设置：

```
SHOW DateStyle;
 DateStyle
-----------
 ISO, MDY
(1 row)
```

显示参数geqo的当前设置：

```
SHOW geqo;
 geqo
------
 on
(1 row)
```

显示所有设置：

```
SHOW ALL;
            name         | setting |                 description
-------------------------+---------+-----------------------------------------------
 allow_system_table_mods | off     | Allows modifications of the structure of ...
    .
    .
    .
 xmloption               | content | Sets whether XML data in implicit parsing ...
 zero_damaged_pages      | off     | Continues processing past damaged page headers.
(196 rows)
```

兼容性

SHOW命令是一种 UXDB扩展。

另见

SET(7)， RESET(7)

名称

SHOW CREATE TABLE — 查询表的建表语句

大纲

SHOW CREATE TABLE *tablename*

描述

通过show create table查询表的建表语句。

参数

tablename

表名称

返回值

表名和其对应的建表语句。

示例

通过show create table查看表，如下所示。

```
show create table test;
   TABLE    |           CREATE TABLE
------------+-----------------------------------
 PUBLIC.TEST | SET SEARCH_PATH = PUBLIC;         +
           | CREATE TABLE PUBLIC.TEST  (       +
           |       TYPE1 CHARACTER VARYING(100)+
           | );
(1 row)
```

名称

START TRANSACTION — 开始一个事务块

大纲

START TRANSACTION [*transaction_mode* [, ...]]

其中 *transaction_mode* 是下列之一：

```
  ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ
UNCOMMITTED }
  READ WRITE | READ ONLY
  [ NOT ] DEFERRABLE
```

描述

这个命令开始一个新的事务块。如果指定了隔离级别、读写模式 或者可延迟模式，新的事务将会具有这些特性，就像执行了 SET TRANSACTION(7)一样。这和 BEGIN(7)命令一样。

参数

这些参数对于这个语句的含义可参考 SET TRANSACTION(7)。

兼容性

在标准中，没有必要发出START TRANSACTION 来开始一个事务块：任何 SQL 命令会隐式地开始一个块。 UXDB的行为可以被视作 在每个命令之后隐式地发出一个没有跟随在 START TRANSACTION（ 或者BEGIN）之后的 COMMIT并且因此通常被称作 “自动提交”。为了方便，其他关系型数据库系统也可能会 提供自动提交特性。

DEFERRABLE *transaction_mode* 是一种UXDB语言扩展。

SQL 标准要求在连续的*transaction_modes*之间有逗号， 但是由于历史原因UXDB允许 省略逗号。

另见SET TRANSACTION(7)的兼容性小节。

另见

BEGIN(7), COMMIT(7), ROLLBACK(7), SAVEPOINT(7), SET TRANSACTION(7)

名称

TRUNCATE — 清空一个表或者一组表

大纲

```
TRUNCATE [ TABLE ] [ ONLY ] name [ * ] [, ... ]
  [ RESTART IDENTITY | CONTINUE IDENTITY ] [ CASCADE | RESTRICT ]
```

描述

TRUNCATE可以从一组表中快速地移除所有行。 它具有和在每个表上执行无条件DELETE相同的 效果，不过它会更快，因为它没有实际扫描表。此外，它会立刻回收磁盘空间， 而不是要求一个后续的VACUUM操作。在大表上 它最有用。

参数

name

要截断的表的名字（可以是模式限定的）。如果在表名前指定了 ONLY，则只会截断该表。如果没有指定ONLY， 该表及其所有后代表（如果有）都会被截断。可选地，可以在表名后指定 *来显式地包括后代表。

RESTART IDENTITY

自动重新开始被截断表的列所拥有的序列。

CONTINUE IDENTITY

不更改序列值。这是默认值。

CASCADE

自动截断所有对任一所提及表有外键引用的表以及任何由于 CASCADE被加入到组中的表。

RESTRICT

如果任一表上具有来自命令中没有列出的表的外键引用，则拒绝截断。这是默认值。

注解

要截断一个表，必须具有其上的TRUNCATE特权。

TRUNCATE在要操作的表上要求一个 ACCESS EXCLUSIVE锁，这会阻塞所有其他在该表上的 并发操作。当指定RESTART IDENTITY时，任何需要被 重新开始的序列也会被排他地锁住。如果要求表上的并发访问，那么 应该使用DELETE命令。

TRUNCATE不能被用在被其他表外键引用的表上，除非那些表也在同一个命令中被阶段。这些情况中的可行性检查将会要求表扫描，并且重点不是为了做扫描。CASCADE选项可以被用来自动地包括所有依赖表 — 但使用它时要非常注意，否则可能丢失数据！特别注意的是，当要被截断的表是一个分区时，兄弟节点分区不会受到影响，但是所有的引用表都发生级联，他们的分区也没有区别。

TRUNCATE将不会引发表上可能存在的任何 ON DELETE触发器。但是它将会引发 ON TRUNCATE触发器。如果在这些表的任意一个 上定义了ON TRUNCATE触发器，那么所有的BEFORE TRUNCATE触发器将在任何截断发生之前 被引发，而所有AFTER TRUNCATE触发器将在最后 一次截断完成并且所有序列被重置之后引发。触发器将以表被处理的顺 序被引发（首先是那些被列在命令中的，然后是由于级联被加入的）。

TRUNCATE不是 MVCC 安全的。截断之后， 如果并发事务使用的是一个在截断发生前取得的快照， 表将对这些并发事务呈现为空。详见第 10.5 节 “提醒”

从表中数据的角度来说，TRUNCATE是事务安全的： 如果所在的事务没有提交，阶段将会被安全地回滚。

在指定了RESTART IDENTITY时，隐含的 ALTER SEQUENCE RESTART操作也会被事务性地完成。也就是说，如果所在事务没有提交，它们也将被回滚。这和 ALTER SEQUENCE RESTART的通常行为不同。注意如果 事务回滚前在被重启序列上还做了额外的序列操作，这些操作在序列上的效果 也将被回滚，但是它们在currval()上的效果不会被回滚。也就 是说，在事务之后，currval()将继续反映在失败事务内得到的 最后一个序列值，即使序列本身可能已经不再与此一致。这和失败事务之后 currval()的通常行为类似。

TRUNCATE当前不支持外部表。 这表示如果一个指定的表具有任何外部的后代表，这个命令将会失败。

示例

截断表bigtable和 fattable：

```
TRUNCATE bigtable, fattable;
```

做同样的事情，并且还重置任何相关联的序列发生器：

```
TRUNCATE bigtable, fattable RESTART IDENTITY;
```

截断表othertable，并且级联地截断任何通过 外键约束引用othertable的表：

```
TRUNCATE othertable CASCADE;
```

兼容性

SQL:2008 标准包括了一个TRUNCATE命令， 语法是TRUNCATE TABLE *tablename*。子句CONTINUE IDENTITY/RESTART IDENTITY 也出现在了该标准中，但是含义有些不同。这个命令的一些并发行为被标准 留给实现来定义，因此如果必要应该考虑上述注解并且与其他实现进行比较。

参见

DELETE(7)

名称

UNLISTEN — 停止监听一个通知

大纲

UNLISTEN { *channel* | * }

描述

UNLISTEN被用来移除一个已经存在的对 NOTIFY事件的注册。 UNLISTEN取消任何已经存在的把当前 UXDB会话作为名为 *channel*的通知 频道的监听者的注册。特殊的通配符*取消当前会话 的所有监听者注册。

NOTIFY(7)包含有关LISTEN 和NOTIFY使用的更深入讨论。

参数

channel

一个通知频道的名称（任何标识符）。

*

所有用于这个会话的当前监听注册都会被清除。

注解

可以 unlisten 没有监听的东西，不会出现警告或者错误。

在每一个会话末尾，会自动执行UNLISTEN *。

一个已经执行了UNLISTEN的事务不能为 两阶段提交做准备。

示例

做一次注册：

```
LISTEN virtual;
NOTIFY virtual;
Asynchronous notification "virtual" received from server process with PID 8448.
```

一旦执行了UNLISTEN，进一步的NOTIFY 消息将被忽略：

```
UNLISTEN virtual;
NOTIFY virtual;
-- no NOTIFY event is received
```

兼容性

SQL 标准中没有UNLISTEN命令。

另见

LISTEN(7), NOTIFY(7)

名称

UPDATE — 更新一个表的行

大纲

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] {table_name [ * ] [ [ AS ] alias ] | subquery} [[NATURAL] join_type JOIN
 {table_name [ * ] [ [ AS ] alias ] | subquery} [ON join_condition | USING ( join_column [, ...] )]
   SET { column_name = { expression | DEFAULT } |
        ( column_name [, ...] ) = [ ROW ] ( { expression | DEFAULT } [, ...] ) |
        ( column_name [, ...] ) = ( sub-SELECT )
       } [, ...]
   [ FROM from_list ]
   [ WHERE condition | WHERE CURRENT OF cursor_name ]
   [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

描述

UPDATE更改满足条件的所有行中指定列的值。只有要被修改的列需要在SET子句中提及，没有被显式修改的列保持它们之前的值。

有两种方法使用包含在数据库其他表中的信息来修改一个表：使用子选择或者在FROM子句中指定额外的表。这种技术只适合特定的环境。

可选的RETURNING子句导致UPDATE基于实际被更新的每一行计算并且返回值。任何使用该表的列以及FROM中提到的其他表的列的表达式都能被计算。计算时会使用该表的列的新（更新后）值。RETURNING列表的语法和SELECT的输出列表相同。

必须拥有该表上的UPDATE特权，或者至少拥有要被更新的列上的该特权。如果任何一列的值需要被*expressions*或者*condition*读取，还必须拥有该列上的SELECT特权。

UPDATE...JOIN...SET用包含在数据库其他表中的信息来修改一个表。

使用UPDATE...JOIN...SET语句时，不能出现FROM子句和WHERE CURRENT OF子句。

参数

with_query

WITH子句允许指定一个或者更多个在 UPDATE中可用其名称引用的子查询。详见 第 4.8 节“WITH查询（公共表表达式）”和SELECT(7)。

table_name

要更新的表的名称（可以是模式限定的）。如果在表名前指定了 ONLY，只会更新所提及表中的匹配行。如果没有指定 ONLY，任何从所提及表继承得到的表中的匹配行也会 被更新。可选地，在表名之后指定*可以显式地指示要 把后代表也包括在内。

alias

目标表的一个替代名称。在提供了一个别名时，它会完全隐藏表的真实名称。例如，给定UPDATE foo AS f，UPDATE语句的剩余部分必须用f而不是foo来引用该表。

subquery

子查询语句，该子查询对象不能作为更新对象。

join_type

连接类型。详见SELECT语法中FROM子句的join_type部分。

column_name

table_name 所指定的表的一列的名称。如果需要，该列名可以用一个子域名称或者 数组下标限定。不要在目标列的说明中包括表的名称 — 例如 UPDATE table_name SET table_name.col = 1是非法的。

expression

要被赋值给该列的一个表达式。该表达式可以使用该表中这一列或者 其他列的旧值。

DEFAULT

将该列设置为它的默认值（如果没有为它指定默认值表达式，默认值 将会为 NULL）。

sub-SELECT

一个SELECT子查询，它产生和在它之前的圆括号中列列表中 一样多的输出列。被执行时，该子查询必须得到不超过一行。如果它得到 一行，其列值会被赋予给目标列。如果它得不到行，NULL 值将被赋予给 目标列。该子查询可以引用被更新表中当前行的旧值。

from_list

表表达式的列表，允许来自其他表的列出现在WHERE 条件和更新表达式中。这类似于可以在 SELECT语句的“FROM 子句”一节中指定的表列表。注意目标表不能出现在*from_list*中，除非想做自连接（这种情况下它必须 以别名出现在*from_list*中）。

condition

一个返回boolean类型值的表达式。让这个 表达式返回true的行将会被更新。

cursor_name

要在WHERE CURRENT OF条件中使用的游标名。 要被更新的是从这个游标中最近取出的行。该游标必须是一个 在UPDATE目标表上的非分组查询。注意 WHERE CURRENT OF不能和一个布尔条件一起 指定。有关对游标使用WHERE CURRENT OF的 更多信息请见DECLARE(7)。

output_expression

在每一行被更新后，要被UPDATE命令计算并且返回 的表达式。该表达式可以使用*table_name*指定 的表或者FROM列出的表中的任何列名。写* 可以返回所有列。

output_name

用于一个被返回列的名称。

输出

成功完成时，一个UPDATE命令返回形如

UPDATE *count*

的命令标签。 *count*是被更新的行数， 包括值没有更改的匹配行。注意，当更新被一个BEFORE UPDATE 触发器抑制时，这个数量可能比匹配 *condition*的行数少。如果 *count*为零，没有行被该查 询更新（这不是一个错误）。

如果UPDATE命令包含一个RETURNING 子句，其结果将类似于一个包含RETURNING列表中定义的 列和值的SELECT语句（在被该命令更新的行上计算） 的结果。

注解

当存在FROM子句时，实际发生的是：目标表被连接到 *from_list*中的表，并且该连接的每一 个输出行表示对目标表的一个更新操作。在使用FROM 时，应该确保该连接对每一个要修改的行产生至多一个输出行。换 句话说，一个目标行不应该连接到来自其他表的多于一行上。如果发生这种情况，则只有一个连接行将被用于更新目标行，但是将使用哪 一行是很难预测的。

由于这种不确定性，只在一个子选择中引用其他表更安全，不过这种 语句通常很难写并且也比使用连接慢。

在分区表的情况下，更新一行有可能导致它不再满足其所在分区的分区约束。此时，如果这个行满足分区树中某个其他分区的分区约束，那么这个行会被移动到那个分区。 如果没有这样的分区，则会发生错误。在后台，行的移动实际上是一次DELETE操作和一次INSERT操作。

在移动的行上的并发UPDATE或DELETE可能会收到序列化失败错误。 假设会话 1 正在分区键上执行UPDATE，同时，对可访问该行的并发会话 2 在此行上执行UPDATE或DELETE操作。 在这种情况下，会话 2 的UPDATE 或 DELETE将检测行移动并引发序列化失败错误（该错误始终返回 SQLSTATE 代码"40001"）。 如果发生这种情况，应用程序可能希望重试事务。 在通常情况下，表没有分区或没有行移动，会话 2 将标识新更新的行，并执行UPDATE/DELETE在此新行版本中。

请注意，虽然行可以从本地分区移动到外表分区（如果外数据包装器支持元组路由），但它们不能从外表分区移动到另一个分区。

示例

把表films的列*kind* 中的单词Drama改成Dramatic：

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
```

在表weather的一行中调整温度项并且 把沉淀物重置为它的默认值：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

执行相同的操作并且返回更新后的项：

```
UPDATE weather SET temp_lo = temp_lo+1, temp_hi = temp_lo+15, prcp = DEFAULT
  WHERE city = 'San Francisco' AND date = '2003-07-03'
  RETURNING temp_lo, temp_hi, prcp;
```

使用另一种列列表语法来做同样的更新：

```
UPDATE weather SET (temp_lo, temp_hi, prcp) = (temp_lo+1, temp_lo+15, DEFAULT)
  WHERE city = 'San Francisco' AND date = '2003-07-03';
```

为管理Acme Corporation账户的销售人员增加销售量，使用 FROM子句语法：

```
UPDATE employees SET sales_count = sales_count + 1 FROM accounts
  WHERE accounts.name = 'Acme Corporation'
  AND employees.id = accounts.sales_person;
```

执行相同的操作，在 WHERE子句中使用子选择：

```
UPDATE employees SET sales_count = sales_count + 1 WHERE id =
  (SELECT sales_person FROM accounts WHERE name = 'Acme Corporation');
```

更新 accounts 表中的联系人姓名以匹配当前被分配的销售员：

```
UPDATE accounts SET (contact_first_name, contact_last_name) =
    (SELECT first_name, last_name FROM salesmen
     WHERE salesmen.id = accounts.sales_id);
```

可以用连接完成类似的结果：

```
UPDATE accounts SET contact_first_name = first_name,
                    contact_last_name = last_name
  FROM salesmen WHERE salesmen.id = accounts.sales_id;
```

不过，如果salesmen.*id*不是一个唯一键， 第二个查询可能会给出令人意外的结果，然而如果有多个*id*匹配， 第一个查询保证会发生错误。还有，如果对于一个特定的 accounts.*sales_id*项没有匹配，第一个查询将 把相应的姓名域设置为 NULL，而第二个查询完全不会更新该行。

更新一个统计表中的统计数据以匹配当前数据：

```
UPDATE summary s SET (sum_x, sum_y, avg_x, avg_y) =
    (SELECT sum(x), sum(y), avg(x), avg(y) FROM data d
     WHERE d.group_id = s.group_id);
```

尝试插入一个新库存项及其库存量。如果该项已经存在，则转而更新 已有项的库存量。要这样做并且不让整个事务失败，可以使用保存点：

```
BEGIN;
-- 其他操作
SAVEPOINT sp1;
INSERT INTO wines VALUES('Chateau Lafite 2003', '24');
-- 假定上述语句由于未被唯一键失败，
-- 那么现在我们发出这些命令：
ROLLBACK TO sp1;
UPDATE wines SET stock = stock + 24 WHERE winename = 'Chateau Lafite 2003';
-- 继续其他操作，并且最终
COMMIT;
```

更改表films中由游标c_films 定位的行的*kind*列：

```
UPDATE films SET kind = 'Dramatic' WHERE CURRENT OF c_films;
```

UPDATE语句可以使用别名来设置表中数据。

在提供了一个别名时，它会完全隐藏表的真实名称，如下所示。

```
update table_name as alias_name set table_name.column_name1 = 3, alias_name.column_name2
 = 'wangwu' where column_name1 = 3; -- ok
update table_name as alias_name set table_name.column_name1 = 3, alias_name.column_name2
 = 'wangwu' where alias_name.column_name1 = 3; -- ok
update table_name as alias_name set table_name.column_name1 = 3, alias_name.column_name2
 = 'wangwu' where table_name.column_name1 = 3;
-- failed
```

在提供别名或没提供别名，剩余的语句都可以直接使用列名方式来引用。

```
update table_name as alias_name set alias_name.column_name1 = 3, column_name2 = 'wangwu'
 where column_name1 = 3;-- ok
update table_name set table_name.column_name1 = 3, column_name2 = 'wangwu' where
 column_name1 = 3;-- ok
update table_name as alias_name set table_name.column_name1 = 3, alias_name.column_name2
 = 'wangwu' where column_name1 = 3; -- failed
```

UPDATE...JOIN...SET语句示例，如下所示。

```
--使用左连接更新目标表
UPDATE t1 LEFT JOIN t2 ON t1.ID = t2.ID SET t1.ID = 4;
--使用交叉连接更新目标表
UPDATE t1 CROSS JOIN t2 SET t2.ID = 6;
--根据子查询更新目标表
UPDATE t1 CROSS JOIN (SELECT * FROM t2) a SET t1.ID = 8;
--多重连接更新表
UPDATE t1 CROSS JOIN (t2 RIGHT JOIN t3 USING(ID)) SET t2.ID = 4;
```

兼容性

这个命令符合SQL标准，不过 FROM和RETURNING子句是 UXDB扩展，把 WITH用于UPDATE也是扩展。

有些其他数据库系统提供了一个FROM选项，在其中在其中目标表可以在FROM中被再次列出。但UXDB不是这样解释FROM的。在移植使用这种扩展的应用时要注意。

根据标准，一个目标列名的圆括号子列表的来源值可以是任意得到正确列数的行值 表达式。UXDB只允许来源值是一个 行构造器或者一个子-SELECT。一个列的 被更新值可以在行构造器的情况中被指定为DEFAULT，但在 子-SELECT的情况中不能这样做。

名称

VACUUM — 垃圾收集并根据需要分析一个数据库

大纲

VACUUM [(*option* [, ...])] [*table_and_columns* [, ...]]
VACUUM [FULL] [FREEZE] [VERBOSE] [ANALYZE] [*table_and_columns* [, ...]]

其中*option*可以是下列之一：

FULL [*boolean*]
FREEZE [*boolean*]
VERBOSE [*boolean*]
ANALYZE [*boolean*]
DISABLE_PAGE_SKIPPING [*boolean*]
SKIP_LOCKED [*boolean*]
INDEX_CLEANUP [*boolean*]
TRUNCATE [*boolean*]

而*table_and_columns*是：

table_name [(*column_name* [, ...])]

描述

VACUUM收回由死亡元组占用的存储空间。在通常的UXDB操作中，被删除或者被更新废弃的元组并没有在物理上从它们的表中移除，它们将一直存在直到一次**VACUUM**被执行。因此有必要周期性地做**VACUUM**，特别是在频繁被更新的表上。

在没有*table_and_columns*列表的情况下，**VACUUM**会处理当前用户具有清理权限的当前数据库中的每一个表和物化视图。如果给出一个列表，**VACUUM**可以只处理列表中的那些表。

VACUUM ANALYZE对每一个选定的表**ANALYZE**。这是两种命令的一种方便的组合形式，可以用于例行的维护脚本。其处理细节可参考ANALYZE(7)。

简单的 **VACUUM**（不带FULL）简单地收回空间并使其可以被重用。这种形式的命令可以和表的普通读写操作并行，因为它不会获得一个排他锁。但是，这种形式中额外的空间并没有被还给操作系统（在大多数情况下），它仅仅被保留在同一个表中以备重用。**VACUUM FULL**将表的整个内容重写到一个新的磁盘文件中，并且不包含额外的空间，这使得没有被使用的空间被还给操作系统。这种形式的命令更慢并且在其被处理时要求在每个表上保持一个排他锁。

当选项列表被包围在圆括号中时，选项可以被写成任何顺序。如果没有圆括号，选项必须严格按照上面所展示的顺序指定。有圆括号的语法被加入，无圆括号的语法则被废弃。

参数

FULL

选择“完全”清理，它可以收回更多空间，并且需要更长时间和表上的排他锁。这种方法还需要额外的磁盘空间，因为它会创建该表的一个新拷贝，并且在操作完成之前都不会释放旧的拷贝。通常这种方法只用于需要从表中收回数量庞大的空间时。

FREEZE

选择激进的元组“冻结”。指定FREEZE 等价于参数vacuum_freeze_min_age和vacuum_freeze_table_age设置为0的 VACUUM。当表被重写时总是会执行激进的冻结， 因此指定FULL时这个选项是多余的。

VERBOSE

为每个表打印一份详细的清理活动报告。

ANALYZE

更新优化器用以决定最有效执行一个查询的方法的统计信息。

DISABLE_PAGE_SKIPPING

通常，VACUUM将基于可见性映射跳过页面。已知所有元组都被冻结的页面总是会被跳过，而那些所有元组对所有事务都可见的页面则可能会被跳过（除非执行的是激进的清理）。此外，除非在执行激进的清理时，一些页面也可能会被跳过，这样可避免等待其他页面完成对其使用。这个选项禁用所有的跳过页面的行为，其意图是只在可见性映射内容被怀疑时使用，这种情况只有在硬件或者软件问题导致数据库损坏时才会发生。

SKIP_LOCKED

规定VACUUM在开始处理关系时不等待任何冲突锁被释放：如果关系不能立即锁定而不等待，则跳过关系。 请注意即使采用此选项，VACUUM在打开关系的索引时仍可能阻塞。 此外，VACUUM ANALYZE在从分区、继承子表和某些类型的外表获取示例行时，仍然可能阻塞。还有，虽然VACUUM通常处理指定分区表的所有分区，但如果分区表上的锁冲突， 此选项将导致VACUUM跳过所有分区。

INDEX_CLEANUP

规定VACUUM尝试删除指向死元组的索引条目。 这通常是所需的行为，并且是默认行为，除非将vacuum_index_cleanup 选项设置为 false，对要被清空的表。 如果需要尽快运行清空操作的话，将此选项设置为 false 可能很有用，例如，为了避免即将发生的事务 ID 回绕[wraparound]。 但是，如果不定期执行索引清理，性能可能会受到影响，因为随着表的修改，索引将累积死元组，并且表本身将累积死行指针，在索引清理完成之前都无法删除。此选项对于没有索引的表无效，如果使用 FULL 选项，则忽略此选项。

TRUNCATE

指定VACUUM尝试截断表末尾的任何空页，并允许将截断页的磁盘空间返回到操作系统。 这通常是所需的行为，并且是默认行为，除非将vacuum_truncate选项设置为 false，对要被清空的表。 将此选项设置为 false 可能有助于避免ACCESS EXCLUSIVE锁定需要截断的表。如果使用FULL选项，则忽略此选项。

boolean

指定打开还是关闭所选选项。可以写入TRUE、ON或1以启用该选项，以及FALSE、OFF或0来禁用它。 在TRUE被假定的情况下，*boolean* 值也可以被省略。

table_name

要清理的表或物化视图的名称（可以有模式修饰）。如果指定的表示一个分区表，则它所有的叶子分区也会被清理。

column_name

要分析的指定列的名称。缺省是所有列。如果指定了一个列的列表，则ANALYZE也必须被指定。

输出

如果声明了VERBOSE，**VACUUM**会发出进度消息来表明当前正在处理哪个表。各种有关这些表的统计信息也会打印出来。

注意

要清理一个表，操作者通常必须是表的拥有者或者超级用户。但是，数据库拥有者被允许清理他们的数据库中除了共享目录之外的所有表（对于共享目录的限制意味着一个真正的数据库范围的**VACUUM**只能被超级用户执行）。**VACUUM**将会跳过执行者不具备清理权限的表。

VACUUM不能在一个事务块内被执行。

对具有GIN索引的表，**VACUUM**（任何形式）也会通过将待处理索引项移动到主要GIN索引结构中的合适位置来完成任何待处理的索引插入。

我们建议经常清理活动的生产数据库（至少每晚一次），以保证移除失效的行。在增加或删除了大量行之后， 对受影响的表执行**VACUUM ANALYZE**命令是一个很好的做法。这样做将把最近的更改更新到系统目录，并且允许 UXDB查询规划器在规划用户查询时做出更好的选择。

日常使用时，不推荐FULL选项，但在特殊情况时它会有用。一个例子是当删除或者更新了一个表中的绝大部分行时，如果希望在物理上收缩表以减少磁盘空间占用并且允许更快的表扫描，则该选项是比较合适的。**VACUUM FULL**通常会比简单**VACUUM**更多地收缩表。

VACUUM会导致I/O流量的大幅度增加，这可能导致其他活动会话性能变差。因此，有时建议使用基于代价的清理延迟特性。

UXDB包括了一个“autovacuum”工具，它可以自动进行例行的清理维护。

例子

清理单一表onek，为优化器分析它并且打印出详细的清理活动报告：

```
VACUUM (VERBOSE, ANALYZE) onek;
```

兼容性

在SQL标准中没有**VACUUM**语句。

名称

VALUES — 计算一个行集合

大纲

```
VALUES ( expression [, ...] ) [, ...]
  [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
  [ LIMIT { count | ALL } ]
  [ OFFSET start [ ROW | ROWS ] ]
  [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

描述

VALUES计算由值表达式指定的一个行值或者 一组行值。更常见的是把它用来生成一个大型命令内的“常量表”， 但是它也可以被独自使用。

当多于一行被指定时，所有行都必须具有相同数量的元素。结果表的列数据类型 由出现在该列的表达式的显式或者推导类型组合决定，决定的规则与 UNION相同（见第 7.5 节“UNION、CASE和相关结构”）。

在大型的命令中，在语法上允许VALUES出现在 SELECT出现的任何地方。因为语法把它当做一个SELECT，可以为一个VALUES 命令使用ORDER BY、 LIMIT（或者等效的FETCH FIRST） 以及OFFSET子句。

参数

expression

要在结果表（行集合）中指定位置计算并且插入的一个常量或者表达式。 在一个出现于INSERT顶层的 VALUES列表中， *expression*可以 被DEFAULT替代以表示应该插入目标列的默认值。 当VALUES出现在其他环境中时，不能使用 DEFAULT。

sort_expression

一个指示如何排序结果行的表达式或者整型常量。这个表达式 可以用column1、column2等来 引用该VALUES结果的列。详见 “ORDER BY 子句”一节。

operator

一个排序操作符。详见 “ORDER BY 子句”一节。

count

要返回的最大行数。详见 “LIMIT 子句”一节。

start

开始返回行之前要跳过的行数。详见 “LIMIT 子句”一节。

注解

应该避免具有大量行的VALUES列表，否则可能会 碰到内存不足失败或者很差的性能。出现在INSERT 中的VALUES是一种特殊情况（因为想要的列类型 可以从INSERT的目标表得知，并且不需要通过扫描 该VALUES列表来推导），因此它可以处理比其他 环境中更大的列表。

示例

一个纯粹的VALUES命令：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

这将返回一个具有两列、三行的表。它实际等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

更常用地，VALUES可以被用在一个大型 SQL 命令中。 在INSERT中最常用：

```
INSERT INTO films (code, title, did, date_prod, kind)
   VALUES ('T_601', 'Yojimbo', 106, '1961-06-16', 'Drama');
```

在INSERT的环境中，一个VALUES列表 的项可以是DEFAULT来指示应该使用该列的默认值而不是指定一个值：

```
INSERT INTO films VALUES
   ('UA502', 'Bananas', 105, DEFAULT, 'Comedy', '82 minutes'),
   ('T_601', 'Yojimbo', 106, DEFAULT, 'Drama', DEFAULT);
```

VALUES也可以被用在可以写子-SELECT 的地方，例如在一个FROM子句中：

```
SELECT f.*
  FROM films f, (VALUES('MGM', 'Horror'), ('UA', 'Sci-Fi')) AS t (studio, kind)
  WHERE f.studio = t.studio AND f.kind = t.kind;

UPDATE employees SET salary = salary * v.increase
  FROM (VALUES(1, 200000, 1.2), (2, 400000, 1.4)) AS v (depno, target, increase)
  WHERE employees.depno = v.depno AND employees.sales >= v.target;
```

注意当VALUES被用在一个FROM子句中时， 需要提供一个AS子句，与SELECT相同。 不需要为所有的列用AS子句指定名称，但是那样做是一种好 习惯（在UXDB中， VALUES的默认列名是column1、 column2等，但在其他数据库系统中可能会不同）。

当在INSERT中使用VALUES时，值都会 被自动地强制为相应目标列的数据类型。当在其他环境中使用时，有必要指定 正确的数据类型。如果项都是带引号的字符串常量，强制第一个就足以为所有 项假设数据类型：

```
SELECT * FROM machines
WHERE ip_address IN (VALUES('192.168.0.1'::inet), ('192.168.0.10'), ('192.168.1.43'));
```

> **提示**
>
> 对于简单的IN测试，最好使用IN的 list-of-scalars形式 而不是写一个上面那样的VALUES查询。标量列表方法的 书写更少并且常常更加高效。

兼容性

VALUES符合 SQL 标准。 LIMIT和OFFSET是 UXDB扩展，另见 SELECT(7)。

另见

INSERT(7), SELECT(7)

部分 II. SQL 语言

这部份描述在UXDB中SQL语言的使用。我们从描述SQL的一般语法开始，然后解释如何创建保存数据的结构、如何填充数据库以及如何查询它。中间的部分列出了在SQL命令中可用的数据类型和函数。剩余的部分则留给对于调优数据性能的重要方面。

这部份的信息被组织成让一个新用户可以从头到尾跟随它来全面理解主题，而不需要多次参考后面的内容。这些章都是自包含的，这样高级用户可以根据他们的选择阅读单独的章。这一部分的信息被以一种叙事的风格展现。需要查看一个特定命令的完整描述的读者应该去看看第 I 部分 “参考”。

这一部分的阅读者应该知道如何连接到一个UXDB数据库并且发出SQL命令。SQL通常使用UXDB的交互式终端uxsql输入，但是其他具有相似功能的程序也可以被使用。

目录

第 1 章 SQL语法

这一章描述了SQL的语法。它构成了理解后续具体介绍如何使用SQL定义和修改数据的章节的基础 。

我们同时建议已经熟悉SQL的用户仔细阅读本章，因为本章包含一些在SQL数据库中实现得不一致的以及UXDB中特有的规则和概念。

1.1. 词法结构

SQL输入由一个命令序列组成。一个命令由一个记号的序列构成，并由一个分号（“;”）终结。输入流的末端也会标志一个命令的结束。具体哪些记号是合法的与具体命令的语法有关。

一个记号可以是一个关键词、一个标识符、一个带引号的标识符、一个literal（或常量）或者一个特殊字符符号。记号通常以空白（空格、制表符、新行）来分隔，但在无歧义时并不强制要求如此（唯一的例子是一个特殊字符紧挨着其他记号）。

例如，下面是一个（语法上）合法的SQL输入：

```
SELECT * FROM MY_TABLE;
UPDATE MY_TABLE SET A = 5;
INSERT INTO MY_TABLE VALUES (3, 'hi there');
```

这是一个由三个命令组成的序列，每一行一个命令（尽管这不是必须地，在同一行中可以有超过一个命令，而且命令还可以被跨行分割）。

另外，注释也可以出现在SQL输入中。它们不是记号，它们和空白完全一样。

根据标识命令、操作符、参数的记号不同，SQL的语法不很一致。最前面的一些记号通常是命令名，因此在上面的例子中我们通常会说一个“SELECT”、一个“UPDATE”和一个“INSERT”命令。但是例如UPDATE命令总是要求一个SET记号出现在一个特定位置，而INSERT则要求一个VALUES来完成命令。每个命令的精确语法规则在第 I 部分 “参考”中介绍。

1.1.1. 标识符和关键词

上例中的SELECT、UPDATE或VALUES记号是关键词的例子，即SQL语言中具有特定意义的词。记号MY_TABLE和A则是标识符的例子。它们标识表、列或者其他数据库对象的名字，取决于使用它们的命令。因此它们有时也被简称为“名字”。关键词和标识符具有相同的词法结构，这意味着我们无法在没有语言知识的前提下区分一个标识符和关键词。一个关键词的完整列表可以在附录 A,中找到。

SQL标识符和关键词必须以一个字母（a-z，也可以是带变音符的字母和非拉丁字母）或一个下划线（_）开始。后续字符可以是字母、下划线（_）、数字（0-9）或美元符号（$）。注意根据SQL标准的字母规定，美元符号是不允许出现在标识符中的，因此它们的使用可能会降低应用的可移植性。SQL标准不会定义包含数字或者以下划线开头或结尾的关键词，因此这种形式的标识符不会与未来可能的标准扩展冲突 。

系统中一个标识符的长度不能超过 NAMEDATALEN-1 字节，在命令中可以写超过此长度的标识符，但是它们会被截断。默认情况下，NAMEDATALEN 的值为64，因此标识符的长度上限为63字节。如果这个限制有问题，可以在src/include/ux_config_manual.h中修改 NAMEDATALEN 常量。

关键词和不被引号修饰的标识符是大小写不敏感的。因此：

UPDATE MY_TABLE SET A = 5;

可以等价地写成：

uPDaTE my_TabLE SeT a = 5;

一个常见的习惯是将关键词写成大写，而名称写成小写，例如：

UPDATE my_table SET a = 5;

这里还有第二种形式的标识符：受限标识符或被引号修饰的标识符。它是由双引号（"）包围的一个任意字符序列。一个受限标识符总是一个标识符而不会是一个关键字。因此"select"可以用于引用一个名为“select”的列或者表，而一个没有引号修饰的select则会被当作一个关键词，从而在本应使用表或列名的地方引起解析错误。在上例中使用受限标识符的例子如下：

UPDATE "my_table" SET "a" = 5;

受限标识符可以包含任何字符，除了代码为0的字符（如果要包含一个双引号，则写两个双引号）。这使得可以构建原本不被允许的表或列的名称，例如包含空格或花号的名字。但是长度限制依然有效。

一种受限标识符的变体允许包括转义的用代码点标识的Unicode字符。这种变体以U&（大写或小写U跟上一个花号）开始，后面紧跟双引号修饰的名称，两者之间没有任何空白，如U&"foo"（注意这里与操作符&似乎有一些混淆，但是在&操作符周围使用空白避免了这个问题） 。在引号内，Unicode字符可以以转义的形式指定：反斜线接上4位16进制代码点号码或者反斜线和加号接上6位16进制代码点号码。例如，标识符"data"可以写成：

U&"d\0061t\+000061"

下面的例子用斯拉夫语字母写出了俄语单词 “slon”（大象）：

U&"\0441\043B\043E\043D"

如果希望使用其他转义字符来代替反斜线，可以在字符串后使用UESCAPE子句，例如：

U&"d!0061t!+000061" UESCAPE '!'

转义字符可以是除了16进制位、加号、单引号、双引号、空白字符之外的任意单个字符。注意转义字符是被写在单引号而不是双引号内。

为了在标识符中包括转义字符本身，将其写两次即可。

Unicode转义语法只有在服务器编码为UTF8时才起效。当使用其他服务器编码时，只有在ASCII范围内（最高到\007F）的编码点才能被使用。4位和6位形式都可以被用来定义UTF-16代理对来组成代码点大于U+FFFF的字符，尽管6位形式的存在使得这种做法变得不必要（代理对并不被直接存储，而是被被绑定到一个单独的代码点然后被编码到UTF-8）。

将一个标识符变得受限同时也使它变成大小写敏感的，反之非受限名称总是被转换成小写形式。例如，标识符FOO、foo和"foo"在UXDB中被认为是相同的，而"Foo"和"FOO"则互 不相同且也不同于前面三个标识符（UXDB将非受限名字转换为小写形式与SQL标准是不兼容 的，SQL标准

中要求将非受限名称转换为大写形式。这样根据标准， foo应该和 "FOO"而不是"foo"相同。如果希望写一个可移植的应用，我们应该总是用引号修饰一个特定名字或者 从不使用 引号修饰）。

1.1.2. 常量

在UXDB中有三种隐式类型常量：字符串、位串和数字。常量也可以被指定显示类型，这可以使得它被更精确地展示以及更有效地处理。这些选择将会在后续小节中讨论。

1.1.2.1. 字符串常量

在SQL中，一个字符串常量是一个由单引号（'）包围的任意字符序列，例如'This is a string'。为了在一个字符串中包括一个单引号，可以写两个相连的单引号，例如'Dianne''s horse'。注意这和一个双引号（"）不同。

两个只由空白及至少一个新行分隔的字符串常量会被连接在一起，并且将作为一个写在一起的字符串常量来对待。例如：

```
SELECT 'foo'
'bar';
```

等同于：

```
SELECT 'foobar';
```

但是：

```
SELECT 'foo'     'bar';
```

则不是合法的语法（这种有些奇怪的行为是SQL指定的，UXDB遵循了该标准）。

1.1.2.2. C风格转义的字符串常量

UXDB也接受“转义”字符串常量，这也是SQL标准的一个扩展。一个转义字符串常量可以通过在开单引号前面写一个字母E（大写或小写形式）来指定，例如E'foo'（当一个转义字符串常量跨行时，只在第一个开引号之前写E）。在一个转义字符串内部，一个反斜线字符（\）会开始一个 C风格的反斜线转义序列，在其中反斜线和后续字符的组合表示一个特殊的字节值（如表 1.1 “反斜线转义序列”中所示）。

表 1.1. 反斜线转义序列

反斜线转义序列	解释
\b	退格
\f	换页
\n	换行
\r	回车
\t	制表符
o, *oo*, *ooo* (*o* = 0 - 7)	八进制字节值
\x*h*, \x*hh* (*h* = 0 - 9, A - F)	十六进制字节值

反斜线转义序列	解释
\u*xxxx*, \U*xxxxxxxx* (*x* = 0 - 9, A - F)	16 或 32-位十六进制 Unicode 字符值

跟随在一个反斜线后面的任何其他字符被当做其字面意思。因此，要包括一个反斜线字符，请写两个反斜线（\\）。在一个转义字符串中包括一个单引号除了普通方法"之外，还可以写成\'。

要负责保证创建的字节序列由服务器字符集编码中合法的字符组成，特别是在使用八进制或十六进制转义时。如果服务器编码为 UTF-8，那么应该使用 Unicode 转义或替代的 Unicode 转义语法（在第 1.1.2.3 节 “带有 Unicode 转义的字符串常量”中解释）。替代方案可能是手工写出 UTF-8 编码字节，这可能会非常麻烦。

只有当服务器编码是UTF8时，Unicode 转义语法才能完全工作。当使用其他服务器编码时，只有在 ASCII 范围（低于\u007F）内的代码点能够被指定。4 位和 8 位形式都能被用来指定 UTF-16 代理对，用来组成代码点超过 U+FFFF 的字符，不过 8 位形式的可用从技术上使得这种做法不再是必须的（当服务器编码为UTF8并使用代理对时，它们首先被结合到一个单一代码点，然后会被用 UTF-8 编码）。

> **注意**
>
> 如果配置参数standard_conforming_strings为off，那么UXDB对常规字符串常量和转义字符串常量中的反斜线转义都识别。不过，该参数的默认值为on，意味着只在转义字符串常量中识别反斜线转义。这种行为更兼容标准，但是可能打断依赖于历史行为（反斜线转义总是会被识别）的应用。作为一种变通，可以设置该参数为off，但是最好迁移到符合新的行为。如果需要使用一个反斜线转义来表示一个特殊字符，为该字符串常量写上一个E。
>
> 在standard_conforming_strings之外，配置参数escape_string_warning和backslash_quote也决定了如何对待字符串常量中的反斜线。

代码零的字符不能出现在一个字符串常量中。

1.1.2.3. 带有 Unicode 转义的字符串常量

UXDB也支持另一种类型的字符串转义语法，它允许用代码点指定任意 Unicode 字符。一个 Unicode 转义字符串常量开始于U&（大写或小写形式的字母 U，后跟花号），后面紧跟着开引号，之间没有任何空白，例如U&'foo'（注意这产生了与操作符&的混淆。在操作符周围使用空白来避免这个问题）。在引号内，Unicode 字符可以通过写一个后跟 4 位十六进制代码点编号或者一个前面有加号的 6 位十六进制代码点编号的反斜线来指定。例如，字符串'data'可以被写为

```
U&'d\0061t\+000061'
```

下面的例子用斯拉夫字母写出了俄语的单词“slon”（大象）：

```
U&'\0441\043B\043E\043D'
```

如果想要一个不是反斜线的转义字符，可以在字符串之后使用UESCAPE子句来指定，例如：

```
U&'d!0061t!+000061' UESCAPE '!'
```

转义字符可以是出一个十六进制位、加号、单引号、双引号或空白字符之外的任何单一字符。

只有当服务器编码是UTF8时，Unicode 转义语法才能完全工作。当使用其他服务器编码时，只有在 ASCII 范围（低于\u007F）内的代码点能够被指定。4 位和 8 位形式都能被用来指定 UTF-16 代理对，用来组成代码点超过 U+FFFF 的字符，不过 8 位形式的可用从技术上使得这种做法不再是必须的（当服务器编码为UTF8并使用代理对时，它们首先被结合到一个单一代码点，然后会被用 UTF-8 编码）。

还有，只有当配置参数standard_conforming_strings被打开时，用于字符串常量的 Unicode 转义语法才能工作。这是因为否则这种语法将迷惑客户端中肯地解析 SQL 语句，进而会导致 SQL 注入以及类似的安全性问题。如果这个参数被设置为关闭，这种语法将被拒绝并且报告一个错误消息。

要在一个字符串中包括一个表示其字面意思的转义字符，把它写两次。

1.1.2.4. 美元引用的字符串常量

虽然用于指定字符串常量的标准语法通常都很方便，但是当字符串中包含了很多单引号或反斜线时很难理解它，因为每一个都需要被双写。要在这种情形下允许可读性更好的查询，UXDB提供了另一种被称为“美元引用”的方式来书写字符串常量。一个美元引用的字符串常量由一个美元符号（$）、一个可选的另个或更多字符的“标签”、另一个美元符号、一个构成字符串内容的任意字符序列、一个美元符号、开始这个美元引用的相同标签和一个美元符号组成。例如，这里有两种不同的方法使用美元引用指定字符串“Dianne’s horse”：

```
$$Dianne's horse$$
$SomeTag$Dianne's horse$SomeTag$
```

注意在美元引用字符串中，单引号可以在不被转义的情况下使用。事实上，在一个美元引用字符串中不需要对字符进行转义：字符串内容总是按其字面意思写出。反斜线不是特殊的，并且美元符号也不是特殊的，除非它们是匹配开标签的一个序列的一部分。

可以通过在每一个嵌套级别上选择不同的标签来嵌套美元引用字符串常量。这最常被用在编写函数定义上。例如：

```
$function$
BEGIN
   RETURN ($1 ~ $q$[\t\r\n\v\\]$q$);
END;
$function$
```

这里，序列q[\t\r\n\v\\]q表示一个美元引用的文字串[\t\r\n\v\\]，当该函数体被UXDB执行时它将被识别。但是因为该序列不匹配外层的美元引用的定界符$function$，它只是一些在外层字符串所关注的常量中的字符而已。

一个美元引用字符串的标签（如果有）遵循一个未被引用标识符的相同规则，除了它不能包含一个美元符号之外。标签是大小写敏感的，因此tagString contenttag是正确的，但是$TAG $String content$tag$不正确。

一个跟着一个关键词或标识符的美元引用字符串必须用空白与之分隔开，否则美元引用定界符可能会被作为前面标识符的一部分。

美元引用不是 SQL 标准的一部分，但是在书写复杂字符串文字方面，它常常是一种比兼容标准的单引号语法更方便的方法。当要表示的字符串常量位于其他常量中时它特别有用，这种情况常常在过程函数定义中出现。如果用单引号语法，上一个例子中的每个反斜线将必须被写成四个反斜线，这在解析原始字符串常量时会被缩减到两个反斜线，并且接着在函数执行期间重新解析内层字符串常量时变成一个。

1.1.2.5. 位串常量

位串常量看起来像常规字符串常量在开引号之前（中间无空白）加了一个B（大写或小写形式），例如B'1001'。位串常量中允许的字符只有0和1。

作为一种选择，位串常量可以用十六进制记号法指定，使用一个前导X（大写或小写形式），例如X'1FF'。这种记号法等价于一个用四个二进制位取代每个十六进制位的位串常量。

两种形式的位串常量可以以常规字符串常量相同的方式跨行继续。美元引用不能被用在位串常量中。

1.1.2.6. 数字常量

在这些一般形式中可以接受数字常量：

digits
digits.[*digits*][e[+-]*digits*]
[*digits*].*digits*[e[+-]*digits*]
*digits*e[+-]*digits*

其中*digits*是一个或多个十进制数字（0 到 9）。如果使用了小数点，在小数点前面或后面必须至少有一个数字。如果存在一个指数标记（e），在其后必须跟着至少一个数字。在该常量中不能嵌入任何空白或其他字符。注意任何前导的加号或减号并不实际被考虑为常量的一部分，它是一个应用到该常量的操作符。

这些是合法数字常量的例子：

```
42
3.5
4.
.001
5e2
1.925e-3
```

如果一个不包含小数点和指数的数字常量的值适合类型integer（32 位），它首先被假定为类型integer。否则如果它的值适合类型bigint（64 位），它被假定为类型bigint。再否则它会被取做类型numeric。包含小数点和/或指数的常量总是首先被假定为类型numeric。

一个数字常量初始指派的数据类型只是类型转换算法的一个开始点。在大部分情况中，常量将被根据上下文自动被强制到最合适的类型。必要时，可以通过造型它来强制一个数字值被解释为一种指定数据类型。例如，可以这样强制一个数字值被当做类型real（float4）：

```
REAL '1.23'  -- string style
1.23::REAL   -- UXDB (historical) style
```

这些实际上只是接下来要讨论的一般造型记号的特例。

1.1.2.7. 其他类型的常量

一种任意类型的一个常量可以使用下列记号中的任意一种输入：

type '*string*'
'*string*'::*type*
CAST ('*string*' AS *type*)

字符串常量的文本被传递到名为*type*的类型的输入转换例程中。其结果是指定类型的一个常量。如果对该常量的类型没有歧义（例如，当它被直接指派给一个表列时），显式类型造型可以被忽略，在那种情况下它会被自动强制。

字符串常量可以使用常规 SQL 记号或美元引用书写。

也可以使用一个类似函数的语法来指定一个类型强制：

typename ('*string*')

但是并非所有类型名都可以用在这种方法中，详见第 1.2.9 节 “类型转换”。

如第 1.2.9 节 “类型转换”中讨论的，::、CAST()以及函数调用语法也可以被用来指定任意表达式的运行时类型转换。要避免语法歧义，*type* '*string*'语法只能被用来指定简单文字常量的类型。*type* '*string*'语法上的另一个限制是它无法对数组类型工作，指定一个数组常量的类型可使用::或CAST()。

CAST()语法符合 SQL。*type* '*string*'语法是该标准的一般化：SQL 指定这种语法只用于一些数据类型，但是UXDB允许它用于所有类型。带有::的语法是UXDB的历史用法，就像函数调用语法一样。

1.1.3. 操作符

一个操作符名是最多NAMEDATALEN-1（默认为 63）的一个字符序列，其中的字符来自下面的列表：

```
+ - * / < > = ~ ! @ # % ^ & | ` ?
```

不过，在操作符名上有一些限制：

- -- and /*不能在一个操作符名的任何地方出现，因为它们将被作为一段注释的开始。
- 一个多字符操作符名不能以+或-结尾，除非该名称也至少包含这些字符中的一个：

  ```
  ~ ! @ # % ^ & | ` ?
  ```

  例如，@-是一个被允许的操作符名，但*-不是。这些限制允许UXDB解析 SQL 兼容的查询而不需要在记号之间有空格。

当使用非 SQL 标准的操作符名时，通常需要用空格分隔相邻的操作符来避免歧义。例如，如果定义了一个名为@的左一元操作符，不能写X*@Y，必须写X* @Y来确保UXDB把它读作两个操作符名而不是一个。

1.1.4. 特殊字符

一些不是数字字母的字符有一种不同于作为操作符的特殊含义。这些字符的详细用法可以在描述相应语法元素的地方找到。这一节只是为了告知它们的存在以及总结这些字符的目的。

- 跟随在一个美元符号（$）后面的数字被用来表示在一个函数定义或一个预备语句中的位置参数。在其他上下文中该美元符号可以作为一个标识符或者一个美元引用字符串常量的一部分。

- 圆括号（()）具有它们通常的含义，用来分组表达式并且强制优先。在某些情况中，圆括号被要求作为一个特定 SQL 命令的固定语法的一部分。
- 方括号（[]）被用来选择一个数组中的元素。更多关于数组的信息见第 5.15 节 “数组”
- 逗号（,）被用在某些语法结构中来分割一个列表的元素。
- 分号（;）结束一个 SQL 命令。它不能出现在一个命令中间的任何位置，除了在一个字符串常量中或者一个被引用的标识符中。
- 冒号（:）被用来从数组中选择“切片”（见第 5.15 节 “数组”）。在某些 SQL 的“方言”（例如嵌入式 SQL）中，冒号被用来作为变量名的前缀。
- 星号（*）被用在某些上下文中标记一个表的所有域或者组合值。当它被用作一个聚集函数的参数时，它还有一种特殊的含义，即该聚集不要求任何显式参数。
- 句点（.）被用在数字常量中，并且被用来分割模式、表和列名。

1.1.5. 注释

一段注释是以双斜线开始并且延伸到行结尾的一个字符序列，例如：

```
-- This is a standard SQL comment
```

另外，也可以使用 C 风格注释块：

```
/* multiline comment
 * with nesting: /* nested block comment */
 */
```

这里该注释开始于/*并且延伸到匹配出现的*/。这些注释块可按照 SQL 标准中指定的方式嵌套，但和 C 中不同。这样我们可以注释掉一大段可能包含注释块的代码。

在进一步的语法分析前，注释会被从输入流中被移除并且实际被替换为空白。

1.1.6. 操作符优先级

表 1.2 “操作符优先级（从高到低）”显示了UXDB中操作符的优先级和结合性。大部分操作符具有相同的优先并且是左结合的。操作符的优先级和结合性被硬写在解析器中。

此外，当使用二元和一元操作符的组合时，有时将需要增加圆括号。例如：

```
SELECT 5 ! - 6;
```

将被解析为：

```
SELECT 5 ! (- 6);
```

因为解析器不知道 — 知道时就为时已晚 — !被定义为一个后缀操作符而不是一个中缀操作符。在这种情况下要得到想要的行为，必须写成：

```
SELECT (5 !) - 6;
```

只是为了扩展性必须付出的代价。

表 1.2. 操作符优先级（从高到低）

操作符/元素	结合性	描述
.	左	表/列名分隔符
::	左	UXDB-风格的类型转换
[ ]	左	数组元素选择
+ -	右	一元加、一元减
^	左	指数
* / %	左	乘、除、模
+ -	左	加、减
（任意其他操作符）	左	所有其他本地以及用户定义的操作符
BETWEEN IN LIKE ILIKE SIMILAR		范围包含、集合成员关系、字符串匹配
< > = <= >= <>		比较操作符
IS ISNULL NOTNULL		IS TRUE、IS FALSE、IS NULL、IS DISTINCT FROM等
NOT	右	逻辑否定
AND	左	逻辑合取
OR	左	逻辑析取

注意该操作符有限规则也适用于与上述内建操作符具有相同名称的用户定义的操作符。例如，如果为某种自定义数据类型定义了一个“+”操作符，它将具有和内建的“+”操作符相同的优先级，不管操作符要做什么。

当一个模式限定的操作符名被用在OPERATOR语法中时，如下面的例子：

SELECT 3 OPERATOR(ux_catalog.+) 4;

OPERATOR结构被用来为“任意其他操作符”获得表 1.2 “操作符优先级（从高到低）”中默认的优先级。不管出现在OPERATOR()中的是哪个指定操作符，这都是真的。

1.2. 值表达式

值表达式被用于各种各样的环境中，例如在SELECT命令的目标列表中、作为INSERT或UPDATE中的新列值或者若干命令中的搜索条件。为了区别于一个表表达式（是一个表）的结果，一个值表达式的结果有时候被称为一个标量。值表达式因此也被称为标量表达式（或者甚至简称为表达式）。表达式语法允许使用算数、逻辑、集合和其他操作从原始部分计算值。

一个值表达式是下列之一：

- 一个常量或文字值
- 一个列引用
- 在一个函数定义体或预备语句中的一个位置参数引用

- 一个下标表达式
- 一个域选择表达式
- 一个操作符调用
- 一个函数调用
- 一个聚集表达式
- 一个窗口函数调用
- 一个类型转换
- 一个排序规则表达式
- 一个标量子查询
- 一个数组构造器
- 一个行构造器
- 另一个在圆括号（用来分组子表达式以及重载优先级）中的值表达式

在这个列表之外，还有一些结构可以被分类为一个表达式，但是它们不遵循任何一般语法规则。这些通常具有一个函数或操作符的语义并且在第 6 章 函数和操作符的合适位置解释。一个例子是IS NULL子句。

我们已经在第 1.1.2 节 “常量”中讨论过常量。下面的小节会讨论剩下的选项。

1.2.1. 列引用

一个列可以以下面的形式被引用：

correlation.columnname

*correlation*是一个表（有可能以一个模式名限定）的名字，或者是在FROM子句中为一个表定义的别名。如果列名在当前索引所使用的表中都是唯一的，关联名称和分隔用的句点可以被忽略（另见第 4 章 查询。

1.2.2. 位置参数

一个位置参数引用被用来指示一个由 SQL 语句外部提供的值。参数被用于 SQL 函数定义和预备查询中。某些客户端库还支持独立于 SQL 命令字符串来指定数据值，在这种情况中参数被用来引用那些线外数据值。一个参数引用的形式是：

$*number*

例如，考虑一个函数dept的定义：

```
CREATE FUNCTION dept(text) RETURNS dept
    AS $$ SELECT * FROM dept WHERE name = $1 $$
    LANGUAGE SQL;
```

这里$1引用函数被调用时第一个函数参数的值。

1.2.3. 下标

如果一个表达式得到了一个数组类型的值，那么可以抽取出该数组值的一个特定元素：

expression[*subscript*]

或者抽取出多个相邻元素（一个“数组切片”）：

expression[*lower_subscript*:*upper_subscript*]

（这里，方括号[ ]表示其字面意思）。每一个下标自身是一个表达式，它必须得到一个整数值。

通常，数组表达式必须被加上括号，但是当要被加下标的表达式只是一个列引用或位置参数时，括号可以被忽略。还有，当原始数组是多维时，多个下标可以被连接起来。例如：

```
mytable.arraycolumn[4]
mytable.two_d_column[17][34]
$1[10:42]
(arrayfunction(a,b))[42]
```

最后一个例子中的圆括号是必需的。详见第 5.15 节 “数组”。

1.2.4. 域选择

如果一个表达式得到一个组合类型（行类型）的值，那么可以抽取该行的指定域

expression.*fieldname*

通常行表达式必须被加上括号，但是当该表达式是仅从一个表引用或位置参数选择时，圆括号可以被忽略。例如：

```
mytable.mycolumn
$1.somecolumn
(rowfunction(a,b)).col3
```

（因此，一个被限定的列引用实际上只是域选择语法的一种特例）。一种重要的特例是从一个组合类型的表列中抽取一个域：

```
(compositecol).somefield
(mytable.compositecol).somefield
```

这里需要圆括号来显示*compositecol*是一个列名而不是一个表名，在第二种情况中则是显示mytable是一个表名而不是一个模式名。

可以通过书写.*来请求一个组合值的所有域：

```
(compositecol).*
```

这种记法的行为根据上下文会有不同，详见第 5.16.5 节 “在查询中使用组合类型”。

1.2.5. 操作符调用

对于一次操作符调用，有三种可能的语法：

expression operator expression（二元中缀操作符）
operator expression（一元前缀操作符）
expression operator（一元后缀操作符）
其中*operator*记号遵循第 1.1.3 节 “操作符”的语法规则，或者是关键词AND、OR和NOT之一，或者是一个如下形式的受限定操作符名：

OPERATOR(*schema.operatorname*)

哪个特定操作符存在以及它们是一元的还是二元的取决于由系统或用户定义的那些操作符。第 6 章 函数和操作符描述了内建操作符。

1.2.6. 函数调用

一个函数调用的语法是一个函数的名称（可能受限于一个模式名）后面跟上封闭于圆括号中的参数列表：

function_name ([*expression* [, *expression* ...]])

例如，下面会计算 2 的平方根：

sqrt(2)

当在一个某些用户不信任其他用户的数据库中发出查询时，在编写函数调用时应遵守第 7.3 节 “函数”中的安全防范措施。

内建函数的列表在第 6 章 函数和操作符中。其他函数可以由用户增加。

参数可以有选择地被附加名称。详见第 1.3 节 “调用函数”。

> **注意**
>
> 一个采用单一组合类型参数的函数可以被有选择地称为域选择语法，并且反过来域选择可以被写成函数的风格。也就是说，记号col(table)和table.col是可以互换的。这种行为是非 SQL 标准的但是在UXDB中被提供，因为它允许函数的使用来模拟“计算域”。详见第 5.16.5 节 “在查询中使用组合类型”。

1.2.7. 聚集表达式

一个聚集表达式表示在由一个查询选择的行上应用一个聚集函数。一个聚集函数将多个输入减少到一个单一输出值，例如对输入的求和或平均。一个聚集表达式的语法是下列之一：

aggregate_name (*expression* [, ...] [*order_by_clause*]) [FILTER (WHERE *filter_clause*)]
aggregate_name (ALL *expression* [, ...] [*order_by_clause*]) [FILTER (WHERE *filter_clause*)]

aggregate_name (DISTINCT *expression* [, ...] [*order_by_clause*]) [FILTER (WHERE *filter_clause*)]
aggregate_name (*) [FILTER (WHERE *filter_clause*)]
aggregate_name ([*expression* [, ...]]) WITHIN GROUP (*order_by_clause*) [FILTER (WHERE *filter_clause*)]

这里*aggregate_name*是一个之前定义的聚集（可能带有一个模式名限定），并且*expression*是任意自身不包含聚集表达式的值表达式或一个窗口函数调用。可选的*order_by_clause*和*filter_clause*描述如下。

第一种形式的聚集表达式为每一个输入行调用一次聚集。第二种形式和第一种相同，因为ALL是默认选项。第三种形式为输入行中表达式的每一个可区分值（或者对于多个表达式是值的可区分集合）调用一次聚集。第四种形式为每一个输入行调用一次聚集，因为没有特定的输入值被指定，它通常只对于count(*)聚集函数有用。最后一种形式被用于有序集聚集函数，其描述如下。

大部分聚集函数忽略空输入，这样其中一个或多个表达式得到空值的行将被丢弃。除非另有说明，对于所有内建聚集都是这样。

例如，count(*)得到输入行的总数。count(f1)得到输入行中f1为非空的数量，因为count忽略空值。而count(distinct f1)得到f1的非空可区分值的数量。

一般地，交给聚集函数的输入行是未排序的。在很多情况中这没有关系，例如不管接收到什么样的输入，min总是产生相同的结果。但是，某些聚集函数（例如array_agg 和string_agg）依据输入行的排序产生结果。当使用这类聚集时，可选的*order_by_clause*可以被用来指定想要的顺序。*order_by_clause*与查询级别的ORDER BY子句（如第 4.5 节 “行排序”所述）具有相同的语法，除非它的表达式总是仅有表达式并且不能是输出列名称或编号。例如：

```
SELECT array_agg(a ORDER BY b DESC) FROM table;
```

在处理多参数聚集函数时，注意ORDER BY出现在所有聚集参数之后。例如，要这样写：

```
SELECT string_agg(a, ',' ORDER BY a) FROM table;
```

而不能这样写：

```
SELECT string_agg(a ORDER BY a, ',') FROM table;  -- 不正确
```

后者在语法上是合法的，但是它表示用两个ORDER BY键来调用一个单一参数聚集函数（第二个是无用的，因为它是一个常量）。

如果在*order_by_clause*之外指定了DISTINCT，那么所有的ORDER BY表达式必须匹配聚集的常规参数。也就是说，不能在DISTINCT列表没有包括的表达式上排序。

> **注意**
>
> 在一个聚集函数中指定DISTINCT以及ORDER BY的能力是一种UXDB扩展。

按照到目前为止的描述，如果一般目的和统计性聚集中 排序是可选的，在要为它排序输入行时可以在该聚集的常规参数 列表中放置ORDER BY。有一个聚集函数的子集叫 做有序集聚集，它要求一个 *order_by_clause*，通常是因为 该聚集的计算只对其输入行的特定顺序有意义。有序

集聚集的典 型例子包括排名和百分位计算。按照上文的最后一种语法，对于 一个有序集聚集，*order_by_clause*被写在 WITHIN GROUP (...)中。 *order_by_clause*中的表达式 会像普通聚集参数一样对每一个输入行计算一次，按照每个 *order_by_clause*的要求排序并 且交给该聚集函数作为输入参数（这和非 WITHIN GROUP *order_by_clause*的情况不同，在其中表达 式的结果不会被作为聚集函数的参数）。如果有在 WITHIN GROUP之前的参数表达式，会把它们称 为直接参数以便与列在 *order_by_clause*中的 聚集参数相区分。与普通聚集参数不同，针对 每次聚集调用只会计算一次直接参数，而不是为每一个输入行 计算一次。这意味着只有那些变量被GROUP BY 分组时，它们才能包含这些变量。这个限制同样适用于根本不在 一个聚集表达式内部的直接参数。直接参数通常被用于百分数 之类的东西，它们只有作为每次聚集计算用一次的单一值才有意 义。直接参数列表可以为空，在这种情况下，写成() 而不是(*)（实际上 UXDB接受两种拼写，但是只有第一 种符合 SQL 标准）。

有序集聚集的调用例子：

```
SELECT percentile_cont(0.5) WITHIN GROUP (ORDER BY income) FROM households;
 percentile_cont
-----------------
           50489
```

这会从表households的 *income*列得到第 50 个百分位或者中位的值。 这里0.5是一个直接参数，对于百分位部分是一个 在不同行之间变化的值的情况它没有意义。

如果指定了FILTER，那么只有对*filter_clause*计算为真的输入行会被交给该聚集函数，其他行会被丢弃。例如：

```
SELECT
  count(*) AS unfiltered,
  count(*) FILTER (WHERE i < 5) AS filtered
FROM generate_series(1,10) AS s(i);
 unfiltered | filtered
------------+----------
         10 |        4
(1 row)
```

预定义的聚集函数在第 6.20 节 “聚集函数”中描述。其他聚集函数可以由用户增加。

一个聚集表达式只能出现在SELECT命令的结果列表或是HAVING子句中。在其他子句（如WHERE）中禁止使用它，因为那些子句的计算在逻辑上是在聚集的结果被形成之前。

当一个聚集表达式出现在一个子查询中（见第 1.2.11 节 “标量子查询”和第 6.22 节 “子查询表达式”），聚集通常在该子查询的行上被计算。但是如果该聚集的参数（以及*filter_clause*，如果有）只包含外层变量则会产生一个异常：该聚集则属于最近的那个外层，并且会在那个查询的行上被计算。该聚集表达式从整体上则是对其所出现于的子查询的一种外层引用，并且在那个子查询的任意一次计算中都作为一个常量。只出现在结果列表或HAVING子句的限制适用于该聚集所属的查询层次。

1.2.8. 窗口函数调用

一个窗口函数调用表示在一个查询选择的行的某个部分上应用一个聚集类的函数。和非窗口聚集函数调用不同，这不会被约束为将被选择的行分组为一个单一的输出行 — 在查询输出中每一个行仍保持独立。不过，窗口函数能够根据窗口函数调用的分组声明（PARTITION BY列表）访问属于当前行所在分组中的所有行。一个窗口函数调用的语法是下列之一：

function_name ([*expression* [, *expression* ...]]) [FILTER (WHERE *filter_clause*)]
 OVER *window_name*
function_name ([*expression* [, *expression* ...]]) [FILTER (WHERE *filter_clause*)] OVER
 (*window_definition*)
function_name (*) [FILTER (WHERE *filter_clause*)] OVER *window_name*
function_name (*) [FILTER (WHERE *filter_clause*)] OVER (*window_definition*)

其中*window_definition*的语法是

[*existing_window_name*]
[PARTITION BY *expression* [, ...]]
[ORDER BY *expression* [ASC | DESC | USING *operator*] [NULLS { FIRST | LAST }] [, ...]]
[*frame_clause*]

可选的*frame_clause*是下列之一

{ RANGE | ROWS | GROUPS } *frame_start* [*frame_exclusion*]
{ RANGE | ROWS | GROUPS } BETWEEN *frame_start* AND *frame_end* [*frame_exclusion*]

其中*frame_start*和*frame_end*可以是下面形式中的一种

UNBOUNDED PRECEDING
offset PRECEDING
CURRENT ROW
offset FOLLOWING
UNBOUNDED FOLLOWING

而*frame_exclusion*可以是下列之一

EXCLUDE CURRENT ROW
EXCLUDE GROUP
EXCLUDE TIES
EXCLUDE NO OTHERS

这里，*expression*表示任何自身不含有窗口函数调用的值表达式。

*window_name*是对定义在查询的WINDOW子句中的一个命名窗口声明的引用。还可以使用在WINDOW子句中定义命名窗口的相同语法在圆括号内给定一个完整的*window_definition*，详见SELECT(7)参考页。值得指出的是，OVER wname并不严格地等价于OVER (wname ...)，后者表示复制并修改窗口定义，并且在被引用窗口声明包括一个帧子句时会被拒绝。

PARTITION BY选项将查询的行分组成为分区，窗口函数会独立地处理它们。PARTITION BY工作起来类似于一个查询级别的GROUP BY子句，不过它的表达式总是只是表达式并且不能是输出列的名称或编号。如果没有PARTITION BY，该查询产生的所有行被当作一个单一分区来处理。ORDER BY选项决定被窗口函数处理的一个分区中的行的顺序。它工作起来类似于一个查询级别的ORDER BY子句，但是同样不能使用输出列的名称或编号。如果没有ORDER BY，行将被以未指定的顺序被处理。

*frame_clause*指定构成窗口帧的行集合，它是当前分区的一个子集，窗口函数将作用在该帧而不是整个分区。帧中的行集合会随着哪一行是当前行而变化。在RANGE、ROWS或者GROUPS模

式中可以指定帧，在每一种情况下，帧的范围都是从*frame_start*到*frame_end*。如果*frame_end*被省略，则末尾默认为CURRENT ROW。

UNBOUNDED PRECEDING的一个*frame_start*表示该帧开始于分区的第一行，类似地UNBOUNDED FOLLOWING的一个*frame_end*表示该帧结束于分区的最后一行。

在RANGE或GROUPS模式中，CURRENT ROW的一个*frame_start*表示帧开始于当前行的第一个平级行（被窗口的ORDER BY子句排序为与当前行等效的行），而CURRENT ROW的一个*frame_end*表示帧结束于当前行的最后一个平级行。在ROWS模式中，CURRENT ROW就表示当前行。

在*offset* PRECEDING以及*offset* FOLLOWING帧选项中，*offset*必须是一个不包含任何变量、聚集函数或者窗口函数的表达式。*offset*的含义取决于帧模式：

- 在ROWS模式中，*offset*必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行之前或者之后指定数量的行。
- 在GROUPS模式中，*offset*也必须得到一个非空、非负的整数，并且该选项表示帧开始于当前行的平级组之前或者之后指定数量的平级组，这里平级组是在ORDER BY顺序中等效的行集合（要使用GROUPS模式，在窗口定义中就必须有一个ORDER BY子句）。
- 在RANGE模式中，这些选项要求ORDER BY子句正好指定一列。*offset*指定当前行中那一列的值与它在该帧中前面或后面的行中的列值的最大差值。*offset*表达式的数据类型会随着排序列的数据类型而变化。对于数字的排序列，它通常是与排序列相同的类型，但对于日期时间排序列它是一个interval。例如，如果排序列是类型date或者timestamp，我们可以写RANGE BETWEEN '1 day' PRECEDING AND '10 days' FOLLOWING。*offset*仍然要求是非空且非负，不过“非负”的含义取决于它的数据类型。

在任何一种情况下，到帧末尾的距离都受限于到分区末尾的距离，因此对于离分区末尾比较近的行来说，帧可能会包含比较少的行。

注意在ROWS以及GROUPS模式中，0 PRECEDING和0 FOLLOWING与CURRENT ROW等效。通常在RANGE模式中，这个结论也成立（只要有一种合适的、与数据类型相关的“零”的含义）。

*frame_exclusion*选项允许当前行周围的行被排除在帧之外，即便根据帧的开始和结束选项应该把它们包括在帧中。EXCLUDE CURRENT ROW会把当前行排除在帧之外。EXCLUDE GROUP会把当前行以及它在顺序上的平级行都排除在帧之外。EXCLUDE TIES把当前行的任何平级行都从帧中排除，但不排除当前行本身。EXCLUDE NO OTHERS只是明确地指定不排除当前行或其平级行的这种默认行为。

默认的帧选项是RANGE UNBOUNDED PRECEDING，它和RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW相同。如果使用ORDER BY，这会把该帧设置为从分区开始一直到当前行的最后一个ORDER BY平级行的所有行。如果不使用ORDER BY，就意味着分区中所有的行都被包括在窗口帧中，因为所有行都成为了当前行的平级行。

限制是*frame_start*不能是UNBOUNDED FOLLOWING、*frame_end*不能是UNBOUNDED PRECEDING，并且在上述*frame_start*和*frame_end*选项的列表中*frame_end*选择不能早于*frame_start*选择出现 — 例如不允许RANGE BETWEEN CURRENT ROW AND *offset* PRECEDING，但允许ROWS BETWEEN 7 PRECEDING AND 8 PRECEDING，虽然它不会选择任何行。

如果指定了FILTER，那么只有对*filter_clause*计算为真的输入行会被交给该窗口函数，其他行会被丢弃。只有是聚集的窗口函数才接受FILTER 。

内建的窗口函数在表 6.68 “通用窗口函数”中介绍。用户可以加入其他窗口函数。此外，任何内建的或者用户定义的通用聚集或者统计性聚集都可以被用作窗口函数（有序集和假想集聚集当前不能被用作窗口函数）。

使用*的语法被用来把参数较少的聚集函数当作窗口函数调用，例如count(*) OVER (PARTITION BY x ORDER BY y)。星号（*）通常不被用于窗口相关的函数。窗口相关的函数不允许在函数参数列表中使用DISTINCT或ORDER BY。

只有在SELECT列表和查询的ORDER BY子句中才允许窗口函数调用。

更多关于窗口函数的信息可以在第 6.21 节 “窗口函数”以及第 4.2.5 节 “窗口函数处理”中找到。

1.2.9. 类型转换

一个类型造型指定从一种数据类型到另一种数据类型的转换。UXDB接受两种等价的类型造型语法：

CAST (*expression* AS *type*)
expression::*type*

CAST语法遵从 SQL，而用::的语法是UXDB的历史用法。

当一个造型被应用到一种未知类型的值表达式上时，它表示一种运行时类型转换。只有已经定义了一种合适的类型转换操作时，该造型才会成功。注意这和常量的造型（如第 1.1.2.7 节 “其他类型的常量”中所示）使用不同。应用于一个未修饰串文字的造型表示一种类型到一个文字常量值的初始赋值，并且因此它将对任意类型都成功（如果该串文字的内容对于该数据类型的输入语法是可接受的）。

如果一个值表达式必须产生的类型没有歧义（例如当它被指派给一个表列），通常可以省略显式类型造型，在这种情况下系统会自动应用一个类型造型。但是，只有对在系统目录中被标记为“OK to apply implicitly”的造型才会执行自动造型。其他造型必须使用显式造型语法调用。这种限制是为了防止出人意料的转换被无声无息地应用。

还可以用像函数的语法来指定一次类型造型：

typename (*expression*)

不过，这只对那些名字也作为函数名可用的类型有效。例如，double precision不能以这种方式使用，但是等效的float8可以。还有，如果名称interval、time和timestamp被用双引号引用，那么由于语法冲突的原因，它们只能以这种风格使用。因此，函数风格的造型语法的使用会导致不一致性并且应该尽可能被避免。

> **注意**
>
> 函数风格的语法事实上只是一次函数调用。当两种标准造型语法之一被用来做一次运行时转换时，它将在内部调用一个已注册的函数来执行该转换。简而言之，这些转换函数具有和它们的输出类型相同的名字，并且因此“函数风格的语法”无非是对底层转换函数的一次直接调用。显然，一个可移植的应用不应当依赖于它。详见CREATE CAST(7)。

1.2.10. 排序规则表达式

COLLATE子句会重载一个表达式的排序规则。它被追加到它适用的表达式：

expr COLLATE *collation*

这里*collation*可能是一个受模式限定的标识符。COLLATE子句比操作符绑得更紧，需要时可以使用圆括号。

如果没有显式指定排序规则，数据库系统会从表达式所涉及的列中得到一个排序规则，如果该表达式没有涉及列，则会默认采用数据库的默认排序规则。

COLLATE子句的两种常见使用是重载ORDER BY子句中的排序顺序，例如：

SELECT a, b, c FROM tbl WHERE ... ORDER BY a COLLATE "C";

以及重载具有区域敏感结果的函数或操作符调用的排序规则，例如：

SELECT * FROM tbl WHERE a > 'foo' COLLATE "C";

注意在后一种情况中，COLLATE子句被附加到我们希望影响的操作符的一个输入参数上。COLLATE子句被附加到该操作符或函数调用的哪个参数上无关紧要，因为被操作符或函数应用的排序规则是考虑所有参数得来的，并且一个显式的COLLATE子句将重载所有其他参数的排序规则（不过，附加非匹配COLLATE子句到多于一个参数是一种错误。）。因此，这会给出和前一个例子相同的结果：

SELECT * FROM tbl WHERE a COLLATE "C" > 'foo';

但是这是一个错误：

SELECT * FROM tbl WHERE (a > 'foo') COLLATE "C";

因为它尝试把一个排序规则应用到>操作符的结果，而它的数据类型是非可排序数据类型boolean。

1.2.11. 标量子查询

一个标量子查询是一种圆括号内的普通SELECT查询，它刚好返回一行一列（关于书写查询可见第 4 章 查询。SELECT查询被执行并且该单一返回值被使用在周围的值表达式中。将一个返回超过一行或一列的查询作为一个标量子查询使用是一种错误（但是如果在一次特定执行期间该子查询没有返回行则不是错误，该标量结果被当做为空）。该子查询可以从周围的查询中引用变量，这些变量在该子查询的任何一次计算中都将作为常量。对于其他涉及子查询的表达式还可见第 6.22 节 “子查询表达式”。

例如，下列语句会寻找每个州中最大的城市人口：

```
SELECT name, (SELECT max(pop) FROM cities WHERE cities.state = states.name)
  FROM states;
```

1.2.12. 数组构造器

一个数组构造器是一个能构建一个数组值并且将值用于它的成员元素的表达式。一个简单的数组构造器由关键词ARRAY、一个左方括号[、一个用于数组元素值的表达式列表（用逗号分隔）以及最后的一个右方括号]组成。例如：

```
SELECT ARRAY[1,2,3+4];
  array
---------
 {1,2,7}
(1 row)
```

默认情况下，数组元素类型是成员表达式的公共类型，使用和UNION或CASE结构（见第 7.5 节“UNION、CASE和相关结构”）相同的规则决定。可以通过显式将数组构造器造型为想要的类型来重载，例如：

```
SELECT ARRAY[1,2,22.7]::integer[];
  array
----------
 {1,2,23}
(1 row)
```

这和把每一个表达式单独地造型为数组元素类型的效果相同。关于造型的更多信息请见第 1.2.9 节 “类型转换”。

多维数组值可以通过嵌套数组构造器来构建。在内层的构造器中，关键词ARRAY可以被忽略。例如，这些语句产生相同的结果：

```
SELECT ARRAY[ARRAY[1,2], ARRAY[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)

SELECT ARRAY[[1,2],[3,4]];
     array
---------------
 {{1,2},{3,4}}
(1 row)
```

因为多维数组必须是矩形的，处于同一层次的内层构造器必须产生相同维度的子数组。任何被应用于外层ARRAY构造器的造型会自动传播到所有的内层构造器。

多维数组构造器元素可以是任何得到一个正确种类数组的任何东西，而不仅仅是一个子-ARRAY结构。例如：

```
CREATE TABLE arr(f1 int[], f2 int[]);

INSERT INTO arr VALUES (ARRAY[[1,2],[3,4]], ARRAY[[5,6],[7,8]]);

SELECT ARRAY[f1, f2, '{{9,10},{11,12}}'::int[]] FROM arr;
                array
```

```
-----------------------------------------------
 {{{1,2},{3,4}},{{5,6},{7,8}},{{9,10},{11,12}}}
(1 row)
```

可以构造一个空数组，但是因为无法得到一个无类型的数组，必须显式地把空数组转换成想要的类型。例如：

```
SELECT ARRAY[]::integer[];
 array
-------
 {}
(1 row)
```

也可以从一个子查询的结果构建一个数组。在这种形式中，数组构造器被写为关键词ARRAY后跟着一个加了圆括号（不是方括号）的子查询。例如：

```
SELECT ARRAY(SELECT oid FROM ux_proc WHERE proname LIKE 'bytea%');
                          array
------------------------------------------------------------------------
 {2011,1954,1948,1952,1951,1244,1950,2005,1949,1953,2006,31,2412,2413}
(1 row)

SELECT ARRAY(SELECT ARRAY[i, i*2] FROM generate_series(1,5) AS a(i));
           array
---------------------------------
 {{1,2},{2,4},{3,6},{4,8},{5,10}}
(1 row)
```

子查询必须返回一个单一列。如果子查询的输出列是非数组类型， 结果的一维数组将为该子查询结果中的每一行有一个元素， 并且有一个与子查询的输出列匹配的元素类型。如果子查询的输出列 是一种数组类型，结果将是同类型的一个数组，但是要高一个维度。 在这种情况下，该子查询的所有行必须产生同样维度的数组，否则结果 就不会是矩形形式。

用ARRAY构建的一个数组值的下标总是从一开始。更多关于数组的信息，请见第 5.15 节 “数组”。

1.2.13. 行构造器

一个行构造器是能够构建一个行值（也称作一个组合类型）并用值作为其成员域的表达式。一个行构造器由关键词ROW、一个左圆括号、用于行的域值的零个或多个表达式（用逗号分隔）以及最后的一个右圆括号组成。例如：

```
SELECT ROW(1,2.5,'this is a test');
```

当在列表中有超过一个表达式时，关键词ROW是可选的。

一个行构造器可以包括语法*rowvalue*.*，它将被扩展为该行值的元素的一个列表，就像在一个顶层SELECT列表（见第 5.16.5 节 “在查询中使用组合类型”中使用.*时发生的事情一样。例如，如果表t有列f1和f2，那么这些是相同的：

```
SELECT ROW(t.*, 42) FROM t;
```

```
SELECT ROW(t.f1, t.f2, 42) FROM t;
```

> **注意**
>
> .*语法不会在行构造器中被扩展，这样写ROW(t.*, 42)会创建一个有两个域的行，其第一个域是另一个行值。新的行为通常更有用。如果需要嵌套行值的旧行为，写内层行值时不要用.*，例如ROW(t, 42)。

默认情况下，由一个ROW表达式创建的值是一种匿名记录类型。如果必要，它可以被造型为一种命名的组合类型 — 或者是一个表的行类型，或者是一种用CREATE TYPE AS创建的组合类型。为了避免歧义，可能需要一个显式造型。例如：

```
CREATE TABLE mytable(f1 int, f2 float, f3 text);

CREATE FUNCTION getf1(mytable) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 不需要造型因为只有一个 getf1() 存在
SELECT getf1(ROW(1,2.5,'this is a test'));
 getf1
-------
     1
(1 row)

CREATE TYPE myrowtype AS (f1 int, f2 text, f3 numeric);

CREATE FUNCTION getf1(myrowtype) RETURNS int AS 'SELECT $1.f1' LANGUAGE SQL;

-- 现在我们需要一个造型来指示要调用哪个函数：
SELECT getf1(ROW(1,2.5,'this is a test'));
ERROR:  function getf1(record) is not unique

SELECT getf1(ROW(1,2.5,'this is a test')::mytable);
 getf1
-------
     1
(1 row)

SELECT getf1(CAST(ROW(11,'this is a test',2.5) AS myrowtype));
 getf1
-------
    11
(1 row)
```

行构造器可以被用来构建存储在一个组合类型表列中的组合值，或者被传递给一个接受组合参数的函数。还有，可以比较两个行值，或者用IS NULL或IS NOT NULL测试一个行，例如：

```
SELECT ROW(1,2.5,'this is a test') = ROW(1, 3, 'not the same');

SELECT ROW(table.*) IS NULL FROM table;  -- detect all-null rows
```

详见第 6.23 节 “行和数组比较”。如第 6.22 节 “子查询表达式”中所讨论的，行构造器也可以被用来与子查询相连接。

1.2.14. 表达式计算规则

子表达式的计算顺序没有被定义。特别地，一个操作符或函数的输入不必按照从左至右或其他任何固定顺序进行计算。

此外，如果一个表达式的结果可以通过只计算其一部分来决定，那么其他子表达式可能完全不需要被计算。例如，如果我们写：

SELECT true OR somefunc();

那么somefunc()将（可能）完全不被调用。如果我们写成下面这样也是一样：

SELECT somefunc() OR true;

注意这和一些编程语言中布尔操作符从左至右的“短路”不同。

因此，在复杂表达式中使用带有副作用的函数是不明智的。在WHERE和HAVING子句中依赖副作用或计算顺序尤其危险，因为在建立一个执行计划时这些子句会被广泛地重新处理。这些子句中布尔表达式（AND/OR/NOT的组合）可能会以布尔代数定律所允许的任何方式被重组。

当有必要强制计算顺序时，可以使用一个CASE结构（见第 6.17 节 “条件表达式和函数”。例如，在一个WHERE子句中使用下面的方法尝试避免除零是不可靠的：

SELECT ... WHERE x > 0 AND y/x > 1.5;

但是这是安全的：

SELECT ... WHERE CASE WHEN x > 0 THEN y/x > 1.5 ELSE false END;

一个以这种风格使用的CASE结构将使得优化尝试失败，因此只有必要时才这样做（在这个特别的例子中，最好通过写y > 1.5*x来回避这个问题）。

不过，CASE不是这类问题的万灵药。上述技术的一个限制是， 它无法阻止常量子表达式的提早计算。当查询被规划而不是被执行时，被标记成 IMMUTABLE的函数和操作符可以被计算。因此

SELECT CASE WHEN x > 0 THEN x ELSE 1/0 END FROM tab;

很可能会导致一次除零失败，因为规划器尝试简化常量子表达式。即便是 表中的每一行都有x > 0（这样运行时永远不会进入到 ELSE分支）也是这样。

虽然这个特别的例子可能看起来愚蠢，没有明显涉及常量的情况可能会发生 在函数内执行的查询中，因为因为函数参数的值和本地变量可以作为常量 被插入到查询中用于规划目的。例如，在PL/uxSQL函数 中，使用一个IF-THEN-ELSE语句来 保护一种有风险的计算比把它嵌在一个CASE表达式中要安全得多。

另一个同类型的限制是，一个CASE无法阻止其所包含的聚集表达式 的计算，因为在考虑SELECT列表或HAVING子句中的 其他表达式之前，会先计算聚集表达式。例如，下面的查询会导致一个除零错误， 虽然看起来好像已经这种情况加以了保护：

```
SELECT CASE WHEN min(employees) > 0
            THEN avg(expenses / employees)
       END
  FROM departments;
```

min()和avg()聚集会在所有输入行上并行地计算， 因此如果任何行有*employees*等于零，在有机会测试 min()的结果之前，就会发生除零错误。取而代之的是，可以使用 一个WHERE或FILTER子句来首先阻止有问题的输入行到达 一个聚集函数。

1.3. 调用函数

UXDB允许带有命名参数的函数被使用位置或命名记号法调用。命名记号法对于有大量参数的函数特别有用，因为它让参数和实际参数之间的关联更明显和可靠。在位置记号法中，书写一个函数调用时，其参数值要按照它们在函数声明中被定义的顺序书写。在命名记号法中，参数根据名称匹配函数参数，并且可以以任何顺序书写。对于每一种记法，还要考虑函数参数类型的效果，这些在第 7.3 节 “函数有介绍。

在任意一种记号法中，在函数声明中给出了默认值的参数根本不需要在调用中写出。但是这在命名记号法中特别有用，因为任何参数的组合都可以被忽略。而在位置记号法中参数只能从右往左忽略。

UXDB也支持混合记号法，它组合了位置和命名记号法。在这种情况中，位置参数被首先写出并且命名参数出现在其后。

下列例子将展示所有三种记号法的用法：

```
CREATE FUNCTION concat_lower_or_upper(a text, b text, uppercase boolean DEFAULT false)
RETURNS text
AS
$$
 SELECT CASE
       WHEN $3 THEN UPPER($1 || ' ' || $2)
       ELSE LOWER($1 || ' ' || $2)
       END;
$$
LANGUAGE SQL IMMUTABLE STRICT;
```

函数concat_lower_or_upper有两个强制参数，a和b。此外，有一个可选的参数uppercase，其默认值为false。a和b输入将被串接，并且根据uppercase参数被强制为大写或小写形式。这个函数的剩余细节对这里并不重要。

1.3.1. 使用位置记号

在UXDB中，位置记号法是给函数传递参数的传统机制。一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World', true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

所有参数被按照顺序指定。结果是大写形式，因为uppercase被指定为true。另一个例子：

```
SELECT concat_lower_or_upper('Hello', 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

这里，uppercase参数被忽略，因此它接收它的默认值false，并导致小写形式的输出。在位置记号法中，参数可以按照从右往左被忽略并且因此而得到默认值。

1.3.2. 使用命名记号

在命名记号法中，每一个参数名都用=> 指定来把它与参数表达式分隔开。例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World');
 concat_lower_or_upper
-----------------------
 hello world
(1 row)
```

再次，参数uppercase被忽略，因此它被隐式地设置为false。使用命名记号法的一个优点是参数可以用任何顺序指定，例如：

```
SELECT concat_lower_or_upper(a => 'Hello', b => 'World', uppercase => true);
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)

SELECT concat_lower_or_upper(a => 'Hello', uppercase => true, b => 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

为了向后兼容性，基于 ″:=″ 的旧语法仍被支持：

```
SELECT concat_lower_or_upper(a := 'Hello', uppercase := true, b := 'World');
 concat_lower_or_upper
-----------------------
 HELLO WORLD
(1 row)
```

1.3.3. 使用混合记号

混合记号法组合了位置和命名记号法。不过，正如已经提到过的，命名参数不能超越位置参数。例如：

```
SELECT concat_lower_or_upper('Hello', 'World', uppercase => true);
 concat_lower_or_upper
```

```
-----------------------
 HELLO WORLD
(1 row)
```

在上述查询中，参数a和b被以位置指定，而uppercase通过名字指定。在这个例子中，这只增加了一点文档。在一个具有大量带默认值参数的复杂函数中，命名的或混合的记号法可以节省大量的书写并且减少出错的机会。

> **注意**
>
> 命名的和混合的调用记号法当前不能在调用聚集函数时使用（但是当聚集函数被用作窗口函数时它们可以被使用）。

第 2 章 数据定义

本章包含了如何创建用来保存数据的数据库结构。在一个关系型数据库中，原始数据被存储在表中，因此本章的主要工作就是解释如何创建和修改表，以及哪些特性可以控制何种数据会被存储在表中。接着，我们讨论表如何被组织成模式，以及如何将权限分配给表。最后，我们将将简短地介绍其他一些影响数据存储的特性，例如继承、表分区、视图、函数和触发器。

2.1. 表基础

关系型数据库中的一个表非常像纸上的一张表：它由行和列组成。列的数量和顺序是固定的，并且每一列拥有一个名字。行的数目是变化的，它反映了在一个给定时刻表中存储的数据量。SQL并不保证表中行的顺序。当一个表被读取时，表中的行将以非特定顺序出现，除非明确地指定需要排序。这些将在第 4 章 查询介绍。此外，SQL不会为行分配唯一的标识符，因此在一个表中可能会存在一些完全相同的行。这是SQL之下的数学模型导致的结果，但并不是所期望的。稍后在本章中我们将看到如何处理这种问题。

每一列都有一个数据类型。数据类型约束着一组可以分配给列的可能值，并且它为列中存储的数据赋予了语义，这样它可以用于计算。例如，一个被声明为数字类型的列将不会接受任何文本串，而存储在这样一列中的数据可以用来进行数学计算。反过来，一个被声明为字符串类型的列将接受几乎任何一种的数据，它可以进行如字符串连接的操作但不允许进行数学计算。

UXDB包括了相当多的内建数据类型，可以适用于很多应用。用户也可以定义他们自己的数据类型。大部分内建数据类型有着显而易见的名称和语义，所以我们将它们的详细解释放在第 5 章 数据类型中。一些常用的数据类型是：用于整数的integer；可以用于分数的numeric；用于字符串的text，用于日期的date，用于一天内时间的time以及可以同时包含日期和时间的timestamp。

要创建一个表，我们要用到CREATE TABLE(7)命令。在这个命令中 我们需要为新表至少指定一个名字、列的名字及数据类型。例如：

```
CREATE TABLE my_first_table (
    first_column text,
    second_column integer
);
```

这将创建一个名为my_first_table的表，它拥有两个列。第一个列名为first_column且数据类型为text；第二个列名为second_column且数据类型为integer。表和列的名字遵循第 1.1.1 节 “标识符和关键词”中解释的标识符语法。类型名称通常也是标识符，但是也有些例外。注意列的列表由逗号分隔并被圆括号包围。

当然，前面的例子是非常不自然的。通常，我们为表和列赋予的名称都会表明它们存储着什么类别的数据。因此让我们再看一个更现实的例子：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

（numeric类型能够存储小数部分，典型的例子是金额。）

> **提示**
>
> 当我们创建很多相关的表时，最好为表和列选择一致的命名模式。例如，一种选择是用单数或复数名词作为表名，每一种都受到一些理论家支持。

一个表能够拥有的列的数据是有限的，根据列的类型，这个限制介于250和1600之间。但是，极少会定义一个接近这个限制的表，即便有也是一个值的商榷的设计。

如果我们不再需要一个表，我们可以通过使用DROP TABLE(7)命令来移除它。例如：

```
DROP TABLE my_first_table;
DROP TABLE products;
```

尝试移除一个不存在的表会引起错误。然而，在SQL脚本中在创建每个表之前无条件地尝试移除它的做法是很常见的，即使发生错误也会忽略之，因此这样的脚本可以在表存在和不存在时都工作得很好（如果喜欢，可以使用DROP TABLE IF EXISTS变体来防止出现错误消息，但这并非标准SQL）。

如果我们需要修改一个已经存在的表，请参考本章稍后的第 2.6 节 “修改表”。

利用到目前为止所讨论的工具，我们可以创建一个全功能的表。本章的后续部分将集中于为表定义增加特性来保证数据完整性、安全性或方便。如果希望现在就去填充表，可以跳过这些直接去第 3 章 数据操纵

2.2. 默认值

一个列可以被分配一个默认值。当一个新行被创建且没有为某些列指定值时，这些列将会被它们相应的默认值填充。一个数据操纵命令也可以显式地要求一个列被置为它的默认值，而不需要知道这个值到底是什么（数据操纵命令详见第 3 章 数据操纵。

如果没有显式指定默认值，则默认值是空值。这是合理的，因为空值表示未知数据。

在一个表定义中，默认值被列在列的数据类型之后。例如：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric DEFAULT 9.99
);
```

默认值可以是一个表达式，它将在任何需要插入默认值的时候被实时计算（不是表创建时）。一个常见的例子是为一个timestamp列指定默认值为CURRENT_TIMESTAMP，这样它将得到行被插入时的时间。另一个常见的例子是为每一行生成一个“序列号” 。这在UXDB可以按照如下方式实现：

```
CREATE TABLE products (
    product_no integer DEFAULT nextval('products_product_no_seq'),
    ...
);
```

这里nextval()函数从一个序列对象第 6.16 节 “序列操作函数”。还有一种特别的速写：

```
CREATE TABLE products (
    product_no SERIAL,
    ...
);
```

SERIAL速写将在第 5.1.4 节 “序数类型”进一步讨论。

2.3. 生成列

生成的列是一个特殊的列，它总是从其他列计算而来。因此说，它对于列就像视图对于表一样。生成列有两种:存储列和虚拟列。 存储生成列在写入(插入或更新)时计算，并且像普通列一样占用存储空间。虚拟生成列不占用存储空间并且在读取时进行计算。 如此看来，虚拟生成列类似于视图，存储生成列类似于物化视图(除了它总是自动更新之外)。 UXDB目前只实现了存储生成列。

建立一个生成列，在 CREATE TABLE中使用 GENERATED ALWAYS AS 子句，例如:

```
CREATE TABLE people (
    ...,
    height_cm numeric,
    height_in numeric GENERATED ALWAYS AS (height_cm / 2.54) STORED
);
```

必须指定关键字 STORED 以选择存储类型的生成列。更多细节请参见 CREATE TABLE(7) 。

生成列不能被直接写入. 在INSERT 或 UPDATE 命令中，不能为生成列指定值，但是可以指定关键字DEFAULT。

考虑列缺省情况和生成列之间的差异。 如果没有提供其他值，列缺省情况下在行被首次插入时计算一次;生成列则在行每次改变时进行更新，并且不能被取代。 列缺省情况下不能引用表的其他列；生成表达式通常会这样做。 列缺省情况下可以使用易失性函数，例如random()或引用当前时间函数；而对于生成列这是不允许的。

生成列和涉及生成列的表的定义有几个限制:

- 生成表达式只能使用不可变函数，并且不能使用子查询或以任何方式引用当前行以外的任何内容。
- 生成表达式不能引用另一个生成列。
- 生成表达式不能引用系统表，除了 tableoid。
- 生成列不能具有列默认或标识定义。
- 生成列不能是分区键的一部分。
- 外部表可以有生成列. 更多细节请参见 CREATE FOREIGN TABLE(7) 。

使用生成列的其他注意事项。

- 生成列保留着有别于其下层的基础列的访问权限。因此，可以对其进行排列以便于从生成列中读取特定的角色，而不是从下层基础列。
- 从概念上讲，生成列在BEFORE 触发器运行后更新。 因此，BEFORE 触发器中的基础列所做的变更将反映在生成列中。 但相反，不允许访问BEFORE 触发器中的生成列。

2.4. 约束

数据类型是一种限制能够存储在表中数据类别的方法。但是对于很多应用来说，它们提供的约束太粗糙。例如，一个包含产品价格的列应该只接受正值。但是没有任何一种标准数据类型只接受正值。另一个问题是我们可能需要根据其他列或行来约束一个列中的数据。例如，在一个包含产品信息的表中，对于每个产品编号应该只有一行。

到目前为止，SQL允许我们在列和表上定义约束。约束让我们能够根据我们的愿望来控制表中的数据。如果一个用户试图在一个列中保存违反一个约束的数据，一个错误会被抛出。即便是这个值来自于默认值定义，这个规则也同样适用。

2.4.1. 检查约束

一个检查约束是最普通的约束类型。它允许我们指定一个特定列中的值必须要满足一个布尔表达式。例如，为了要求正值的产品价格，我们可以使用：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

如所见，约束定义就和默认值定义一样跟在数据类型之后。默认值和约束之间的顺序没有影响。一个检查约束有关键字CHECK以及其后的包围在圆括号中的表达式组成。检查约束表达式应该涉及到被约束的列，否则该约束也没什么实际意义。

我们也可以给与约束一个独立的名称。这会使得错误消息更为清晰，同时也允许我们在需要更改约束时能引用它。语法为：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CONSTRAINT positive_price CHECK (price > 0)
);
```

要指定一个命名的约束，请在约束名称标识符前使用关键词CONSTRAINT，然后把约束定义放在标识符之后（如果没有以这种方式指定一个约束名称，系统将会为我们选择一个）。

一个检查约束也可以引用多个列。例如我们存储一个普通价格和一个打折后的价格，而我们希望保证打折后的价格低于普通价格：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

前两个约束看起来很相似。第三个则使用了一种新语法。它并没有依附在一个特定的列，而是作为一个独立的项出现在逗号分隔的列列表中。列定义和这种约束定义可以以混合的顺序出现在列表中。

我们将前两个约束称为列约束，而第三个约束为表约束，因为它独立于任何一个列定义。列约束也可以写成表约束，但反过来不行，因为一个列约束只能引用它所依附的那一个列（UXDB并不强制要求这个规则，但是如果我们希望表定义能够在其他数据库系统中工作，那就应该遵循它）。上述例子也可以写成：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CHECK (price > discounted_price)
);
```

甚至是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

这只是口味的问题。

表约束也可以用列约束相同的方法来指定名称：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0),
    CONSTRAINT valid_discount CHECK (price > discounted_price)
);
```

需要注意的是，一个检查约束在其检查表达式值为真或空值时被满足。因为当任何操作数为空时大部分表达式将计算为空值，所以它们不会阻止被约束列中的控制。为了保证一个列不包含控制，可以使用下一节中的非空约束。

> **注意**
>
> UXDB不支持引用表数据以外的要检查的新增或更新的行的CHECK约束。虽然违反此规则的CHECK约束在简单测试中看起来能工作，它不能保证数据库不会达到约束条件为假(false)的状态（由于涉及的其他行随后发生了更改）。 这将导致数据库转储和重新加载失败。 即使完整的数据库状态与约束一致，重新加载也可能失败，因为行未按照满足约束的顺序加

载。 如果可能的话，使用UNIQUE, EXCLUDE,或 FOREIGN KEY约束以表示跨行和跨表限制。

如果希望的是在插入行时的时候对其他行进行一次性检查，而不是持续维护的一致性保证，一个自定义的trigger可以用于实现这个功能。 （此方法避免了转储/重新加载问题，因为ux_dump不会重新安装触发器直到重新加载数据之后，因此不会在转储/重新加载期间强制执行检查。）

注意

UXDB假定CHECK约束的条件是不可变的，也就是说，它们始终为同一输入行提供相同的结果。 这个假设是仅在插入或更新行时,而不是在其他时间检查CHECK约束的原因。 （上面关于不引用其他表数据的警告实际上是此限制的特殊情况。）

打破此假设的常见方法的一个示例是在 CHECK表达式中引用用户定义的函数，然后更改该函数的行为。 UXDB不会禁止那样，但它不会注意到现在表中是否有行违反了CHECK约束。这将导致后续数据库转储和重新加载失败。 处理此类更改的建议方法是删除约束（使用ALTER TABLE），调整函数定义，然后重新添加约束，从而对所有表行进行重新检查。

2.4.2. 非空约束

一个非空约束仅仅指定一个列中不会有空值。语法例子：

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric
);
```

一个非空约束总是被写成一个列约束。一个非空约束等价于创建一个检查约束CHECK (*column_name* IS NOT NULL)，但在UXDB中创建一个显式的非空约束更高效。这种方式创建的非空约束的缺点是我们无法为它给予一个显式的名称。

当然，一个列可以有多于一个的约束，只需要将这些约束一个接一个写出：

```
CREATE TABLE products (
    product_no integer NOT NULL,
    name text NOT NULL,
    price numeric NOT NULL CHECK (price > 0)
);
```

约束的顺序没有关系，因为并不需要决定约束被检查的顺序。

NOT NULL约束有一个相反的情况：NULL约束。这并不意味着该列必须为空，进而肯定是无用的。相反，它仅仅选择了列可能为空的默认行为。SQL标准中并不存在NULL约束，因此它不能被用于可移植的应用中（UXDB中加入它是为了和某些其他数据库系统兼容）。但是某些用户喜欢它，因为它使得在一个脚本文件中可以很容易的进行约束切换。例如，初始时我们可以：

```
CREATE TABLE products (
```

```
    product_no integer NULL,
    name text NULL,
    price numeric NULL
);
```

然后可以在需要的地方插入NOT关键词。

> 提示
>
> 在大部分数据库中多数列应该被标记为非空。

2.4.3. 唯一约束

唯一约束保证\在一列中或者一组列中保存的数据在表中所有行间是唯一的。写成一个列约束的语法是：

```
CREATE TABLE products (
    product_no integer UNIQUE,
    name text,
    price numeric
);
```

写成一个表约束的语法是：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric,
    UNIQUE (product_no)
);
```

当写入表约束时。

要为一组列定义一个唯一约束，把它写作一个表级约束，列名用逗号分隔：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    UNIQUE (a, c)
);
```

这指定这些列的组合值在整个表的范围内是唯一的，但其中任意一列的值并不需要是（一般也不是）唯一的。

我们可以通常的方式为一个唯一索引命名：

```
CREATE TABLE products (
    product_no integer CONSTRAINT must_be_different UNIQUE,
    name text,
    price numeric
```

);

增加一个唯一约束会在约束中列出的列或列组上自动创建一个唯一B-tree索引。只覆盖某些行的唯一性限制不能被写为一个唯一约束，但可以通过创建一个唯一的部分索引来强制这种限制。

通常，如果表中有超过一行在约束所包括列上的值相同，将会违反唯一约束。但是在这种比较中，两个空值被认为是不同的。这意味着即便存在一个唯一约束，也可以存储多个在至少一个被约束列中包含空值的行。这种行为符合SQL标准，但我们听说一些其他SQL数据库可能不遵循这个规则。所以在开发需要可移植的应用时应注意这一点。

2.4.4. 主键

一个主键约束表示可以用作表中行的唯一标识符的一个列或者一组列。这要求那些值都是唯一的并且非空。因此，下面的两个表定义接受相同的数据：

```
CREATE TABLE products (
    product_no integer UNIQUE NOT NULL,
    name text,
    price numeric
);
```

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

主键也可以包含多于一个列，其语法和唯一约束相似：

```
CREATE TABLE example (
    a integer,
    b integer,
    c integer,
    PRIMARY KEY (a, c)
);
```

增加一个主键将自动在主键中列出的列或列组上创建一个唯一B-tree索引。并且会强制这些列被标记为NOT NULL。

一个表最多只能有一个主键（可以有任意数量的唯一和非空约束，它们可以达到和主键几乎一样的功能，但只能有一个被标识为主键）。关系数据库理论要求每一个表都要有一个主键。但UXDB中并未强制要求这一点，但是最好能够遵循它。

主键对于文档和客户端应用都是有用的。例如，一个允许修改行值的 GUI 应用可能需要知道一个表的主键，以便能唯一地标识行。如果定义了主键，数据库系统也有多种方法来利用主键。例如，主键定义了外键要引用的默认目标列。

2.4.5. 外键

一个外键约束指定一列（或一组列）中的值必须匹配出现在另一个表中某些行的值。我们说这维持了两个关联表之间的引用完整性。

例如我们有一个使用过多次的产品表：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

让我们假设我们还有一个存储这些产品订单的表。我们希望保证订单表中只包含真正存在的产品的订单。因此我们在订单表中定义一个引用产品表的外键约束：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

现在就不可能创建包含不存在于产品表中的*product_no*值（非空）的订单。

我们说在这种情况下，订单表是*引用表*而产品表是*被引用表*。相应地，也有引用和被引用列的说法。

我们也可以把上述命令简写为：

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

因为如果缺少列的列表，则被引用表的主键将被用作被引用列。

一个外键也可以约束和引用一组列。照例，它需要被写成表约束的形式。下面是一个例子：

```
CREATE TABLE t1 (
  a integer PRIMARY KEY,
  b integer,
  c integer,
  FOREIGN KEY (b, c) REFERENCES other_table (c1, c2)
);
```

当然，被约束列的数量和类型应该匹配被引用列的数量和类型。

按照前面的方式，我们可以为一个外键约束命名。

一个表可以有超过一个的外键约束。这被用于实现表之间的多对多关系。例如我们有关于产品和订单的表，但我们现在希望一个订单能包含多种产品（这在上面的结构中是不允许的）。我们可以使用这种表结构：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);
```

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products,
    order_id integer REFERENCES orders,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

注意在最后一个表中主键和外键之间有重叠。

我们知道外键不允许创建与任何产品都不相关的订单。但如果一个产品在一个引用它的订单创建之后被移除会发生什么？SQL允许我们处理这种情况。直观上，我们有几种选项：

- 不允许删除一个被引用的产品
- 同时也删除引用产品的订单
- 其他？

为了说明这些，让我们在上面的多对多关系例子中实现下面的策略：当某人希望移除一个仍然被一个订单引用（通过order_items）的产品时　，我们组织它。如果某人移除一个订单，订单项也同时被移除：

```
CREATE TABLE products (
    product_no integer PRIMARY KEY,
    name text,
    price numeric
);

CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    shipping_address text,
    ...
);

CREATE TABLE order_items (
    product_no integer REFERENCES products ON DELETE RESTRICT,
    order_id integer REFERENCES orders ON DELETE CASCADE,
    quantity integer,
    PRIMARY KEY (product_no, order_id)
);
```

限制删除或者级联删除是两种最常见的选项。RESTRICT阻止删除一个被引用的行。NO ACTION表示在约束被检察时如果有任何引用行存在，则会抛出一个错误，这是我们没有指定任何东西时的默认行为（这两种选择的本质不同在于NO　ACTION允许检查被推迟到事务的最后，而RESTRICT则不会）。CASCADE指定当一个被引用行被删除后，引用它的行也应该被自动删除。还有其他两种选项：SET NULL和SET DEFAULT。这些将导致在被引用行被删除后，引用行中的引用列被置为空值或它们的默认值。注意这些并不会是我们免于遵守任何约束。例如，如果一个动作指定了SET DEFAULT，但是默认值不满足外键约束，操作将会失败。

与ON DELETE相似，同样有ON UPDATE可以用在一个被引用列被修改（更新）的情况，可选的动作相同。在这种情况下，CASCADE意味着被引用列的更新值应该被复制到引用行中。

正常情况下，如果一个引用行的任意一个引用列都为空，则它不需要满足外键约束。如果在外键定义中加入了MATCH　FULL，一个引用行只有在它的所有引用列为空时才不需要满足外键约束（因此空和非空值的混合肯定会导致MATCH　FULL约束失败）。如果不希望引用行能够避开外键约束，将引用行声明为NOT NULL。

一个外键所引用的列必须是一个主键或者被唯一约束所限制。这意味着被引用列总是拥有一个索引（位于主键或唯一约束之下的索引），因此在其上进行的一个引用行是否匹配的检查将会很高效。由于从被引用表中DELETE一行或者UPDATE一个被引用列将要求对引用表进行扫描以得到匹配旧值的行，在引用列上建立合适的索引也会大有益处。由于这种做法并不是必须的，而且创建索引也有很多种选择，所以外键约束的定义并不会自动在引用列上创建索引。

更多关于更新和删除数据的信息请见第　3　章　数据操纵外键约束的语法描述请参考CREATE TABLE(7)。

2.4.5.1.　兼容mysql外键约束开关

控制外键约束的失效与生效。　外键约束开关FOREIGN_KEY_CHECKS默认为on开启外键约束，当开关关闭时跳过对应的外键检查操作，如：UPDATE、INSERT　、DELETE、DROP、ADD　CONSTRAINT操作。该开关仅在UXDB数据库mysql运行模式下有效。

控制外键约束失效与生效有如下两种方式。

- set FOREIGN_KEY_CHECKS = on; 或 set FOREIGN_KEY_CHECKS = off;
- alter system set FOREIGN_KEY_CHECKS = on; 或 alter system set FOREIGN_KEY_CHECKS = off;

2.4.6.　排他约束

排他约束保证如果将任何两行的指定列或表达式使用指定操作符进行比较，至少其中一个操作符比较将会返回否或空值。语法是：

```
CREATE TABLE circles (
    c circle,
    EXCLUDE USING gist (c WITH &&)
);
```

详见CREATE TABLE ... CONSTRAINT ... EXCLUDE。

增加一个排他约束将在约束声明所指定的类型上自动创建索引。

2.5.　系统列

每一个表都拥有一些由系统隐式定义的system columns。因此，这些列的名字不能像用户定义的列一样使用（注意这种限制与名称是否为关键词没有关系，即便用引号限定一个名称也无法绕过这种限制）。　事实上用户不需要关心这些列，只需要知道它们存在即可。

tableoid

包含这一行的表的OID。该列是特别为从继承层次（见第　2.10　节　“继承”）中选择的查询而准备，因为如果没有它将很难知道一行来自于哪个表。*tableoid*可以与ux_class的*oid*列进行连接来获得表的名称。

xmin

插入该行版本的事务身份（事务ID）。一个行版本是一个行的一个特别版本，对一个逻辑行的每一次更新都将创建一个新的行版本。

cmin

插入事务中的命令标识符（从0开始）。

xmax

删除事务的身份（事务ID），对于未删除的行版本为0。对于一个可见的行版本，该列值也可能为非零。这通常表示删除事务还没有提交，或者一个删除尝试被回滚。

cmax

删除事务中的命令标识符，或者为0。

ctid

行版本在其表中的物理位置。注意尽管*ctid*可以被用来非常快速地定位行版本，但是一个行的*ctid*会在被更新或者被**VACUUM FULL**移动时改变。因此，*ctid*不能作为一个长期行标识符。应使用主键来标识逻辑行。

事务标识符也是32位量。在一个历时长久的数据库中事务ID同样会绕回。但如果采取适当的维护过程，这不会是一个致命的问题。但是，长期（超过10亿个事务）依赖事务ID的唯一性是不明智的。

命令标识符也是32位量。这对一个事务中包含的SQL命令设置了一个硬极限： 2^{32}（40亿）。在实践中，该限制并不是问题 — 注意该限制只是针对SQL命令的数目而不是被处理的行数。同样，只有真正 修改了数据库内容的命令才会消耗一个命令标识符。

2.6. 修改表

当我们已经创建了一个表并意识到犯了一个错误或者应用需求发生改变时，我们可以移除表并重新创建它。但如果表中已经被填充数据或者被其他数据库对象引用（例如有一个外键约束），这种做法就显得很不方便。因此，UXDB提供了一族命令来对已有的表进行修改。注意这和修改表中所包含的数据是不同的，这里要做的是对表的定义或者说结构进行修改。

利用这些命令，我们可以：

- 增加列
- 移除列
- 增加约束
- 移除约束
- 修改默认值
- 修改列数据类型
- 重命名列
- 重命名表

所有这些动作都由ALTER TABLE(7)命令执行，其参考页面中包含更详细的信息。

2.6.1. 增加列

要增加一个列，可以使用这样的命令：

ALTER TABLE products ADD COLUMN description text;

新列将被默认值所填充（如果没有指定DEFAULT子句，则会填充空值）。

> **提示**
>
> 从 UXDB 2.1.1.2版本开始，添加一个具有常量默认值的列不再意味着在执行ALTER TABLE 语句时需要更新表的每一行。 相反，默认值将在下次访问该行时返回，并在表被重写时应用，从而使得ALTER TABLE即使在大表上也非常快。
>
> 但是，如果默认值是可变的（例如clock_timestamp()），则每一行需要被ALTER TABLE被执行时计算的值更新。 为避免潜在的长时间的更新操作，特别是如果想要用大多数非默认值填充列，那么最好添加没有默认值的列，再用 UPDATE插入正确的值，然后按照下面所述添加任何期望的默认值。

也可以同时为列定义约束，语法：

ALTER TABLE products ADD COLUMN description text CHECK (description <> '');

事实上CREATE TABLE中关于一列的描述都可以应用在这里。记住不管怎样，默认值必须满足给定的约束，否则ADD将会失败。也可以先将新列正确地填充好，然后再增加约束（见后文）。

2.6.2. 移除列

为了移除一个列，使用如下的命令：

ALTER TABLE products DROP COLUMN description;

列中的数据将会消失。涉及到该列的表约束也会被移除。然而，如果该列被另一个表的外键所引用，UXDB不会安静地移除该约束。我们可以通过增加CASCADE来授权移除任何依赖于被删除列的所有东西：

ALTER TABLE products DROP COLUMN description CASCADE;

关于这个操作背后的一般性机制请见第 2.14 节 “依赖跟踪”。

2.6.3. 增加约束

为了增加一个约束，可以使用表约束的语法，例如：

```
ALTER TABLE products ADD CHECK (name <> '');
ALTER TABLE products ADD CONSTRAINT some_name UNIQUE (product_no);
ALTER TABLE products ADD FOREIGN KEY (product_group_id) REFERENCES product_groups;
```

要增加一个不能写成表约束的非空约束，可使用语法：

ALTER TABLE products ALTER COLUMN product_no SET NOT NULL;

该约束会立即被检查，所以表中的数据必须在约束被增加之前就已经符合约束。

2.6.4. 移除约束

为了移除一个约束首先需要知道它的名称。如果在创建时已经给它指定了名称，那么事情就变得很容易。否则约束的名称是由系统生成的，我们必须先找出这个名称。uxsql的命令\d 表名将会对此有所帮助，其他接口也会提供方法来查看表的细节。因此命令是：

```
ALTER TABLE products DROP CONSTRAINT some_name;
```

（如果处理的是自动生成的约束名称，如$2，别忘了用双引号使它变成一个合法的标识符。）

和移除一个列相似，如果需要移除一个被某些别的东西依赖的约束，也需要加上CASCADE。一个例子是一个外键约束依赖于被引用列上的一个唯一或者主键约束。

这对除了非空约束之外的所有约束类型都一样有效。为了移除一个非空约束可以用：

```
ALTER TABLE products ALTER COLUMN product_no DROP NOT NULL;
```

（回忆一下，非空约束是没有名称的，所以不能用第一种方式。）

2.6.5. 更改列的默认值

要为一个列设置一个新默认值，使用命令：

```
ALTER TABLE products ALTER COLUMN price SET DEFAULT 7.77;
```

注意这不会影响任何表中已经存在的行，它只是为未来的INSERT命令改变了默认值。

要移除任何默认值，使用：

```
ALTER TABLE products ALTER COLUMN price DROP DEFAULT;
```

这等同于将默认值设置为空值。相应的，试图删除一个未被定义的默认值并不会引发错误，因为默认值已经被隐式地设置为空值。

2.6.6. 修改列的数据类型

为了将一个列转换为一种不同的数据类型，使用如下命令：

```
ALTER TABLE products ALTER COLUMN price TYPE numeric(10,2);
```

只有当列中的每一个项都能通过一个隐式类型转换为新的类型时该操作才能成功。如果需要一种更复杂的转换，应该加上一个USING子句来指定应该如何把旧值转换为新值。

UXDB将尝试把列的默认值转换为新类型，其他涉及到该列的任何约束也是一样。但是这些转换可能失败或者产生奇特的结果。因此最好在修改类型之前先删除该列上所有的约束，然后在修改完类型后重新加上相应修改过的约束。

2.6.7. 重命名列

要重命名一个列：

ALTER TABLE products RENAME COLUMN product_no TO product_number;

2.6.8. 重命名表

要重命名一个表：

ALTER TABLE products RENAME TO items;

2.7. 权限

一旦一个对象被创建，它会被分配一个所有者。所有者通常是执行创建语句的角色。对于大部分类型的对象，初始状态下只有所有者（或者超级用户）能够对该对象做任何事情。为了允许其他角色使用它，必须分配权限。

有多种不同的权限：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES、TRIGGER、CREATE、CONNECT、TEMPORARY、EXECUTE及USAGE。可以应用于一个特定对象的权限随着对象的类型（表、函数等）而不同。 有关这些权限含义的更多详细信息请参阅下文。后续的章节将介绍如何使用这些权限。

修改或销毁一个对象的权力通常是只有所有者才有的权限。

一个对象可以通过该对象类型相应的**ALTER**命令来重新分配所有者，例如

ALTER TABLE *table_name* OWNER TO *new_owner*;

超级用户总是可以做到这点，普通角色只有同时是对象的当前所有者（或者是拥有角色的一个成员）以及新拥有角色的一个成员时才能做同样的事。

要分配权限，可以使用GRANT(7)命令。例如，如果joe是一个已有角色，而accounts是一个已有表，更新该表的权限可以按如下方式授权：

GRANT UPDATE ON accounts TO joe;

用ALL取代特定权限会把与对象类型相关的所有权限全部授权。

一个特殊的名为PUBLIC的“角色”可以用来向系统中的每一个角色授予一个权限。同时，在数据库中有很多用户时可以设置“组”角色来帮助管理权限。

为了撤销一个权限，使用REVOKE(7) 命令：

REVOKE ALL ON accounts FROM PUBLIC;

对象拥有者的特殊权限（即执行**DROP**、**GRANT**、**REVOKE**等的权力）总是隐式地属于拥有者，并且不能被授予或撤销。但是对象拥有者可以选择撤销他们自己的普通权限，例如把一个表变得对他们自己和其他人只读。

一般情况下，只有对象拥有者（或者超级用户）可以授予或撤销一个对象上的权限。但是可以在授予权限时使用“with grant option”来允许接收人将权限转授给其他人。如果后来授予选项被撤销，则所有从接收人那里获得的权限（直接或者通过授权链获得）都将被撤销。更多详情请见GRANT(7)和REVOKE(7)参考页。

The available privileges are:

SELECT

允许 SELECT(7) 从任何列、或特定的列、表、视图、物化视图、或其他类似表格的对象。也允许使用 COPY(7) TO. 还需要这个权限来引用UPDATE(7) 或 DELETE(7)中现有的列值。对于序列，这个权限还允许使用currval 函数。对于大对象，此权限允许读取对象。

INSERT

允许将新行的 INSERT(7) 加入表、视图等等。 可以在特定列上授予，在这种情况下INSERT命令中只有那些列可以被分配（其他列将因此而收到默认值）。 还允许使用COPY(7) FROM。

UPDATE

允许 UPDATE(7) 更新任何列、或指定列、表、视图等等。（实际上，任何有效的UPDATE命令也需要SELECT权限，因为它必须引用表列来确定要更新的行，和/或计算列的新值。）SELECT ... FOR UPDATE和SELECT ... FOR SHARE除了SELECT权限外，还需要至少一列上的这个权限。 对于序列，这个权限允许使用 nextval 和 setval 函数。对于大对象，此权限允许写入或截断对象。

DELETE

允许 DELETE(7) 从表、视图等等中删除行. （实际上，任何有效的DELETE命令也需要SELECT权限，因为它必须引用表列来确定要删除的行。）

TRUNCATE

允许在表、视图等等上 TRUNCATE(7) 。

REFERENCES

允许创建引用表或表的特定列的外键约束。

TRIGGER

允许在表、视图等等上创建触发器。

CREATE

对于数据库，允许在数据库中创建新的模式和发布。

对于模式，允许在模式中创建新对象。要重命名现有对象，必须拥有对象 and所包含模式的此权限。

对于表空间，允许在表空间中创建表、索引和临时文件，并允许创建将表空间作为默认表空间的数据库。（注意，撤销此特权不会更改已有对象的位置。）

CONNECT

允许受让者连接到数据库。此权限在连接启动时进行检查(加之ux_hba.conf施加的任何约束).

TEMPORARY

允许在使用数据库时创建临时表。

EXECUTE

允许调用函数或过程，包括使用在函数之上实现的任何运算符。这是适用于函数和过程的唯一权限类型。

USAGE

对于程序语言，允许使用语言来创建该语言的函数。 这是适用于过程语言的唯一权限类型。

对于模式，允许访问模式中包含的对象（假设对象自己的权限要求也已得到满足）。 从本质上讲，这允许受让者“look up”模式中的对象。如果没有此权限，仍可以看到对象名称，例如通过查询系统目录。 此外，在撤消此权限后，现有会话可能还具有以前执行过此查找的语句，因此这不是阻止对象访问的彻底安全的方法。

对于序列，允许使用currval 和 nextval 函数.

对于类型和域，允许在创建表、函数和其他模式对象时使用类型或域。 （注意，此权限不控制类型的全部 “usage” ，例如查询中出现的类型的值。 它仅防止创建依赖于类型的对象。 此权限的主要目的是控制哪些用户可以对类型创建依赖项，这可能会防止所有者以后更改类型。 ）

对于外部数据包装器，允许使用外部数据包装器创建新服务器。

对于外部服务器，允许使用服务器创建外部表。受让者还可以创建、更改或删除与该服务器关联的自己的用户映射。

其他命令所需的权限罗列在相应命令的参考页上。

在创建对象时，UXDB默认将某些类型对象的权限授予PUBLIC。 默认情况下，在表、表列、序列、外部数据包装器、外部服务器、大型对象、模式或表空间上，不向PUBLIC授予权限。 对于其他类型的对象，授予 PUBLIC的默认权限如下所示： 针对数据库的CONNECT和TEMPORARY（创建临时表）权限; 针对函数和程序的EXECUTE权限;以及针对语言和数据类型（包括域）的USAGE权限。 当然，对象所有者可以REVOKE默认权限和特别授予的权限。 （为了最大程度的安全性，在创建对象的同一事务中发出REVOKE;那么就没有其他用户能够使用该对象的窗口。） 此外，可以使用ALTER DEFAULT PRIVILEGES(7)命令取代这些默认权限设置。

表 2.1 “ACL 权限缩写”显示了ACL（访问控制列表）值中用于这些权限类型的单字母缩写。 将在下面列出的psql命令的输出中，或者在查看系统目录的 ACL 列时看到这些字母。

表 2.1. ACL 权限缩写

权限	缩写	适用对象类型
SELECT	r（“读”）	LARGE OBJECT, SEQUENCE, TABLE (and table-like objects), table column
INSERT	a（“增补”）	TABLE, table column
UPDATE	w（“写”）	LARGE OBJECT, SEQUENCE, TABLE, table column
DELETE	d	TABLE
TRUNCATE	D	TABLE
REFERENCES	x	TABLE, table column
TRIGGER	t	TABLE

权限	缩写	适用对象类型
CREATE	C	DATABASE, SCHEMA, TABLESPACE
CONNECT	c	DATABASE
TEMPORARY	T	DATABASE
EXECUTE	X	FUNCTION, PROCEDURE
USAGE	U	DOMAIN, FOREIGN DATA WRAPPER, FOREIGN SERVER, LANGUAGE, SCHEMA, SEQUENCE, TYPE

表 2.2 “访问权限摘要”使用上面所示的缩写总结了每种类型 SQL 对象可用的权限. 它还显示可用于检查每种对象类型的特权设置的 uxsql 命令。

表 2.2. 访问权限摘要

对象类型	所有权限	默认 **PUBLIC** 权限	uxsql 命令
DATABASE	CTc	Tc	\l
DOMAIN	U	U	\dD+
FUNCTION or PROCEDURE	X	X	\df+
FOREIGN DATA WRAPPER	U	none	\dew+
FOREIGN SERVER	U	none	\des+
LANGUAGE	U	U	\dL+
LARGE OBJECT	rw	none	
SCHEMA	UC	none	\dn+
SEQUENCE	rwU	none	\dp
TABLE (and table-like objects)	arwdDxt	none	\dp
Table column	arwx	none	\dp
TABLESPACE	C	none	\db+
TYPE	U	U	\dT+

已授予特定对象的权限显示为aclitem项的列表，其中每个aclitem项描述了特定授予者授予给一个被授与者的权限。 例如，calvin=r*w/hobbes 指明角色calvin具有SELECT（r）权限和授予选项（*）以及不可授予权限UPDATE（w），均由角色hobbes授予。 如果calvin对由其他授予人授予的同一对象也具有一些权限，那将显示为单独的aclitem条目。 aclitem 中的空受赠方字段代表PUBLIC。

例如，假设用户miriam创建了表mytable并且：

```
GRANT SELECT ON mytable TO PUBLIC;
GRANT SELECT, UPDATE, INSERT ON mytable TO admin;
GRANT SELECT (col1), UPDATE (col1) ON mytable TO miriam_rw;
```

则 uxsql的 \dp 命令将显示:

```
=> \dp mytable
                              Access privileges
 Schema | Name  | Type |   Access privileges   |  Column privileges  | Policies
--------+-------+------+-----------------------+---------------------+----------
 public | mytable | table | miriam=arwdDxt/miriam+| col1:               +|
        |       |       | =r/miriam             +|  miriam_rw=rw/miriam |
        |       |       | admin=arw/miriam      |                     |
(1 row)
```

如果“Access privileges”列对于给定对象为空，则表示该对象具有默认权限（也就是说，它在相关系统目录中的权限条目为空）。 默认权限始终包含所有者的所有权限，并且可以包括PUBLIC 的一些权限，具体取决于对象类型，如上所述。 对象上的第一个GRANT或REVOKE将实例化默认权限（例如，生成miriam_arwdDxt/miriam），然后根据指定的请求修改它们。 类似的，只有具有非默认特权的列的条目才显示在“Column privileges”中。（注意：为此目的，“default privileges”始终表示对象类型的内置缺省权限。其权限受ALTER DEFAULT PRIVILEGES 命令影响的对象将始终显示一个显式权限条目，其中包含 ALTER。）

注意所有者的隐式授予选项没有在访问权限显示中标记。仅当授予选项被显式授予给某人时才会出现*。

2.8. 行安全性策略

除可以通过GRANT(7)使用 SQL 标准的 特权系统之外，表还可以具有 行安全性策略，它针对每一个用户限制哪些行可以 被普通的查询返回或者可以被数据修改命令插入、更新或删除。这种 特性也被称为行级安全性。默认情况下，表不具有 任何策略，这样用户根据 SQL 特权系统具有对表的访问特权，对于 查询或更新来说其中所有的行都是平等的。

当在一个表上启用行安全性时（使用 ALTER TABLE ... ENABLE ROW LEVEL SECURITY），所有对该表选择行或者修改行的普通访问都必须被一条 行安全性策略所允许（不过，表的拥有者通常不服从行安全性策略）。如果 表上不存在策略，将使用一条默认的否定策略，即所有的行都不可见或者不能 被修改。应用在整个表上的操作不服从行安全性，例如TRUNCATE和 REFERENCES。

行安全性策略可以针对特定的命令、角色或者两者。一条策略可以被指定为 适用于ALL命令，或者SELECT、 INSERT、UPDATE、或者DELETE。 可以为一条给定策略分配多个角色，并且通常的角色成员关系和继承规则也适用。

要指定哪些行根据一条策略是可见的或者是可修改的，需要一个返回布尔结果 的表达式。对于每一行，在计算任何来自用户查询的条件或函数之前，先会计 算这个表达式（这条规则的唯一例外是leakproof函数， 它们被保证不会泄露信息，优化器可能会选择在行安全性检查之前应用这类 函数）。使该表达式不返回true的行将不会被处理。可以指定 独立的表达式来单独控制哪些行可见以及哪些行被允许修改。策略表达式会作 为查询的一部分运行并且带有运行该查询的用户的特权，但是安全性定义者函数 可以被用来访问对调用用户不可用的数据。

具有BYPASSRLS属性的超级用户和角色在访问一个表时总是 可以绕过行安全性系统。表拥有者通常也能绕过行安全性，不过表拥有者 可以选择用ALTER TABLE ... FORCE ROW LEVEL SECURITY来服从行安全性。

启用和禁用行安全性以及向表增加策略是只有表拥有者具有的特权。

策略的创建可以使用CREATE POLICY(7)命令，策略的修改 可以使用ALTER POLICY(7)命令，而策略的删除可以使用 DROP POLICY(7)命令。要为一个给定表启用或者禁用行 安全性，可以使用ALTER TABLE(7)命令。

每一条策略都有名称并且可以为一个表定义多条策略。由于策略是表相 关的，一个表的每一条策略都必须有一个唯一的名称。不同的表可以拥有 相同名称的策略。

当多条策略适用于一个给定的查询时，会把它们用OR（对宽容性策略，默认的策略类型）或者AND（对限制性策略）组合在一起。这和给定角色拥有它作为成员的所有角色的特权的规则类似。宽容性策略和限制性策略在下文将会进一步讨论。

作为一个简单的例子，这里是如何在account关系上 创建一条策略以允许只有managers角色的成员能访问行， 并且只能访问它们账户的行：

```
CREATE TABLE accounts (manager text, company text, contact_email text);

ALTER TABLE accounts ENABLE ROW LEVEL SECURITY;

CREATE POLICY account_managers ON accounts TO managers
   USING (manager = current_user);
```

上面的策略隐含地提供了一个与其该约束适用于被一个命令选择的行（这样一个经理不能SELECT、UPDATE或者DELETE属于其他经理的已有行）以及被一个命令修改的行（这样属于其他经理的行不能通过INSERT或者UPDATE创建）。

如果没有指定角色或者使用了特殊的用户名PUBLIC， 则该策略适用于系统上所有的用户。要允许所有用户访问users 表中属于他们自己的行，可以使用一条简单的策略：

```
CREATE POLICY user_policy ON users
   USING (user_name = current_user);
```

这个例子的效果和前一个类似。

为了对增加到表中的行使用与可见行不同的策略，可以组合多条策略。这一对策略将允许所有用户查看users表中的所有行，但只能修改他们自己的行：

```
CREATE POLICY user_sel_policy ON users
   FOR SELECT
   USING (true);
CREATE POLICY user_mod_policy ON users
   USING (user_name = current_user);
```

在一个SELECT命令中，这两条规则被用OR组合在一起，最终的效应就是所有的行都能被选择。在其他命令类型中，只有第二条策略适用，这样其效果就和以前相同。

也可以用ALTER TABLE命令禁用行安全性。禁用行安全性 不会移除定义在表上的任何策略，它们只是被简单地忽略。然后该表中的所有 行都是可见的并且可修改，服从于标准的 SQL 特权系统。

下面是一个较大的例子，它展示了这种特性如何被用于生产环境。表 passwd模拟了一个 Unix 口令文件：

```
-- 简单的口令文件例子
CREATE TABLE passwd (
  user_name          text UNIQUE NOT NULL,
  pwhash             text,
```

```
  uid               int  PRIMARY KEY,
  gid               int  NOT NULL,
  real_name           text NOT NULL,
  home_phone          text,
  extra_info        text,
  home_dir            text NOT NULL,
  shell             text NOT NULL
);

CREATE ROLE admin;  -- 管理员
CREATE ROLE bob;    -- 普通用户
CREATE ROLE alice;  -- 普通用户

-- 填充表
INSERT INTO passwd VALUES
  ('admin','xxx',0,0,'Admin','111-222-3333',null,'/root','/bin/dash');
INSERT INTO passwd VALUES
  ('bob','xxx',1,1,'Bob','123-456-7890',null,'/home/bob','/bin/zsh');
INSERT INTO passwd VALUES
  ('alice','xxx',2,1,'Alice','098-765-4321',null,'/home/alice','/bin/zsh');

-- 确保在表上启用行级安全性
ALTER TABLE passwd ENABLE ROW LEVEL SECURITY;

-- 创建策略
-- 管理员能看见所有行并且增加任意行
CREATE POLICY admin_all ON passwd TO admin USING (true) WITH CHECK (true);
-- 普通用户可以看见所有行
CREATE POLICY all_view ON passwd FOR SELECT USING (true);
-- 普通用户可以更新它们自己的记录，但是限制普通用户可用的 shell
CREATE POLICY user_mod ON passwd FOR UPDATE
  USING (current_user = user_name)
  WITH CHECK (
    current_user = user_name AND
    shell IN ('/bin/bash','/bin/sh','/bin/dash','/bin/zsh','/bin/tcsh')
  );

-- 允许管理员有所有普通权限
GRANT SELECT, INSERT, UPDATE, DELETE ON passwd TO admin;
-- 用户只在公共列上得到选择访问
GRANT SELECT
  (user_name, uid, gid, real_name, home_phone, extra_info, home_dir, shell)
  ON passwd TO public;
-- 允许用户更新特定行
GRANT UPDATE
  (pwhash, real_name, home_phone, extra_info, shell)
  ON passwd TO public;
```

对于任意安全性设置来说，重要的是测试并确保系统的行为符合预期。 使用上述的例子，下面展示了权限系统工作正确：

```
-- admin 可以看到所有的行和域
uxdb=> set role admin;
```

```
SET
uxdb=> table passwd;
 user_name | pwhash | uid | gid | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+--------+-----+-----+-----------+--------------+------------+-------------+-----------
 admin     | xxx    |   0 |   0 | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | xxx    |   1 |   1 | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | xxx    |   2 |   1 | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

-- 测试 Alice 能做什么
uxdb=> set role alice;
SET
uxdb=> table passwd;
ERROR:  permission denied for relation passwd
uxdb=> select user_name,real_name,home_phone,extra_info,home_dir,shell from passwd;
 user_name | real_name |  home_phone  | extra_info | home_dir    |   shell
-----------+-----------+--------------+------------+-------------+-----------
 admin     | Admin     | 111-222-3333 |            | /root       | /bin/dash
 bob       | Bob       | 123-456-7890 |            | /home/bob   | /bin/zsh
 alice     | Alice     | 098-765-4321 |            | /home/alice | /bin/zsh
(3 rows)

uxdb=> update passwd set user_name = 'joe';
ERROR:  permission denied for relation passwd
-- Alice 被允许更改她自己的 real_name，但不能改其他的
uxdb=> update passwd set real_name = 'Alice Doe';
UPDATE 1
uxdb=> update passwd set real_name = 'John Doe' where user_name = 'admin';
UPDATE 0
uxdb=> update passwd set shell = '/bin/xx';
ERROR:  new row violates WITH CHECK OPTION for "passwd"
uxdb=> delete from passwd;
ERROR:  permission denied for relation passwd
uxdb=> insert into passwd (user_name) values ('xxx');
ERROR:  permission denied for relation passwd
-- Alice 可以更改她自己的口令；行级安全性会悄悄地阻止更新其他行
uxdb=> update passwd set pwhash = 'abc';
UPDATE 1
```

目前为止所有构建的策略都是宽容性策略，也就是当多条策略都适用时会被适用“OR”布尔操作符组合在一起。而宽容性策略可以被用来仅允许在预计情况中对行的访问，这比将宽容性策略与限制性策略（记录必须通过这类策略并且它们会被“AND”布尔操作符组合起来）组合在一起更简单。在上面的例子之上，我们增加一条限制性策略要求通过一个本地Unix套接字连接过来的管理员访问passwd表的记录：

```
CREATE POLICY admin_local_only ON passwd AS RESTRICTIVE TO admin
    USING (ux_catalog.inet_client_addr() IS NULL);
```

然后，由于这条限制性规则的存在，我们可以看到从网络连接进来的管理员将无法看到任何记录：

```
=> SELECT current_user;
```

```
 current_user
--------------
 admin
(1 row)

=> select inet_client_addr();
 inet_client_addr
------------------
 127.0.0.1
(1 row)

=> SELECT current_user;
 current_user
--------------
 admin
(1 row)

=> TABLE passwd;
 user_name | pwhash | uid | gid | real_name | home_phone | extra_info | home_dir | shell
-----------+--------+-----+-----+-----------+------------+------------+----------+-------
(0 rows)

=> UPDATE passwd set pwhash = NULL;
UPDATE 0
```

参照完整性检查（例如唯一或逐渐约束和外键引用）总是会绕过行级安全性以保证数据完整性得到维护。在开发模式和行级安全性时必须避免“隐通道”通过这类参照完整性检查泄露信息。

在某些环境中确保行安全性没有被应用很重要。例如，在做备份时，如果　行安全性悄悄地导致某些行被从备份中忽略掉，这会是灾难性的。在这类　情况下，可以设置row_security配置参数为 off。这本身不会绕过行安全性，它所做的是如果任何结果会 被一条策略过滤掉，就会抛出一个错误。然后错误的原因就可以被找到并且 修复。

在上面的例子中，策略表达式只考虑了要被访问的行中的当前值。这是最简单并且表现最好的情况。如果可能，最好设计行安全性应用以这种方式工作。如果需要参考其他行或者其他表来做出策略的决定，可以在策略表达式中通过使用子-SELECT或者包含SELECT的函数来实现。不过要注意这类访问可能会导致竞争条件，在不注意的情况下这可能会导致信息泄露。作为一个例子，考虑下面的表设计：

```
-- 特权组的定义
CREATE TABLE groups (group_id int PRIMARY KEY,
                     group_name text NOT NULL);

INSERT INTO groups VALUES
  (1, 'low'),
  (2, 'medium'),
  (5, 'high');

GRANT ALL ON groups TO alice;  -- alice 是管理员
GRANT SELECT ON groups TO public;

-- 用户的特权级别的定义
CREATE TABLE users (user_name text PRIMARY KEY,
                    group_id int NOT NULL REFERENCES groups);
```

```
INSERT INTO users VALUES
  ('alice', 5),
  ('bob', 2),
  ('mallory', 2);

GRANT ALL ON users TO alice;
GRANT SELECT ON users TO public;

-- 保存要被保护的信息的表
CREATE TABLE information (info text,
                          group_id int NOT NULL REFERENCES groups);

INSERT INTO information VALUES
  ('barely secret', 1),
  ('slightly secret', 2),
  ('very secret', 5);

ALTER TABLE information ENABLE ROW LEVEL SECURITY;

-- 对于安全性 group_id 大于等于一行的 group_id 的用户，
-- 这一行应该是可见的/可更新的
CREATE POLICY fp_s ON information FOR SELECT
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));
CREATE POLICY fp_u ON information FOR UPDATE
  USING (group_id <= (SELECT group_id FROM users WHERE user_name = current_user));

-- 我们只依赖于行级安全性来保护信息表
GRANT ALL ON information TO public;
```

现在假设alice希望更改“有一点点秘密” 的信息，但是觉得mallory不应该看到该行中的新 内容，因此她这样做：

```
BEGIN;
UPDATE users SET group_id = 1 WHERE user_name = 'mallory';
UPDATE information SET info = 'secret from mallory' WHERE group_id = 2;
COMMIT;
```

这看起来是安全的，没有窗口可供mallory看到 “对 mallory 保密”的字符串。不过，这里有一种 竞争条件。如果mallory正在并行地做：

```
SELECT * FROM information WHERE group_id = 2 FOR UPDATE;
```

并且她的事务处于READ COMMITTED模式，她就可能看到 “s对 mallory 保密”的东西。如果她的事务在alice 做完之后就到达信息行，这就会发生。它会阻塞等待 alice的事务提交，然后拜FOR UPDATE子句所赐 取得更新后的行内容。不过，对于来自users的隐式 SELECT，它不会取得一个已更新的行， 因为子-SELECT没有FOR UPDATE，相反 会使用查询开始时取得的快照读取users行。因此， 策略表达式会测试mallory的特权级别的旧值并且允许她看到 被更新的行。

有多种方法能解决这个问题。一种简单的答案是在行安全性策略中的 子-SELECT里使用SELECT ... FOR SHARE。 不过，这要求在被引用表（这里是users）上授予 UPDATE特权给

受影响的用户，这可能不是我们想要的（ 但是另一条行安全性策略可能被应用来阻止它们实际使用这个特权，或者 子-SELECT可能被嵌入到一个安全性定义者函数中）。 还有，在被引用的表上过多并发地使用行共享锁可能会导致性能问题， 特别是表更新比较频繁时。另一种解决方案（如果被引用表上的更新 不频繁就可行）是在更新被引用表时对它取一个排他锁，这样就没有 并发事务能够检查旧的行值了。或者我们可以在提交对被引用表的更新 之后、在做依赖于新安全性情况的更改之前等待所有并发事务结束。

更多细节请见CREATE POLICY(7) 和ALTER TABLE(7)。

2.9. 模式

一个UXDB数据库集簇中包含一个或更多命名的数据库。用户和用户组被整个集簇共享，但没有其他数据在数据库之间共享。任何给定客户端连接只能访问在连接中指定的数据库中的数据。

> **注意**
>
> 一个集簇的用户并不必拥有访问集簇中每一个数据库的权限。用户名的共享意味着不可能在同一个集簇中出现重名的不同用户，例如两个数据库中都有叫joe的用户。但系统可以被配置为只允许joe访问某些数据库。

一个数据库包含一个或多个命名模式，模式中包含着表。模式还包含其他类型的命名对象，包括数据类型、函数和操作符。相同的对象名称可以被用于不同的模式中二不会出现冲突，例如schema1和myschema都可以包含名为mytable的表。和数据库不同，模式并不是被严格地隔离：一个用户可以访问他们所连接的数据库中的所有模式内的对象，只要他们有足够的权限。

下面是一些使用方案的原因：

- 允许多个用户使用一个数据库并且不会互相干扰。
- 将数据库对象组织成逻辑组以便更容易管理。
- 第三方应用的对象可以放在独立的模式中，这样它们就不会与其他对象的名称发生冲突。

模式类似于操作系统层的目录，但是模式不能嵌套。

2.9.1. 创建模式

要创建一个模式，可使用CREATE SCHEMA(7)命令，并且给出选择的模式名称。例如：

```
CREATE SCHEMA myschema;
```

在一个模式中创建或访问对象，需要使用由模式名和表名构成的限定名，模式名和表名之间以点号分隔：

schema.table

在任何需要一个表名的地方都可以这样用，包括表修改命令和后续章节要讨论的数据访问命令（为了简洁我们在这里只谈到表，但是这种方式对其他类型的命名对象同样有效，例如类型和函数）。

事实上，还有更加通用的语法：

database.schema.table

也可以使用，但是目前它只是在形式上与SQL标准兼容。如果我们写一个数据库名称，它必须是我们正在连接的数据库。

因此，如果要在一个新模式中创建一个表，可用：

```
CREATE TABLE myschema.mytable (
 ...
);
```

要删除一个为空的模式（其中的所有对象已经被删除），可用：

```
DROP SCHEMA myschema;
```

要删除一个模式以及其中包含的所有对象，可用：

```
DROP SCHEMA myschema CASCADE;
```

有关于此的更一般的机制请参见第 2.14 节 “依赖跟踪”。

我们常常希望创建一个由其他人所拥有的模式（因为这是将用户动作限制在良定义的名字空间中的方法之一）。其语法是：

```
CREATE SCHEMA schema_name AUTHORIZATION user_name;
```

我们甚至可以省略模式名称，在此种情况下模式名称将会使用用户名，参见第 2.9.6 节 “使用模式”。

以ux_开头的模式名被保留用于系统目的，所以不能被用户所创建。

2.9.2. 公共模式

在前面的小节中，我们创建的表都没有指定任何模式名称。默认情况下这些表（以及其他对象）会自动的被放入一个名为“public”的模式中。任何新数据库都包含这样一个模式。因此，下面的命令是等效的：

```
CREATE TABLE products ( ... );
```

以及：

```
CREATE TABLE public.products ( ... );
```

2.9.3. 模式搜索路径

限定名写起来很冗长，通常最好不要把一个特定模式名拉到应用中。因此，表名通常被使用非限定名来引用，它只由表名构成。系统将沿着一条搜索路径来决定该名称指的是哪个表，搜索路径

是一个进行查看的模式列表。　搜索路径中第一个匹配的表将被认为是所需要的。如果在搜索路径中没有任何匹配，即使在数据库的其他模式中存在匹配的表名也将会报告一个错误。

在不同方案中创建命名相同的对象的能力使得编写每次都准确引用相同对象的查询变得复杂。这也使得用户有可能更改其他用户查询的行为，不管是出于恶意还是无意。由于未经限定的名称在查询中以及在UXDB内部的广泛使用，在search_path中增加一个方案实际上是信任所有在该方案中具有CREATE特权的用户。在运行一个普通查询时，恶意用户可以在搜索路径中的以方案中创建能够夺取控制权并且执行任意SQL函数的对象，而这些事情就像是在执行一样。

搜索路径中的第一个模式被称为当前模式。除了是第一个被搜索的模式外，如果CREATE TABLE命令没有指定模式名，它将是新创建表所在的模式。

要显示当前搜索路径，使用下面的命令：

```
SHOW search_path;
```

在默认设置下这将返回：

```
 search_path
-------------
 "$user", public
```

第一个元素说明一个和当前用户同名的模式会被搜索。如果不存在这个模式，该项将被忽略。第二个元素指向我们已经见过的公共模式。

搜索路径中的第一个模式是创建新对象的默认存储位置。这就是默认情况下对象会被创建在公共模式中的原因。当对象在任何其他没有模式限定的环境中被引用（表修改、数据修改或查询命令）时，搜索路径将被遍历直到一个匹配对象被找到。因此，在默认配置中，任何非限定访问将只能指向公共模式。

要把新模式放在搜索路径中，我们可以使用：

```
SET search_path TO myschema,public;
```

（我们在这里省略了$user，因为我们并不立即需要它）。然后我们可以删除该表而无需使用方案进行限定：

```
DROP TABLE mytable;
```

同样，由于myschema是路径中的第一个元素，新对象会被默认创建在其中。

我们也可以这样写：

```
SET search_path TO myschema;
```

这样我们在没有显式限定时再也不必去访问公共模式了。公共模式没有什么特别之处，它只是默认存在而已，它也可以被删除。

其他操作模式搜索路径的方法请见第　6.25　节　“系统信息函数和运算符”。

搜索路径对于数据类型名称、函数名称和操作符名称的作用与表名一样。数据类型和函数名称可以使用和表名完全相同的限定方式。如果我们需要在一个表达式中写一个限定的操作符名称，我们必须写成一种特殊的形式：

OPERATOR(*schema.operator*)

这是为了避免句法歧义。例如：

```
SELECT 3 OPERATOR(ux_catalog.+) 4;
```

实际上我们通常都会依赖于搜索路径来查找操作符，因此没有必要去写如此“丑陋”的东西。

2.9.4. 模式和权限

默认情况下，用户不能访问不属于他们的方案中的任何对象。要允许这种行为，模式的拥有者必须在该模式上授予USAGE权限。为了允许用户使用方案中的对象，可能还需要根据对象授予额外的权限。

一个用户也可以被允许在其他某人的模式中创建对象。要允许这种行为，模式上的CREATE权限必须被授予。注意在默认情况下，所有人都拥有在public模式上的CREATE和USAGE权限。这使得用户能够连接到一个给定数据库并在它的public模式中创建对象。回收这一特权的使用模式调用：

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

（第一个“public”是方案，第二个“public”指的是“每一个用户”。第一种是一个标识符，第二种是一个关键词，所以两者的大小写不同。请回想第 1.1.1 节 “标识符和关键词”中的指导方针。）

2.9.5. 系统目录模式

除public和用户创建的模式之外，每一个数据库还包括一个ux_catalog模式，它包含了系统表和所有内建的数据类型、函数以及操作符。ux_catalog总是搜索路径的一个有效部分。如果没有在路径中显式地包括该模式，它将在路径中的模式之前被搜索。这保证了内建的名称总是能被找到。然而，如果我们希望用用户定义的名称重载内建的名称，可以显式的将ux_catalog放在搜索路径的末尾。

由于系统表名称以ux_开头，最好还是避免使用这样的名称，以避免和未来新版本中 可能出现的系统表名发生冲突。系统表将继续采用以ux_开头的方式，这样它们不会 与非限制的用户表名称冲突。

2.9.6. 使用模式

模式能够以多种方式组织数据.`secure schema usage pattern`防止不受信任的用户更改其他用户查询的行为。 当数据库不使用安全模式使用方式时，希望安全地查询该数据库的用户将在每个会话开始时采取保护操作。 具体的说，他们将通过设置search_path到空字符串或在其它情况下从search_path中删除非超级用户可写的模式来开始每个会话。 默认配置可以很容易的支持一些使用模式。

- 将普通用户约束在其私有的方案中。要实现这一点，发出REVOKE CREATE ON SCHEMA public FROM PUBLIC，并且为每一个用户创建一个用其用户名命名的方案。 回想一下以$user开头的默认搜索路径，该路径解析为用户名。 因此，如果每个用户都有单独的模式，则默认情况下他们访问自己的模式。 在不受信任的用户已经登录的数据库中采用此模式后，请考虑审计名字类似于模式ux_catalog中的对象的公共模式。 此方式是一种安全模式的使用方

式，除非不受信任的用户是数据库所有者或拥有CREATEROLE权限，在这种情况下没有安全模式使用方式存在。

- 从默认搜索路径中删除公共模式，通过修改uxsinodb.conf或通过发出ALTER ROLE ALL SET search_path ="$user"。 每一个都保留在公共模式中创建对象的能力，但是只有符合资格的名称才会选择这些对象。 虽然符合资格的表引用是可以的，但是要调用公共模式中的函数will be unsafe or unreliable。 如果在公共模式中创建函数或扩展，请改用第一个方式。 否则，与第一个模式一样，这是安全的，除非不受信任的用户是数据库所有者或拥有CREATEROLE权限。

- 保持默认。所有用户都隐式地访问公共模式。这模拟了方案根本不可用的情况，可以用于从无模式感知的世界平滑过渡。 但是，这绝不是一个安全的模式。只有当数据库仅有单个用户或者少数相互信任的用户时，才可以接受。

对于任何一种模式，为了安装共享的应用（所有人都要用其中的表，第三方提供的额外函数，等等），可把它们放在单独的方案中。记住授予适当的特权以允许其他用户访问它们。然后用户可以通过以方案名限定名称的方式来引用这些额外的对象，或者他们可以把额外的方案放在自己的搜索路径中。

2.9.7. 可移植性

在SQL标准中，在由不同用户拥有的同一个模式中的对象是不存在的。此外，某些实现不允许创建与拥有者名称不同名的模式。事实上，在那些仅实现了标准中基本模式支持的数据库中，模式和用户的概念是等同的。因此，很多用户认为限定名称实际上是由*user_name.table_name*组成的。如果我们为每一个用户都创建了一个模式，UXDB实际也是这样认为的。

同样，在SQL标准中也没有public模式的概念。为了最大限度的与标准一致，我们不应使用（甚至是删除）public模式。

当然，某些SQL数据库系统可能根本没有实现方案，或者提供允许跨数据库访问的名字空间。如果需要使用这样一些系统，最好不要使用方案。

2.10. 继承

UXDB实现了表继承，这对数据库设计者来说是一种有用的工具（SQL:1999及其后的版本定义了一种类型继承特性，但和这里介绍的继承有很大的不同）。

让我们从一个例子开始：假设我们要为城市建立一个数据模型。每一个州有很多城市，但是只有一个首府。我们希望能够快速地检索任何特定州的首府城市。这可以通过创建两个表来实现：一个用于州首府，另一个用于不是首府的城市。然而，当我们想要查看一个城市的数据（不管它是不是一个首府）时会发生什么？继承特性将有助于解决这个问题。我们可以将capitals表定义为继承自cities表：

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- in feet
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

在这种情况下，capitals表继承了它的父表cities的所有列。州首府还有一个额外的列*state*用来表示它所属的州。

在UXDB中，一个表可以从0个或者多个其他表继承，而对一个表的查询则可以引用一个表的所有行或者该表的所有行加上它所有的后代表。默认情况是后一种行为。例如，下面的查询将查找所有海拔高于500尺的城市的名称，包括州首府：

```
SELECT name, altitude
  FROM cities
  WHERE altitude > 500;
```

对于来自UXDB教程的例子数据，它将返回：

```
  name    | altitude
----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

在另一方面，下面的查询将找到海拔超过500尺且不是州首府的所有城市：

```
SELECT name, altitude
  FROM ONLY cities
  WHERE altitude > 500;

  name    | altitude
----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

这里的ONLY关键词指示查询只被应用于cities上，而其他在继承层次中位于cities之下的其他表都不会被该查询涉及。很多我们已经讨论过的命令（如SELECT、UPDATE和DELETE）都支持ONLY关键词。

我们也可以在表名后写上一个*来显式地将后代表包括在查询范围内：

```
SELECT name, altitude
  FROM cities*
  WHERE altitude > 500;
```

写*不是必需的，因为这种行为总是默认的。

在某些情况下，我们可能希望知道一个特定行来自于哪个表。每个表中的系统列*tableoid*可以告诉我们行来自于哪个表：

```
SELECT c.tableoid, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

将会返回：

```
 tableoid |   name    | altitude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

（如果重新生成这个结果，可能会得到不同的OID数字。）通过与ux_class进行连接可以看到实际的表名：

```
SELECT p.relname, c.name, c.altitude
FROM cities c, ux_class p
WHERE c.altitude > 500 AND c.tableoid = p.oid;
```

将会返回：

```
 relname |   name    | altitude
---------+-----------+----------
 cities  | Las Vegas |     2174
 cities  | Mariposa  |     1953
 capitals| Madison   |      845
```

另一种得到同样效果的方法是使用regclass别名类型， 它将象征性地打印出表的 OID：

```
SELECT c.tableoid::regclass, c.name, c.altitude
FROM cities c
WHERE c.altitude > 500;
```

继承不会自动地将来自INSERT或COPY命令的数据传播到继承层次中的其他表中。在我们的例子中，下面的INSERT语句将会失败：

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('Albany', NULL, NULL, 'NY');
```

我们也许希望数据能被以某种方式被引入到capitals表中，但是这不会发生：INSERT总是向指定的表中插入。在某些情况下，可以通过使用一个规则来将插入动作重定向。但是这对上面的情况并没有帮助，因为cities表根本就不包含*state*列，因而这个命令将在触发规则之前就被拒绝。

父表上的所有检查约束和非空约束都将自动被它的后代所继承，除非显式地指定了NO INHERIT子句。其他类型的约束（唯一、主键和外键约束）则不会被继承。

一个表可以从超过一个的父表继承，在这种情况下它拥有父表们所定义的列的并集。任何定义在子表上的列也会被加入到其中。如果在这个集合中出现重名列，那么这些列将被“合并”，这样在子表中只会有一个这样的列。重名列能被合并的前提是这些列必须具有相同的数据类型，否则会导致错误。可继承的检查约束和非空约束会以类似的方式被合并。例如，如果合并成一个合并列的任一列定义被标记为非空，则该合并列会被标记为非空。如果检查约束的名称相同，则他们会被合并，但如果它们的条件不同则合并会失败。

表继承通常是在子表被创建时建立，使用CREATE TABLE(7)语句的INHERITS子句。一个已经被创建的表也可以另外一种方式增加一个新的父亲关系，使用ALTER TABLE(7)的INHERIT变体。要这样做，新的子表必须已经包括和父表相同名称和数据类型的列。子表还必须包括和父表相同的检

查约束和检查表达式。相似地，一个继承链接也可以使用ALTER TABLE的 NO INHERIT变体从一个子表中移除。动态增加和移除继承链接可以用于实现表划分（见第 2.11 节 “表分区”）。

一种创建一个未来将被用做子女的新表的方法是在CREATE TABLE中使用LIKE子句。这将创建一个和源表具有相同列的新表。如果源表上定义有任何CHECK约束，LIKE的INCLUDING CONSTRAINTS选项可以用来让新的子表也包含和父表相同的约束。

当有任何一个子表存在时，父表不能被删除。当子表的列或者检查约束继承于父表时，它们也不能被删除或修改。如果希望移除一个表和它的所有后代，一种简单的方法是使用CASCADE选项删除父表（见第 2.14 节 “依赖跟踪”）。

ALTER TABLE(7)将会把列的数据定义或检查约束上的任何变化沿着继承层次向下传播。同样，删除被其他表依赖的列只能使用CASCADE选项。ALTER TABLE对于重名列的合并和拒绝遵循与CREATE TABLE同样的规则。

继承的查询仅在附表上执行访问权限检查。例如，在cities表上授予UPDATE权限也隐含着通过cities访问时在capitals表中更新行的权限。 这保留了数据（也）在父表中的样子。但是如果没有额外的授权，则不能直接更新capitals表。 此规则的两个例外是TRUNCATE 和 LOCK TABLE，总是检查子表的权限，不管它们是直接处理还是通过在父表上执行的那些命令递归处理。

以类似的方式，父表的行安全性策略（见第 2.8 节 “行安全性策略”）适用于继承查询期间来自于子表的行。 只有当子表在查询中被明确提到时，其策略（如果有）才会被应用，在那种情况下，附着在其父表上的任何策略都会被忽略。

外部表（见第 2.12 节 “外部数据”）也可以是继承层次 中的一部分，即可以作为父表也可以作为子表，就像常规表一样。如果 一个外部表是继承层次的一部分，那么任何不被该外部表支持的操作也 不被整个层次所支持。

2.10.1. 警告

注意并非所有的SQL命令都能工作在继承层次上。用于数据查询、数据修改或模式修改（例如SELECT、UPDATE、DELETE、大部分ALTER TABLE的变体，但INSERT或ALTER TABLE ... RENAME不在此列）的命令会默认将子表包含在内并且支持ONLY记号来排除子表。负责数据库维护和调整的命令（如REINDEX、VACUUM）只工作在独立的、物理的表上并且不支持在继承层次上的递归。每个命令相应的行为请参见它们的参考页（SQL 命令）。

继承特性的一个严肃的限制是索引（包括唯一约束）和外键约束值应用在单个表上而非它们的继承子女。在外键约束的引用端和被引用端都是这样。因此，按照上面的例子：

- 如果我们声明cities.*name*为UNIQUE或者PRIMARY KEY，这将不会阻止capitals表中拥有和cities中城市同名的行。而且这些重复的行将会默认显示在cities的查询中。事实上，capitals在默认情况下是根本不能拥有唯一约束的，并且因此能够包含多个同名的行。我们可以为capitals增加一个唯一约束，但这无法阻止相对于cities的重复。

- 相似地，如果我们指定cities.*name* REFERENCES某个其他表，该约束不会自动地传播到capitals。在此种情况下，我们可以变通地在capitals上手工创建一个相同的REFERENCES约束。

- 指定另一个表的列REFERENCES cities(name)将允许其他表包含城市名称，但不会包含首府名称。这对于这个例子不是一个好的变通方案。

某些未为继承层次结构实现的功能是为声明性分区实现的。在决定使用旧继承进行分区是否对应用程序有用时，需要非常注意。

2.11. 表分区

UXDB支持基本的表划分。本小节介绍为何以及怎样把划分实现为数据库设计的一部分。

2.11.1. 概述

划分指的是将逻辑上的一个大表分成一些小的物理上的片。划分有很多益处：

- 在某些情况下查询性能能够显著提升，特别是当那些访问压力大的行在一个分区或者少数几个分区时。划分可以取代索引的主导列、减小索引尺寸以及使索引中访问压力大的部分更有可能被放在内存中。
- 当查询或更新访问一个分区的大部分行时，可以通过该分区上的一个顺序扫描来取代分散到整个表上的索引和随机访问，这样可以改善性能。
- 如果批量操作的需求是在分区设计时就规划好的，则批量装载和删除可以通过增加或者去除分区来完成。执行ALTER TABLE DETACH PARTITION或者使用DROP TABLE删除一个分区远快于批量操作。这些命令也完全避免了批量DELETE导致的VACUUM开销。
- 很少使用的数据可以被迁移到便宜且较慢的存储介质上。

当一个表非常大时，划分所带来的好处是非常值得的。一个表何种情况下会从划分获益取决于应用，一个经验法则是当表的尺寸超过了数据库服务器物理内存时，划分会为表带来好处。

UXDB对下列分区形式提供了内建支持：

范围划分

表被根据一个关键列或一组列划分为“范围”，不同的分区的范围之间没有重叠。例如，我们可以根据日期范围划分，或者根据特定业务对象的标识符划分。

列表划分

通过显式地列出每一个分区中出现的键值来划分表。

哈希分区

通过为每个分区指定模数和余数来对表进行分区。每个分区所持有的行都满足：分区键的值除以为其指定的模数将产生为其指定的余数。

如果应用需要使用上面所列之外的分区形式，可以使用诸如继承和UNION ALL视图之类的替代方法。这些方法很灵活，但是却缺少内建声明式分区的一些性能优势。

2.11.2. 声明式划分

UXDB提供了一种方法指定如何把一个表划分成称为分区的片段。被划分的表被称作*分区表*。这种说明由*分区方法*以及要被用作*分区键*的列或者表达式列表组成。

所有被插入到分区表的行将被基于分区键的值路由到*分区*中。每个分区都有一个由其*分区边界*定义的数据子集。当前支持的分区方法是范围、列表以及哈希。

分区本身也可能被定义为分区表，这种用法被称为*子分区*。分区可以有自己的与其他分区不同的索引、约束以及默认值。创建分区表及分区的更多细节请见CREATE TABLE(7)。

无法把一个常规表转换成分区表，反之亦然。不过，可以把一个包含数据的常规表或者分区表作为分区加入到另一个分区表，或者从分区表中移走一个分区并且把它变成一个独立的表。有关ATTACH PARTITION和DETACH PARTITION子命令的内容请见ALTER TABLE(7)。

个体分区在内部以继承的方式链接到分区表，不过无法对声明式分区表或其分区使用继承的某些一般特性（下文讨论）。例如，分区不能有除其所属分区表之外的父表，一个常规表也不能从分区表继承使得后者成为其父表。这意味着分区表及其分区不会参与到与常规表的继承关系中。由于分区表及其分区组成的分区层次仍然是一种继承层次，所有第 2.10 节 “继承”中所述的继承的普通规则也适用，不过有一些例外，尤其是：

- 分区表的CHECK约束和NOT NULL约束总是会被其所有的分区所继承。不允许在分区表上创建标记为NO INHERIT的CHECK约束。
- 只要分区表中不存在分区，则支持使用ONLY仅在分区表上增加或者删除约束。一旦分区存在，那样做就会导致错误，因为当分区存在时是不支持仅在分区表上增加或删除约束的。不过，分区表本身上的约束可以被增加（如果它们不出现在父表中）和删除。
- 由于分区表并不直接拥有任何数据，尝试在分区表上使用TRUNCATE ONLY将总是返回错误。
- 分区不能有在父表中不存在的列。在使用CREATE TABLE创建分区时不能指定列，在事后使用ALTER TABLE时也不能为分区增加列。只有当表的列正好匹配父表时，才能使用ALTER TABLE ... ATTACH PARTITION将它作为分区加入。
- 如果NOT NULL约束在父表中存在，那么就不能删除分区的列上的对应的NOT NULL约束。

分区也可以是外部表，不过它们有一些普通表没有的限制，详情请见CREATE FOREIGN TABLE(7)。

更新行的分区键可能导致它满足另一个不同的分区的分区边界，进而被移动到那个分区中。

2.11.2.1. 例子

假定我们正在为一个大型的冰激凌公司构建数据库。该公司每天测量最高温度以及每个区域的冰激凌销售情况。概念上，我们想要一个这样的表：

```
CREATE TABLE measurement (
    city_id         int not null,
    logdate         date not null,
    peaktemp        int,
    unitsales       int
);
```

我们知道大部分查询只会访问上周的、上月的或者上季度的数据，因为这个表的主要用途是为管理层准备在线报告。为了减少需要被存放的旧数据量，我们决定只保留最近3年的数据。在每个月的开始我们将去除掉最早的那个月的数据。在这种情况下我们可以使用分区技术来帮助我们满足对measurement表的所有不同需求。

要在这种情况下使用声明式分区，可采用下面的步骤：

1. 通过指定PARTITION BY子句把measurement表创建为分区表，该子句包括分区方法（这个例子中是RANGE）以及用作分区键的列列表。

   ```
   CREATE TABLE measurement (
       city_id         int not null,
       logdate         date not null,
       peaktemp        int,
       unitsales       int
   ) PARTITION BY RANGE (logdate);
   ```

可能需要决定在分区键中使用多列进行范围分区。当然，这通常会导致较大数量的分区，其中每一个个体都比较小。另一方面，使用较少的列可能会导致粗粒度的分区策略得到较少数量的分区。如果条件涉及这些列中的一部分或者全部，访问分区表的查询将不得不扫描较少的分区。例如，考虑一个使用列*lastname*和*firstname*（按照这样的顺序）作为分区键进行范围分区的表。

2. 创建分区。每个分区的定义必须指定对应于父表的分区方法和分区键的边界。注意，如果指定的边界使得新分区的值会与已有分区中的值重叠，则会导致错误。向父表中插入无法映射到任何现有分区的数据将会导致错误，这种情况下应该手工增加一个合适的分区。

   分区以普通UXDB表（或者可能是外部表）的方式创建。可以为每个分区单独指定表空间和存储参数。

   没有必要创建表约束来描述分区的分区边界条件。相反，只要需要引用分区约束时，分区约束会自动地隐式地从分区边界说明中生成。

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
   FOR VALUES FROM ('2006-02-01') TO ('2006-03-01');

CREATE TABLE measurement_y2006m03 PARTITION OF measurement
   FOR VALUES FROM ('2006-03-01') TO ('2006-04-01');

...
CREATE TABLE measurement_y2007m11 PARTITION OF measurement
   FOR VALUES FROM ('2007-11-01') TO ('2007-12-01');

CREATE TABLE measurement_y2007m12 PARTITION OF measurement
   FOR VALUES FROM ('2007-12-01') TO ('2008-01-01')
   TABLESPACE fasttablespace;

CREATE TABLE measurement_y2008m01 PARTITION OF measurement
   FOR VALUES FROM ('2008-01-01') TO ('2008-02-01')
   WITH (parallel_workers = 4)
   TABLESPACE fasttablespace;
```

   为了实现子分区，在创建分区的命令中指定PARTITION BY子句，例如：

```
CREATE TABLE measurement_y2006m02 PARTITION OF measurement
   FOR VALUES FROM ('2006-02-01') TO ('2006-03-01')
   PARTITION BY RANGE (peaktemp);
```

   在创建了measurement_y2006m02的分区之后，任何被插入到measurement中且被映射到measurement_y2006m02的数据（或者直接被插入到measurement_y2006m02的数据，假定它满足这个分区的分区约束）将被基于*peaktemp*列进一步重定向到measurement_y2006m02的一个分区。指定的分区键可以与父亲的分区键重叠，不过在指定子分区的边界时要注意它接受的数据集合是分区自身边界允许的数据集合的一个子集，系统不会尝试检查事情情况是否如此。

3. 在分区表的键列上创建一个索引，还有其他需要的索引（键索引并不是必需的，但是大部分场景中它都能很有帮助）。这会自动在每个分区上创建一个索引，并且后来创建或者附着的任何分区也将会包含索引。

```
CREATE INDEX ON measurement (logdate);
```

4. 确保enable_partition_pruning配置参数在uxsinodb.conf中没有被禁用。如果被禁用，查询将不会按照想要的方式被优化。

在上面的例子中，我们会每个月创建一个新分区，因此写一个脚本来自动生成所需的DDL会更好。

2.11.2.2. 分区维护

通常在初始定义分区表时建立的分区并非保持静态不变。移除旧分区的数据并且为新数据周期性地增加新分区的需求比比皆是。分区的最大好处之一就是可以通过操纵分区结构来近乎瞬时地执行这类让人头痛的任务，而不是物理地去除大量数据。

移除旧数据最简单的选择是删除掉不再需要的分区：

```
DROP TABLE measurement_y2006m02;
```

这可以非常快地删除数百万行记录，因为它不需要逐个删除每个记录。不过要注意上面的命令需要在父表上拿到ACCESS EXCLUSIVE锁。

另一种通常更好的选项是把分区从分区表中移除，但是保留它作为一个独立的表：

```
ALTER TABLE measurement DETACH PARTITION measurement_y2006m02;
```

这允许在它被删除之前在其数据上执行进一步的操作。例如，这通常是一种使用COPY、ux_dump或类似工具备份数据的好时候。这也是把数据聚集成较小的格式、执行其他数据操作或者运行报表的好时机。

类似地，我们可以增加一个新分区来处理新数据。我们可以在分区表中创建一个空分区，就像上面创建的初始分区那样：

```
CREATE TABLE measurement_y2008m02 PARTITION OF measurement
   FOR VALUES FROM ('2008-02-01') TO ('2008-03-01')
   TABLESPACE fasttablespace;
```

另外一种选择是，有时候在分区结构之外创建新表更加方便，然后将它作为一个合适的分区。这允许先对数据进行装载、检查和转换，然后再让它们出现在分区表中：

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS)
  TABLESPACE fasttablespace;

ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );

\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work

ALTER TABLE measurement ATTACH PARTITION measurement_y2008m02
   FOR VALUES FROM ('2008-02-01') TO ('2008-03-01' );
```

在运行ATTACH PARTITION命令之前，推荐在要被挂接的表上创建一个CHECK约束来匹配期望的分区约束。 这样，系统将能够跳过扫描来验证隐式分区约束。 没有CHECK约束，将扫描表以

验证分区约束，同时对该分区持有ACCESS EXCLUSIVE锁定，并在父表上持有SHARE UPDATE EXCLUSIVE锁。 在完成ATTACH PARTITION后，可能需要删除冗余CHECK约束。

如上所述，可以在分区的表上创建索引，并自动将其应用于整个层次结构。 这非常便利，因为不仅现有分区将变为索引，而且将来创建的任何分区都将变为索引。 一个限制是，在创建这样一个分区索引时，不可能同时使用CONCURRENTLY限定符。 为了克服长时间锁，可以对分区表使用CREATE INDEX ON ONLY ;这样的索引被标记为无效，并且分区不会自动应用该索引。分区上的索引可以使用CONCURRENTLY分别的创建。 然后使用ALTER INDEX .. ATTACH PARTITIONattached到父索引。 一旦所有分区的索引附加到父索引，父索引将自动标记为有效。例如：

```
CREATE INDEX measurement_usls_idx ON ONLY measurement (unitsales);

CREATE INDEX measurement_usls_200602_idx
   ON measurement_y2006m02 (unitsales);
ALTER INDEX measurement_usls_idx
   ATTACH PARTITION measurement_usls_200602_idx;
...
```

该技术也可以与UNIQUE 和PRIMARY KEY 约束一起试用；当创建约束时隐式创建索引。例如：

```
ALTER TABLE ONLY measurement ADD UNIQUE (city_id, logdate);

ALTER TABLE measurement_y2006m02 ADD UNIQUE (city_id, logdate);
ALTER INDEX measurement_city_id_logdate_key
   ATTACH PARTITION measurement_y2006m02_city_id_logdate_key;
...
```

2.11.2.3. 限制

分区表有下列限制：

- 没有办法创建跨越所有分区的排除约束，只可能单个约束每个叶子分区。
- 分区表上的惟一约束必须包括所有分区键列。存在此限制是因为UXDB只能每个分区中分别强制实施唯一性。
- 如果必要，必须在个体分区上定义BEFORE ROW触发器，分区表上不需要。
- 不允许在同一个分区树中混杂临时关系和持久关系。因此，如果分区表是持久的，则其分区也必须是持久的，反之亦然。在使用临时关系时，分区数的所有成员都必须来自于同一个会话。

2.11.3. 使用继承实现

虽然内建的声明式分区适合于大部分常见的用例，但还是有一些场景需要更加灵活的方法。分区可以使用表继承来实现，这能够带来一些声明式分区不支持的特性，例如：

- 对声明式分区来说，分区必须具有和分区表正好相同的列集合，而在表继承中，子表可以有父表中没有出现过的额外列。
- 表继承允许多继承。
- 声明式分区仅支持范围、列表以及哈希分区，而表继承允许数据按照用户的选择来划分（不过注意，如果约束排除不能有效地剪枝子表，查询性能可能会很差）。

- 在使用声明式分区时，一些操作比使用表继承时要求更长的持锁时间。例如，向分区表中增加分区或者从分区表移除分区要求在父表上取得一个ACCESS EXCLUSIVE锁，而在常规继承的情况下一个SHARE UPDATE EXCLUSIVE锁就足够了。

2.11.3.1. 例子

我们使用上面用过的同一个measurement表。为了使用继承实现分区，可使用下面的步骤：

1. 创建“主”表，所有的“子”表都将从它继承。这个表将不包含数据。不要在这个表上定义任何检查约束，除非想让它们应用到所有的子表上。同样，在这个表上定义索引或者唯一约束也没有意义。对于我们的例子来说，主表是最初定义的measurement表。
2. 创建数个“子”表，每一个都从主表继承。通常，这些表将不会在从主表继承的列集合之外增加任何列。正如声明性分区那样，这些表就是普通的UXDB表（或者外部表）。

```
CREATE TABLE measurement_y2006m02 () INHERITS (measurement);
CREATE TABLE measurement_y2006m03 () INHERITS (measurement);
...
CREATE TABLE measurement_y2007m11 () INHERITS (measurement);
CREATE TABLE measurement_y2007m12 () INHERITS (measurement);
CREATE TABLE measurement_y2008m01 () INHERITS (measurement);
```

3. 为子表增加不重叠的表约束来定义每个分区允许的键值。

   典型的例子是：

```
CHECK ( x = 1 )
CHECK ( county IN ( 'Oxfordshire', 'Buckinghamshire', 'Warwickshire' ))
CHECK ( outletID >= 100 AND outletID < 200 )
```

   确保约束能保证不同子表允许的键值之间没有重叠。设置范围约束的常见错误：

```
CHECK ( outletID BETWEEN 100 AND 200 )
CHECK ( outletID BETWEEN 200 AND 300 )
```

   这是错误的，因为不清楚键值200属于哪一个子表。

   像下面这样创建子表会更好：

```
CREATE TABLE measurement_y2006m02 (
   CHECK ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2006m03 (
   CHECK ( logdate >= DATE '2006-03-01' AND logdate < DATE '2006-04-01' )
) INHERITS (measurement);

...
CREATE TABLE measurement_y2007m11 (
   CHECK ( logdate >= DATE '2007-11-01' AND logdate < DATE '2007-12-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2007m12 (
```

```
  CHECK ( logdate >= DATE '2007-12-01' AND logdate < DATE '2008-01-01' )
) INHERITS (measurement);

CREATE TABLE measurement_y2008m01 (
  CHECK ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
) INHERITS (measurement);
```

4. 对于每个子表，在键列上创建一个索引，以及任何想要的其他索引。

```
CREATE INDEX measurement_y2006m02_logdate ON measurement_y2006m02 (logdate);
CREATE INDEX measurement_y2006m03_logdate ON measurement_y2006m03 (logdate);
CREATE INDEX measurement_y2007m11_logdate ON measurement_y2007m11 (logdate);
CREATE INDEX measurement_y2007m12_logdate ON measurement_y2007m12 (logdate);
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2008m01 (logdate);
```

5. 我们希望我们的应用能够使用INSERT INTO measurement ...并且数据将被重定向到合适的分区表。我们可以通过为主表附加一个合适的触发器函数来实现这一点。如果数据将只被增加到最后一个分区，我们可以使用一个非常简单的触发器函数：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO measurement_y2008m01 VALUES (NEW.*);
  RETURN NULL;
END;
$$
LANGUAGE pluxsql;
```

完成函数创建后，我们创建一个调用该触发器函数的触发器：

```
CREATE TRIGGER insert_measurement_trigger
  BEFORE INSERT ON measurement
  FOR EACH ROW EXECUTE FUNCTION measurement_insert_trigger();
```

我们必须在每个月重新定义触发器函数，这样它才会总是指向当前的子表。而触发器的定义则不需要被更新。

我们也可能希望插入数据时服务器会自动地定位应该加入数据的子表。我们可以通过一个更复杂的触发器函数来实现之，例如：

```
CREATE OR REPLACE FUNCTION measurement_insert_trigger()
RETURNS TRIGGER AS $$
BEGIN
  IF ( NEW.logdate >= DATE '2006-02-01' AND
       NEW.logdate < DATE '2006-03-01' ) THEN
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
  ELSIF ( NEW.logdate >= DATE '2006-03-01' AND
        NEW.logdate < DATE '2006-04-01' ) THEN
    INSERT INTO measurement_y2006m03 VALUES (NEW.*);
  ...
  ELSIF ( NEW.logdate >= DATE '2008-01-01' AND
        NEW.logdate < DATE '2008-02-01' ) THEN
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

```
    ELSE
        RAISE EXCEPTION 'Date out of range.  Fix the measurement_insert_trigger() function!';
    END IF;
    RETURN NULL;
END;
$$
LANGUAGE pluxsql;
```

触发器的定义和以前一样。注意每一个IF测试必须准确地匹配它的子表的CHECK约束。

当该函数比单月形式更加复杂时，并不需要频繁地更新它，因为可以在需要的时候提前加入分支。

> **注意**
>
> 在实践中，如果大部分插入都会进入最新的子表，最好先检查它。为了简洁，我们为触发器的检查采用了和本例中其他部分一致的顺序。

把插入重定向到一个合适的子表中的另一种不同方法是在主表上设置规则而不是触发器。例如：

```
CREATE RULE measurement_insert_y2006m02 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2006-02-01' AND logdate < DATE '2006-03-01' )
DO INSTEAD
    INSERT INTO measurement_y2006m02 VALUES (NEW.*);
...
CREATE RULE measurement_insert_y2008m01 AS
ON INSERT TO measurement WHERE
    ( logdate >= DATE '2008-01-01' AND logdate < DATE '2008-02-01' )
DO INSTEAD
    INSERT INTO measurement_y2008m01 VALUES (NEW.*);
```

规则的开销比触发器大很多，但是这种开销是每个查询只有一次，而不是每行一次，因此这种方法可能对批量插入的情况有优势。不过，在大部分情况下，触发器方法将提供更好的性能。

注意COPY会忽略规则。如果想要使用COPY插入数据，则需要拷贝到正确的子表而不是直接放在主表中。COPY会引发触发器，因此在使用触发器方法时可以正常使用它。

规则方法的另一个缺点是，如果规则集合无法覆盖插入日期，则没有简单的方法能够强制产生错误，数据将会无声无息地进入到主表中。

6. 确认constraint_exclusion配置参数在uxsinodb.conf中没有被禁用，否则将会不必要地访问子表。

如我们所见，一个复杂的表层次可能需要大量的DDL。在上面的例子中，我们可能为每个月创建一个新的子表，因此编写一个脚本来自动生成所需要的DDL可能会更好。

2.11.3.2. 继承分区的维护

要快速移除旧数据，只需要简单地去掉不再需要的子表：

```
DROP TABLE measurement_y2006m02;
```

要从继承层次表中去掉子表，但还是把它当做一个表保留：

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

要增加一个新子表来处理新数据，可以像上面创建的原始子表那样创建一个空的子表：

```
CREATE TABLE measurement_y2008m02 (
   CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' )
) INHERITS (measurement);
```

或者，用户可能想要创建新子表并且在将它加入到表层次之前填充它。这可以允许数据在被父表上的查询可见之前对数据进行装载、检查以及转换。

```
CREATE TABLE measurement_y2008m02
  (LIKE measurement INCLUDING DEFAULTS INCLUDING CONSTRAINTS);
ALTER TABLE measurement_y2008m02 ADD CONSTRAINT y2008m02
  CHECK ( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01' );
\copy measurement_y2008m02 from 'measurement_y2008m02'
-- possibly some other data preparation work
ALTER TABLE measurement_y2008m02 INHERIT measurement;
```

2.11.3.3. 提醒

下面的提醒适用于用继承实现的分区：

- 没有自动的方法啊验证所有的CHECK约束之间是否互斥。编写代码来产生子表以及创建和修改相关对象比手写命令要更加安全。
- 索引和外键约束适用于单个表而不是其继承子级，因此它们有一些caveats 需要注意。
- 这里展示的模式假定行的键列值从不改变，或者说改变不足以让行移动到另一个分区。由于CHECK约束的存在，尝试那样做的**UPDATE**将会失败。如果需要处理那种情况，可以在子表上放置适当的更新触发器，但是那会使对结构的管理更加复杂。
- 如果使用手工的**VACUUM**或者**ANALYZE**命令，不要忘记需要在每个子表上单独运行它们。这样的命令：

  ```
  ANALYZE measurement;
  ```

  将只会处理主表。
- 带有ON CONFLICT子句的**INSERT**语句不太可能按照预期工作，因为只有在指定的目标关系而不是其子关系上发生唯一违背时才会采取ON CONFLICT行动。
- 将会需要触发器或者规则将行路由到想要的子表中，除非应用明确地知道分区的模式。编写触发器可能会很复杂，并且会比声明式分区在内部执行的元组路由慢很多。

2.11.4. 分区剪枝

分区剪枝是一种提升声明式分区表性能的查询优化技术。例如：

```
SET enable_partition_pruning = on;                -- the default
```

SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';

如果没有分区剪枝，上面的查询将会扫描measurement表的每一个分区。如果启用了分区剪枝，规划器将会检查每个分区的定义并且检验该分区是否因为不包含符合查询WHERE子句的行而无需扫描。当规划器可以证实这一点时，它会把分区从查询计划中排除（剪枝）。

通过使用EXPLAIN命令和enable_partition_pruning配置参数，可以展示剪枝掉分区的计划与没有剪枝的计划之间的差别。对这种类型的表设置，一种典型的未优化计划是：

```
SET enable_partition_pruning = off;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                 QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate  (cost=188.76..188.77 rows=1 width=8)
   ->  Append  (cost=0.00..181.05 rows=3085 width=0)
         ->  Seq Scan on measurement_y2006m02  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2006m03  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
...
         ->  Seq Scan on measurement_y2007m11  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2007m12  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
         ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
               Filter: (logdate >= '2008-01-01'::date)
```

某些或者全部的分区可能会使用索引扫描取代全表顺序扫描，但是这里的重点是根本不需要扫描较老的分区来回答这个查询。当我们启用分区剪枝时，我们会得到一个便宜很多的计划，而它能给出相同的答案：

```
SET enable_partition_pruning = on;
EXPLAIN SELECT count(*) FROM measurement WHERE logdate >= DATE '2008-01-01';
                                 QUERY PLAN
-----------------------------------------------------------------------------------------
 Aggregate  (cost=37.75..37.76 rows=1 width=8)
   ->  Seq Scan on measurement_y2008m01  (cost=0.00..33.12 rows=617 width=0)
         Filter: (logdate >= '2008-01-01'::date)
```

注意，分区剪枝仅由分区键隐式定义的约束所驱动，而不是由索引的存在驱动。因此，没有必要在键列上定义索引。是否需要为一个给定分区创建索引取决于预期的查询扫描该分区时会扫描大部分还是小部分。后一种情况下索引的帮助会比前者大。

不仅在给定查询的规划期间可以执行分区剪枝，在其执行期间也能执行分区剪枝。 这非常有用，因为如果子句中包含查询规划时值未知的表达式时，这可以剪枝掉更多的分区； 例如在PREPARE语句中定义的参数会使用从子查询拿到的值，或者嵌套循环连接内侧关系上的参数化值。 执行期间的分区剪枝可能在下列任何时刻执行：

- 在查询计划的初始化期间。对于执行的初始化阶段就已知值的参数，可以在这里执行分区剪枝。这个阶段中被剪枝掉的分区将不会显示在查询的EXPLAIN或EXPLAIN ANALYZE结果中。通过观察EXPLAIN输出的“Subplans Removed”属性，可以确定被剪枝掉的分区数。

- 在查询计划的实际执行期间。这里可以使用只有在实际查询执行时才能知道的值执行分区剪枝。这包括来自子查询的值以及来自执行时参数的值（例如来自于参数化嵌套循环连接的参

数）。由于在查询执行期间这些参数的值可能会改变多次，所以只要分区剪枝使用到的执行参数发生改变，就会执行一次分区剪枝。要判断分区是否在这个阶段被剪枝，需要仔细地观察EXPLAIN ANALYZE输出中的loops属性。 对应于不同分区的子计划可以具有不同的值，这取决于在执行期间每个分区被修剪的次数。 如果每次都被剪枝，有些分区可能会显示为(never executed)。

可以使用enable_partition_pruning设置禁用分区剪枝。

> **注意**
>
> 执行时间分区裁剪当前只针对Append和MergeAppend节点类型。 它还没有为ModifyTable节点类型实现，但有可能会在将来发布的 UXDB中更改。

2.11.5. 分区和约束排除

约束排除是一种与分区剪枝类似的查询优化技术。虽然它主要被用于使用传统继承方法实现的分区上，但它也可以被用于其他目的，包括用于声明式分区。

约束排除以非常类似于分区剪枝的方式工作，不过它使用每个表的CHECK约束 — 这也是它得名的原因 — 而分区剪枝使用表的分区边界，分区边界仅存在于声明式分区的情况中。另一点不同之处是约束排除仅在规划时应用，在执行时不会尝试移除分区。

由于约束排除使用CHECK约束，这导致它比分区剪枝要慢，但有时候可以被当作一种优点加以利用：因为甚至可以在声明式分区的表上（在分区边界之外）定义约束，约束排除可能可以从查询计划中消去额外的分区。

constraint_exclusion的默认（也是推荐的）设置不是on也不是off，而是一种被称为partition的中间设置，这会导致该技术仅被应用于可能工作在继承分区表上的查询。on设置导致规划器检查所有查询中的CHECK约束，甚至是那些不太可能受益的简单查询。

下列提醒适用于约束排除：

- 约束排除仅适用于查询规划期间，和分区裁剪不同，在查询执行期间也可以应用。
- 只有查询的WHERE子句包含常量（或者外部提供的参数）时，约束排除才能有效果。例如，针对一个非不变函数（如CURRENT_TIMESTAMP）的比较不能被优化，因为规划器不知道该函数的值在运行时会落到哪个子表中。
- 保持分区约束简单化，否则规划器可能无法验证哪些子表可能不需要被访问。如前面的例子所示，对列表分区使用简单的等值条件，对范围分区使用简单的范围测试。一种好的经验规则是分区约束应该仅包含分区列与常量使用B-树的可索引操作符的比较，因为只有B-树的可索引列才允许出现在分区键中。
- 约束排除期间会检查父表的所有子表上的所有约束，因此大量的子表很可能明显地增加查询规划时间。因此，传统的基于继承的分区可以很好地处理上百个子表，不要尝试使用上千个子表。

2.11.6. 声明分区最佳实践

应该谨慎地选择如何划分表，因为查询规划和执行的性能可能会受到不良设计的负面影响。

最重要的设计决策之一是列或者如和对数据进行分区的。 通常最佳选择是按列或列集合进行分区，这些列最常出现在分区表上执行的查询的 WHERE子句中。 WHERE子句项与分区键匹配并兼容，可用于裁剪不需要的分区。 但是，可能会被迫根据PRIMARY KEY或UNIQUE约束的要求

做出其他决策。 在规划分区策略时，删除不需要的数据也是需要考虑的一个因素。 可以相当快地分离整个分区，因此采用这样方式设计分区策略可能是有益的，既把一次删除的所有数据都放在单个分区中。

选择表应该划分的分区的目标数量也是一个重要的决策。 没有足够的分区可能意味着索引仍然太大，数据位置仍然较差，这可能导致缓存命中率很低。 但是，将表划分为太多的分区也会导致问题。 在查询规划和执行期间，过多的分区可能意味着查询计划时间较长，内存消耗也更高。 在选择如何划分表时，考虑将来可能发生的更改也很重要。 例如，如果您选择为每个客户提供一个分区，而您目前只有少量的大客户，那么，如果几年后您发现自己有大量的小客户，那么就要考虑这种影响。 在这种情况下，最好选择按HASH分区并且选择合理数量的分区，而不是尝试按 LIST 进行分区，并希望客户数量的增长不会超出按数据分区的实际范围。

子分区可用于进一步划分预期会比其他分区更大的分区，尽管过多的子分区很容易导致大量分区，并可能导致前一段中提到的相同问题。

考虑查询计划和执行期间的分区开销也很重要。 查询规划器通常能够很好地处理多达几千个分区的分区层次结构，前提是典型的查询允许查询规划器裁剪除了少量分区之外的所有分区。 规划器执行分区修剪后保留更多分区时，规划时间会变长，内存消耗会更高。对于UPDATE 和DELETE命令尤其如此。 担心拥有大量分区的另一个原因是，服务器的内存消耗可能会在一段时间内显著增加，特别是如果许多会话接触大量分区。 这是因为每个分区都需要将其元数据加载到接触它的每个会话的本地内存中。

对于数据仓库类型工作负载，使用比 OLTP 类型工作负载更多的分区数量很有意义。 通常，在数据仓库中，查询计划时间不太值得关注，因为大多数处理时间都花在查询执行期间。 对于这两种类型的工作负载，尽早做出正确的决策非常重要，因为重新分区大量数据可能会非常缓慢。模拟预期工作负载通常有利于优化分区策略。永远不要假设更多的分区比更少的分区更好，反之亦然。

2.12. 外部数据

UXDB实现了部分的SQL/MED规定，允许我们使用普通SQL查询来访问位于UXDB之外的数据。这种数据被称为外部数据（注意这种用法不要和外键混淆，后者是数据库中的一种约束）。

外部数据可以在一个外部数据包装器的帮助下被访问。一个外部数据包装器是一个库，它可以与一个外部数据源通讯，并隐藏连接到数据源和从它获取数据的细节。在contrib模块中有一些外部数据包装器。其他类型的外部数据包装器可以在第三方产品中找到。如果这些现有的外部数据包装器都不能满足需要，可以自己编写一个。

要访问外部数据，我们需要建立一个外部服务器对象，它根据它所支持的外部数据包装器所使用的一组选项定义了如何连接到一个特定的外部数据源。接着我们需要创建一个或多个外部表，它们定义了外部数据的结构。一个外部表可以在查询中像一个普通表一样地使用，但是在UXDB服务器中外部表没有存储数据。不管使用什么外部数据包装器，UXDB会要求外部数据包装器从外部数据源获取数据，或者在更新命令的情况下传送数据到外部数据源。

访问远程数据可能需要在外部数据源的授权。这些信息通过一个用户映射提供，它基于当前的UXDB角色提供了附加的数据例如用户名和密码。

更多信息请见 CREATE FOREIGN DATA WRAPPER(7)、 CREATE SERVER(7)、 CREATE USER MAPPING(7)、 CREATE FOREIGN TABLE(7)、以及 IMPORT FOREIGN SCHEMA(7)。

2.13. 其他数据库对象

表是一个关系型数据库结构中的核心对象，因为它们承载了我们的数据。但是它们并不是数据库中的唯一一种对象。有很多其他种类的对象可以被创建来使得数据的使用和刮泥更加方便或高效。在本章中不会讨论它们，但是我们在会给出一个列表：

- 视图
- 函数、过程和操作符
- 数据类型和域
- 触发器和重写规则

2.14. 依赖跟踪

当我们创建一个涉及到很多具有外键约束、视图、触发器、函数等的表的复杂数据库结构时，我们隐式地创建了一张对象之间的依赖关系网。例如，具有一个外键约束的表依赖于它所引用的表。

为了保证整个数据库结构的完整性，UXDB确保我们无法删除仍然被其他对象依赖的对象。例如，尝试删除第 2.4.5 节 “外键”中的产品表会导致一个如下的错误消息，因为有订单表依赖于产品表：

```
DROP TABLE products;

ERROR:  cannot drop table products because other objects depend on it
DETAIL:  constraint orders_product_no_fkey on table orders depends on table products
HINT:  Use DROP ... CASCADE to drop the dependent objects too.
```

该错误消息包含了一个有用的提示：如果我们不想一个一个去删除所有的依赖对象，我们可以执行：

```
DROP TABLE products CASCADE;
```

这样所有的依赖对象将被移除，同样依赖于它们的任何对象也会被递归删除。在这种情况下，订单表不会被移除，但是它的外键约束会被移除。之所以在这里会停下，是因为没有什么依赖着外键约束（如果希望检查DROP ... CASCADE会干什么，运行不带CASCADE的DROP并阅读DETAIL输出）。

UXDB中的几乎所有DROP命令都支持CASCADE。当然，其本质的区别随着对象的类型而不同。我们也可以用RESTRICT代替CASCADE来获得默认行为，它将阻止删除任何被其他对象依赖的对象。

> **注意**
>
> 根据SQL标准，在DROP命令中指定RESTRICT或CASCADE是被要求的。但没有哪个数据库系统真正强制了这个规则，但是不同的系统中两种默认行为都是可能的。

如果一个DROP命令列出了多个对象，只有在存在指定对象构成的组之外的依赖关系时才需要CASCADE。例如，如果发出命令DROP TABLE tab1, tab2且存在从tab2到tab1的外键引用，那么就不需要CASCADE即可成功执行。

对于用户定义的函数，UXDB会追踪与函数外部可见性质相关的依赖性，例如它的参数和结果类型，但不追踪检查函数体才能知道的依赖性。例如，考虑这种情况：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow',
                        'green', 'blue', 'purple');

CREATE TABLE my_colors (color rainbow, note text);

CREATE FUNCTION get_color_note (rainbow) RETURNS text AS
  'SELECT note FROM my_colors WHERE color = $1'
  LANGUAGE SQL;
```

UXDB将会注意到get_color_note函数依赖于rainbow类型：删掉该类型会强制删除该函数，因为该函数的参数类型就无法定义了。但是UXDB不会认为get_color_note依赖于my_colors表，因此即使该表被删除也不会删除这个函数。虽然这种方法有缺点，但是也有好处。如果该表丢失，这个函数在某种程度上仍然是有效的，但是执行它会导致错误。创建一个同名的新表将允许该函数重新有效。

2.15. 隐藏列

2.15.1. rowid

- 类型介绍

  UXDB数据库的rowid是为了兼容达梦rowid伪列所做功能适配，用来标识数据库基表中每一条记录的唯一键值。本次实现，还新增了数据类型：rowid。该类型是一个大范围整数的数字类型，其范围和功能等同bigint类型。

  新增GUC参数：create_as_parallel。该参数的作用：控制在执行create ... as或者select ... into命令时，数据库可以并行的条件下，是否走并行。如果设置为off，表示为不并行，此时rowid功能正常；设置为on时，表示为走并行，此时无法保证rowid功能正常。该参数默认值为off。

- 示例

```
---创建字段为rowid类型的表并插入数据查询
CREATE TABLE TEST(ID ROWID);
INSERT INTO TEST SELECT GENERATE_SERIES(1, 1000000);
SELECT * FROM TEST WHERE ID > 999995;
   ID
---------
  999996
  999997
  999998
  999999
 1000000
(5 rows)

---创建普通表，查询rowid隐藏列
CREATE TABLE TEST2(ID INT);
INSERT INTO TEST2 SELECT GENERATE_SERIES(25, 30);
SELECT ROWID, * FROM TEST2;
 ROWID | ID
-------+----
     1 | 25
     2 | 26
```

```
    3 | 27
    4 | 28
    5 | 29
    6 | 30
(6 rows)

---利用rowid隐藏列进行数据更新或删除
UPDATE TEST2 SET ID = 31 WHERE ROWID = 1;
DELETE FROM TEST2 WHERE ROWID = 6;
SELECT ROWID, * FROM TEST2;
 ROWID | ID
-------+----
    2 | 26
    3 | 27
    4 | 28
    5 | 29
    1 | 31
(5 rows)

---利用rowid隐藏列进行数据查询
SELECT * FROM TEST2 WHERE ROWID BETWEEN 2 AND 5;
 ID
----
 26
 27
 28
 29
(4 rows)
SELECT * FROM TEST2 WHERE ROWID >= 3;
 ID
----
 27
 28
 29
(3 rows)

---创建分区表，查询rowid隐藏列
CREATE TABLE PATEST(DA DATE NOT NULL) PARTITION BY RANGE (DA);
CREATE TABLE PATEST_CH1 PARTITION OF PATEST FOR VALUES FROM
 ('2022/03/01') TO ('2022/04/01');
CREATE TABLE PATEST_CH2 PARTITION OF PATEST FOR VALUES FROM
 ('2022/04/01') TO ('2022/05/01');
INSERT INTO PATEST SELECT GENERATE_SERIES(TIMESTAMP'2022/03/01',
 TIMESTAMP'2022/04/30', '1 day');

SELECT ROWID, * FROM PATEST_CH1 LIMIT 5;
 ROWID |     DA
-------+------------
    1 | 2022-03-01
    2 | 2022-03-02
    3 | 2022-03-03
    4 | 2022-03-04
    5 | 2022-03-05
(5 rows)
```

```
SELECT ROWID, * FROM PATEST_CH2 LIMIT 5;
 ROWID |     DA
-------+------------
    32 | 2022-04-01
    33 | 2022-04-02
    34 | 2022-04-03
    35 | 2022-04-04
    36 | 2022-04-05
(5 rows)

---开启强制并行，create_as_parallel为默认值
set force_parallel_mode to on;
create table test(id int);
insert into test select generate_series(1, 5);
create table test1 as select * from test;

select rowid, * from test1;
 ROWID | ID
-------+----
     1 |  1
     2 |  2
     3 |  3
     4 |  4
     5 |  5
(5 rows)

---开启强制并行，设置create_as_parallel为on
set force_parallel_mode to on;
set create_as_parallel to on;
create table test(id int);
insert into test select generate_series(1, 5);
select * into test1 from test;

select rowid, * from test1;
 ROWID | ID
-------+----
     0 |  1
     0 |  2
     0 |  3
     0 |  4
     0 |  5
(5 rows)
```

第 3 章 数据操纵

前面的章节讨论了如何创建表和其他结构来保存数据。现在是时候给表填充数据了。本章涉及如何插入、更新和删除表数据。在接下来的一章将最终解释如何把丢失已久的数据从数据库中抽取出来。

3.1. 插入数据

当一个表被创建后，它不包含数据。在数据库可以有点用之前要做的第一件事就是向里面插入数据。数据在概念上是以每次一行地方式被插入的。可以每次插入多行，但是却没有办法一次插入少于一行的数据。即使只知道几个列的值，那么必须创建一个完整的行。

要创建一个新行，使用INSERT(7)命令。这条命令要求提供表的名字和其中列的值。例如，考虑第 2 章 数据定义的产品表：

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric
);
```

一个插入一行的命令将是：

```
INSERT INTO products VALUES (1, 'Cheese', 9.99);
```

数据的值是按照这些列在表中出现的顺序列出的，并且用逗号分隔。通常，数据的值是文字（常量），但也允许使用标量表达式。

上面的语法的缺点是必须知道表中列的顺序。要避免这个问题，也可以显式地列出列。例如，下面的两条命令都有和上文那条 命令一样的效果：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99);
INSERT INTO products (name, price, product_no) VALUES ('Cheese', 9.99, 1);
```

许多用户认为明确列出列的名字是个好习惯。

如果没有获得所有列的值，那么可以省略其中的一些。在这种情况下，这些列将被填充为它们的缺省值。例如：

```
INSERT INTO products (product_no, name) VALUES (1, 'Cheese');
INSERT INTO products VALUES (1, 'Cheese');
```

第二种形式是UXDB的一个扩展。它从使用给出的值从左开始填充列，有多少个给出的列值就填充多少个列，其他列的将使用缺省值。

为了保持清晰，也可以显式地要求缺省值，用于单个的列或者用于整个行：

```
INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', DEFAULT);
INSERT INTO products DEFAULT VALUES;
```

可以在一个命令中插入多行：

```
INSERT INTO products (product_no, name, price) VALUES
    (1, 'Cheese', 9.99),
    (2, 'Bread', 1.99),
    (3, 'Milk', 2.99);
```

也可以插入查询的结果（可能没有行、一行或多行）：

```
INSERT INTO products (product_no, name, price)
  SELECT product_no, name, price FROM new_products
    WHERE release_date = 'today';
```

这提供了用于计算要插入的行的SQL查询机制（第 4 章 查询的全部功能。

> **提示**
>
> 在一次性插入大量数据时，考虑使用COPY(7)命令。它不如INSERT(7)命令那么灵活，但是更高效。 参考第 11.4 节 “填充一个数据库”获取更多有关批量装载性能的信息。

3.2. 更新数据

修改已经存储在数据库中的数据的行为叫做更新。可以更新单个行，也可以更新表中所有的行，还可以更新其中的一部分行。 我们可以独立地更新每个列，而其他的列则不受影响。

要更新现有的行，使用UPDATE(7)命令。这需要提供三部分信息：

1. 表的名字和要更新的列名
2. 列的新值
3. 要更新的是哪（些）行

我们在第 2 章 数据定义说过，SQL 通常并不为行提供唯一标识符。因此我们无法总是直接指定需要更新哪一行。但是，我们可以通过指定一个被更新的行必须满足的条件。只有在表里面存在主键的时候（不管声明它还是不声明它），我们才能可靠地通过选择一个匹配主键的条件来指定一个独立的行。图形化的数据库访问工具就靠这允许我们独立地更新某些行。

例如，这条命令把所有价格为5的产品的价格更新为10：

```
UPDATE products SET price = 10 WHERE price = 5;
```

这样做可能导致零行、一行或者更多行被更新。如果我们试图做一个不匹配任何行的更新，那也不算错误。

让我们仔细看看这个命令。首先是关键字UPDATE， 然后跟着表名字。和平常一样，表名字也可以是用模式限定的， 否则会从路径中查找它。然后是关键字SET， 后面跟着列名、一个等号以及新的列值。新的列值可以是任意标量表达式， 而不仅仅是常量。例如，如果想把所有产品的价格提高 10%，可以用：

```
UPDATE products SET price = price * 1.10;
```

如所见，用于新值的表达式也可以引用行中现有的值。我们还忽略了WHERE子句。如果我们忽略了这个子句， 那么就意味着表中的所有行都要被更新。如果出现了WHERE子句， 那么只有匹配它后面的条件的行被更新。请注意在SET子句中的等号是一个赋值， 而在WHERE子句中的等号是比较，不过这样并不会导致任何歧义。当然WHERE条件不一定非得是等值测试。许多其他操作符也都可以使用（参阅第 6 章 函数和操作符。但是表达式必须得出一个布尔结果。

还可以在一个UPDATE命令中更新更多的列， 方法是在SET子句中列出更多赋值。例如：

```
UPDATE mytable SET a = 5, b = 3, c = 1 WHERE a > 0;
```

3.3. 删除数据

到目前为止我们已经解释了如何向表中增加数据以及如何改变数据。剩下的是讨论如何删除不再需要的数据。和前面增加数据一样，也只能从表中整行整行地删除数据。在前面的一节里我们解释了 SQL 不提供直接访问单个行的方法。因此，删除行只能是通过指定被删除行必须匹配的条件进行。如果在表上有一个主键，那么可以指定准确的行。但是也可以删除匹配条件的一组行，或者可以一次从表中删除所有的行。

可以使用DELETE(7)命令删除行，它的语法和UPDATE命令非常类似。例如，要从产品表中删除所有价格为 10 的产品，使用：

```
DELETE FROM products WHERE price = 10;
```

如果只是写：

```
DELETE FROM products;
```

那么表中所有行都会被删除！程序员一定要注意。

3.4. 从修改的行中返回数据

有时在修改行的操作过程中获取数据很有用。INSERT、 UPDATE和DELETE命令都有一个支持这个的可选的 RETURNING子句。使用RETURNING 可以避免执行额外的数据库查询来收集数据，并且在否则难以可靠地识别修改的行时尤其有用。

所允许的RETURNING子句的内容与SELECT命令的输出列表相同 （请参阅第 4.3 节 “选择列表”）。它可以包含命令的目标表的列名， 或者包含使用这些列的值表达式。一个常见的简写是RETURNING *， 它按顺序选择目标表的所有列。

在INSERT中，可用于RETURNING的数据是插入的行。 这在琐碎的插入中并不是很有用，因为它只会重复客户端提供的数据。 但依赖于计算出的默认值时可以非常方便。例如，当使用serial列来提供唯一标识符时， RETURNING可以返回分配给新行的ID：

```
CREATE TABLE users (firstname text, lastname text, id serial primary key);

INSERT INTO users (firstname, lastname) VALUES ('Joe', 'Cool') RETURNING id;
```

RETURNING子句对于INSERT ... SELECT也非常有用。

在UPDATE中，可用于RETURNING的数据是被修改行的新内容。 例如：

```
UPDATE products SET price = price * 1.10
  WHERE price <= 99.99
  RETURNING name, price AS new_price;
```

在DELETE中，可用于RETURNING的数据是删除行的内容。例如：

```
DELETE FROM products
  WHERE obsoletion_date = 'today'
  RETURNING *;
```

如果目标表上有触发器，可用于RETURNING　的数据是被触发器修改的行。因此，检查由触发器计算的列是 RETURNING的另一个常见用例。

第 4 章 查询

前面的章节解释了如何创建表、如何用数据填充它们 以及如何操纵那些数据。现在我们终于可以讨论如何从数据库中检索数据了。

4.1. 概述

从数据库中检索数据的过程或命令叫做查询。在 SQL 里SELECT(7)命令用于指定查询。SELECT命令的一般语法是

[WITH *with_queries*] SELECT *select_list* FROM *table_expression* [*sort_specification*]

下面几个小节描述选择列表、表表达式和排序声明的细节。WITH查询等高级特性将在最后讨论。

一个简单类型的查询的形式：

SELECT * FROM table1;

假设有一个表叫做table1，这条命令将table1中检索所有行和所有用户定义的列（检索的方法取决于客户端应用。例如，uxsql程序将在屏幕上显示一个 ASCII 形式的表格， 而客户端库将提供函数来从检索结果中抽取单个值）。 选择列表声明*意味着所有表表达式提供的列。 一个选择列表也可以选择可用列的一个子集或者在使用它们之前对列进行计算。例如，如果table1有叫做a、b和c的列（可能还有其他），那么可以用下面的查询：

SELECT a, b + c FROM table1;

（假设b和c都是数字数据类型）。 参阅第 4.3 节 “选择列表获取更多细节。

FROM table1是一种非常简单的表表达式：它只读取了一个表。通常，表表达式可以是基本表、连接和子查询组成的复杂结构。 但也可以省略整个表表达式而把SELECT命令当做一个计算器：

SELECT 3 * 4;

如果选择列表里的表达式返回变化的结果，那么这就更有用了。例如，可以用这种方法调用函数：

SELECT random();

4.2. 表表达式

表表达式计算一个表。该表表达式包含一个FROM子句，该子句后面可以根据需要选用WHERE、GROUP BY和HAVING子句。最简单的表表达式只是引用磁盘上的一个表，一个所谓的基本表，但是我们可以用更复杂的表表达式以多种方法修改或组合基本表。

表表达式里可选的WHERE、GROUP BY和HAVING子句指定一系列对源自FROM子句的表的转换操作。所有这些转换最后生成一个虚拟表，它提供行传递给选择列表计算查询的输出行。

4.2.1. FROM子句

“FROM 子句”一节从一个用逗号分隔的表引用列表中的一个或更多个其它表中生成一个表。

FROM *table_reference* [, *table_reference* [, ...]]

表引用可以是一个表名字（可能有模式限定）或者是一个生成的表， 例如子查询、一个JOIN结构或者这些东西的复杂组合。如果在FROM子句中引用了多于一个表， 那么它们被交叉连接（即构造它们的行的笛卡尔积，见下文）。FROM列表的结果是一个中间的虚拟表，该表可以进行由WHERE、GROUP BY和HAVING子句指定的转换，并最后生成全局的表表达式结果。

如果一个表引用是一个简单的表名字并且它是表继承层次中的父表，那么该表引用将产生该表和它的后代表中的行，除非在该表名字前面放上ONLY关键字。但是，这种引用只会产生出现在该命名表中的列 — 在子表中增加的列都会被忽略。

除了在表名前写ONLY，可以在表名后面写上*来显式地指定要包括所有的后代表。没有实际的理由再继续使用这种语法，因为搜索后代表现在总是默认行为。

4.2.1.1. 连接表

一个连接表是根据特定的连接类型的规则从两个其它表（真实表或生成表）中派生的表。目前支持内连接、外连接和交叉连接。一个连接表的一般语法是：

T1 join_type T2 [*join_condition*]

所有类型的连接都可以被链在一起或者嵌套：*T1*和*T2*都可以是连接表。在JOIN子句周围可以使用圆括号来控制连接顺序。如果不使用圆括号，JOIN子句会从左至右嵌套。

连接类型

交叉连接

T1 CROSS JOIN *T2*

对来自于*T1*和*T2*的行的每一种可能的组合（即笛卡尔积），连接表将包含这样一行：它由所有*T1*里面的列后面跟着所有*T2*里面的列构成。如果两个表分别有 N 和 M 行，连接表将有 N * M 行。

FROM *T1* CROSS JOIN *T2*等效于FROM *T1* INNER JOIN *T2* ON TRUE（见下文）。它也等效于FROM *T1*,*T2*。

> **注意**
>
> 当多于两个表出现时，后一种等效并不严格成立，因为JOIN比逗号绑得更紧。例如FROM *T1* CROSS JOIN *T2* INNER JOIN *T3* ON *condition*和FROM *T1*,*T2* INNER JOIN *T3* ON *condition*并不完全相同，因为第一种情况中的*condition*可以引用*T1*，但在第二种情况中却不行。

条件连接

T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN *T2* ON *boolean_expression*
T1 { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN *T2* USING (*join column list*)
T1 NATURAL { [INNER] | { LEFT | RIGHT | FULL } [OUTER] } JOIN *T2*

INNER和OUTER对所有连接形式都是可选的。INNER是缺省；LEFT、RIGHT和FULL指示一个外连接。

连接条件在ON或USING子句中指定， 或者用关键字NATURAL隐含地指定。连接条件决定来自两个源表中的哪些行是“匹配”的，这些我们将在后文详细解释。

可能的条件连接类型是：

INNER JOIN

对于 T1 的每一行 R1，生成的连接表都有一行对应 T2 中的每一个满足和 R1 的连接条件的行。

LEFT OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。因此，生成的连接表里为来自 T1 的每一行都至少包含一行。

RIGHT OUTER JOIN

首先，执行一次内连接。然后，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。因此，生成的连接表里为来自 T2 的每一行都至少包含一行。

FULL OUTER JOIN

首先，执行一次内连接。然后，为 T1 中每一个无法在连接条件上匹配 T2 里任何一行的行返回一个连接行，该连接行中 T2 的列用空值补齐。同样，为 T2 中每一个无法在连接条件上匹配 T1 里任何一行的行返回一个连接行，该连接行中 T1 的列用空值补齐。

ON子句是最常见的连接条件的形式：它接收一个和WHERE子句里用的一样的布尔值表达式。 如果两个分别来自*T1*和*T2*的行在ON表达式上运算的结果为真，那么它们就算是匹配的行。

USING是个缩写符号，它允许利用特殊的情况：连接的两端都具有相同的连接列名。它接受共享列名的一个逗号分隔列表，并且为其中每一个共享列构造一个包含等值比较的连接条件。例如用USING (a, b)连接*T1*和*T2*会产生连接条件ON *T1*.a = *T2*.a AND *T1*.b = *T2*.b。

更进一步，JOIN USING的输出会废除冗余列：不需要把匹配上的列都打印出来，因为它们必须具有相等的值。不过JOIN ON会先产生来自*T1*的所有列，后面跟上所有来自*T2*的列；而JOIN USING会先为列出的每一个列对产生一个输出列，然后先跟上来自*T1*的剩余列，最后跟上来自*T2*的剩余列。

最后，NATURAL是USING的缩写形式：它形成一个USING列表， 该列表由那些在两个表里都出现了的列名组成。和USING一样，这些列只在输出表里出现一次。如果不存在公共列，NATURAL JOIN的行为将和JOIN ... ON TRUE一样产生交叉集连接。

> **注意**
>
> USING对于连接关系中的列改变是相当安全的，因为只有被列出的列会被组合成连接条件。NATURAL的风险更大，因为如果其中一个关系的模式

> 改变会导致出现一个新的匹配列名，就会导致连接将新列也组合成连接条件。

为了解释这些问题，假设我们有一个表t1：

```
 num | name
-----+------
   1 | a
   2 | b
   3 | c
```

和t2：

```
 num | value
-----+-------
   1 | xxx
   3 | yyy
   5 | zzz
```

然后我们用不同的连接方式可以获得各种结果：

```
=> SELECT * FROM t1 CROSS JOIN t2;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   1 | a    |   3 | yyy
   1 | a    |   5 | zzz
   2 | b    |   1 | xxx
   2 | b    |   3 | yyy
   2 | b    |   5 | zzz
   3 | c    |   1 | xxx
   3 | c    |   3 | yyy
   3 | c    |   5 | zzz
(9 rows)

=> SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
   1 | a    |   1 | xxx
   3 | c    |   3 | yyy
(2 rows)

=> SELECT * FROM t1 INNER JOIN t2 USING (num);
 num | name | value
-----+------+-------
   1 | a    | xxx
   3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 NATURAL INNER JOIN t2;
 num | name | value
```

```
-----+------+-------
  1 | a    | xxx
  3 | c    | yyy
(2 rows)

=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
  1 | a    |   1 | xxx
  2 | b    |     |
  3 | c    |   3 | yyy
(3 rows)

=> SELECT * FROM t1 LEFT JOIN t2 USING (num);
 num | name | value
-----+------+-------
  1 | a    | xxx
  2 | b    |
  3 | c    | yyy
(3 rows)

=> SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
  1 | a    |   1 | xxx
  3 | c    |   3 | yyy
    |      |   5 | zzz
(3 rows)

=> SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
 num | name | num | value
-----+------+-----+-------
  1 | a    |   1 | xxx
  2 | b    |     |
  3 | c    |   3 | yyy
    |      |   5 | zzz
(4 rows)
```

用ON指定的连接条件也可以包含与连接不直接相关的条件。这种功能可能对某些查询很有用，但是需要我们仔细想清楚。例如：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num AND t2.value = 'xxx';
 num | name | num | value
-----+------+-----+-------
  1 | a    |   1 | xxx
  2 | b    |     |
  3 | c    |     |
(3 rows)
```

注意把限制放在WHERE子句中会产生不同的结果：

```
=> SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num WHERE t2.value = 'xxx';
 num | name | num | value
```

```
-----+------+-----+-------
  1 | a    |   1 | xxx
(1 row)
```

这是因为放在ON子句中的一个约束在连接之前被处理，而放在WHERE子句中的一个约束是在连接之后被处理。这对内连接没有关系，但是对于外连接会带来麻烦。

4.2.1.2. 表和列别名

可以给一个表或复杂的表引用指定一个临时的名字，用于剩下的查询中引用那些派生的表。这被叫做*表别名*。

要创建一个表别名，我们可以写：

FROM *table_reference* AS *alias*

或者

FROM *table_reference alias*

AS关键字是可选的。别名可以是任意标识符。

表别名的典型应用是给长表名赋予比较短的标识符， 好让连接子句更易读。例如：

SELECT * FROM some_very_long_table_name s JOIN another_fairly_long_name a ON s.id = a.num;

到这里，别名成为当前查询的表引用的新名称 — 我们不再能够用该表最初的名字引用它了。因此，下面的用法是不合法的：

SELECT * FROM my_table AS m WHERE my_table.a > 5; -- 错误

表别名主要用于简化符号，但是当把一个表连接到它自身时必须使用别名，例如：

SELECT * FROM people AS mother JOIN people AS child ON mother.id = child.mother_id;

此外，如果一个表引用是一个子查询，则必须要使用一个别名（见第 4.2.1.3 节 “子查询”）。

圆括弧用于解决歧义。在下面的例子中，第一个语句将把别名b赋给my_table的第二个实例，但是第二个语句把别名赋给连接的结果：

```
SELECT * FROM my_table AS a CROSS JOIN my_table AS b ...
SELECT * FROM (my_table AS a CROSS JOIN my_table) AS b ...
```

另外一种给表指定别名的形式是给表的列赋予临时名字，就像给表本身指定别名一样：

FROM *table_reference* [AS] *alias* (*column1* [, *column2* [, ...]])

如果指定的列别名比表里实际的列少，那么剩下的列就没有被重命名。这种语法对于自连接或子查询特别有用。

如果用这些形式中的任何一种给一个JOIN子句的输出附加了一个别名， 那么该别名就在JOIN的作用下隐去了其原始的名字。例如：

```
SELECT a.* FROM my_table AS a JOIN your_table AS b ON ...
```

是合法 SQL，但是：

```
SELECT a.* FROM (my_table AS a JOIN your_table AS b ON ...) AS c
```

是不合法的：表别名a在别名c外面是看不到的。

4.2.1.3. 子查询

子查询指定了一个派生表，它必须被包围在圆括弧里并且必须被赋予一个表别名（参阅第 4.2.1.2 节 “表和列别名”）。例如：

```
FROM (SELECT * FROM table1) AS alias_name
```

这个例子等效于FROM table1 AS alias_name。更有趣的情况是在子查询里面有分组或聚集的时候， 子查询不能被简化为一个简单的连接。

一个子查询也可以是一个VALUES列表：

```
FROM (VALUES ('anne', 'smith'), ('bob', 'jones'), ('joe', 'blow'))
    AS names(first, last)
```

再次的，这里要求一个表别名。为VALUES列表中的列分配别名是可选的，但是选择这样做是一个好习惯。更多信息可参见第 4.7 节 “VALUES列表”。

4.2.1.4. 表函数

表函数是那些生成一个行集合的函数，这个集合可以是由基本数据类型（标量类型）组成， 也可以是由复合数据类型（表行）组成。它们的用法类似一个表、视图或者在查询的FROM子句里的子查询。表函数返回的列可以像一个表列、视图或者子查询那样被包含
在SELECT、JOIN或WHERE子句里。

也可以使用ROWS FROM语法将平行列返回的结果组合成表函数； 这种情况下结果行的数量是最大一个函数结果的数量，较小的结果会用空值来填充。

```
function_call [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
ROWS FROM( function_call [, ... ] ) [WITH ORDINALITY] [[AS] table_alias [(column_alias [, ... ])]]
```

如果指定了WITH ORDINALITY子句，一个额外的 bigint类型的列将会被增加到函数的结果列中。这个列对 函数结果集的行进行编号，编号从 1 开始（这是对 SQL 标准语法 UNNEST ... WITH ORDINALITY的一般化）。默认情 况下，序数列被称为ordinality，但也可以通过使用一个 AS子句给它分配一个不同的列名。

调用特殊的表函数UNNEST可以使用任意数量的数组参数， 它会返回对应的列数，就好像在每一个参数上单独调用 UNNEST（第 6.18 节 “数组函数和操作符”）并且使用 ROWS FROM结构把它们组合起来。

UNNEST(*array_expression* [, ...]) [WITH ORDINALITY] [[AS] *table_alias* [(*column_alias* [, ...])]]

如果没有指定*table_alias*，该函数名将被用作 表名。在ROWS FROM()结构的情况中，会使用第一个函数名。

如果没有提供列的别名，那么对于一个返回基数据类型的函数，列名也与该函数 名相同。对于一个返回组合类型的函数，结果列会从该类型的属性得到名称。

例子：

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);

CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
   SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

SELECT * FROM foo
   WHERE foosubid IN (
                     SELECT foosubid
                     FROM getfoo(foo.fooid) z
                     WHERE z.fooid = foo.fooid
                   );

CREATE VIEW vw_getfoo AS SELECT * FROM getfoo(1);

SELECT * FROM vw_getfoo;
```

有时侯，定义一个能够根据它们被调用方式返回不同列集合的表函数是很有用的。为了支持这些，表函数可以被声明为返回伪类型record。 如果在查询里使用这样的函数，那么我们必须在查询中指定所预期的行结构，这样系统才知道如何分析和规划该查询。这种语法是这样的：

```
function_call [AS] alias (column_definition [, ... ])
function_call AS [alias] (column_definition [, ... ])
ROWS FROM( ... function_call AS (column_definition [, ... ]) [, ... ] )
```

在没有使用ROWS FROM()语法时， *column_definition*列表会取代无法附着在 FROM项上的列别名列表，列定义中的名称就起到列别名的作用。 在使用ROWS FROM()语法时， 可以为每一个成员函数单独附着一个 *column_definition*列表；或者在只有一个成员 函数并且没有WITH ORDINALITY子句的情况下，可以在 ROWS FROM()后面写一个 *column_definition*列表来取代一个列别名列表。

考虑下面的例子：

```
SELECT *
   FROM dblink('dbname=mydb', 'SELECT proname, prosrc FROM ux_proc')
     AS t1(proname name, prosrc text)
   WHERE proname LIKE 'bytea%';
```

dblink函数（dblink模块的一部分）执行一个远程的查询。它被声明为返回record，因为它可能会被用于任何类型的查询。 实际的列集必须在调用它的查询中指定，这样分析器才知道类似*这样的东西应该扩展成什么样子。

4.2.1.5. LATERAL子查询

可以在出现于FROM中的子查询前放置关键词LATERAL。这允许它们引用前面的FROM项提供的列（如果没有LATERAL，每一个子查询将被独立计算，并且因此不能被其他FROM项交叉引用）。

出现在FROM中的表函数的前面也可以被放上关键词LATERAL，但对于函数该关键词是可选的，在任何情况下函数的参数都可以包含对前面的FROM项提供的列的引用。

一个LATERAL项可以出现在FROM列表顶层，或者出现在一个JOIN树中。在后一种情况下，如果它出现在JOIN的右部，那么它也可以引用在JOIN左部的任何项。

如果一个FROM项包含LATERAL交叉引用，计算过程如下：对于提供交叉引用列的FROM项的每一行，或者多个提供这些列的多个FROM项的行集合，LATERAL项将被使用该行或者行集中的列值进行计算。得到的结果行将和它们被计算出来的行进行正常的连接。对于来自这些列的源表的每一行或行集，该过程将重复。

LATERAL的一个简单例子：

```
SELECT * FROM foo, LATERAL (SELECT * FROM bar WHERE bar.id = foo.bar_id) ss;
```

这不是非常有用，因为它和一种更简单的形式得到的结果完全一样：

```
SELECT * FROM foo, bar WHERE bar.id = foo.bar_id;
```

在必须要使用交叉引用列来计算那些即将要被连接的行时，LATERAL是最有用的。一种常用的应用是为一个返回集合的函数提供一个参数值。例如，假设vertices(polygon)返回一个多边形的顶点集合，我们可以这样标识存储在一个表中的多边形中靠近的顶点：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1, polygons p2,
    LATERAL vertices(p1.poly) v1,
    LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

这个查询也可以被写成：

```
SELECT p1.id, p2.id, v1, v2
FROM polygons p1 CROSS JOIN LATERAL vertices(p1.poly) v1,
    polygons p2 CROSS JOIN LATERAL vertices(p2.poly) v2
WHERE (v1 <-> v2) < 10 AND p1.id != p2.id;
```

或者写成其他几种等价的公式（正如以上提到的，LATERAL关键词在这个例子中并不是必不可少的，但是我们在这里使用它是为了使表述更清晰）。

有时候也会很特别地把LEFT JOIN放在一个LATERAL子查询的前面，这样即使LATERAL子查询对源行不产生行，源行也会出现在结果中。例如，如果get_product_names()返回一个制造商制造的产品的名字，但是某些制造商在我们的表中目前没有制造产品，我们可以找出哪些制造商是这样：

```
SELECT m.name
FROM manufacturers m LEFT JOIN LATERAL get_product_names(m.id) pname ON true
WHERE pname IS NULL;
```

4.2.2. WHERE子句

“WHERE 子句”一节的语法是

WHERE *search_condition*

这里的*search_condition*是任意返回一个boolean类型值的值表达式（参阅第 1.2 节 “值表达式”）。

在完成对FROM子句的处理之后，生成的虚拟表的每一行都会对根据搜索条件进行检查。 如果该条件的结果是真，那么该行被保留在输出表中；否则（也就是说，如果结果是假或空）就把它抛弃。搜索条件通常至少要引用一些在FROM子句里生成的列；虽然这不是必须的，但如果不引用这些列，那么WHERE子句就没什么用了。

> **注意**
>
> 内连接的连接条件既可以写在WHERE子句也可以写在JOIN子句里。例如，这些表表达式是等效的：
>
> FROM a, b WHERE a.id = b.id AND b.val > 5
>
> 和：
>
> FROM a INNER JOIN b ON (a.id = b.id) WHERE b.val > 5
>
> 或者可能还有：
>
> FROM a NATURAL JOIN b WHERE b.val > 5
>
> 想用哪个只是一个风格问题。FROM子句里的JOIN语法可能不那么容易移植到其它SQL数据库管理系统中。 对于外部连接而言没有选择：它们必须在FROM子句中完成。 外部连接的ON或USING子句不等于WHERE条件，因为它导致最终结果中行的增加（对那些不匹配的输入行）和减少。

这里是一些WHERE子句的例子：

```
SELECT ... FROM fdt WHERE c1 > 5

SELECT ... FROM fdt WHERE c1 IN (1, 2, 3)

SELECT ... FROM fdt WHERE c1 IN (SELECT c1 FROM t2)

SELECT ... FROM fdt WHERE c1 IN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10)
```

SELECT ... FROM fdt WHERE c1 BETWEEN (SELECT c3 FROM t2 WHERE c2 = fdt.c1 + 10) AND 100

SELECT ... FROM fdt WHERE EXISTS (SELECT c1 FROM t2 WHERE c2 > fdt.c1)

在上面的例子里，fdt是从FROM子句中派生的表。 那些不符合WHERE子句的搜索条件的行会被从fdt中删除。请注意我们把标量子查询当做一个值表达式来用。 和任何其它查询一样，子查询里可以使用复杂的表表达式。同时还请注意fdt在子查询中也被引用。只有在c1也是作为子查询输入表的生成表的列时，才必须把c1限定成fdt.c1。但限定列名字可以增加语句的清晰度，即使有时候不是必须的。这个例子展示了一个外层查询的列名范围如何扩展到它的内层查询。

4.2.3. GROUP BY和HAVING子句

在通过了WHERE过滤器之后，生成的输入表可以使用GROUP BY子句进行分组，然后用HAVING子句删除一些分组行。

SELECT *select_list*
  FROM ...
  [WHERE ...]
  GROUP BY *grouping_column_reference* [, *grouping_column_reference*]...

“GROUP BY 子句”一节被用来把表中在所列出的列上具有相同值的行分组在一起。 这些列的列出顺序并没有什么关系。其效果是把每组具有相同值的行组合为一个组行，它代表该组里的所有行。 这样就可以删除输出里的重复和/或计算应用于这些组的聚集。例如：

=> **SELECT * FROM test1;**
 x | y
---+---
 a | 3
 c | 2
 b | 5
 a | 1
(4 rows)

=> **SELECT x FROM test1 GROUP BY x;**
 x
---
 a
 b
 c
(3 rows)

在第二个查询里，我们不能写成SELECT * FROM test1 GROUP BY x， 因为列y里没有哪个值可以和每个组相关联起来。被分组的列可以在选择列表中引用是因为它们在每个组都有单一的值。

通常，如果一个表被分了组，那么没有在GROUP BY中列出的列都不能被引用，除非在聚集表达式中被引用。 一个用聚集表达式的例子是：

=> **SELECT x, sum(y) FROM test1 GROUP BY x;**
 x | sum
---+-----

```
 a |  4
 b |  5
 c |  2
(3 rows)
```

这里的sum是一个聚集函数，它在整个组上计算出一个单一值。有关可用的聚集函数的更多信息可以在第 6.20 节 “聚集函数”。

> **提示**
>
> 没有聚集表达式的分组实际上计算了一个列中可区分值的集合。我们也可以用DISTINCT子句实现（参阅第 4.3.3 节 DISTINCT”）。

这里是另外一个例子：它计算每种产品的总销售额（而不是所有产品的总销售额）：

```
SELECT product_id, p.name, (sum(s.units) * p.price) AS sales
    FROM products p LEFT JOIN sales s USING (product_id)
    GROUP BY product_id, p.name, p.price;
```

在这个例子里，列product_id、p.name和p.price必须在GROUP BY子句里， 因为它们都在查询的选择列表里被引用到（但见下文）。列s.units不必在GROUP BY列表里，因为它只是在一个聚集表达式（sum(...)）里使用，它代表一组产品的销售额。对于每种产品，这个查询都返回一个该产品的所有销售额的总和行。

如果产品表被建立起来，例如product_id是主键，那么在上面的例子中用product_id来分组就够了，因为名称和价格都是函数依赖于产品ID，并且关于为每个产品ID分组返回哪个名称和价格值就不会有歧义。

在严格的 SQL 里，GROUP BY只能对源表的列进行分组，但UXDB把这个扩展为也允许GROUP BY去根据选择列表中的列分组。也允许对值表达式进行分组，而不仅是简单的列名。

如果一个表已经用GROUP BY子句分了组，然后又只对其中的某些组感兴趣， 那么就可以用HAVING子句，它很象WHERE子句，用于从结果中删除一些组。其语法是：

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression
```

在HAVING子句中的表达式可以引用分组的表达式和未分组的表达式（后者必须涉及一个聚集函数）。

例子：

```
=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
 x | sum
---+-----
 a |  4
 b |  5
(2 rows)

=> SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
 x | sum
---+-----
 a |  4
```

```
 b |  5
(2 rows)
```

再次，一个更现实的例子：

```
SELECT product_id, p.name, (sum(s.units) * (p.price - p.cost)) AS profit
    FROM products p LEFT JOIN sales s USING (product_id)
    WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
    GROUP BY product_id, p.name, p.price, p.cost
    HAVING sum(p.price * s.units) > 5000;
```

在上面的例子里，WHERE子句用那些非分组的列选择数据行（表达式只是对那些最近四周发生的销售为真）。 而HAVING子句限制输出为总销售收入超过 5000 的组。请注意聚集表达式不需要在查询中的所有地方都一样。

如果一个查询包含聚集函数调用，但是没有GROUP BY子句，分组仍然会发生：结果是一个单一行（或者根本就没有行，如果该单一行被HAVING所消除）。它包含一个HAVING子句时也是这样，即使没有任何聚集函数调用或者GROUP BY子句。

4.2.4. GROUPING SETS、CUBE和ROLLUP

使用分组集的概念可以实现比上述更加复杂的分组操作。由 FROM和WHERE子句选出的数据被按照每一个指定 的分组集单独分组，按照简单GROUP BY子句对每一个分组计算 聚集，然后返回结果。例如：

```
=> SELECT * FROM items_sold;
 brand | size | sales
-------+------+-------
 Foo   | L    |  10
 Foo   | M    |  20
 Bar   | M    |  15
 Bar   | L    |  5
(4 rows)

=> SELECT brand, size, sum(sales) FROM items_sold GROUP BY GROUPING SETS ((brand),
 (size), ());
 brand | size | sum
-------+------+-----
 Foo   |      |  30
 Bar   |      |  20
       | L    |  15
       | M    |  35
       |      |  50
(5 rows)
```

GROUPING SETS的每一个子列表可以指定一个或者多个列或者表达式， 它们将按照直接出现在GROUP BY子句中同样的方式被解释。一个空的 分组集表示所有的行都要被聚集到一个单一分组（即使没有输入行存在也会被输出） 中，这就像前面所说的没有GROUP BY子句的聚集函数的情况一样。

对于分组列或表达式没有出现在其中的分组集的结果行，对分组列或表达式的引用会 被空值所替代。要区分一个特定的输出行来自于哪个分组，请见 表 6.66 “分组操作”。

UXDB 中提供了一种简化方法来指定两种常用类型的分组集。下面形式的子句

ROLLUP (*e1*, *e2*, *e3*, ...)

表示给定的表达式列表及其所有前缀（包括空列表），因此它等效于

```
GROUPING SETS (
   ( e1, e2, e3, ... ),
   ...
   ( e1, e2 ),
   ( e1 ),
   ( )
)
```

这通常被用来分析历史数据，例如按部门、区和公司范围计算的总薪水。

下面形式的子句

CUBE (*e1*, *e2*, ...)

表示给定的列表及其可能的子集（即幂集）。因此

CUBE (a, b, c)

等效于

```
GROUPING SETS (
   ( a, b, c ),
   ( a, b    ),
   ( a,    c ),
   ( a       ),
   (    b, c ),
   (    b    ),
   (       c ),
   (         )
)
```

CUBE或ROLLUP子句中的元素可以是表达式或者 圆括号中的元素子列表。在后一种情况中，对于生成分组集的目的来说，子列 表被当做单一单元来对待。例如：

CUBE ((a, b), (c, d))

等效于

```
GROUPING SETS (
   ( a, b, c, d ),
   ( a, b       ),
   (       c, d ),
   (            )
```

)

并且

ROLLUP (a, (b, c), d)

等效于

```
GROUPING SETS (
   ( a, b, c, d ),
   ( a, b, c    ),
   ( a          ),
   (            )
)
```

CUBE和ROLLUP可以被直接用在 GROUP BY子句中，也可以被嵌套在一个 GROUPING SETS子句中。如果一个 GROUPING SETS子句被嵌套在另一个同类子句中， 效果和把内层子句的所有元素直接写在外层子句中一样。

如果在一个GROUP BY子句中指定了多个分组项，那么最终的 分组集列表是这些项的叉积。例如：

GROUP BY a, CUBE (b, c), GROUPING SETS ((d), (e))

等效于

```
GROUP BY GROUPING SETS (
   (a, b, c, d), (a, b, c, e),
   (a, b, d),    (a, b, e),
   (a, c, d),    (a, c, e),
   (a, d),       (a, e)
)
```

> **注意**
>
> 在表达式中，结构(a, b)通常被识别为一个 a 行构造器。在 GROUP BY子句中，这不会在表达式的顶层应用，并且 (a, b)会按照上面所说的被解析为一个表达式的列表。如果出于 某种原因在分组表达式中需要一个行构造器，请使用 ROW(a, b)。

4.2.5. 窗口函数处理

如果查询包含任何窗口函数（见第 6.21 节 “窗口函数”和第 1.2.8 节 “窗口函数调用”，这些函数将在任何分组、聚集和HAVING过滤被执行之后被计算。也就是说如果查询使用了任何聚集、GROUP BY或HAVING，则窗口函数看到的行是分组行而不是来自于FROM/WHERE的原始表行。

当多个窗口函数被使用，所有在窗口定义中有句法上等效的PARTITION BY和ORDER BY子句的窗口函数被保证在数据上的同一趟扫描中计算。因此它们将会看到相同的排序顺序，即使ORDER BY没有唯一地决定一个顺序。但是，对于具有不同PARTITION BY或ORDER BY定义

的函数的计算没有这种保证（在这种情况中，在多个窗口函数计算之间通常要求一个排序步骤，并且并不保证保留行的顺序，即使它的ORDER BY把这些行视为等效的）。

目前，窗口函数总是要求排序好的数据，并且这样查询的输出总是被根据窗口函数的PARTITION BY/ORDER BY子句的一个或者另一个排序。但是，我们不推荐依赖于此。如果希望确保结果以特定的方式排序，请显式使用顶层的ORDER BY子句。

4.3. 选择列表

如前面的小节说明的那样， 在SELECT命令里的表表达式构造了一个中间的虚拟表， 方法可能有组合表、视图、消除行、分组等等。这个表最后被选择列表传递下去处理。选择列表判断中间表的哪个列是实际输出。

4.3.1. 选择列表项

最简单的选择列表类型是*，它发出表表达式生成的所有列。否则，一个选择列表是一个逗号分隔的值表达式的列表（和在第 1.2 节 “值表达式里定义的一样）。 例如，它可能是一个列名的列表：

```
SELECT a, b, c FROM ...
```

列名字a、b和c要么是在FROM子句里引用的表中列的实际名字，要么是像第 4.2.1.2 节 “表和列别名” 里解释的那样的别名。在选择列表里可用的名字空间和在WHERE子句里的一样， 除非使用了分组，这时候它和HAVING子句一样。

如果超过一个表有同样的列名，那么还必须给出表名字，如：

```
SELECT tbl1.a, tbl2.a, tbl1.b FROM ...
```

在使用多个表时，要求一个特定表的所有列也是有用的：

```
SELECT tbl1.*, tbl2.a FROM ...
```

更多有关*table_name*.*记号的内容请参考第 5.16.5 节 “在查询中使用组合类型”

如果将任意值表达式用于选择列表，那么它在概念上向返回的表中增加了一个新的虚拟列。 值表达式为结果的每一行进行一次计算，对任何列引用替换行的值。 不过选择列表中的这个表达式并非一定要引用来自FROM子句中表表达式里面的列，例如它也可以是任意常量算术表达式。

4.3.2. 列标签

选择列表中的项可以被赋予名字，用于进一步的处理。 例如为了在一个ORDER BY子句中使用或者为了客户端应用显示。例如：

```
SELECT a AS value, b + c AS sum FROM ...
```

如果没有使用AS指定输出列名，那么系统会分配一个缺省的列名。对于简单的列引用， 它是被引用列的名字。对于函数调用，它是函数的名字。对于复杂表达式，系统会生成一个通用的名字。

只有在新列无法匹配任何UXDB关键词（见附录 A,）时，AS关键词是可选的。为了避免一个关键字的意外匹配，可以使用双引号来修饰列名。例如，VALUE是一个关键字，所以下面的语句不会工作：

SELECT a value, b + c AS sum FROM ...

但是这个可以：

SELECT a "value", b + c AS sum FROM ...

为了防止未来可能的关键词增加，我们推荐总是写AS或者用双引号修饰输出列名。

> **注意**
>
> 输出列的命名和在FROM子句里的命名是不一样的 （参阅第 4.2.1.2 节“表和列别名”）。 它实际上允许对同一个列命名两次，但是在选择列表中分配的名字是要传递下去的名字。

4.3.3. DISTINCT

在处理完选择列表之后，结果表可以可选的删除重复行。我们可以直接在SELECT后面写上DISTINCT关键字来指定：

SELECT DISTINCT *select_list* ...

（如果不用DISTINCT可以用ALL关键词来指定获得的所有行的缺省行为）。

显然，如果两行里至少有一个列有不同的值，那么我们认为它是可区分的。空值在这种比较中被认为是相同的。

另外，我们还可以用任意表达式来判断什么行可以被认为是可区分的：

SELECT DISTINCT ON (*expression* [, *expression* ...]) *select_list* ...

这里*expression*是任意值表达式，它为所有行计算。如果一个行集合里所有表达式的值是一样的， 那么我们认为它们是重复的并且因此只有第一行保留在输出中。请注意这里的一个集合的“第一行”是不可预料的， 除非在足够多的列上对该查询排了序，保证到达DISTINCT过滤器的行的顺序是唯一的（DISTINCT ON处理是发生在ORDER BY排序后面的）。

DISTINCT ON子句不是 SQL 标准的一部分， 有时候有人认为它是一个糟糕的风格，因为它的结果是不可判定的。 如果有选择的使用GROUP BY和在FROM中的子查询，那么我们可以避免使用这个构造， 但是通常它是更方便的候选方法。

4.4. 组合查询

两个查询的结果可以用集合操作并、交、差进行组合。语法是

query1 UNION [ALL] *query2*
query1 INTERSECT [ALL] *query2*
query1 EXCEPT [ALL] *query2*

*query1*和*query2*都是可以使用以上所有特性的查询。集合操作也可以嵌套和级连，例如

query1 UNION *query2* UNION *query3*

实际执行的是：

(*query1* UNION *query2*) UNION *query3*

UNION有效地把*query2*的结果附加到*query1*的结果上（不过我们不能保证这就是这些行实际被返回的顺序）。此外，它将删除结果中所有重复的行，　　就象DISTINCT做的那样，除非使用了UNION ALL。

INTERSECT返回那些同时存在于*query1*和*query2*的结果中的行，除非声明了INTERSECT　ALL，否则所有重复行都被消除。

EXCEPT返回所有在*query1*的结果中但是不在*query2*的结果中的行（有时侯这叫做两个查询的差）。同样的，除非声明了EXCEPT ALL，否则所有重复行都被消除。

为了计算两个查询的并、交、差，这两个查询必须是“并操作兼容的”，也就意味着它们都返回同样数量的列，　并且对应的列有兼容的数据类型，如第　7.5　节　“UNION、CASE和相关结构”中描述的那样。

4.5.　行排序

在一个查询生成一个输出表之后（在处理完选择列表之后），还可以选择性地对它进行排序。如果没有选择排序，那么行将以未指定的顺序返回。　这时候的实际顺序将取决于扫描和连接计划类型以及行在磁盘上的顺序，但是肯定不能依赖这些东西。一种特定的顺序只能在显式地选择了排序步骤之后才能被保证。

ORDER BY子句指定了排序顺序：

```
SELECT select_list
  FROM table_expression
  ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
       [, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

排序表达式可以是任何在查询的选择列表中合法的表达式。一个例子是：

```
SELECT a, b FROM table1 ORDER BY a + b, c;
```

当多于一个表达式被指定，后面的值将被用于排序那些在前面值上相等的行。每一个表达式后可以选择性地放置一个ASC或DESC关键词来设置排序方向为升序或降序。ASC顺序是默认值。升序会把较小的值放在前面，而“较小”则由<操作符定义。相似地，降序则由>操作符定义。[1]

[1]　事实上，UXDB为表达式的数据类型使用默认B-tree操作符类来决定ASC和DESC的排序顺序。照惯例，数据类型将被建立，这样<和>操作符负责这个排序顺序，但是一个用户定义的数据类型的设计者可以选择做些不同的设置。

NULLS FIRST和NULLS LAST选项将可以被用来决定在排序顺序中，空值是出现在非空值之前或者出现在非空值之后。默认情况下，排序时空值被认为比任何非空值都要大，即NULLS FIRST是DESC顺序的默认值，而不是NULLS LAST的默认值。

注意顺序选项是对每一个排序列独立考虑的。例如ORDER BY x, y DESC表示ORDER BY x ASC, y DESC，而和ORDER BY x DESC, y DESC不同。

一个*sort_expression*也可以是列标签或者一个输出列的编号，如：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum;
SELECT a, max(b) FROM table1 GROUP BY a ORDER BY 1;
```

两者都根据第一个输出列排序。注意一个输出列的名字必须孤立，即它不能被用在一个表达式中 — 例如，这是不正确的：

```
SELECT a + b AS sum, c FROM table1 ORDER BY sum + c;        -- 错误
```

该限制是为了减少混淆。如果一个ORDER BY项是一个单一名字并且匹配一个输出列名或者一个表表达式的列，仍然会出现混淆。在这种情况中输出列将被使用。只有在使用AS来重命名一个输出列来匹配某些其他表列的名字时，这才会导致混淆。

ORDER BY可以被应用于UNION、INTERSECT或EXCEPT组合的结果，但是在这种情况中它只被允许根据输出列名或编号排序，而不能根据表达式排序。

4.6. LIMIT和OFFSET

LIMIT和OFFSET允许只检索查询剩余部分产生的行的一部分：

```
SELECT select_list
    FROM table_expression
    [ ORDER BY ... ]
    [ LIMIT { number | ALL } ] [ OFFSET number ]
[ OFFSET number] [ LIMIT {number | ALL }]
[ LIMIT number,rows]
```

如果给出了一个限制计数，那么会返回数量不超过该限制的行（但可能更少些，因为查询本身可能生成的行数就比较少）。LIMIT ALL的效果和省略LIMIT子句一样，就像是LIMIT带有 NULL 参数一样。

OFFSET说明在开始返回行之前忽略多少行。OFFSET 0的效果和省略OFFSET子句是一样的，并且LIMIT NULL的效果和省略LIMIT子句一样，就像是OFFSET带有 NULL 参数一样。

如果OFFSET和LIMIT都出现了， 那么在返回LIMIT个行之前要先忽略OFFSET行。

如果使用LIMIT，那么用一个ORDER BY子句把结果行约束成一个唯一的顺序是很重要的。否则就会拿到一个不可预料的该查询的行的子集。要的可能是第十到第二十行，但以什么顺序的第十到第二十？除非指定了ORDER BY，否则顺序是不知道的。

查询优化器在生成查询计划时会考虑LIMIT，因此如果给定LIMIT和OFFSET，那么很可能收到不同的规划（产生不同的行顺序）。因此，使用不同的LIMIT/OFFSET值选择查询结果的不同子

集将生成不一致的结果，除非用ORDER BY强制一个可预测的顺序。这并非bug， 这是一个很自然的结果，因为 SQL 没有许诺把查询的结果按照任何特定的顺序发出，除非用了ORDER BY来约束顺序。

被OFFSET子句忽略的行仍然需要在服务器内部计算；因此，一个很大的OFFSET的效率可能还是不够高。

分页查询语法在不同场景下查询数据。

- 场景一：LIMIT...OFFSET....

SELECT id, name, gender, score FROM students ORDER BY score DESC LIMIT 3 OFFSET 2;

```
 id | name | gender | score
----+------+--------+-------
  1 | 小明 | M      |    90
  9 | 小王 | M      |    89
  3 | 小军 | M      |    88
(3 rows)
```

- 场景二：LIMIT...，.....

SELECT id, name, gender, score FROM students ORDER BY score DESC LIMIT 2, 3;

```
 id | name | gender | score
----+------+--------+-------
  1 | 小明 | M      |    90
  9 | 小王 | M      |    89
  3 | 小军 | M      |    88
(3 rows)
```

- 场景二：OFFSET...LIMIT.....

SELECT id, name, gender, score FROM students ORDER BY score DESC OFFSET 2 LIMIT 3;

```
 id | name | gender | score
----+------+--------+-------
  1 | 小明 | M      |    90
  9 | 小王 | M      |    89
  3 | 小军 | M      |    88
(3 rows)
```

- 场景四：LIMIT 表达式

SELECT id, name, gender, score FROM students ORDER BY score DESC LIMIT 2+2 OFFSET 2;

```
id | name | gender | score
----+------+--------+-------
  1 | 小明 | M      |    90
  9 | 小王 | M      |    89
  3 | 小军 | M      |    88
 10 | 小丽 | F      |    88
```

(4 rows)

- 场景五：OFFSET...

SELECT id, name, gender, score FROM students ORDER BY score DESC OFFSET 2;

```
 id | name | gender | score
----+------+--------+-------
  1 | 小明 | M      |    90
  9 | 小王 | M      |    89
  3 | 小军 | M      |    88
 10 | 小丽 | F      |    88
  7 | 小林 | M      |    85
  5 | 小白 | F      |    81
  4 | 小米 | F      |    73
  6 | 小冰 | M      |    55
(8 rows)
```

4.7. VALUES列表

VALUES提供了一种生成“常量表”的方法，它可以被使用在一个查询中而不需要实际在磁盘上创建一个表。语法是：

VALUES (*expression* [, ...]) [, ...]

每一个被圆括号包围的表达式列表生成表中的一行。列表都必须具有相同数据的元素（即表中列的数目），并且在每个列表中对应的项必须具有可兼容的数据类型。分配给结果的每一列的实际数据类型使用和UNION相同的规则确定（参见第 7.5 节 “UNION、CASE和相关结构”）。

一个例子：

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

将会返回一个有两列三行的表。它实际上等效于：

```
SELECT 1 AS column1, 'one' AS column2
UNION ALL
SELECT 2, 'two'
UNION ALL
SELECT 3, 'three';
```

在默认情况下，UXDB将column1、column2等名字分配给一个VALUES表的列。这些列名不是由SQL标准指定的，并且不同的数据库系统的做法也不同，因此通常最好使用表别名列表来重写这些默认的名字，像这样：

```
=> SELECT * FROM (VALUES (1, 'one'), (2, 'two'), (3, 'three')) AS t (num,letter);
 num | letter
-----+--------
   1 | one
   2 | two
```

```
  3 | three
(3 rows)
```

在句法上，后面跟随着表达式列表的VALUES列表被视为和

SELECT *select_list* FROM *table_expression*

一样，并且可以出现在SELECT能出现的任何地方。例如，可以把它用作UNION的一部分，或者附加一个*sort_specification*（ORDER BY、LIMIT和/或OFFSET）给它。VALUES最常见的用途是作为一个INSERT命令的数据源，以及作为一个子查询。

更多信息请见VALUES(7)。

4.8. WITH查询（公共表表达式）

WITH提供了一种方式来书写在一个大型查询中使用的辅助语句。这些语句通常被称为公共表表达式或CTE，它们可以被看成是定义只在一个查询中存在的临时表。在WITH子句中的每一个辅助语句可以是一个SELECT、INSERT、UPDATE或DELETE，并且WITH子句本身也可以被附加到一个主语句，主语句也可以是SELECT、INSERT、UPDATE或DELETE。

4.8.1. WITH中的SELECT

WITH中SELECT的基本价值是将复杂的查询分解称为简单的部分。一个例子：

```
WITH regional_sales AS (
   SELECT region, SUM(amount) AS total_sales
   FROM orders
   GROUP BY region
), top_regions AS (
   SELECT region
   FROM regional_sales
   WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
      product,
      SUM(quantity) AS product_units,
      SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

它只显示在高销售区域每种产品的销售总额。WITH子句定义了两个辅助语句regional_sales和top_regions，其中regional_sales的输出用在top_regions中而top_regions的输出用在主SELECT查询。这个例子可以不用WITH来书写，但是我们必须要用两层嵌套的子SELECT。使用这种方法要更简单些。

可选的RECURSIVE修饰符将WITH从单纯的句法便利变成了一种在标准SQL中不能完成的特性。通过使用RECURSIVE，一个WITH查询可以引用它自己的输出。一个非常简单的例子是计算从1到100的整数合的查询：

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

一个递归WITH查询的通常形式总是一个*非递归项*，然后是UNION（或者UNION ALL），再然后是一个*递归项*，其中只有递归项能够包含对于查询自身输出的引用。这样一个查询可以被这样执行：

过程 4.1. 递归查询求值

1. 计算非递归项。对UNION（但不对UNION ALL），抛弃重复行。把所有剩余的行包括在递归查询的结果中，并且也把它们放在一个临时的*工作表*中。

2. 只要工作表不为空，重复下列步骤：

   a. 计算递归项，用当前工作表的内容替换递归自引用。对UNION（不是UNION ALL），抛弃重复行以及那些与之前结果行重复的行。将剩下的所有行包括在递归查询的结果中，并且也把它们放在一个临时的*中间表*中。

   b. 用中间表的内容替换工作表的内容，然后清空中间表。

> **注意**
>
> 严格来说，这个处理是迭代而不是递归，但是RECURSIVE是SQL标准委员会选择的术语。

在上面的例子中，工作表在每一步只有一个行，并且它在连续的步骤中取值从1到100。在第100步，由于WHERE子句导致没有输出，因此查询终止。

递归查询通常用于处理层次或者树状结构的数据。一个有用的例子是这个用于找到一个产品的直接或间接部件的查询，只要给定一个显示了直接包含关系的表：

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part, p.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

在使用递归查询时，确保查询的递归部分最终将不返回元组非常重要，否则查询将会无限循环。在某些时候，使用UNION替代UNION ALL可以通过抛弃与之前输出行重复的行来达到这个目的。不过，经常有循环不涉及到完全重复的输出行：它可能只需要检查一个或几个域来看相同点之前是否达到过。处理这种情况的标准方法是计算一个已经访问过值的数组。例如，考虑下面这个使用*link*域搜索表graph的查询：

```
WITH RECURSIVE search_graph(id, link, data, depth) AS (
    SELECT g.id, g.link, g.data, 1
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1
    FROM graph g, search_graph sg
    WHERE g.id = sg.link
)
SELECT * FROM search_graph;
```

如果*link*关系包含环，这个查询将会循环。因为我们要求一个“depth”输出，仅仅将UNION ALL 改为UNION不会消除循环。反过来在我们顺着一个特定链接路径搜索时，我们需要识别我们是否再次到达了一个相同的行。我们可以项这个有循环倾向的查询增加两个列*path*和*cycle*：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[g.id],
      false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || g.id,
      g.id = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

除了阻止环，数组值对于它们自己的工作显示到达任何特定行的“path”也有用。

在通常情况下如果需要检查多于一个域来识别一个环，请用行数组。例如，如果我们需要比较域*f1*和*f2*：

```
WITH RECURSIVE search_graph(id, link, data, depth, path, cycle) AS (
    SELECT g.id, g.link, g.data, 1,
      ARRAY[ROW(g.f1, g.f2)],
      false
    FROM graph g
  UNION ALL
    SELECT g.id, g.link, g.data, sg.depth + 1,
      path || ROW(g.f1, g.f2),
      ROW(g.f1, g.f2) = ANY(path)
    FROM graph g, search_graph sg
    WHERE g.id = sg.link AND NOT cycle
)
SELECT * FROM search_graph;
```

> **提示**
>
> 在通常情况下只有一个域需要被检查来识别一个环，可以省略ROW()语法。这允许使用一个简单的数组而不是一个组合类型数组，可以获得效率。

> **提示**
>
> 递归查询计算算法使用宽度优先搜索顺序产生它的输出。可以通过让外部查询ORDER BY一个以这种方法构建的“path”，用来以深度优先搜索顺序显示结果。

当不确定查询是否可能循环时，一个测试查询的有用技巧是在父查询中放一个LIMIT。例如，这个查询没有LIMIT时会永远循环：

```
WITH RECURSIVE t(n) AS (
    SELECT 1
  UNION ALL
    SELECT n+1 FROM t
)
SELECT n FROM t LIMIT 100;
```

这会起作用，因为UXDB的实现只计算WITH查询中被父查询实际取到的行。不推荐在生产中使用这个技巧，因为其他系统可能以不同方式工作。同样，如果让外层查询排序递归查询的结果或者把它们连接成某种其他表，这个技巧将不会起作用，因为在这些情况下外层查询通常将尝试取得WITH查询的所有输出。

WITH查询的一个有用的特性是在每一次父查询的执行中它们通常只被计算一次，即使它们被父查询或兄弟WITH查询引用了超过一次。 因此，在多个地方需要的昂贵计算可以被放在一个WITH查询中来避免冗余工作。另一种可能的应用是阻止不希望的多个函数计算产生副作用。但是，从另一方面来看，优化器不能将来自父查询的约束下推到乘法引用WITH查询，因为当他应该只影响一个时它可能会影响所有使用WITH查询的输出的使用。 乘法引用WITH查询通常将会被按照所写的方式计算，而不抑制父查询以后可能会抛弃的行（但是，如上所述，如果对查询的引用只请求有限数目的行，计算可能会提前停止）。

但是，如果 WITH 查询是非递归和边际效应无关的（就是说，它是一个SELECT包含没有可变函数），则它可以合并到父查询中，允许两个查询级别的联合优化。 默认情况下，这发生在如果父查询仅引用 WITH查询一次的时候，而不是它引用WITH查询多于一次时。 可以超越控制这个决策，通过指定 MATERIALIZED 来强制分开计算 WITH 查询，或者通过指定 NOT MATERIALIZED来强制它被合并到父查询中。 后一种选择存在重复计算WITH查询的风险，但它仍然能提供净节省，如果WITH查询的每个使用只需要WITH查询的完整输出的一小部分。

这些规则的一个简单示例是

```
WITH w AS (
    SELECT * FROM big_table
)
SELECT * FROM w WHERE key = 123;
```

这个 WITH 查询将被合并，生成相同的执行计划为

```
SELECT * FROM big_table WHERE key = 123;
```

特别是，如果在*key*上有一个索引，它可能只用于获取具有 key = 123的行。 另一方面，在

```
WITH w AS (
    SELECT * FROM big_table
```

```
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

WITH查询将被物化，生成一个big_table的临时拷贝，然后与其自身 — 联合，这样将不能从索引上获得任何好处。 如果写成下面的形式，这个查询将被执行得更有效率。

```
WITH w AS NOT MATERIALIZED (
   SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

所以父查询的限制可以直接应用于big_table的扫描。

一个NOT MATERIALIZED 可能不理想的例子为

```
WITH w AS (
   SELECT key, very_expensive_function(val) as f FROM some_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.f = w2.f;
```

在这里，WITH查询的物化确保very_expensive_function每个表行只计算一次，而不是两次。

以上的例子只展示了和SELECT一起使用的WITH，但是它可以被以相同的方式附加在INSERT、UPDATE或DELETE上。在每一种情况中，它实际上提供了可在主命令中引用的临时表。

4.8.2. WITH中的数据修改语句

可以在WITH中使用数据修改语句（INSERT、UPDATE或DELETE）。这允许在同一个查询中执行多个而不同操作。一个例子：

```
WITH moved_rows AS (
   DELETE FROM products
   WHERE
      "date" >= '2010-10-01' AND
      "date" < '2010-11-01'
   RETURNING *
)
INSERT INTO products_log
SELECT * FROM moved_rows;
```

这个查询实际上从products把行移动到products_log。WITH中的DELETE删除来自products的指定行，以它的RETURNING子句返回它们的内容，并且接着主查询读该输出并将它插入到products_log。

上述例子中好的一点是WITH子句被附加给INSERT，而没有附加给INSERT的子SELECT。这是必需的，因为数据修改语句只允许出现在附加给顶层语句的WITH子句中。不过，普通WITH可见性规则应用，这样才可能从子SELECT中引用到WITH语句的输出。

正如上述例子所示，WITH中的数据修改语句通常具有RETURNING子句（见第 3.4 节 “从修改的行中返回数据”）。它是RETURNING子句的输出，不是数据修改语句的目标表，它形成了剩

余查询可以引用的临时表。如果一个WITH中的数据修改语句缺少一个RETURNING子句，则它形不成临时表并且不能在剩余的查询中被引用。但是这样一个语句将被执行。一个非特殊使用的例子：

```
WITH t AS (
    DELETE FROM foo
)
DELETE FROM bar;
```

这个例子将从表foo和bar中移除所有行。被报告给客户端的受影响行的数目可能只包括从bar中移除的行。

数据修改语句中不允许递归自引用。在某些情况中可以采取引用一个递归WITH的输出来操作这个限制，例如：

```
WITH RECURSIVE included_parts(sub_part, part) AS (
    SELECT sub_part, part FROM parts WHERE part = 'our_product'
  UNION ALL
    SELECT p.sub_part, p.part
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
DELETE FROM parts
  WHERE part IN (SELECT part FROM included_parts);
```

这个查询将会移除一个产品的所有直接或间接子部件。

WITH中的数据修改语句只被执行一次，并且总是能结束，而不管主查询是否读取它们所有（或者任何）的输出。注意这和WITH中SELECT的规则不同：正如前一小节所述，直到主查询要求SELECT的输出时，SELECT才会被执行。

The sub-statements in WITH中的子语句被和每一个其他子语句以及主查询并发执行。因此在使用WITH中的数据修改语句时，指定更新的顺序实际是以不可预测的方式发生的。所有的语句都使用同一个snapshot执行（参见第 10 章 并发控制，因此它们不能“看见”在目标表上另一个执行的效果。这减轻了行更新的实际顺序的不可预见性的影响，并且意味着RETURNING数据是在不同WITH子语句和主查询之间传达改变的唯一方法。其例子

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM products;
```

外层SELECT可以返回在UPDATE动作之前的原始价格，而在

```
WITH t AS (
    UPDATE products SET price = price * 1.05
    RETURNING *
)
SELECT * FROM t;
```

外部SELECT将返回更新过的数据。

在一个语句中试图两次更新同一行是不被支持的。只会发生一次修改，但是该办法不能很容易地（有时是不可能）可靠地预测哪一个会被执行。这也应用于删除一个已经在同一个语句中被更新过的行：只有更新被执行。因此通常应该避免尝试在一个语句中尝试两次修改同一个行。尤其是防止书写可能影响被主语句或兄弟子语句修改的相同行。这样一个语句的效果将是不可预测的。

当前，在WITH中一个数据修改语句中被用作目标的任何表不能有条件规则、ALSO规则或INSTEAD规则，这些规则会扩展成为多个语句。

4.8.3. WITH中的PROCEDURE、FUNCTION

- 描述

  WITH ［PROCEDURE］ FUNCTION 子句用于在SQL语句中临时声明并定义函数或存储过程，这些函数或存储过程，可以在with子句内被引用。相比模式级别定义的函数后存储过程，通过WITH［PROCEDURE］FUNCTION定义的函数在解析时拥有更高的优先级。

  和公用表表达式CTE类似，通过WITH［PROCEDURE］FUNCTION定义的函数或存储过程对象不会存入系统表中，且仅在当前with语句中有效。

- 语法

```
WITH {
[ PROCEDURE name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
  { LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
  } ...
]

 FUNCTION  func_name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ]
 [, ...] ] )
[ RETURNS rettype
    | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
  | TRANSFORM { FOR TYPE type_name } [, ... ]
  | WINDOW
  | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
  | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
  | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
  | PARALLEL { UNSAFE | RESTRICTED | SAFE }
  | COST execution_cost
  | ROWS result_rows
  | SUPPORT support_function
  | SET configuration_parameter { TO value | = value | FROM CURRENT }
  | AS 'definition'
  | AS 'obj_file', 'link_symbol'
  }
} [; ...]
SELECT  [ LATERAL ] func_name ( [ argument [, ...] ] )
/
```

- 参数

  [PROCEDURE] FUNCTION

  指明需要定义的是函数还是存储过程。

  func_name、[*argmode*]等参数

  请参见CREATE FUNCTION(7)和CREATE PROCEDURE(7)。

- 注解

  1. 同一with子句支持同时创建多个function、procedure，中间以分号(;)分隔，末尾以斜杠符号(/)结束输入。

  2. procedure支持嵌套使用，但必须由function调用，不可单独定义使用。

- 支持模式

  该功能支持的模式：标准模式、兼容(oracle)模式、MYSQL模式、安全模式。

- 示例

  定义并使用function，如下所示。

```
with function with_func1(aa numeric) returns numeric
as $$
begin
return aa + 20;
end;
$$ language pluxsql;
select with_func1(3::numeric);
/
 with_func1
------------
         23
(1 row)
```

  定义procedure并在function中调用，如下所示。

```
create table with_tb1 (id int);

with
procedure with_proce1(aa int)
AS $$
insert into with_tb1 values (aa);
$$ language SQL;
function with_func1(cc int) returns int
as $$
begin
call with_proce1(cc);
return cc;
end;
$$ language pluxsql;
```

```
select with_func1(1);
/

 with_func1
------------
          1
(1 row)

select * from with_tb1;
 id
----
  1
(1 row)
```

- 兼容性

  oracle与达梦均支持with function用法，达梦不支持存储过程(procedure)定义使用。

4.9. pivot行列转换

- 概述

  PIVOT查询为将行转换为列，以交叉表格式生成结果。在转换过程中聚合数据，新的列表示不同范围的聚合数据， PIVOT 操作的输出通常比初始数据行包含更多的列和更少的行。

- 语法

  SELECT *column_clause*
  FROM <*table-expr*>
  PIVOT
  (
  pivot_clause
  pivot_for_clause
  pivot_in_clause
)

  其中 *column_clause* 是：
  column1, column2(,... ...)或者*
  为*或者列名，若为列名，则必须为table-expr中子列

  其中 *table-expr* 是：
  table-expr 可以是基本表或者子查询

  其中 *pivot_clause* 是：
  aggregate_function (*expr*) [[AS] *alias*]
  [, *aggregate_function* (*expr*) [[AS] *alias*] ... ...]
  aggregate_function为聚集函数比如sum，max，可为单列/多列聚合，支持as 重命名，as重命名子句可缺省

  其中 *pivot_for_clause* 是：
  for (*column1* [, *column2* , ...])

  其中 *pivot_in_clause* 是：

in (*value1* [as *alias*] [, *value2* [as *alias*] , ...])
指定的value值，行转列之后的新列的列名

- 参数

  pivot_clause

  单列/多列的聚集函数。

  pivot_for_clause

  列名，且该列名必须要在table-expr中出现。

  pivot_in_clause

  常量表达式，为字符串/数值，非常量表达式不可用。

- 返回值

  返回行转列之后的视图。

- 支持模式

  支持标准模式、兼容模式、安全模式，Linux平台和Windows平台可用。

- 示例

  1. 创建表，并插入数据。

     ```
     create table sales(product VARCHAR(32) not null,id int,country VARCHAR(20),channel
      int,
     quarter  int,amount_sold int, quantity_sold int);

     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('noodles',1,'China',3,01,7,100);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('noodles',2,'China',3,02,7,150);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('noodles',2,'China',4,02,7,200);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('noodles',2,'China',5,03,7,300);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('noodles',3,'China',9,04,7,400);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('KFC',1,'America',3,01,20,100);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('KFC',2,'America',4,02,20,200);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('KFC',3,'America',5,03,20,300);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('KFC',3,'America',9,04,20,400);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('pizza',1,'japan',3,01,15,100);
     insert into sales
      (product,id,country,channel,quarter,amount_sold,quantity_sold)values('pizza',1,'japan',4,02,15,200);
     ```

```
insert into sales
 (product,id,country,channel,quarter,amount_sold,quantity_sold)values('pizza',1,'japan',5,03,15,300);
insert into sales
 (product,id,country,channel,quarter,amount_sold,quantity_sold)values('pizza',4,'japan',9,04,15,400);

create table uxdbc_oracle_pivot_0001_table01(id1 int not null,name varchar2(15) not null,
 sex char(10), phone varchar2(15), salary float);

create table uxdbc_oracle_pivot_0001_table02(id2 int not null,name varchar2(15) not null,
 depart varchar2(15),entime int);

insert into uxdbc_oracle_pivot_0001_table01 values(0001,'张
三','man','0123-54589521',2500);
insert into uxdbc_oracle_pivot_0001_table02 values(0001,'张三','A3001',1);
insert into uxdbc_oracle_pivot_0001_table01 values(0002,'李
明','woman','0123-54589521',2500);
insert into uxdbc_oracle_pivot_0001_table02 values(0002,'李明','A3001',2);
insert into uxdbc_oracle_pivot_0001_table01 values(0003,'李四','man','',3000);
insert into uxdbc_oracle_pivot_0001_table02 values(0003,'李四','A3002',2);
```

2. 验证单列聚合函数。

```
SELECT *
FROM sales PIVOT(SUM(amount_sold)
FOR CHANNEL IN(3,4,5,9))
ORDER BY product,id,quarter,quantity_sold;
```

PRODUCT	ID	COUNTRY	QUARTER	QUANTITY_SOLD	3	4	5	9
KFC	1	America	1	100	20			
KFC	2	America	2	200		20		
KFC	3	America	3	300			20	
KFC	3	America	4	400				20
noodles	1	China	1	100	7			
noodles	2	China	2	150	7			
noodles	2	China	2	200		7		
noodles	2	China	3	300			7	
noodles	3	China	4	400				7
pizza	1	japan	1	100	15			
pizza	1	japan	2	200		15		
pizza	1	japan	3	300			15	
pizza	4	japan	4	400				15

(13 rows)

3. 验证多列聚合函数与聚合函数重命名。

```
SELECT *
FROM (SELECT product, channel, amount_sold, quantity_sold FROM sales) S
PIVOT
```

```
(SUM(amount_sold) AS sums, SUM(quantity_sold) AS sumq
FOR channel IN(5, 4, 3, 9)) ORDER BY product;
```

```
PRODUCT | 5_SUMS | 5_SUMQ | 4_SUMS | 4_SUMQ | 3_SUMS | 3_SUMQ | 9_SUMS | 9_SUMQ
--------+--------+--------+--------+--------+--------+--------+--------+--------
KFC     |     20 |    300 |     20 |    200 |     20 |    100 |     20 |    400
noodles |      7 |    300 |      7 |    200 |     14 |    250 |      7 |    400
pizza   |     15 |    300 |     15 |    200 |     15 |    100 |     15 |    400
(3 rows)
```

4. 验证for-in多列与as重命名。

```
SELECT *
  FROM (SELECT product, channel, quarter, quantity_sold FROM  sales) S
PIVOT(SUM(quantity_sold) AS sums, SUM(quantity_sold)
   FOR (CHANNEL, quarter)
IN((5,3) as col1,(4,2),(3,1) col3,(3,2),(9,4)))
ORDER BY product;
```

```
PRODUCT | COL1_SUMS | COL1 | 4_2_SUMS | 4_2 | COL3_SUMS | COL3 | 3_2_SUMS | 3_2 | 9_4_SUMS | 9_4
--------+-----------+------+----------+-----+-----------+------+----------+-----+----------+-----
KFC     |       300 |  300 |      200 | 200 |       100 |  100 |          |     |      400 | 400
noodles |       300 |  300 |      200 | 200 |       100 |  100 |      150 | 150 |      400 | 400
pizza   |       300 |  300 |      200 | 200 |       100 |  100 |          |     |      400 | 400
(3 rows)
```

5. 验证外连接时使用pivot的结合性-1。

```
select * from
(
 uxdbc_oracle_pivot_0001_table01
 LEFT JOIN
 uxdbc_oracle_pivot_0001_table02
 pivot(sum(entime) FOR depart in ('A3001')) s
 on s.id2 = uxdbc_oracle_pivot_0001_table01.id1
) ORDER BY id1, id2;
```

```
 ID1 | NAME |    SEX    |     PHONE     | SALARY | ID2 | NAME | 'A3001'
-----+------+-----------+---------------+--------+-----+------+---------
   1 | 张三 | man       | 0123-54589521 |   2500 |   1 | 张三 |       1
   2 | 李明 | woman     | 0123-54589521 |   2500 |   2 | 李明 |       2
   3 | 李四 | man       |               |   3000 |   3 | 李四 |
(3 rows)
```

6. 验证外连接时使用pivot的结合性-2。

```
select * from
(
 uxdbc_oracle_pivot_0001_table01
 LEFT JOIN
 uxdbc_oracle_pivot_0001_table02
```

```
 pivot(sum(entime) FOR name in (’张三’,’李四’,’李明’)) s
 on s.id2 = uxdbc_oracle_pivot_0001_table01.id1
)pivot (sum(salary) FOR sex IN ('man')) ORDER BY id1, id2;

 ID1 | NAME |     PHONE     | ID2 | DEPART | ’张三’ | ’李四’ | ’李明’ | 'man'
-----+------+---------------+-----+--------+--------+--------+--------+-------
   1 | 张三 | 0123-54589521 |   1 | A3001  |      1 |        |        | 2500
   2 | 李明 | 0123-54589521 |   2 | A3001  |        |        |      2 |
   3 | 李四 |               |   3 | A3002  |        |      2 |        | 3000
(3 rows)
```

7. 验证外连接时使用pivot的结合性-3。

```
select * from
(
 uxdbc_oracle_pivot_0001_table01
 pivot(sum(salary) FOR sex in ('man')) s
 pivot(max(name) FOR phone IN ('0123-54589521')) s1
 LEFT JOIN
 uxdbc_oracle_pivot_0001_table02
 on s1.id1 = uxdbc_oracle_pivot_0001_table02.id2
) ORDER BY id1, id2;

 ID1 | 'man' | '0123-54589521' | ID2 | NAME | DEPART | ENTIME
-----+-------+-----------------+-----+------+--------+--------
   1 |  2500 | 张三            |   1 | 张三 | A3001  |      1
   2 |       | 李明            |   2 | 李明 | A3001  |      2
   3 |  3000 |                 |   3 | 李四 | A3002  |      2
(3 rows)
```

8. 验证pivot作子查询。

```
select * from uxdbc_oracle_pivot_0001_table02 pivot(sum(id2)
FOR depart in ('A3001','A3002')) pivot(sum(entime)
 FOR name in (’张三’,’李四’)) ORDER BY ″’张三’″;

 'A3001' | 'A3002' | ’张三’ | ’李四’
---------+---------+--------+--------
       1 |         |      1 |
         |       3 |        |      2
       2 |         |        |
(3 rows)
```

9. 验证pivot在多个table-expr成员存在的情况下的优先结合性。

```
select * from
uxdbc_oracle_pivot_0001_table01
pivot(sum(salary) FOR sex in ('man')),
uxdbc_oracle_pivot_0001_table02
ORDER BY id1,id2;

 ID1 | NAME |     PHONE     | 'man' | ID2 | NAME | DEPART | ENTIME
```

```
-----+------+---------------+-------+-----+------+--------+--------
  1 | 张三 | 0123-54589521 |  2500 |   1 | 张三 | A3001  |      1
  1 | 张三 | 0123-54589521 |  2500 |   2 | 李明 | A3001  |      2
  1 | 张三 | 0123-54589521 |  2500 |   3 | 李四 | A3002  |      2
  2 | 李明 | 0123-54589521 |       |   1 | 张三 | A3001  |      1
  2 | 李明 | 0123-54589521 |       |   2 | 李明 | A3001  |      2
  2 | 李明 | 0123-54589521 |       |   3 | 李四 | A3002  |      2
  3 | 李四 |               |  3000 |   1 | 张三 | A3001  |      1
  3 | 李四 |               |  3000 |   2 | 李明 | A3001  |      2
  3 | 李四 |               |  3000 |   3 | 李四 | A3002  |      2
(9 rows)
```

第 5 章 数据类型

UXDB有着丰富的本地数据类型可用。用户可以使用CREATE TYPE(7)命令为 UXDB增加新的数据类型。

表 5.1 “数据类型”显示了所有内建的普通数据类型。大部分在“别名”列里列出的可选名字都是因历史原因 被UXDB在内部使用的名字。另外，还有一些内部使用的或者废弃的类型也可以用，但没有在这里列出。

表 5.1. 数据类型

名字	别名	描述
bigint	int8	有符号的8字节整数
bigserial	serial8	自动增长的8字节整数
bit [(*n*)]		定长位串
bit varying [(*n*)]	varbit [(*n*)]	变长位串
boolean	bool	逻辑布尔值（真/假）
box		平面上的普通方框
bytea		二进制数据（“字节数组”）
character [(*n*)]	char [(*n*)]	定长字符串
character varying [(*n*)]	varchar [(*n*)]	变长字符串
cidr		IPv4或IPv6网络地址
circle		平面上的圆
date		日历日期（年、月、日）
double precision	float8	双精度浮点数（8字节）
inet		IPv4或IPv6主机地址
integer	int, int4	有符号4字节整数
interval [*fields*] [(*p*)]		时间段
json		文本 JSON 数据
jsonb		二进制 JSON 数据，已分解
line		平面上的无限长的线
lseg		平面上的线段
macaddr		MAC（Media Access Control）地址
macaddr8		MAC（Media Access Control）地址（EUI-64格式）
money		货币数量
numeric [(*p*, *s*)]	decimal [(*p*, *s*)]	可选择精度的精确数字
path		平面上的几何路径
ux_lsn		UXDB日志序列号
point		平面上的几何点
polygon		平面上的封闭几何路径

名字	别名	描述
real	float4	单精度浮点数（4字节）
smallint	int2	有符号2字节整数
smallserial	serial2	自动增长的2字节整数
serial	serial4	自动增长的4字节整数
text		变长字符串
time [(*p*)] [without time zone]		一天中的时间（无时区）
time [(*p*)] with time zone	timetz	一天中的时间，包括时区
timestamp [(*p*)] [without time zone]		日期和时间（无时区）
timestamp [(*p*)] with time zone	timestamptz	日期和时间，包括时区
tsquery		文本搜索查询
tsvector		文本搜索文档
txid_snapshot		用户级别事务ID快照
uuid		通用唯一标识码
xml		XML数据

兼容性

下列类型（或者及其拼写）是SQL指定的：bigint、bit、bit varying、boolean、char、character varying、character、varchar、date、double precision、integer、interval、numeric、decimal、real、smallint、time（有时区或无时区）、timestamp（有时区或无时区）、xml。

每种数据类型都有一个由其输入和输出函数决定的外部表现形式。许多内建的类型有明显的格式。不过，许多类型要么是UXDB所特有的（例如几何路径），要么可能是有几种不同的格式（例如日期和时间类型）。 有些输入和输出函数是不可逆的，即输出函数的结果和原始输入比较时可能丢失精度。

允许插入空串基础数据类型仅支持在oracle模式下使用。

表 5.2. 允许插入空串基础数据类型

类型		允许空串插入	展现形式
数值类型（N）	smallint:int2	允许	null值
	integer:int4	允许	null值
	bigint:int8	允许	null值
	number:numeric(m,n)	允许	null值
	real:float4（单精度浮点数）	允许	null值
	double precision:float8(双精度浮点数)	允许	null值

类型		允许空串插入	展现形式
字符类型（S）	char(n)：bpchar	允许	null值
	char:char(1)	允许	null值
	varchar	允许	null值
	varchar2	允许	null值
	nvarchar2	允许	null值
日期/时间类型（D/T）	timestamp	允许	null值
	timestamptz	允许	null值
	date	允许	null值
	time	允许	null值
	timetz	允许	null值
	interval（时间段）	允许	null值
布尔类型（B）	boolean:bool	允许	null值
位串类型（V）	bit(n)	允许	null值
	varbit(n)	允许	null值
二进制数据类型（B）	binary	允许	null值
	varbinary	允许	null值
	longvarbinary	允许	null值
	raw	允许	null值
JSON类型（A）	json	允许	null值
	jsonb	允许	null值
其他类型	oid	允许	null值
	blob	允许	null值
	clob	允许	null值

5.1. 数字类型

数字类型由2、4或8字节的整数以及4或8字节的浮点数和可选精度小数组成。表 5.3 “数字类型”列出了所有可用类型。

表 5.3. 数字类型

名字	存储尺寸	描述	范围
smallint	2字节	小范围整数	-32768 to +32767
integer	4字节	整数的典型选择	-2147483648 to +2147483647
bigint	8字节	大范围整数	-9223372036854775808 to +9223372036854775807
decimal	可变	用户指定精度，精确	最高小数点前131072位，以及小数点后16383位

名字	存储尺寸	描述	范围
numeric	可变	用户指定精度，精确，同number类型	最高小数点前131072位，以及小数点后8191位
real	4字节	可变精度，不精确	6位十进制精度
double precision	8字节	可变精度，不精确，同double类型	15位十进制精度
smallserial	2字节	自动增加的小整数	1到32767
serial	4字节	自动增加的整数	1到2147483647
bigserial	8字节	自动增长的大整数	1到9223372036854775807

数字类型常量的语法在第 1.1.2 节 “常量”里描述。数字类型有一整套对应的数学操作符和函数。相关信息请参考 第 6 章 函数和操作符下面的几节详细描述这些类型。

5.1.1. 整数类型

类型smallint、integer和bigint存储各种范围的全部是数字的数，也就是没有小数部分的数字。试图存储超出范围以外的值将导致一个错误。

常用的类型是integer，因为它提供了在范围、存储空间和性能之间的最佳平衡。一般只有在磁盘空间紧张的时候才使用 smallint类型。而只有在integer的范围不够的时候才使用bigint。

SQL只声明了整数类型integer（或int）、smallint和bigint。类型int2、int4和int8都是扩展，也在许多其它SQL数据库系统中使用。

5.1.2. 任意精度数字

类型numeric可以存储非常多位的数字。我们特别建议将它用于货币金额和其它要求计算准确的数量。numeric值的计算在可能的情况下会得到准确的结果，例如加法、减法、乘法。不过，numeric类型上的算术运算比整数类型或者下一节描述的浮点数类型要慢很多。

在随后的内容里，我们使用了下述术语：一个numeric的**precision**（精度）是整个数中有效位的总数，也就是小数点两边的位数。numeric的**scale**（刻度）是小数部分的数字位数，也就是小数点右边的部分。因此数字 23.5141 的精度为6而刻度为4。可以认为整数的刻度为零。

numeric列的最大精度和最大刻度都是可以配置的。要声明一个类型为numeric的列，可以用下面的语法：

NUMERIC(*precision*, *scale*)

精度必须为正数，刻度可以为零或者正数，取值范围为-1000到8191。另外：

NUMERIC(*precision*)

选择比例为 0 。如果使用

NUMERIC

创建一个列时不使用精度或刻度，则该列可以存储任何精度和刻度的数字值，并且值的范围最多可以到实现精度的上限。一个这种列将不会把输入值转化成任何特定的刻度，而带有刻度声明的

numeric列将把输入值转化为该刻度（SQL标准要求缺省的刻度是 0，即转化成整数精度。我们觉得这样做有点没用。如果关心移植性，那最好总是显式声明精度和刻度）。

> **注意**
>
> 显式指定类型精度时的最大允许精度为 1000，没有指定精度的NUMERIC受到表 5.3 “数字类型”中描述的限制所控制。

如果一个要存储的值的刻度比列声明的刻度高，那么系统将尝试圆整（四舍五入）该值到指定的分数位数。 然后，如果小数点左边的位数超过了声明的精度减去声明的刻度，那么抛出一个错误。

数字值在物理上是以不带任何前导或者后缀零的形式存储。 因此，列上声明的精度和刻度都是最大值，而不是固定分配的 （在这个方面，numeric类型更类似于varchar(*n*)， 而不像char(*n*)）。 实际存储要求是每四个十进制位组用两个字节，再加上三到八个字节的开销。

除了普通的数字值之外，numeric类型允许特殊值NaN， 表示“不是一个数字”。任何在 NaN上面的操作都生成另外一个NaN。 如果在 SQL 命令里把这些值当作一个常量写，必须在其周围放上单引号，例如UPDATE table SET x = 'NaN'。在输入时，字串NaN被识别为大小写无关。

> **注意**
>
> 在“不是一个数字”概念的大部分实现中，NaN被认为不等于任何其他数字值（包括NaN）。为了允许numeric值可以被排序和使用基于树的索引，UXDB把NaN值视为相等，并且比所有非NaN值都要大。

类型decimal和numeric是等效的。两种类型都是SQL标准的一部分。

在对值进行圆整时，numeric类型会圆到远离零的整数，而（在大部分机器上）real和double precision类型会圆到最近的偶数上。例如：

```
SELECT x,
  round(x::numeric) AS num_round,
  round(x::double precision) AS dbl_round
FROM generate_series(-3.5, 3.5, 1) as x;
  x   | num_round | dbl_round
------+-----------+-----------
 -3.5 |        -4 |        -4
 -2.5 |        -3 |        -2
 -1.5 |        -2 |        -2
 -0.5 |        -1 |        -0
  0.5 |         1 |         0
  1.5 |         2 |         2
  2.5 |         3 |         2
  3.5 |         4 |         4
(8 rows)
```

5.1.3. 浮点类型

数据类型real和double precision是不精确的、变精度的数字类型，double precision与double类型相同。 在所有当前支持的平台上，这些类型是IEEE标准 754 二进制浮点算术（分别对应单精度和双精度）的实现， 一直到下层处理器、操作系统和支持它的编译器。

不准确意味着一些值不能准确地转换成内部格式并且是以近似的形式存储的，因此存储和检索一个值可能出现一些缺失。 处理这些错误以及这些错误是如何在计算中传播的主题属于数学和计算机科学的一个完整的分支， 我们不会在这里进一步讨论它，这里的讨论仅限于如下几点：

- 如果要求准确的存储和计算（例如计算货币金额），应使用numeric类型。
- 如果想用这些类型做任何重要的复杂计算，尤其是那些对范围情况（无穷、下溢）严重依赖的事情，那应该仔细评估实现。
- 用两个浮点数值进行等值比较不可能总是按照期望地进行。

在所有当前支持的平台上，real类型的范围是 1E-37 to 1E+37 ，精度至少是 6 位小数。 double precision类型的范围是 1E-307 to 1E+308 ，精度至少是 15 位数字。 太大或者太小的值都会导致错误。 如果输入数字的精度太高，那么可能发生四舍五入。 太接近零的数字，如果不能体现出与零的区别就会导致下溢错误。

默认情况下，浮点值以其最短精确的十进制表示的文本形式输出；所产生的十进制值与相同二进制精度的任何其他的值表示相比，更接近于真实存储的二进制值。 （但是，当前输出值永远不会精确地处于两个可表示的值之间，以免输入程序不能正确遵守舍近取整法则。） 对于float8值，此值最多使用 17 个有效十进制数字，对于float4值，最多使用9个数字。

> **注意**
>
> 生成这种最短精确的输出格式比历史的四舍五入的格式要快得多。

设置extra_float_digits位任何大于 0 的值将选择最短精确格式。

> **注意**
>
> 需要更精确值的应用需要设置extra_float_digits为3以获取更精确值。

除了普通的数字值之外，浮点类型还有几个特殊值：

Infinity
-Infinity
NaN

这些分别代表 IEEE 754 特殊值“infinity”、“negative infinity”以及“not-a-number”，如果在 SQL 命令里把这些数值当作常量写，必须在它们周围放上单引号，例如UPDATE table SET x = '-Infinity'。 在输入时，这些字符串是以大小写不敏感的方式识别的。

> **注意**
>
> IEEE754指定NaN不应该与任何其他浮点值（包括NaN）相等。为了允许浮点值被排序或者在基于树的索引中使用，UXDB将NaN值视为相等，并且比所有非NaN值要更大。

UXDB还支持 SQL 标准表示法float和float(*p*)用于声明非精确的数字类型。在这里，*p*指定以二进制位表示的最低可接受精度。 在选取real类型的时候，UXDB接受float(1)到float(24)，在选取double precision的时候，接受float(25)到float(53)。在允许范围之外的*p*值将导致一个错误。没有指定精度的float将被当作是double precision。

5.1.4. 序数类型

> **注意**
>
> 这一节描述了UXDB特有的创建一个自增列的方法。另一种方法是使用SQL标准的标识列特性，它在CREATE TABLE(7)中描述。

smallserial、serial和bigserial类型不是真正的类型，它们只是为了创建唯一标识符列而存在的方便符号（类似其它一些数据库中支持的AUTO_INCREMENT属性）。 在目前的实现中，下面一个语句：

```
CREATE TABLE tablename (
    colname SERIAL
);
```

等价于以下语句：

```
CREATE SEQUENCE tablename_colname_seq AS integer;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

因此，我们就创建了一个整数列并且把它的缺省值安排为从一个序列发生器取值。应用了一个NOT NULL约束以确保空值不会被插入（在大多数情况下可能还希望附加一个UNIQUE或者PRIMARY KEY约束避免意外地插入重复的值，但这个不是自动发生的）。最后，该序列被标记为“属于”该列，这样当列或表被删除时该序列也会被删除。

> **注意**
>
> 因为smallserial、serial和bigserial是用序列实现的，所以即使没有删除过行，在出现在列中的序列值可能有“空洞”或者间隙。如果一个从序列中分配的值被用在一行中，即使该行最终没有被成功地插入到表中，该值也被“用掉”了。例如，当插入事务回滚时就会发生这种情况。更多信息参见第 6.16 节 “序列操作函数”中的nextval()。

要使用serial列插入序列的下一个数值到表中， 请指定serial列应该被赋予其缺省值。我们可以通过在INSERT语句中把该列排除在列列表之外来实现，也可以通过使用DEFAULT关键字来实现。

类型名serial和serial4是等效的： 两个都创建integer列。类型名bigserial和serial8也一样，只不过它们创建一个 bigint列。如果预计在表的生存期中使用的标识符数目超过2^{31} 个，那么应该使用bigserial。类型名smallserial和serial2也以相同方式工作，只不过它们创建一个smallint列。

为一个serial列创建的序列在所属的列被删除的时候自动删除。可以在不删除列的情况下删除序列，但是这会强制删除该列的默认值表达式。

5.2. 货币类型

money类型存储固定小数精度的货币数字，参阅表 5.4 “货币类型”。小数的精度由数据库的guc_lc_monetary设置决定。表中展示的范围假设有两个小数位。可接受的输入格式很多，包括

整数和浮点数文字，以及常用的货币格式，如'$1,000.00'。 输出通常是最后一种形式，但和区域相关。

表 5.4. 货币类型

名字	存储尺寸	描述	范围
money	8 bytes	货币额	-92233720368547758.08到 +92233720368547758.07

由于这种数据类型的输出是区域敏感的，因此将money数据装入到一个具有不同lc_monetary设置的数据库是不起作用的。为了避免这种问题，在恢复一个转储到一个新数据库中之前，应确保新数据库的lc_monetary设置和被转储数据库的相同或者具有等效值。

数据类型numeric、int和bigint的值可以被造型成money。从数据类型real和double precision的转换可以通过先造型成numeric来实现，例如：

```
SELECT '12.34'::float8::numeric::money;
```

但是，我们不推荐这样做。浮点数不应该被用来处理货币，因为浮点数可能会有圆整错误。

一个money值可以在不损失精度的情况下被造型成numeric。转换到其他类型可能会丢失精度，并且必须采用两个阶段完成：

```
SELECT '52093.89'::money::numeric::float8;
```

一个money值被一个整数值除的除法结果会被截去分数部分。要得到圆整的结果，可以除以一个浮点值，或者在除法之前把money转换成numeric然后在除法之后转回money（如果要避免精度丢失的风险则后者更好）。当一个money值被另一个money值除时，结果是double precision（即一个纯数字，而不是金额），在除法中货币单位被约掉了。

5.3. 字符类型

表 5.5. 字符类型

名字	描述
character varying(*n*), varchar(*n*)	有限制的变长
character(*n*), char(*n*)	定长，空格填充
text	无限变长
long	存储可变长字符串，同text类型

表 5.5 “字符类型”显示了在UXDB里可用的一般用途的字符类型。

long类型与text类型相同，属于字符类型。存储可变长字符串，能被存储的最长的字符串是1GB。

SQL定义了两种基本的字符类型：character varying(*n*)和character(*n*)，其中*n*是一个正整数。两种类型都可以存储最多*n*个字符长的串。试图存储更长的串到这些类型的列里会产生一个错误，除非超出长度的字符都是空白，这种情况下该串将被截断为最大长度（这个看上去有点怪异的例外是SQL标准要求的）。 如果要存储的串比声明的长度短，类型为character的值将会用空白填满；而类型为character varying的值将只是存储短些的串。

如果我们明确地把一个值造型成character varying(*n*)或者character(*n*)， 那么超长的值将被截断成*n*个字符，而不会抛出错误（这也是SQL标准的要求）。

varchar(*n*)和char(*n*)的概念分别是character varying(*n*)和character(*n*)的别名。没有长度声明词的character等效于character(1)。如果不带长度说明词使用character varying，那么该类型接受任何长度的串。后者是一个UXDB的扩展。

另外，UXDB提供text类型，它可以存储任何长度的串。尽管类型text不是SQL标准，但是许多其它 SQL 数据库系统也有它。

类型character的值物理上都用空白填充到指定的长度*n*， 并且以这种方式存储和显示。不过，拖尾的空白被当作是没有意义的，并且在比较两个 character类型值时不会考虑它们。在空白有意义的排序规则中，这种行为可能会 产生意料之外的结果，例如SELECT 'a '::CHAR(2) collate "C" < E'a\n'::CHAR(2)会返回真（即便C区域会认为一个空格比新行更大）。当把一个character值转换成其他 字符串类型之一时，拖尾的空白会被移除。请注意，在character varying和text值里， 结尾的空白语意上是有含义的，并且在使用模式匹配（如LIKE和正则表达式）时也会被考虑。

这些类型的存储需求是 4 字节加上实际的字串，如果是 character 的话再加上填充的字节。长的字串将会自动被系统压缩， 因此在磁盘上的物理需求可能会更少些。长的数值也会存储在后台表里面，这样它们就不会干扰对短字段值的快速访问。 不管怎样，允许存储的最长字串大概是 1 GB。 （允许在数据类型声明中出现的的 n 的最大值比这还小。 修改这个行为没有甚么意义，因为在多字节编码下字符和字节的数目可能差别很大。 如果想存储没有特定上限的长字串，那么使用 text 或者没有长度声明词的character varying， 而不要选择一个任意长度限制。） 一个短串（最长126字节）的存储要求是1个字节外加实际的串，该串在character情况下包含填充的空白。长一些的串在前面需要4个字节而不是1个字节。长串会被系统自动压缩，这样在磁盘上的物理需求可能会更少。非常长的值也会被存储在背景表中，这样它们不会干扰对较短的列值的快速访问。在任何情况下，能被存储的最长的字符串是1GB（数据类型定义中*n*能允许的最大值比这个值要小。修改它没有用处，因为对于多字节字符编码来说，字符的数量和字节数可能完全不同。如果想要存储没有指定上限的长串，使用text或没有长度声明的character varying，而不是给出一个任意长度限制）。

> **提示**
>
> 这三种类型之间没有性能差别，只不过是在使用填充空白的类型的时候需要更多存储尺寸，以及在存储到一个有长度约束的列时需要少量额外CPU周期来检查长度。虽然在某些其它的数据库系统里，character(*n*)有一定的性能优势，但在UXDB里没有。事实上，character(*n*)通常是这三种类型之中最慢的一个，因为它需要额外的存储开销。在大多数情况下，应该使用text或者character varying。

请参考第 1.1.2.1 节 “字符串常量”获取关于串文本的语法的信息，以及参阅第 6 章 函数和操作符获取关于可用操作符和函数的信息。 数据库的字符集决定用于存储文本值的字符集。

例 5.1. 使用字符类型

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1;

 a   | char_length
------+-------------
```

```
 ok   |         2


CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR:  value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- explicit truncation
SELECT b, char_length(b) FROM test2;

   b   | char_length
-------+-------------
 ok    |           2
 good  |           5
 too l |           5
```

函数char_length在第 6.4 节 “字符串函数和操作符”中讨论。

在UXDB里另外还有两种定长字符类型，在表 5.6 “特殊字符类型”里显示。 name类型只用于在内部系统目录中存储标识符并且不是给一般用户使用的。该类型长度当前定为 64 字节（63 可用字符加结束符）但在C源代码应该使用常量 NAMEDATALEN引用。这个长度是在编译的时候设置的（因而可以为特殊用途调整）。类型"char"（注意引号）和 char(1)是不一样的，它只用了一个字节的存储空间。它在系统内部用于系统目录当做简化的枚举类型用。

表 5.6. 特殊字符类型

名字	存储尺寸	描述
"char"	1字节	单字节内部类型
name	64字节	用于对象名的内部类型

5.4. 二进制数据类型

bytea数据类型允许存储二进制串，参见表 5.7 “二进制数据类型”。

表 5.7. 二进制数据类型

名字	存储尺寸	描述
bytea	1或4字节外加真正的二进制串	变长二进制串
raw		同bytea数据类型，为可变长度二进制串
varbinary		可变长度二进制数据类型
longvarbinary		加长版可变二进制类型数据

二进制串是一个八位位组（或字节）的序列。 二进制串和字符串的区别有两个： 首先，二进制串明确允许存储零值的字节以及其它“不可打印的”字节（通常是位于十进制范围32到126之外的字节）。 字符串不允许零字节，并且也不允许那些对于数据库的选定字符集编码是非法的任何其它字节值或者字节值序列。 第二，对二进制串的操作会处理实际上的字节，而字符串的处理和取决于区域设置。 简单说，二进制字串适用于存储那些程序员认为是“裸字节”的数据，而字符串适合存储文本。

bytea类型支持两种用于输入和输出的格式：“十六进制”格式和UXDB的历史的“转义”格式。在输入时这两种格式总是会被接受。输出格式则取决于配置参数bytea_output，其默认值为十六进制。

SQL标准定义了一种不同的二进制串类型， 叫做BLOB或者BINARY LARGE OBJECT。其输入格式和bytea不同，但是提供的函数和操作符大多一样。

5.4.1. bytea的十六进制格式

“十六进制”格式将二进制数据编码为每个字节2个十六进制位，最高有效位在前。整个串以序列\x开头（用以和转义格式区分）。在某些情景中，开头的反斜线可能需要通过双写来转义，详见第 1.1.2.1 节 “字符串常量”。 作为输入，十六进制位可以是大写也可以是小写，在位对之间可以有空白（但是在位对内部以及开头的\x序列中不能有空白）。十六进制格式和很多外部应用及协议相兼容，并且其转换速度要比转义格式更快，因此人们更愿意用它。

例子：

```
SELECT '\xDEADBEEF';
```

5.4.2. bytea的转义格式

“转义”格式是bytea类型的传统UXDB格式。它采用将二进制串表示成ASCII字符序列的方法，而将那些无法用ASCII字符表示的字节转换成特殊的转义语句。从应用的角度来看，如果将字节表示为字符有意义，那么这种表示将很方便。但是在实际中，这常常是令人困扰的，因为它使二进制串和字符串之间的区别变得模糊，并且这种特别的转义机制也有点难于处理。因此这种格式可能会在大部分新应用中避免使用。

在转义模式下输入bytea值时，某些值的字节必须被转义，而所有的字节值都可以被转义。通常，要转义一个字节，需要把它转换成与它的三位八进制值， 并且前导一个反斜线。反斜线本身（十进制字节值92）也可以用双写的反斜线表示。表 5.8 “bytea文字转义字节”显示了必须被转义的字符，并给出了可以使用的替代转义序列。

表 5.8. bytea文字转义字节

十进制字节值	描述	转义输入表示	例子	十六进制表示
0	0字节	'\000'	SELECT '\000'::bytea;	\x00
39	单引号	''''或'\047'	SELECT ''''::bytea;	\x27
92	反斜线	'\\'或'\134'	SELECT '\\'::bytea;	\x5c
0到31和127到255	“不可打印的”字节	'*xxx*'（八进制值）	SELECT '\001'::bytea;	\x01

转义“不可打印的”字节的要求取决于区域设置。在某些实例中，可以不理睬它们，让它们保持未转义的状态。

如表 5.8 “bytea文字转义字节”中所示，要求单引号必须写两次的原因对任何SQL命令中的字符串常量都是一样的。 文字解析器消耗最外层的单引号，并缩减成对的单引号为一个普通数据字符。 bytea输入函数看到的只是一个单引号，它将其视为普通数据字符。 但是，bytea输入函数将反斜杠视为特殊字符，表 5.8 “bytea文字转义字节”中显示的其他行为由该函数实现。

在某些情况下，反斜杠必须加倍，如上所示，因为通用的字符串文字解析器也会 将一对反斜杠减少为一个数据字符；请参阅第 1.1.2.1 节 “字符串常量”。

Bytea字节默认被输出为hex格式。如果把bytea_output改为escape，“不可打印的”字节会被转换成与之等效的三位八进制值并且前置一个反斜线。大部分“可打印的”字节被输出为它们在客户端字符集中的标准表示形式，例如：

```
SET bytea_output = 'escape';

SELECT 'abc \153\154\155 \052\251\124'::bytea;
    bytea
----------------
 abc klm *\251T
```

十进制值为92（反斜线）的字节在输出时被双写。详情请见表　5.9 “bytea输出转义字节”。

表　5.9.　bytea输出转义字节

十进制字节值	描述	转义的输出表示	例子	输出结果
92	反斜线	\\	SELECT '\134'::bytea;	\\
0到31和127到255	“不可打印的”字节	*xxx*（八进制值）	SELECT '\001'::bytea;	\001
32到126	“可打印的”字节	客户端字符集表示	SELECT '\176'::bytea;	~

根据使用的UXDB前端，在转义和未转义bytea串方面可能需要做额外的工作。例如，如果接口自动翻译换行和回车，可能也不得不转义它们。

5.4.3.　raw

- 类型介绍

  同bytea数据类型，为可变长度二进制串。

  特殊字符的表现形式参见表　5.8 “bytea文字转义字节”。

- 示例

  CREATE TABLE test_raw(ID INT, CONTENT RAW);

  创建表，插入数据，并查看结果。

```
CREATE  TABLE test_raw(ID INT, CONTENT RAW);
INSERT INTO test_raw VALUES(1,'TO BE BOTH A SPEAKER OF WORDS AND A DOER
 OF DEEDS');
INSERT INTO test_raw VALUES(2,0X4D414E41474552);
INSERT INTO test_raw VALUES(3,'0X4D414E41474552');
INSERT INTO test_raw VALUES(3,'1');
SELECT * FROM test_raw;
 ID |                                CONTENT
----+-------------------------------------------------------------------------------------------------
  1 | \x544f20424520424f5448204120535045414b4552204f4620574f52445320414e442041
```

```
  20444f4552204f46204445454453
 2 | \x307834443431344534314343734353532
 3 | \x305834443431344534314343734353532
 3 | \x31
(4 rows)
```

5.4.4. varbinary

- 类型介绍

  可变长度二进制数据类型。

- 示例

  Create table test(v1 varbinary(100),v2 varbinary);

  varbinary(n)和varbinary均可声明数据类型，n为正整数。

  创建表，插入数据，并查看结果。

```
CREATE TABLE test_varbinary(PW VARBINARY(100));
INSERT INTO test_varbinary VALUES('123');
INSERT INTO test_varbinary VALUES(0X123);
INSERT INTO test_varbinary VALUES('0X123');
INSERT INTO test_varbinary VALUES(NULL);
INSERT INTO test_varbinary VALUES('');
INSERT INTO test_varbinary VALUES(123);
INSERT INTO test_varbinary VALUES('MANAGER');

SELECT * FROM test_varbinary;
       PW
------------------
 0x313233
 0x0123
 0x3058313233


 0x313233
 0x4d414e41474552
(7 rows)
SELECT * FROM test_varbinary WHERE PW = 'MANAGER';
       PW
------------------
 0x4d414e41474552
(1 row)
SELECT * FROM test_varbinary WHERE PW = '0X6D616E61676572';
 PW
----
(0 rows)
```

5.4.5. longvarbinary

加长版可变二进制类型数据，使用方法参见第 5.4.4 节 “varbinary”。

5.5. 日期/时间类型

UXDB支持SQL中所有的日期和时间类型，如表 5.10 “日期/时间类型”所示。这些数据类型上可用的操作如第 6.9 节 “时间/日期函数和操作符”所述。日期根据公历来计算，即使对于该历法被引入之前的年份也一样。

表 5.10. 日期/时间类型

名字	存储尺寸	描述	最小值	最大值	解析度
timestamp [(*p*)] [without time zone]	8字节	包括日期和时间（无时区）	4713 BC	294276 AD	1微秒
timestamp [(*p*)] with time zone	8字节	包括日期和时间，有时区	4713 BC	294276 AD	1微秒
date	4字节	日期（没有一天中的时间）	4713 BC	5874897 AD	1日
time [(*p*)] [without time zone]	8字节	一天中的时间（无日期）	00:00:00	24:00:00	1微秒
time [(*p*)] with time zone	12字节	仅仅是一天中的时间（没有日期），带有时区	00:00:00+1459	24:00:00-1459	1微秒
interval [*fields*] [(*p*)]	16字节	时间间隔	-178000000年	178000000年	1微秒

> **注意**
>
> SQL要求只写timestamp等效于timestamp without time zone，并且UXDB鼓励这种行为。timestamptz被接受为timestamp with time zone的一种简写，这是一种UXDB的扩展。

time、timestamp和interval接受一个可选的精度值 *p*，这个精度值声明在秒域中小数点之后保留的位数。缺省情况下，在精度上没有明确的边界。*p*允许的范围是从 0 到 6。

timestamp类型支持插入0::int。

interval类型有一个附加选项，它可以通过写下面之一的短语来限制存储的fields的集合：

YEAR
MONTH
DAY
HOUR
MINUTE
SECOND
YEAR TO MONTH
DAY TO HOUR

DAY TO MINUTE
DAY TO SECOND
HOUR TO MINUTE
HOUR TO SECOND
MINUTE TO SECOND

注意如果*fields*和*p*被指定，*fields*必须包括SECOND，因为精度只应用于秒。

类型time with time zone是 SQL 标准定义的，但是该定义显示出了一些会影响可用性的性质。在大多数情况下， date、time、timestamp without time zone和timestamp with time zone的组合就应该能提供任何应用所需的全范围的日期/时间功能。

5.5.1. 日期/时间输入

日期和时间的输入可以接受几乎任何合理的格式，包括 ISO 8601、SQL-兼容的、UXDB和其他的形式。 对于一些格式，日期输入里的日、月和年的顺序会让人混淆， 并且支持指定所预期的这些域的顺序。把DateStyle参数设置为MDY，就是选择“月－日－年”的解释，设置为DMY就是 “日－月－年”，而YMD是 “年－月－日”。

UXDB在处理日期/时间输入上比SQL标准要求的更灵活。

请记住任何日期或者时间的文字输入需要由单引号包围，就象一个文本字符串一样。参考第 1.1.2.7 节 “其他类型的常量”获取更多信息。SQL要求下面的语法

type [(*p*)] '*value*'

其中*p*是一个可选的精度声明，它给出了在秒域中的小数位数目。精度可以被指定给time、timestamp和interval类型，并且可以取从0到6的值。这允许前文所述的值。如果在一个常数声明中没有指定任何精度，它将默认取文字值的精度（但不能超过6位）。

5.5.1.1. 日期

表 5.11 “日期输入”显示了date类型可能的输入方式。

表 5.11. 日期输入

例子	描述
1999-01-08	ISO 8601; 任何模式下的1月8日 （推荐格式）
January 8, 1999	在任何datestyle输入模式下都无歧义
1/8/1999	MDY模式中的1月8日；DMY模式中的8月1日
1/18/1999	MDY模式中的1月18日；在其他模式中被拒绝
01/02/03	MDY模式中的2003年1月2日； DMY模式中的2003年2月1日； YMD模式中的2001年2月3日
1999-Jan-08	任何模式下的1月8日
Jan-08-1999	任何模式下的1月8日
08-Jan-1999	任何模式下的1月8日
99-Jan-08	YMD模式中的1月8日，否则错误
08-Jan-99	1月8日，除了在YMD模式中错误

例子	描述
Jan-08-99	1月8日，除了在YMD模式中错误
19990108	ISO 8601；任何模式中的1999年1月8日
990108	ISO 8601；任何模式中的1999年1月8日
1999.008	年和一年中的日子
J2451187	儒略日期
January 8, 99 BC	公元前99年

5.5.1.2. 时间

当日时间类型是time [(*p*)] without time zone和time [(*p*)] with time zone。 只写time等效于time without time zone。

这些类型的有效输入由当日时间后面跟着可选的时区组成（参阅表 5.12 “时间输入”和表 5.13 “时区输入”）。 如果在time without time zone的输入中指定了时区，那么它会被无声地忽略。也可以指定一个日期但是它会被忽略，除非使用了一个涉及到夏令时规则的时区，例如America/New_York。在这种情况下，为了判断是应用了标准时间还是夏令时时间，要求指定该日期。适当的时区偏移被记录在time with time zone值中。

表 5.12. 时间输入

例子	描述
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	和04:05一样，AM并不影响值
04:05 PM	和16:05一样，输入的小时必须为 <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	缩写指定的时区
2003-04-12 04:05:06 America/New_York	全名指定的时区

表 5.13. 时区输入

例子	描述
PST	缩写（太平洋标准时间）
America/New_York	完整时区名
PST8PDT	POSIX风格的时区声明
-8:00	PST的ISO-8601偏移
-800	PST的ISO-8601偏移

例子	描述
-8	PST的ISO-8601偏移
zulu	UTC的军方缩写
z	zulu的短形式

参考第 5.5.3 节 “ 时区”可以了解如何指定时区。

5.5.1.3. 时间戳

时间戳类型的有效输入由一个日期和时间的串接组成，后面跟着一个可选的时区，一个可选的AD或者BC（另外，AD/BC可以出现在时区前面，但这个顺序并非最佳）。 因此：

1999-01-08 04:05:06

和：

1999-01-08 04:05:06 -8:00

和：

199901080405

都是有效的值，它遵循ISO 8601 标准。另外，使用广泛的格式：

January 8 04:05:06 1999 PST

空字符也被支持。

SQL标准通过“+”或者“-”符号的存在以及时间后面的时区偏移来区分timestamp without time zone和timestamp with time zone文字。因此，根据标准，

TIMESTAMP '2004-10-19 10:23:54'

是一个timestamp without time zone， 而

TIMESTAMP '2004-10-19 10:23:54+02'

是一个timestamp with time zone。UXDB从来不会在确定文字串的类型之前检查其内容，因此会把上面两个都看做是 timestamp without time zone。因此要保证把上面的文字当作timestamp with time zone看待， 就要给它正确的显式类型：

TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02'

如果一个文字已被确定是timestamp without time zone，UXDB将不声不响忽略任何其中指出的时区。 即，结果值是从输入值的日期/时间域衍生出来的，并且没有就时区进行调整。

对于timestamp with time zone，内部存储的值总是 UTC （全球统一时间，以前也叫格林威治时间GMT）。如果一个输入值有明确的时区声明， 那么它将用该时区合适的偏移量转换成 UTC。如果在输入串里没有时区声明， 那么它就被假设是在系统的TimeZone参数里的那个时区，然后使用这个 timezone时区的偏移转换成 UTC。

如果一个timestamp with time zone值被输出，那么它总是从 UTC 转换成当前的timezone时区，并且显示为该时区的本地时间。要看其它时区的时间，要么修改timezone，要么使用AT TIME ZONE构造（参阅第 6.9.3 节 “AT TIME ZONE”）。

在timestamp without time zone和timestamp with time zone之间的转换通常假设timestamp without time zone值应该以timezone本地时间的形式接受或者写出。为该转换指定一个不同的可以用AT TIME ZONE。

5.5.1.4. 特殊值

为了方便，UXDB支持一些特殊日期/时间输入值，如表 5.14 “特殊日期/时间输入”所示。这些值中infinity和-infinity被在系统内部以特殊方式表示并且将被原封不动地显示。但是其他的仅仅只是概念上的速写，当被读到的时候会被转换为正常的日期/时间值（特殊地，now及相关串在被读到时立刻被转换到一个指定的时间值）。在作为常量在SQL命令中使用时，所有这些值需要被包括在单引号内。

表 5.14. 特殊日期/时间输入

输入串	合法类型	描述
epoch	date, timestamp	1970-01-01 00:00:00+00（Unix系统时间 0）
infinity	date, timestamp	比任何其他时间戳都晚
-infinity	date, timestamp	比任何其他时间戳都早
now	date, time, timestamp	当前事务的开始时间
today	date, timestamp	今日午夜（00:00）
tomorrow	date, timestamp	明日午夜（00:00）
yesterday	date, timestamp	昨日午夜（00:00）
allballs	time	00:00:00.00 UTC

下列SQL-兼容的函数可以被用来为相应的数据类型获得当前时间值：CURRENT_DATE、CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME、LOCALTIMESTAMP。后四种接受一个可选的亚秒精度声明（参见第 6.9.4 节 “当前日期/时间”）。注意这些是SQL函数并且在数据输入串中不被识别。

5.5.2. 日期/时间输出

时间/日期类型的输出格式可以设成四种风格之一： ISO 8601、SQL（Ingres）、传统的UXDB（Unix的date格式）或 German 。缺省是ISO格式（ISO标准要求使用 ISO 8601 格式。ISO输出格式的名字是历史偶然）。表 5.15 “日期/时间输出风格”显示了每种输出风格的例子。date和time类型的 输出通常只有日期或时间部分和例子中一致。不过，UXDB风格输出的是ISO格式的只有日期的值。

表 5.15. 日期/时间输出风格

风格声明	描述	例子
ISO	ISO 8601, SQL标准	1997-12-17 07:37:16-08
SQL	传统风格	12/17/1997 07:37:16.00 PST

风格声明	描述	例子
UXDB	原始风格	Wed Dec 17 07:37:16 1997 PST
German	地区风格	17.12.1997 07:37:16.00 PST

> **注意**
>
> ISO 8601指定使用大写字母T来分隔日期和时间。UXDB在输入上接受这种格式，但是在输出时它采用一个空格而不是T，如上所示。和一些其他数据库系统一样，这是为了可读性以及与RFC 3339的一致性。

SQL和UXDB风格中，如果DMY域顺序被指定，“日”将出现在“月”之前，否则“月”出现在“日”之前（有关该设置如何影响输入值的解释，请参考第 5.5.1 节 “ 日期/时间输入”）。表 5.16 “日期顺序习惯”给出了例子。

表 5.16. 日期顺序习惯

datestyle设置	输入顺序	例子输出
SQL, DMY	日/月/年	17/12/1997 15:37:16.00 CET
SQL, MDY	月/日/年	12/17/1997 07:37:16.00 PST
UXDB, DMY	日/月/年	Wed 17 Dec 07:37:16 1997 PST

日期/时间风格可以由用户使用SET datestyle命令选取，在uxsinodb.conf配置文件里的参数DateStyle设置或者在服务器或客户端的UXDATESTYLE环境变量里设置。

格式化函数to_char（见第 6.8 节 “数据类型格式化函数”）也可以作为一个更灵活的方式来格式化日期/时间输出。

5.5.3. 时区

时区和时区习惯不仅仅受地球几何形状的影响，还受到政治决定的影响。 到了19世纪，全球的时区变得稍微标准化了些，但是还是易于遭受随意的修改，部分是因为夏时制规 则。UXDB使用广泛使用的 IANA (Olson) 时区数据库来得到有关历史时区规则的信息。对于未来的时间，我们假设关于一个给定时区的最新已知 规则将会一直持续到无穷远的未来。

UXDB努力在典型使用中与SQL标准的定义相兼容。但SQL标准在日期和时间类型和功能上有一些奇怪的混淆。两个显而易见的问题是：

- 尽管date类型与时区没有联系，而time类型却可以有。 然而，现实世界的时区只有在与时间和日期都关联时才有意义， 因为偏移（时差）可能因为实行类似夏时制这样的制度而在一年里有所变化。

- 缺省的时区会指定一个到UTC的数字常量偏移（时差）。因此，当跨DST边界做日期/时间算术时， 我们根本不可能适应于夏时制时间。

为了克服这些困难，我们建议在使用时区的时候，使用那些同时包含日期和时间的日期/时间类型。我们不建议使用类型 time with time zone （尽管UXDB出于遗留应用以及与SQL标准兼容性的考虑支持这个类型）。 UXDB假设用于任何类型的本地时区都只包含日期或时间。

在系统内部，所有时区相关的日期和时间都用UTC存储。它们在被显示给客户端之前会被转换成由TimeZone配置参数指定的本地时间。

UXDB允许使用三种不同形式指定时区：

- 一个完整的时区名字，例如America/New_York。能被识别的时区名字被列在ux_timezone_names视图中。UXDB用广泛使用的 IANA 时区数据来实现该目的，因此相同的时区名字也可以在其他软件中被识别。
- 一个时区缩写，例如PST。这样一种声明仅仅定义了到UTC的一个特定偏移，而不像完整时区名那样指出整套夏令时转换日期规则。能被识别的缩写被列在ux_timezone_abbrevs视图中。不能将配置参数TimeZone或log_timezone设置成一个时区缩写，但是可以在日期/时间输入值和AT TIME ZONE操作符中使用时区缩写。
- 除了时区名和缩写，UXDB将接受POSIX-风格的 时区声明，形式为*STDoffset*或 *STDoffsetDST*，其中*STD*是一个区域缩写、*offset*是从UTC西 起的以小时计的数字偏移量、*DST*是一个可选的夏令时区域缩 写（被假定为给定偏移量提前一小时）。例如，如果EST5EDT还不是一 个被识别的区域名，它可以被接受并且可能和美国东海岸时间的功效相同。在这种语法中， 一个时区缩写可以是一个字母的字符串或者由尖括号（<>）包围 的任意字符串。当一个夏令时区域缩写出现时，会假定根据 IANA 时区数据库的 posixrules条目中使用的同一个夏令时转换规则使用它。 在一个标准的UXDB安装中， posixrules和US/Eastern相同， 因此POSIX-风格的时区声明遵循美国的夏令时规则。如果需要，可以通过替换 posixrules文件来调整这种行为。

简而言之，在缩写和全称之间是有不同的：缩写表示从UTC开始的一个特定偏移量， 而很多全称表示一个本地夏令时规则并且因此具有两种可能的UTC偏移量。例如， 2014-06-04 12:00 America/New_York表示纽约本地时间的中午， 这个特殊的日期是东部夏令时间（UTC-4）。因此2014-06-04 12:00 EDT 指定的是同一个时间点。但是2014-06-04 12:00 EST指定东部标准时间的中午（UTC-5），不管在那个日期夏令时是否生效。

更要命的是，某些行政区已经使用相同的时区缩写在不同的时间表示不同的 UTC 偏移量。例如， 在莫斯科MSK在某些年份表示 UTC+3 而在另一些年份表示 UTC+4。 UXDB 会根据在指定的日期它们到底表示什么（或者最近表示什么） 来解释这种缩写。但是，正如上面的EST例子所示，这并不是必须和那一天的本地 标准时间相同。

应该注意到POSIX-风格的时区特性可能导致伪造的输入被接受，因为它没有对区域缩写合理性的检查。例如SET TIMEZONE TO FOOBAR0将会正常工作，让系统实际使用一个相当奇怪的UTC缩写。另一个需要记住的问题是在POSIX时区名中，正值的偏移量被用于格林威治以西的位置。在其他情况下，UXDB将遵循 ISO-8601 惯例，认为正值的时区偏移量是格林威治以东。

在所有情况下，时区名及其缩写都是大小写不敏感的。

时区名和缩写都不是硬写在服务器中的，它们是从存储在安装目录下的.../share/timezone/和.../share/timezonesets/子目录中获取的。

TimeZone配置参数可以在文件uxsinodb.conf中被设置，或者使用标准方法设置。同时也有一些特殊的方法来设置它：

- SQL命令SET TIME ZONE为会话设置时区。它是SET TIMEZONE TO的另一种拼写，它更加符合SQL的语法。
- libuxsql客户端使用UXTZ环境变量来通过连接发送一个SET TIME ZONE命令给服务器。

5.5.4. 间隔输入

interval值可以使用下列语法书写：

[@] *quantity unit* [*quantity unit*...] [*direction*]

其中*quantity*是一个数字（很可能是有符号的）； *unit*是毫秒、 millisecond、second、minute、hour、day、 week、month、year、 decade、century、millennium 或者缩写或者这些单位的复数； *direction*可以是ago或者为空。At符号（@）是一个可选的噪声。不同单位的数量通过合适的符号计数被隐式地添加。ago对所有域求反。如果IntervalStyle被设置为uxdb_verbose，该语法也被用于间隔输出。

日、小时、分钟和秒的数量可以不适用显式的单位标记指定。例如，'1 12:59:10'被读作'1 day 12 hours 59 min 10 sec'。同样，一个年和月的组合可以使用一个横线指定，例如'200-10'被读作'200年10个月'（这些较短的形式事实上是SQL标准唯一许可的形式，并且在IntervalStyle被设置为sql_standard时用于输出）。

间隔值也可以被写成 ISO 8601 时间间隔，使用该标准4.4.3.2小节的“带标志符的格式”或者4.4.3.3小节的“替代格式”。带标志符的格式看起来像这样：

P *quantity unit* [*quantity unit* ...] [T [*quantity unit* ...]]

该串必须以一个P开始，并且可以包括一个引入当日时间单位的T。可用的单位缩写在表 5.17 “ISO 8601 间隔单位缩写”中给出。单位可以被忽略，并且可以以任何顺序指定，但是小于一天的单位必须出现在T之后。特别地，M的含义取决于它出现在T之前还是之后。

表 5.17. ISO 8601 间隔单位缩写

缩写	含义
Y	年
M	月（在日期部分中）
W	周
D	日
H	小时
M	分钟（在时间部分中）
S	秒

如果使用替代格式：

P [*years-months-days*] [T *hours*:*minutes*:*seconds*]

串必须以P开始，并且一个T分隔间隔的日期和时间部分。其值按照类似于 ISO 8601日期的数字给出。

在用一个域声明书写一个间隔常量时，或者为一个用域声明定义的间隔列赋予一个串时，对于为标记的量的解释依赖于域。例如INTERVAL '1' YEAR被解读成1年，而INTERVAL '1'表示1秒。同样，域声明允许的最后一个有效域“右边”的域值会被无声地丢弃掉。例如书写INTERVAL '1 day 2:03:04' HOUR TO MINUTE将会导致丢弃秒域，而不是日域。

根据SQL标准，一个间隔值的所有域都必须由相同的符号，这样一个领头的负号将会应用到所有域；例如在间隔文字'-1 2:03:04'中的负号会被应用于日、小时、分钟和秒部分。UXDB允许域具有不同的符号，并且在习惯上认为以文本表示的每个域具有独立的符号，因此在这个例子中小时、分钟和秒部分被认为是正值。如果IntervalStyle被设置为sql_standard，则一个领头的符号将被认为是应用于所有域（但是仅当没有额外符号出现）。否则将使用传统的UXDB解释。为了避免混淆，我们推荐在任何域为负值时为每一个域都附加一个显式的符号。

在冗长的输入格式中，以及在更紧凑输入格式的某些域中，域值可以有分数部分；例如'1.5 week'或'01:02:03.45'。这样的输入被转换为合适的月数、日数和秒数用于存储。当这样会导致月和日中的分数时，分数被加到低序域中，使用的转换因子是1月=30日和1日=24小时。例如，'1.5 month'会变成1月和15日。只有秒总是在输出时被显示为分数。

表　5.18 “间隔输入”展示了一些有效interval输入的例子。

表　5.18.　间隔输入

例子	描述
1-2	SQL标准格式：1年2个月
3 4:05:06	SQL标准格式：3日4小时5分钟6秒
1 year 2 months 3 days 4 hours 5 minutes 6 seconds	UXDB格式：1年2个月3日4小时5分钟6秒钟
P1Y2M3DT4H5M6S	“带标志符的”ISO 8601 格式：含义同上
P0001-02-03T04:05:06	ISO 8601 的“替代格式”：含义同上

在内部，interval值被存储为months、days以及seconds。之所以这样做是因为一个月中的天数是变化的，并且在涉及到夏令时调整时一天可以有23或者25个小时。months以及days域是整数，而seconds域可以存储分数。因为区间通常是从常量字符串或者timestamp减法创建而来，这种存储方法在大部分情况下都很好，但是也可能导致预料之外的结果：

```
SELECT EXTRACT(hours from '80 minutes'::interval);
 date_part
-----------
         1

SELECT EXTRACT(days from '80 hours'::interval);
 date_part
-----------
         0
```

函数justify_days和justify_hours可以用来调整溢出其正常范围之外的days和hours。

5.5.5.　间隔输出

间隔类型的输出格式可以被设置为四种风格之一：sql_standard、uxdb、uxdb_verbose或iso_8601，设置方法使用SET　intervalstyle命令。默认值为uxdb格式。表　5.19 “间隔输出风格例子”展示了每种输出风格的例子。

如果间隔值符合SQL标准的限制（仅年-月或仅日-时间，没有正负值部分的混合），sql_standard风格为间隔文字串产生符合SQL标准规范的输出。否则输出将看起来像一个标准的年-月文字串跟着一个日-时间文字串，并且带有显式添加的符号以区分混合符号的间隔。

iso_8601风格的输出匹配在ISO 8601标准的“带标志符的格式”。

表　5.19.　间隔输出风格例子

风格声明	年-月间隔	日-时间间隔	混合间隔
sql_standard	1-2	3 4:05:06	-1-2 +3 -4:05:06

风格声明	年-月间隔	日-时间间隔	混合间隔
uxdb	1 year 2 mons	3 days 04:05:06	-1 year -2 mons +3 days -04:05:06
uxdb_verbose	@ 1 year 2 mons	@ 3 days 4 hours 5 mins 6 secs	@ 1 year 2 mons -3 days 4 hours 5 mins 6 secs ago
iso_8601	P1Y2M	P3DT4H5M6S	P-1Y-2M3DT-4H-5M-6S

5.6. 布尔类型

UXDB提供标准的SQL类型boolean，参见表 5.20 “布尔数据类型”。boolean可以有多个状态：“true（真）”、“false（假）”和第三种状态“unknown（未知）”，未知状态由SQL空值表示。

表 5.20. 布尔数据类型

名字	存储字节	描述
boolean	1字节	状态为真或假

在SQL查询中，布尔常量可以表示为SQL关键字TRUE, FALSE, 和 NULL.

boolean 类型的数据类型输入函数接受这些字符串表示“真”状态:

true
yes
on
1
下面这些表示“假” 状态:

false
no
off
0
这些字符串的唯一前缀也可以接受，例如t 或 n. 前端或尾部的空格将被忽略，并且大小写不敏感。

boolean类型的数据类型输出函数总是发出 t 或 f，如例 5.2 “使用boolean类型”所示。

例 5.2. 使用boolean类型

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic est');
INSERT INTO test1 VALUES (FALSE, 'non est');
SELECT * FROM test1;
 a |    b
---+---------
 t | sic est
 f | non est

SELECT * FROM test1 WHERE a;
```

```
 a |    b
---+---------
 t | sic est
```

在SQL查询中优先使用关键字TRUE　和　FALSE来写布尔常数(SQL-兼容)。　但是也可以使用遵循第　1.1.2.7　节　“其他类型的常量”中描述的通用字符串文字常量句法的字符串来表达，例如'yes'::boolean.

注意语法分析程序会把TRUE　和　FALSE　自动理解为boolean类型，但是不包括NULL　，因为它可以是任何类型的。　因此在某些语境中也许要将　NULL　转化为显示boolean类型，例如NULL::boolean.　反过来，上下文中的字符串文字布尔值也可以不转换，当语法分析程序能够断定文字必定为boolean类型时。

5.7.　枚举类型

枚举（enum）类型是由一个静态、值的有序集合构成的数据类型。它们等效于很多编程语言所支持的enum类型。枚举类型的一个例子可以是一周中的日期，或者一个数据的状态值集合。

5.7.1.　枚举类型的声明

枚举类型可以使用CREATE TYPE(7)命令创建，例如：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

一旦被创建，枚举类型可以像很多其他类型一样在表和函数定义中使用：

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
CREATE TABLE person (
    name text,
    current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
 name | current_mood
------+--------------
 Moe  | happy
(1 row)
```

5.7.2.　排序

一个枚举类型的值的排序是该类型被创建时所列出的值的顺序。枚举类型的所有标准的比较操作符以及相关聚集函数都被支持。例如：

```
INSERT INTO person VALUES ('Larry', 'sad');
INSERT INTO person VALUES ('Curly', 'ok');
SELECT * FROM person WHERE current_mood > 'sad';
 name  | current_mood
-------+--------------
 Moe   | happy
 Curly | ok
```

```
(2 rows)

SELECT * FROM person WHERE current_mood > 'sad' ORDER BY current_mood;
 name  | current_mood
-------+--------------
 Curly | ok
 Moe   | happy
(2 rows)

SELECT name
FROM person
WHERE current_mood = (SELECT MIN(current_mood) FROM person);
 name
-------
 Larry
(1 row)
```

5.7.3. 类型安全性

每一种枚举数据类型都是独立的并且不能和其他枚举类型相比较。看这样一个例子：

```
CREATE TYPE happiness AS ENUM ('happy', 'very happy', 'ecstatic');
CREATE TABLE holidays (
    num_weeks integer,
    happiness happiness
);
INSERT INTO holidays(num_weeks,happiness) VALUES (4, 'happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (6, 'very happy');
INSERT INTO holidays(num_weeks,happiness) VALUES (8, 'ecstatic');
INSERT INTO holidays(num_weeks,happiness) VALUES (2, 'sad');
ERROR:  invalid input value for enum happiness: "sad"
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood = holidays.happiness;
ERROR:  operator does not exist: mood = happiness
```

如果确实需要做这样的事情，可以写一个自定义的操作符或者在查询中加上显式造型：

```
SELECT person.name, holidays.num_weeks FROM person, holidays
  WHERE person.current_mood::text = holidays.happiness::text;
 name | num_weeks
------+-----------
 Moe  |         4
(1 row)
```

5.7.4. 实现细节

枚举标签是大小写敏感的，因此'happy'与'HAPPY'是不同的。标签中的空格也是有意义的。

尽管枚举类型的主要目的是用于值的静态集合，但也有方法在现有枚举类型中增加新值和重命名值（见ALTER TYPE(7)）。不能从枚举类型中去除现有的值，也不能更改这些值的排序顺序，如果要那样做可以删除并且重建枚举类型。

一个枚举值在磁盘上占据4个字节。一个枚举值的文本标签的长度受限于NAMEDATALEN设置，该设置被编译在UXDB中，在标准编译下它表示最多63字节。

从内部枚举值到文本标签的翻译被保存在系统目录ux_enum中。可以直接查询该目录。

5.8. 几何类型

几何数据类型表示二维的空间物体。表 5.21 “几何类型”展示了UXDB中可以用的几何类型。

表 5.21. 几何类型

名字	存储尺寸	表示	描述
point	16字节	平面上的点	(x,y)
line	32字节	无限长的线	{A,B,C}
lseg	32字节	有限线段	((x1,y1),(x2,y2))
box	32字节	矩形框	((x1,y1),(x2,y2))
path	16+16n字节	封闭路径（类似于多边形）	((x1,y1),...)
path	16+16n字节	开放路径	[(x1,y1),...]
polygon	40+16n字节	多边形（类似于封闭路径）	((x1,y1),...)
circle	24字节	圆	<(x,y),r>（中心点和半径）

我们有一系列丰富的函数和操作符可用来进行各种几何操作， 如缩放、平移、旋转和计算相交等 它们在第 6.11 节 “几何函数和操作符”中解释。

5.8.1. 点

点是几何类型的基本二维构造块。用下面的语法描述point类型的值：

$$(x,y)$$
$$x,y$$

其中x和y分别是坐标，都是浮点数。

点使用第一种语法输出。

5.8.2. 线

线由线性方程$Ax + By + C = 0$ 表示，其中A和B都不为零。类型line 的值采用以下形式输入和输出：

$$\{A,B,C\}$$

另外，还可以用下列任一形式输入：

$$[(x1,y1),(x2,y2)]$$

((*x1* , *y1*) , (*x2* , *y2*))
(*x1* , *y1*) , (*x2* , *y2*)
x1 , *y1* , *x2* , *y2*

其中 (*x1*,*y1*) 和 (*x2*,*y2*) 是线上不同的两点。

5.8.3. 线段

线段用一对线段的端点来表示。lseg类型的值用下面的语法声明：

[(*x1* , *y1*) , (*x2* , *y2*)]
((*x1* , *y1*) , (*x2* , *y2*))
(*x1* , *y1*) , (*x2* , *y2*)
x1 , *y1* , *x2* , *y2*

其中(*x1*,*y1*) 和 (*x2*,*y2*) 是线段的端点。

线段使用第一种语法输出。

5.8.4. 方框

方框用其对角的点对表示。box类型的值使用下面的语法指定：

((*x1* , *y1*) , (*x2* , *y2*))
(*x1* , *y1*) , (*x2* , *y2*)
x1 , *y1* , *x2* , *y2*

其中(*x1*,*y1*) 和 (*x2*,*y2*) 是方框的对角点。

方框使用第二种语法输出。

在输入时可以提供任意两个对角，但是值将根据需要被按顺序记录为右上角和左下角。

5.8.5. 路径

路径由一系列连接的点组成。路径可能是开放的，也就是认为列表中第一个点和最后一个点没有被连接起来；也可能是封闭的，这时认为第一个和最后一个点被连接起来。

path类型的值用下面的语法声明：

[(*x1* , *y1*) , ... , (*xn* , *yn*)]
((*x1* , *y1*) , ... , (*xn* , *yn*))
(*x1* , *y1*) , ... , (*xn* , *yn*)
(*x1* , *y1* , ... , *xn* , *yn*)
x1 , *y1* , ... , *xn* , *yn*

其中的点是组成路径的线段的端点。方括弧（[]）表示一个开放的路径，圆括弧（()）表示一个封闭的路径。如第三种到第五种语法所示，当最外面的圆括号被忽略时，路径将被假定为封闭。

路径的输出使用第一种或第二种语法。

5.8.6. 多边形

多边形由一系列点代表（多边形的顶点）。多边形和封闭路径很像，但是存储方式不一样而且有自己的一套支持例程。

polygon类型的值用下列语法声明：

$$((x1,y1),\ldots,(xn,yn))$$
$$(x1,y1),\ldots,(xn,yn)$$
$$(x1,y1\ ,\ldots,\ xn,yn)$$
$$x1,y1\ ,\ldots,\ xn,yn$$

其中的点是组成多边形边界的线段的端点。

多边形的输出使用第一种语法。

5.8.7. 圆

圆由一个圆心和一个半径代表。circle类型的值用下面的语法指定：

$$<(x,y),r>$$
$$((x,y),r)$$
$$(x,y),r$$
$$x,y\ ,r$$

其中(x,y)是圆心，而r是圆的半径。

圆的输出用第一种语法。

5.9. 网络地址类型

UXDB提供用于存储 IPv4、IPv6 和 MAC 地址的数据类型，如表 5.22 “网络地址类型”所示。用这些数据类型存储网络地址比用纯文本类型好，因为这些类型提供输入错误检查以及特殊的操作符和函数（见第 6.12 节 “网络地址函数和操作符”）

表 5.22. 网络地址类型

名字	存储尺寸	描述
cidr	7或19字节	IPv4和IPv6网络
inet	7或19字节	IPv4和IPv6主机以及网络
macaddr	6字节	MAC地址
macaddr8	8 bytes	MAC地址（EUI-64格式）

在对inet或者cidr数据类型进行排序的时候， IPv4 地址将总是排在 IPv6 地址前面，包括那些封装或者是映射在 IPv6 地址里 的 IPv4 地址，例如 ::10.2.3.4 或者 ::ffff::10.4.3.2。

5.9.1. inet

inet在一个数据域里保存一个 IPv4 或 IPv6 主机地址，以及一个可选的它的子网。 子网由主机地址中表示的网络地址位数表示（“网络掩码”）。 如果网络掩码为 32 并且地址是 IPv4 ，那么该值不表示任何子网，只是一台主机。在 IPv6 中地址长度是 128 位，因此 128 位指定一个唯一的主机地址。 请注意如果想只接受网络地址，应该使用cidr类型而不是inet。

该类型的输入格式是地址/y，其中地址是一个 IPv4 或者 IPv6 地址，*y*是网络掩码的位数。如果*y*部分缺失， 则网络掩码对 IPv4 而言是 32，对 IPv6 而言是 128，所以该值表示只有一台主机。在显示时，如果*y*部分指定一个单台主机，它将不会被显示出来。

5.9.2. cidr

cidr类型保存一个 IPv4 或 IPv6 网络地址声明。其输入和输出遵循无类的互联网域路由（Classless Internet Domain Routing）习惯。声明一个网络的格式是地址/y，其中*address*是IPv4 或 IPv6 网络地址而*y*是网络掩码的位数。如果省略*y*， 那么掩码部分用旧的有类的网络编号系统进行计算，否则它将至少大到足以包括写在输入中的所有字节。声明一个在其指定的掩码右边置了位的网络地址会导致错误。

表 5.23 “cidr类型输入例子”展示了一些例子。

表 5.23. cidr类型输入例子

cidr输入	cidr输出	**abbrev(cidr)**
192.168.100.128/25	192.168.100.128/25	192.168.100.128/25
192.168/24	192.168.0.0/24	192.168.0/24
192.168/25	192.168.0.0/25	192.168.0.0/25
192.168.1	192.168.1.0/24	192.168.1/24
192.168	192.168.0.0/24	192.168.0/24
128.1	128.1.0.0/16	128.1/16
128	128.0.0.0/16	128.0/16
128.1.2	128.1.2.0/24	128.1.2/24
10.1.2	10.1.2.0/24	10.1.2/24
10.1	10.1.0.0/16	10.1/16
10	10.0.0.0/8	10/8
10.1.2.3/32	10.1.2.3/32	10.1.2.3/32
2001:4f8:3:ba::/64	2001:4f8:3:ba::/64	2001:4f8:3:ba::/64
2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1/128	2001:4f8:3:ba:2e0:81ff:fe22:d1f1
::ffff:1.2.3.0/120	::ffff:1.2.3.0/120	::ffff:1.2.3/120
::ffff:1.2.3.0/128	::ffff:1.2.3.0/128	::ffff:1.2.3.0/128

5.9.3. inet vs. cidr

inet和cidr类型之间的本质区别是inet接受右边有非零位的网络掩码， 而cidr不接受。例如，192.168.0.1/24对inet是有效的，但对cidr是无效的。

> **提示**
>
> 如果不喜欢inet或cidr值的输出格式，可以尝试函数host、text和abbrev。

5.9.4. macaddr

macaddr类型存储 MAC 地址，也就是以太网卡硬件地址 （尽管 MAC 地址还用于其它用途）。可以接受下列格式的输入：

'08:00:2b:01:02:03'
'08-00-2b-01-02-03'
'08002b:010203'
'08002b-010203'
'0800.2b01.0203'
'0800-2b01-0203'
'08002b010203'
这些例子指定的都是同一个地址。对于位a到f，大小写都可以接受。输出总是使用展示的第一种形式。

IEEE Std 802-2001 指定第二种展示的形式（带有连字符）作为MAC地址的标准形式，并且指定第一种形式（带有分号）作为位翻转的记号，因此 08-00-2b-01-02-03 = 01:00:4D:08:04:0C。这种习惯目前已经被广泛地忽略，并且它只与废弃的网络协议（如令牌环）相关。UXDB 没有对位翻转做任何规定，并且所有可接受的格式都使用标准的LSB顺序。

剩下的五种输入格式不属于任何标准。

5.9.5. macaddr8

macaddr8类型以EUI-64格式存储MAC地址，例如以太网卡的硬件地址（尽管MAC地址也被用于其他目的）。这种类型可以接受6字节和8字节长度的MAC地址，并且将它们存储为8字节长度的格式。以6字节格式给出的MAC地址被存储为8字节长度格式的方式是吧第4和第5字节分别设置为FF和FE。 注意IPv6使用一种修改过的EUI-64格式，其中从EUI-48转换过来后的第7位应该被设置为一。函数macaddr8_set7bit被用来做这种修改。 一般而言，任何由16进制数（字节边界上）对构成的输入（可以由':'、'-'或者'.'统一地分隔）都会被接受。16进制数的数量必须是16（8字节）或者12（6字节）。前导和拖尾的空格会被忽略。 下面是可以被接受的输入格式的例子：

'08:00:2b:01:02:03:04:05'
'08-00-2b-01-02-03-04-05'
'08002b:0102030405'
'08002b-0102030405'
'0800.2b01.0203.0405'
'0800-2b01-0203-0405'
'08002b01:02030405'
'08002b0102030405'
这些例子都指定相同的地址。数字a到f的大小写形式都被接受。输出总是以上面显示的第一种形式。 上述的后六种输入格式不属于任何标准。 要把EUI-48格式的传统48位MAC地址转换成修改版EUI-64格式（包括在IPv6地址中作为主机部分），可以使用下面的macaddr8_set7bit：

```
SELECT macaddr8_set7bit('08:00:2b:01:02:03');

   macaddr8_set7bit
-------------------------
 0a:00:2b:ff:fe:01:02:03
(1 row)
```

5.10. 位串类型

位串就是一串 1 和 0 的串。它们可以用于存储和可视化位掩码。我们有两种类型的 SQL 位类型：bit(*n*)和bit varying(*n*)，其中 *n*是一个正整数。

bit类型的数据必须准确匹配长度*n*；试图存储短些或者长一些的位串都是错误的。bit varying数据是最长*n*的变长类型，更长的串会被拒绝。写一个没有长度的bit等效于 bit(1)，没有长度的bit varying意味着没有长度限制。

> **注意**
>
> 如果我们显式地把一个位串值转换成bit(*n*)，那么它的右边将被截断或者在右边补齐零，直到刚好*n*位，而且不会抛出任何错误。类似地，如果我们显式地把一个位串数值转换成bit varying(*n*)，如果它超过了*n*位，那么它的右边将被截断。

请参考第 1.1.2.5 节 “位串常量”获取有关位串常量的语法的信息。还有一些位逻辑操作符和串操作函数可用，请见第 6.6 节 “位串函数和操作符”。

例 5.3. 使用位串类型

```
CREATE TABLE test (a BIT(3), b BIT VARYING(5));
INSERT INTO test VALUES (B'101', B'00');
INSERT INTO test VALUES (B'10', B'101');

ERROR:  bit string length 2 does not match type bit(3)

INSERT INTO test VALUES (B'10'::bit(3), B'101');
SELECT * FROM test;

 a | b
-----+-----
 101 | 00
 100 | 101
```

一个位串值对于每8位的组需要一个字节，外加总共5个或8个字节，这取决于串的长度（但是长值可能被压缩或者移到线外，如第 5.3 节 “字符类型”中对字符串的解释一样）。

5.11. 文本搜索类型

UXDB提供两种数据类型，它们被设计用来支持全文搜索，全文搜索是一种在自然语言的文档集合中搜索以定位那些最匹配一个查询的文档的活动。tsvector类型表示一个为文本搜索优化的形式下的文档，tsquery类型表示一个文本查询。第 9 章 全文搜索提供了对于这种功能的详细解释，并且第 6.13 节 “文本搜索函数和操作符”总结了相关的函数和操作符。

5.11.1. tsvector

一个tsvector值是一个排序的可区分词位的列表，词位是被正规化合并了同一个词的不同变种的词（详见第 9 章 全文搜索。排序和去重是在输入期间自动完成的，如下例所示：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector;
```

```
                    tsvector
-----------------------------------------------------
 'a' 'and' 'ate' 'cat' 'fat' 'mat' 'on' 'rat' 'sat'
```

要表示包含空白或标点的词位，将它们用引号包围：

```
SELECT $$the lexeme '    ' contains spaces$$::tsvector;
                tsvector
-------------------------------------------
 '    ' 'contains' 'lexeme' 'spaces' 'the'
```

（我们在这个例子中使用美元符号包围的串文字并且下一个用来避免在文字中包含双引号记号产生的混淆）。嵌入的引号和反斜线必须被双写：

```
SELECT $$the lexeme 'Joe"s' contains a quote$$::tsvector;
                  tsvector
-----------------------------------------------
 'Joe"s' 'a' 'contains' 'lexeme' 'quote' 'the'
```

可选的，整数位置可以被附加给词位：

```
SELECT 'a:1 fat:2 cat:3 sat:4 on:5 a:6 mat:7 and:8 ate:9 a:10 fat:11 rat:12'::tsvector;
                              tsvector
-------------------------------------------------------------------------------
 'a':1,6,10 'and':8 'ate':9 'cat':3 'fat':2,11 'mat':7 'on':5 'rat':12 'sat':4
```

一个位置通常表示源词在文档中的定位。位置信息可以被用于邻近排名。位置值可以从 1 到 16383，更大的数字会被 16383。对于相同的词位出现的重复位置将被丢弃。

具有位置的词位可以进一步地被标注一个权重，它可以是A、 B、C或D。 D是默认值并且因此在输出中不会显示：

```
SELECT 'a:1A fat:2B,4C cat:5D'::tsvector;
          tsvector
-----------------------------
 'a':1A 'cat':5 'fat':2B,4C
```

权重通常被用来反映文档结构，例如将主题词标记成与正文词不同。文本搜索排名函数可以为不同的权重标记器分配不同的优先级。

了解tsvector类型本身并不执行任何词正规化这一点很重要，它假定给它的词已经被恰当地为应用正规化过。例如，

```
SELECT 'The Fat Rats'::tsvector;
     tsvector
--------------------
 'Fat' 'Rats' 'The'
```

对于大部分英语文本搜索应用，上面的词将会被认为是非正规化的，但是tsvector并不在乎这一点。原始文档文本通常应该经过to_tsvector以恰当地为搜索正规化其中的词：

```
SELECT to_tsvector('english', 'The Fat Rats');
  to_tsvector
-----------------
 'fat':2 'rat':3
```

再次地，详情请参阅第 9 章 全文搜索

5.11.2. tsquery

一个tsquery值存储要用于搜索的词位，并且使用布尔操作符&（AND）、|（OR）和!（NOT）来组合它们，还有短语搜索操作符<->（FOLLOWED BY）。也有一种 FOLLOWED BY 操作符的变体<*N*>，其中*N*是一个整数常量，它指定要搜索的两个词位之间的距离。<->等效于<1>。

圆括号可以被用来强制对操作符分组。如果没有圆括号，!（NOT）的优先级最高，其次是<->（FOLLOWED BY），然后是&（AND），最后是|（OR）。

这里有一些例子：

```
SELECT 'fat & rat'::tsquery;
    tsquery
---------------
 'fat' & 'rat'

SELECT 'fat & (rat | cat)'::tsquery;
          tsquery
----------------------------
 'fat' & ( 'rat' | 'cat' )

SELECT 'fat & rat & ! cat'::tsquery;
        tsquery
------------------------
 'fat' & 'rat' & !'cat'
```

可选地，一个tsquery中的词位可以被标注一个或多个权重字母，这将限制它们只能和具有那些权重之一的tsvector词位相匹配：

```
SELECT 'fat:ab & cat'::tsquery;
    tsquery
------------------
 'fat':AB & 'cat'
```

此外，一个tsquery中的词位可以被标注为*来指定前缀匹配：

```
SELECT 'super:*'::tsquery;
  tsquery
-----------
 'super':*
```

这个查询将匹配一个tsvector中以“super”开头的任意词。

词位的引号规则和之前描述的tsvector中的词位相同；并且，正如tsvector，任何请求的词正规化必须在转换到tsquery类型之前完成。to_tsquery函数可以方便地执行这种正规化：

```
SELECT to_tsquery('Fat:ab & Cats');
    to_tsquery
------------------
 'fat':AB & 'cat'
```

注意to_tsquery将会以和其他词同样的方式处理前缀，这也意味着下面的比较会返回真：

```
SELECT to_tsvector( 'postgraduate' ) @@ to_tsquery( 'uxdb:*' );
 ?column?
----------
 t
```

因为uxdb会被处理成postgr：

```
SELECT to_tsvector( 'postgraduate' ), to_tsquery( 'uxdb:*' );
  to_tsvector  | to_tsquery
---------------+------------
 'postgradu':1 | 'postgr':*
```

这会匹配postgraduate被处理后的形式。

5.12. UUID类型

数据类型uuid存储由RFC 4122、ISO/IEC 9834-8:2005以及相关标准定义的通用唯一标识符（UUID）（某些系统将这种数据类型引用为全局唯一标识符GUID）。这种标识符是一个128位的量，它由一个精心选择的算法产生，该算法能保证在已知空间中任何其他使用相同算法的人能够产生同一个标识符的可能性非常非常小。因此，对于分布式系统，这些标识符相比序列生成器而言提供了一种很好的唯一性保障，序列生成器只能在一个数据库中保证唯一。

一个UUID被写成一个小写十六进制位的序列，该序列被连字符分隔成多个组：首先是一个8位组，接下来是三个4位组，最后是一个12位组。总共的32位（十六进制位）表示了128个二进制位。一个标准形式的UUID类似于：

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

UXDB也接受另一种输入形式： 使用大写位、标准格式被花括号包围、忽略某些或者全部连字符、在任意4位组后面增加一个连字符。例如：

```
A0EEBC99-9C0B-4EF8-BB6D-6BB9BD380A11
{a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11}
a0eebc999c0b4ef8bb6d6bb9bd380a11
a0ee-bc99-9c0b-4ef8-bb6d-6bb9-bd38-0a11
{a0eebc99-9c0b4ef8-bb6d6bb9-bd380a11}
```

输出总是采用标准形式。

UXDB为UUID提供了存储和比较函数，但是核心数据库不包含任何用于产生UUID的函数，因为没有一个单一算法能够很好地适应每一个应用。uuid-ossp模块提供了实现一些标准算法的函数。uxcrypto模块也为随机 UUID 提供了一个生成函数。 此外，UUID可以由客户端应用产生，或者由通过服务器端函数调用的其他库生成。

5.13. XML类型

xml数据类型可以被用来存储XML数据。它比直接在一个text域中存储XML数据的优势在于，它会检查输入值的结构是不是良好，并且有支持函数用于在其上执行类型安全的操作，参见第 6.14 节 “XML 函数”。使用这种数据类型要求在安装时用configure --with-libxml选项编译。

xml类型可以存储格式良好的遵循XML标准定义的“文档”，以及“内容”片段，它是通过引用更宽泛的“document node” XQuery 和 XPath 数据模型来定义的。 大致上说，这意味着内容片段中可以有多于一个的顶层元素或字符节点。 表达式*xmlvalue* IS DOCUMENT可以被用来评估一个特定的xml值是一个完整文档或者仅仅是一个文档片段。

5.13.1. 创建XML值

要从字符数据中生成一个xml类型的值，可以使用函数xmlparse:

XMLPARSE ({ DOCUMENT | CONTENT } *value*)

例子：

```
XMLPARSE (DOCUMENT '<?xml version="1.0"?><book><title>Manual</title><chapter>...</
chapter></book>')
XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')
```

然而根据SQL标准这是唯一将字符串转换为XML值的方法，UXDB特有的语法：

```
xml '<foo>bar</foo>'
'<foo>bar</foo>'::xml
```

也可以被使用。

即便输入值指定了一个文档类型声明（DTD），xml类型也不根据DTD来验证输入值。目前也没有内建的支持用于根据其他XML模式语言（如XML模式）来进行验证。

作为一个逆操作，从xml产生一个字符串可以使用函数xmlserialize:

XMLSERIALIZE ({ DOCUMENT | CONTENT } *value* AS *type*)

*type*可以是 character、character varying或 text（或者其中之一的一个别名）。再次地，根据SQL标准，这也是在xml类型和字符类型间做转换的唯一方法，但是UXDB也允许简单地造型这些值。

当一个字符串不是使用XMLPARSE造型成xml或者不是使用XMLSERIALIZE从xml造型得到，对于DOCUMENT和CONTENT两者的选择是根据“XML option” 会话配置参数决定的，它可以使用标准命令来设置：

SET XML OPTION { DOCUMENT | CONTENT };

或者是更具有UXDB风格的语法

SET xmloption TO { DOCUMENT | CONTENT };

默认值是CONTENT，因此所有形式的XML数据都被允许。

5.13.2. 编码处理

在客户端、服务器以及其中流过的XML数据上处理多字符编码时必须要注意。在使用文本模式向服务器传递查询以及向客户端传递查询结果（在普通模式）时，UXDB将所有在客户端和服务器之间传递的字符数据转换为目标端的字符编码。这也包括了表示XML值的串，正如上面的例子所述。这也通常意味着由于字符数据会在客户端和服务器之间传递时被转换成其他编码，包含在XML数据中的编码声明可能是无效的，因为内嵌的编码声明没有被改变。为了处理这种行为，包含在表示xml类型输入的字符串中包含的编码声明会被忽略，并且其内容被假定为当前服务器的编码。接着，为了正确处理，XML数据的字符串必须以当前客户端编码从客户端发出。客户端负责在把文档发送给服务器之前将它们转换为当前客户端编码，或者适当地调整客户端编码。在输出时，xml类型的值将不会有一个编码声明，并且客户端将会假设所有数据都是当前客户端编码。

在使用二进制模式传送查询参数给服务器以及传回查询结果给客户端时，不会执行编码转换，因此情况就有所不同。在这种情况下，XML数据中的编码声明将被注意到，并且如果缺少编码声明时该数据会被假定为UTF-8（由于XML标准的要求，注意UXDB不支持UTF-16）。在输出时，数据将会有一个编码声明来指定客户端编码，除非客户端编码为UTF-8（这种情况下编码声明会被忽略）。

不用说，在UXDB中处理XML数据产生错误的可能性更小，并且在XML数据编码、客户端编码和服务器编码三者相同时效率更高。因为XML数据在内部是以UTF-8处理的，如果服务器编码也是UTF-8时，计算效率将会最高。

> **注意**
>
> 当服务器编码不是UTF-8时，某些XML相关的函数可能在非ASCII数据上完全无法工作。尤其在xmltable()和xpath()上，这是一个已知的问题。

5.13.3. 访问XML值

xml数据类型有些不同寻常，因为它不提供任何比较操作符。这是因为对于XML数据不存在良定义的和通用的比较算法。这种状况造成的后果就是，无法通过比较一个xml和一个搜索值来检索行。XML值因此通常应该伴随着一个独立键值域，如一个ID。另一种比较XML值的方案是将它们先转换为字符串，但注意字符串比较对于XML比较方法没有什么帮助。

由于没有可以用于xml数据类型的比较操作符，因此无法直接在这种类型上创建索引。如果需要在XML中快速的搜索，可能的解决方案包括将表达式造型为一个字符串类型然后索引之，或者在一个XPath表达式上索引。当然，实际的查询必须被调整为使用被索引的表达式。

UXDB中的文本搜索功能也可以被用来加速XML数据的全文搜索。但是，所需的预处理支持目前在UXDB发布中还不可用。

5.14. JSON 类型

根据RFC 7159 中的说明，JSON 数据类型是用来存储 JSON（JavaScript Object Notation） 数据的。这种数据也可以被存储为text，但是 JSON 数据类型的 优势在于能强制要求每个被存储

的值符合 JSON 规则。也有很多 JSON 相关的函 数和操作符可以用于存储在这些数据类型中的数据，见 第 6.15 节 “JSON 函数和操作符”。

UXDB 提供存储JSON数据的两种类型：json 和 jsonb。 为了实现这些数据类型高效的查询机制，UXDB还在第 5.14.6 节 “jsonpath Type”中提供了jsonpath数据类型描述。

json 和 jsonb数据类型接受几乎完全相同的值集合作为输入。 主要的实际区别之一是效率。json数据类型存储输入文本的精准拷贝，处理函数必须在每 次执行时必须重新解析该数据。而jsonb数据被存储在一种分解好的 二进制格式中，它在输入时要稍慢一些，因为需要做附加的转换。但是 jsonb在处理时要快很多，因为不需要解析。jsonb也支 持索引，这也是一个令人瞩目的优势。

由于json类型存储的是输入文本的准确拷贝，其中可能会保留在语法 上不明显的、存在于记号之间的空格，还有 JSON 对象内部的键的顺序。还有， 如果一个值中的 JSON 对象包含同一个键超过一次，所有的键/值对都会被保留（ 处理函数会把最后的值当作有效值）。相反，jsonb不保留空格、不 保留对象键的顺序并且不保留重复的对象键。如果在输入中指定了重复的键，只有 最后一个值会被保留。

通常，除非有特别特殊的需要（例如遗留的对象键顺序假设），大多数应用应该 更愿意把 JSON 数据存储为jsonb。

UXDB对每个数据库只允许一种 字符集编码。因此 JSON 类型不可能严格遵守 JSON 规范，除非数据库编码 是 UTF8。尝试直接包括数据库编码中无法表示的字符将会失败。反过来，能 在数据库编码中表示但是不在 UTF8 中的字符是被允许的。

RFC 7159 允许 JSON 字符串包含\u*XXXX* 所标记的 Unicode 转义序列。在json类型的输入函数中，不管数据库 编码如何都允许 Unicode 转义，并且只检查语法正确性（即，跟在\u 后面的四个十六进制位）。但是，jsonb的输入函数更加严格：它不允 许非 ASCII 字符的 Unicode 转义（高于U+007F的那些），除非数据 库编码是 UTF8。jsonb类型也拒绝\u0000（因为 UXDB的text类型无法表示 它），并且它坚持使用 Unicode 代理对来标记位于 Unicode 基本多语言平面之外 的字符是正确的。合法的 Unicode 转义会被转换成等价的 ASCII 或 UTF8 字符进 行存储，这包括把代理对折叠成一个单一字符。

> **注意**
>
> 很多第 6.15 节 “JSON 函数和操作符”中描述的 JSON 处理函数将把 Unicode 转义转换成常规字符，并且将因此抛出和刚才所描述的同样类型的错误（即使它们 的输入是类型json而不是jsonb）。json的 输入函数不做这些检查是由来已久的，不过它确实允许将 JSON Unicode 转义简单的（不经处理）存储在一个非 UTF8 数据库编码中。通常，最好尽可能避免在一个非 UTF8 数据库编码的 JSON 中混入 Unicode 转义。

在把文本 JSON 输入转换成jsonb时，RFC 7159描述 的基本类型会被有效地映射到原生的 UXDB类型（如 表 5.24 “JSON 基本类型和相应的UXDB类型”中所示）。因此，在合法 jsonb数据的组成上有一些次要额外约束，它们不适合 json类型和抽象意义上的 JSON，这些约束对应于有关哪些东西不 能被底层数据类型表示的限制。尤其是，jsonb将拒绝位于 UXDB numeric数据类型范 围之外的数字，而json则不会。这类实现定义的限制是 RFC 7159 所允许的。不过，实际上这类问题更可能发生在其他实 现中，因为把 JSON 的number基本类型表示为 IEEE 754 双精度浮点 是很常见的（这也是RFC 7159 明确期待和允许的）。当在这类系 统间使用 JSON 作为一种交换格式时，应该考虑丢失数字精度的风险。

相反地，如表中所述，有一些 JSON 基本类型输入格式上的次要限制并不适用于相 应的UXDB类型。

表 5.24. JSON 基本类型和相应的UXDB类型

JSON 基本类型	UXDB类型	注释
string	text	不允许\u0000，如果数据库编码不是 UTF8，非 ASCII Unicode 转义也是这样
number	numeric	不允许NaN 和 infinity值
boolean	boolean	只接受小写true和false拼写
null	(无)	SQL NULL是一个不同的概念

5.14.1. JSON 输入和输出语法

RFC 7159 中定义了 JSON 数据类型的输入/输出语法。

下列都是合法的json（或者jsonb）表达式：

```
-- 简单标量/基本值
-- 基本值可以是数字、带引号的字符串、true、false或者null
SELECT '5'::json;

-- 有零个或者更多元素的数组（元素不需要为同一类型）
SELECT '[1, 2, "foo", null]'::json;

-- 包含键值对的对象
-- 注意对象键必须总是带引号的字符串
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json;

-- 数组和对象可以被任意嵌套
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

如前所述，当一个 JSON 值被输入并且接着不做任何附加处理就输出时， json会输出和输入完全相同的文本，而jsonb 则不会保留语义上没有意义的细节（例如空格）。例如，注意下面的不同：

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
                      json
-------------------------------------------------
 {"bar": "baz", "balance": 7.77, "active":false}
(1 row)

SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
                      jsonb
-------------------------------------------------
 {"bar": "baz", "active": false, "balance": 7.77}
(1 row)
```

值得一提的一种语义上无意义的细节是，在jsonb中数据会被按照底层 numeric类型的行为来打印。实际上，这意味着用E记号 输入的数字被打印出来时就不会有该记号，例如：

```
SELECT '{"reading": 1.230e-5}'::json, '{"reading": 1.230e-5}'::jsonb;
       json        |        jsonb
-------------------+-------------------------
 {"reading": 1.230e-5} | {"reading": 0.00001230}
(1 row)
```

不过，如这个例子所示，jsonb将会保留拖尾的小数点后的零，即便这 对于等值检查等目的来说是语义上无意义的。

对于可用于构造和处理 JSON 值的内置函数和运算符的列表，参见 第 6.15 节 “JSON 函数和操作符”。

5.14.2. 设计 JSON 文档

将数据表示为 JSON 比传统关系数据模型要灵活得多，在需求不固定时 这种优势更加令人感兴趣。在同一个应用里非常有可能有两种方法共存 并且互补。不过，即便是在要求最大灵活性的应用中，我们还是推荐 JSON 文档有固定的结构。该结构通常是非强制的（尽管可能会强制一 些业务规则），但是有一个可预测的结构会使书写概括一个表中的 “文档”（数据）集合的查询更容易。

当被存储在表中时，JSON 数据也像其他数据类型一样服从相同的并发 控制考虑。尽管存储大型文档是可行的，但是要记住任何更新都在整行 上要求一个行级锁。为了在更新事务之间减少锁争夺，可考虑把 JSON 文档限制到一个可管理的尺寸。理想情况下，JSON 文档应该每个表示 一个原子数据，业务规则命令不会进一步把它们划分成更小的可独立修 改的数据。

5.14.3. jsonb 包含和存在

测试包含是jsonb的一种重要能力。对 json类型没有平行的功能集。包含测试会测试一个 jsonb文档是否被包含在另一个文档中。除了特别注解 之外，这些例子都会返回真：

```
-- 简单的标量/基本值只包含相同的值：
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;

-- 右边的数字被包含在左边的数组中：
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;

-- 数组元素的顺序没有意义，因此这个例子也返回真：
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;

-- 重复的数组元素也没有关系：
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;

-- 右边具有一个单一键值对的对象被包含在左边的对象中：
SELECT '{"product": "UXDB", "version": 9.4, "jsonb": true}'::jsonb @> '{"version": 9.4}'::jsonb;

-- 右边的数组不会被认为包含在左边的数组中，
-- 即使其中嵌入了一个相似的数组：
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  -- 得到假

-- 但是如果同样也有嵌套，包含就成立：
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;

-- 类似的，这个例子也不会被认为是包含：
```

```
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;  -- 得到假

-- 包含一个顶层键和一个空对象：
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
```

一般原则是被包含的对象必须在结构和数据内容上匹配包含对象，这种匹配　可以是从包含对象中丢弃了不匹配的数组元素或者对象键值对之后成立。但　是记住做包含匹配时数组元素的顺序是没有意义的，并且重复的数组元素实 际也只会考虑一次。

结构必须匹配的一般原则有一种特殊情况，一个数组可以包含一个基本值：

```
-- 这个数组包含基本字符串值：
SELECT '["foo", "bar"]'::jsonb @> '"bar"'::jsonb;

-- 反之不然，下面的例子会报告“不包含”：
SELECT '"bar"'::jsonb @> '["bar"]'::jsonb;  -- 得到假
```

jsonb还有一个*存在*操作符，它是包含的一种 变体：它测试一个字符串（以一个text值的形式给出）是否出　现在jsonb值顶层的一个对象键或者数组元素中。除非特别注解，　下面这些例子返回真：

```
-- 字符串作为一个数组元素存在：
SELECT '["foo", "bar", "baz"]'::jsonb ? 'bar';

-- 字符串作为一个对象键存在：
SELECT '{"foo": "bar"}'::jsonb ? 'foo';

-- 不考虑对象值：
SELECT '{"foo": "bar"}'::jsonb ? 'bar';  -- 得到假

-- 和包含一样，存在必须在顶层匹配：
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar'; -- 得到假

-- 如果一个字符串匹配一个基本 JSON 字符串，它就被认为存在：
SELECT '"foo"'::jsonb ? 'foo';
```

当涉及很多键或元素时，JSON 对象比数组更适合于做包含或存在测试，　因为它们不像数组，进行搜索时会进行内部优化，并且不需要被线性搜索。

提示

由于　JSON　的包含是嵌套的，因此一个恰当的查询可以跳过对子对象的显式选择。　例如，假设我们在顶层有一个*doc*列包含着对象，大部分对象包含着tags域，其中有子对象的数组。这个查询会找到其中出现了　同时包含"term":"paris"和"term":"food"的子对象　的项，而忽略任何位于tags数组之外的这类键：

```
SELECT doc->'site_name' FROM websites
  WHERE doc @> '{"tags":[{"term":"paris"}, {"term":"food"}]}';
```

可以用下面的查询完成同样的事情：

```
SELECT doc->'site_name' FROM websites
  WHERE doc->'tags' @> '[{"term":"paris"}, {"term":"food"}]';
```

但是后一种方法灵活性较差，并且常常也效率更低。

在另一方面，JSON 的存在操作符不是嵌套的：它将只在 JSON 值的顶层 查找指定的键或数组元素。

第 6.15 节 “JSON 函数和操作符”中记录了多个包含和存在操作符，以及 所有其他 JSON 操作符和函数。

5.14.4. jsonb 索引

GIN 索引可以被用来有效地搜索在大量jsonb文档（数据）中出现 的键或者键值对。提供了两种 GIN “操作符类”，它们在性能和灵活 性方面做出了不同的平衡。

jsonb的默认 GIN 操作符类支持使用@>、 ?、?&以及?|操作符的查询（这些 操作符实现的详细语义请见表 6.51 “额外的jsonb操作符”）。 使用这种操作符类创建一个索引的例子：

```
CREATE INDEX idxgin ON api USING gin (jdoc);
```

非默认的 GIN 操作符类jsonb_path_ops只支持索引 @>操作符。使用这种操作符类创建一个索引的例子：

```
CREATE INDEX idxginp ON api USING gin (jdoc jsonb_path_ops);
```

考虑这样一个例子：一个表存储了从一个第三方 Web 服务检索到的 JSON 文档，并且有一个模式定义。一个典型的文档：

```
{
  "guid": "9c36adc1-7fb5-4d5b-83b4-90356a46061a",
  "name": "Angela Barton",
  "is_active": true,
  "company": "Magnafone",
  "address": "178 Howard Place, Gulf, Washington, 702",
  "registered": "2009-11-07T08:53:22 +08:00",
  "latitude": 19.793713,
  "longitude": 86.513373,
  "tags": [
    "enim",
    "aliquip",
    "qui"
  ]
}
```

我们把这些文档存储在一个名为api的表的名为 *jdoc*的jsonb列中。如果在这个列上创建一个 GIN 索引，下面这样的查询就能利用该索引：

```
-- 寻找键 "company" 有值 "Magnafone" 的文档
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"company": "Magnafone"}';
```

不过，该索引不能被用于下面这样的查询，因为尽管操作符? 是可索引的，但它不能直接被应用于被索引列*jdoc*：

```
-- 寻找这样的文档：其中的键 "tags" 包含键或数组元素 "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc -> 'tags' ? 'qui';
```

但是，通过适当地使用表达式索引，上述查询也能使用一个索引。 如果对"tags"键中的特定项的查询很常见，可能值得 定义一个这样的索引：

```
CREATE INDEX idxgintags ON api USING gin ((jdoc -> 'tags'));
```

现在，WHERE 子句 jdoc -> 'tags' ? 'qui' 将被识别为可索引操作符?在索引表达式jdoc -> 'tags' 上的应用（更多有关表达式索引的信息可见第 8.7 节 “表达式索引”。

此外，GIN 索引支持 @@ 和 @?运算符，以执行 jsonpath 匹配。

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*] == "qui"';
```

```
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @@ '$.tags[*] ? (@ == "qui")';
```

GIN 索引从jsonpath中提取如下格式的语句： *accessors_chain* = *const*。 存取器链可能由.key[*], 和 [*index*] 存取器组成。 jsonb_ops 此外还支持 .* 和 .** 存取器。

另一种查询的方法是利用包含，例如：

```
-- 寻找这样的文档：其中键 "tags" 包含数组元素 "qui"
SELECT jdoc->'guid', jdoc->'name' FROM api WHERE jdoc @> '{"tags": ["qui"]}';
```

*jdoc*列上的一个简单 GIN 索引就能支持这个查询。 但是注意这样一个索引将会存储*jdoc*列中每一个键 和值的拷贝，然而前一个例子的表达式索引只存储tags 键下找到的数据。虽然简单索引的方法更加灵活（因为它支持有关任 意键的查询），定向的表达式索引更小并且搜索速度比简单索引更快。

尽管jsonb_path_ops操作符类只支持用 @>，@@和@?操作符的查询，但它比起默认的操作符类 jsonb_ops有更客观的性能优势。一个 jsonb_path_ops索引通常也比一个相同数据上的jsonb_ops要小得多，并且搜索的专一性更好，特 别是当查询包含频繁出现在该数据中的键时。因此，其上的搜索操作 通常比使用默认操作符类的搜索表现更好。

jsonb_ops和jsonb_path_ops GIN 索引之间的技术区别是前者为数据中的每一个键和值创建独立的索引项， 而后者值为该数据中的每个值创建索引项。 [1] 基本上，每一个jsonb_path_ops索引项是其所对应的值和 键的哈希。例如要索引{"foo": {"bar": "baz"}}，将创建一个 单一的索引项，它把所有三个foo、bar、 和baz合并到哈希值中。因此一个查找这个结构的包含查询可能 导致极度详细的索引搜索。但是根本没有办法找到foo是否作为 一个键出现。在另一方面，一个jsonb_ops会创建三个索引 项分别表示foo、bar和baz。那么要 做同样的包含查询，它将会查找包含所有三个项的行。虽然 GIN 索引能够相当 有效地执行这种 AND 搜索，它仍然不如等效的jsonb_path_ops搜索那样详细和快速（特别是如果有大量 行包含三个索引项中的任意一个时）。

[1] 对于这种目的，术语“值”包括数组元素，尽管 JSON 的术语有时 认为数组元素与对象内的值不同。

jsonb_path_ops方法的一个不足是它不会为不包含任何值 的 JSON 结构创建索引项，例如{"a": {}}。如果需要搜索包 含这样一种结构的文档，它将要求一次全索引扫描，那就非常慢。 因此jsonb_path_ops不适合经常执行这类搜索的应用。

jsonb也支持btree和hash索引。 这通常值用于检查完整 JSON 文档等值非常重要的场合。jsonb数据的btree顺序很少有人关系，但是为了完整性其顺序是：

对象 > 数组 > 布尔 > 数字 > 字符串 > 空值

带有 n 对的对象 > 带有 n - 1 对的对象

带有 n 个元素的数组 > 带有 n - 1 个元素的数组

具有相同数量对的对象这样比较：

key-1, *value-1*, *key-2* ...

注意对象键被按照它们的存储顺序进行比较，特别是由于较短的键被存储在 较长的键之前，这可能导致结果不直观，例如：

{ "aa": 1, "c": 1} > {"b": 1, "d": 1}

相似地，具有相同元素数量的数组按照以下顺序比较：

element-1, *element-2* ...

基本 JSON 值的比较会使用低层UXDB 数据类型相同的比较规则进行。字符串的比较会使用默认的数据库排序规则。

5.14.5. 转换

有一些附加的扩展可以为不同的过程语言实现jsonb类型的转换。

PL/Perl的扩展被称作jsonb_plperl和jsonb_plperlu。如果使用它们，jsonb值会视情况被映射为Perl的数组、哈希和标量。

PL/Python的扩展被称作jsonb_plpythonu、jsonb_plpython2u和jsonb_plpython3u。如果使用它们，jsonb值会视情况被映射为Python的词典、列表和标量。

5.14.6. jsonpath Type

在UXDB中，jsonpath类型实现支持SQL/JSON 路径语言以有效地查询 JSON 数据。 它提供了已解析的SQL/JSON路径表达式的二进制表示，该表达式指定路径引擎从JSON数据中检索的项，以便使用SQL/JSON查询函数进行进一步处理。

SQL/JSON 路径谓词和运算符的语义通常遵循 SQL。同时，为了提供使用 JSON 数据的最自然方法，SQL/JSON 路径语法使用一些 JavaScript 约定：

- 小数点 (.) 用于成员访问.
- 方括号 ([]) 用于数组访问.

- 与从 1 开始的常规 SQL 数组不同，SQL/JSON 数组是 0 相对的。

SQL/JSON路径表达式通常以SQL字符串文字形式写入SQL查询中，因此它必须用单引号括起来，并且值中需要的任何单引号都必须是双引号(参见 第 1.1.2.1 节 “字符串常量”。 某些形式的路径表达式需要其中的字符串文本。这些嵌入的字符串文本遵循JavaScript/ECMAScript约定:它们必须用双引号括起来，并且反斜杠转义可以用于表示其他难以输入的字符。 特别是，在嵌入字符串文本中编写双引号的方法为\"，并且要编写反斜杠本身，必须写\\。 包括在JSON字符串中识别的其他特殊的反斜杠序列: \b, \f, \n, \r, \t, \v 对于各种 ASCII 控制字符，以及由它的4个六位数编码点标识标识的 Unicode 字符\u*NNNN*。 反斜杠语法还包括 JSON 不允许的两个案例: \x*NN* 对于只用两个十六进制数字编写的字符代码，以及\u{*N*...} 对于用 1 到 6 个十六进制数字编写的字符代码。

路径表达式由一系列路径元素组成，可以如下所示:

- JSON基本类型的路径文字:Unicode文本、数字、真、假或空.
- Path variables listed in 表 5.25 “jsonpath 变量”中列出的路径变量。
- 表 5.26 “jsonpath Accessors”中列出的访问器运算符。
- 第 6.15.2.3 节 “SQL/JSON路径操作符和方法”中列出的jsonpath 运算符和方法。
- 括号，可用于提供筛选器表达式或定义路径计算的顺序。

有关使用jsonpath具有 SQL/JSON 查询函数的表达式的详细信息，参见第 6.15.2 节 “SQL/JSON路径语言”。

表 5.25. jsonpath 变量

变量	描述
$	表示要查询的 JSON 文本的变量(context item).
$varname	命名变量。其值可以由参数*vars*多个JSON处理函数设置。 详细信息请参见 表 6.53 “JSON处理” 和它的注释。
@	表示筛选器表达式中路径计算结果的变量。

表 5.26. jsonpath Accessors

访问器运算符	描述
.*key* ."$*varname*"	返回具有指定密钥的对象成员的成员访问器。如果键名称是以 $ 开头的命名变量，或者不符合标识符的 JavaScript 规则，则必须将其包含在双引号中作为字符串文本。
.*	通配符成员访问器，该访问器返回位于当前对象顶层的所有成员的值。
.**	递归通配符成员访问器，它处理当前对象JSON层次结构的所有级别，并返回所有成员值，而不管它们的嵌套级别如何。 这是 UXDB SQL/JSON 标准的扩展。
.**{*level*} .**{*start_level* to *end_level*}	与 .** 相同，但在 JSON 层次结构的嵌套级别上具有筛选器。嵌套级别指定为整数。 零级别对应于当前对象。要访问最低嵌套级别，可以

访问器运算符	描述
	使用last关键字。 这是 UXDB SQL/JSON 标准的扩展。
[*subscript*, ...]	数组元素访问器. *subscript* 能够以两种形式给出：*index* 或 *start_index* 到 *end_index*。 第一个窗体按其索引返回单个数组元素。第二个窗体按索引范围返回数组切片，包括对应于提供的元素*start_indexend_index*。 指定的*index*可以是整数，也可以是返回单个数值的表达式，该数值将自动转换为整数。 零索引对应于第一个数组元素。还可以使用last 关键字来表示最后一个数组元素，这对于处理未知长度的数组很有用。
[*]	返回所有数组元素的通配符数组元素访问器。

5.15. 数组

UXDB允许一个表中的列定义为变长多维数组。可以创建任何内建或用户定义的基类、枚举类型、组合类型或者域的数组。

5.15.1. 数组类型的定义

为了展示数组类型的使用，我们创建这样一个表：

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

如上所示，一个数组数据类型可以通过在数组元素的数据类型名称后面加上方括号（[]）来命名。上述命令将创建一个名为sal_emp的表，它有一个类型为text的列（*name*），一个表示雇员的季度工资的一维integer类型数组（*pay_by_quarter*），以及一个表示雇员每周日程表的二维text类型数组（*schedule*）。

CREATE TABLE的语法允许指定数组的确切大小，例如：

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

然而，当前的实现忽略任何提供的数组尺寸限制，即其行为与未指定长度的数组相同。

当前的实现也不会强制所声明的维度数。一个特定元素类型的数组全部被当作是相同的类型，而不论其尺寸或维度数。因此，在CREATE TABLE中声明数组的尺寸或维度数仅仅只是文档而已，它并不影响运行时的行为。

另一种符合SQL标准的语法是使用关键词ARRAY，可以用来定义一维数组。*pay_by_quarter*可以这样定义：

```
  pay_by_quarter  integer ARRAY[4],
```

或者，不指定数组尺寸：

```
  pay_by_quarter  integer ARRAY,
```

但是和前面一样，UXDB在任何情况下都不会强制尺寸限制。

5.15.2. 数组值输入

要把一个数组值写成一个文字常数，将元素值用花括号包围并用逗号分隔（如果懂C，这和初始化结构的C语法没什么两样）。在任意元素值周围可以使用双引号，并且在元素值包含逗号或花括号时必须这样做（更多细节如下所示）。因此，一个数组常量的一般格式如下：

```
'{ val1 delim val2 delim ... }'
```

这里*delim*是类型的定界符，记录在类型的ux_type项中。在UXDB发行提供的标准数据类型中，所有的都使用一个逗号（,），除了类型box使用一个分号（;）。每个*val*可以是数组元素类型的一个常量，也可以是一个子数组。一个数组常量的例子是：

```
'{{1,2,3},{4,5,6},{7,8,9}}'
```

该常量是一个二维的，3乘3数组，它由3个整数子数组构成。

要设置一个数组常量的一个元素为NULL，在该元素值处写NULL（任何NULL的大写或小写变体都有效）。如果需要一个真正的字符串值“NULL”，必须在它两边放上双引号。

（这些种类的数组常数实际是第 1.1.2.7 节 “其他类型的常量”中讨论的一般类型常量的一种特殊形式。常数最初被当做一个字符串，然后被传给数组的输入转换例程。有必要时可能需要一个显式的类型指定。）

现在我们可以展示一些INSERT语句：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"training", "presentation"}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"breakfast", "consulting"}, {"meeting", "lunch"}}');
```

前两个插入的结果看起来像这样：

```
SELECT * FROM sal_emp;
 name |     pay_by_quarter      |              schedule
-------+---------------------------+---------------------------------------
 Bill | {10000,10000,10000,10000} | {{meeting,lunch},{training,presentation}}
```

```
 Carol | {20000,25000,25000,25000} | {{breakfast,consulting},{meeting,lunch}}
(2 rows)
```

多维数组的每一维都必须有相匹配的长度。不匹配会造成错误，例如：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {"meeting"}}');
ERROR:  multidimensional arrays must have array expressions with matching dimensions
```

ARRAY构造器语法也可以被用于：

```
INSERT INTO sal_emp
    VALUES ('Bill',
    ARRAY[10000, 10000, 10000, 10000],
    ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);

INSERT INTO sal_emp
    VALUES ('Carol',
    ARRAY[20000, 25000, 25000, 25000],
    ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

注意数组元素是普通SQL常数或表达式，例如，字符串文字使用单引号而不是双引号包围，因为双引号可以出现在一个数组文字中。ARRAY构造器语法的详细讨论请见第 1.2.12 节 “数组构造器”。

5.15.3. 访问数组

现在，我们可以在该表上运行一些查询。首先，我们展示如何访问一个数组中的一个元素。下面的查询检索在第二季度工资发生变化的雇员的名字：

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <> pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

数组下标写在方括号内。默认情况下，UXDB为数组使用了一种从1开始的编号习惯，即一个具有*n*个元素的数组从array[1]开始，结束于array[*n*]。

下面的查询检索所有员工第三季度的工资：

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
```

```
(2 rows)
```

我们也可以访问一个数组的任意矩形切片或者子数组。一个数组切片可以通过在一个或多个数组维度上指定`下界:上界`来定义例如，下面的查询检索Bill在本周头两天日程中的第一项：

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

      schedule
------------------------
 {{meeting},{training}}
(1 row)
```

如果任何维度被写成一个切片，即包含一个冒号，那么所有的维度都被看成是切片对待。其中任何只有一个数字（无冒号）的维度被视作是从1到指定的数字。例如，下面例子中的[2]被认为是[1:2]：

```
SELECT schedule[1:2][2] FROM sal_emp WHERE name = 'Bill';

               schedule
-------------------------------------------
 {{meeting,lunch},{training,presentation}}
(1 row)
```

为了避免和非切片情况搞混，最好在所有的维度上都使用切片语法，例如[1:2][1:1]而不是[2][1:1]。

可以省略一个切片说明符的*lower-bound*或者 *upper-bound*（亦可两者都省略），缺失的 边界会被数组下标的上下限所替代。例如：

```
SELECT schedule[:2][2:] FROM sal_emp WHERE name = 'Bill';

      schedule
------------------------
 {{lunch},{presentation}}
(1 row)

SELECT schedule[:][1:1] FROM sal_emp WHERE name = 'Bill';

      schedule
------------------------
 {{meeting},{training}}
(1 row)
```

如果数组本身为空或者任何一个下标表达式为空，访问数组下标表达式将会返回空值。如果下标超过了数组边界，下标表达式也会返回空值（这种情况不会抛出错误）。例如，如果schedule目前具有的维度是[1:3][1:2]，那么引用schedule[3][3]将得到NULL。相似地，使用错误的下标号引用一个数组会得到空值而不是错误。

如果数组本身或者任何一个下标表达式为空，则一个数组切片表达式也会得到空值。但是，在其他情况例如选择一个完全位于当前数组边界之外的切片时，一个切片表达式会得到一个空（零维）数组而不是空值（由于历史原因，这并不符合非切片行为）。 如果所请求的切片和数组边界重叠，那么它会被缩减为重叠的区域而不是返回空。

任何数组值的当前维度可以使用array_dims函数获得：

```
SELECT array_dims(schedule) FROM sal_emp WHERE name = 'Carol';

 array_dims
------------
 [1:2][1:2]
(1 row)
```

array_dims产生一个text结果，它便于人类阅读但是不便于程序读取。 Dimensions can also be retrieved with 也可以通过array_upper和array_lower来获得维度，它们将分别返回一个指定数组的上界和下界：

```
SELECT array_upper(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_upper
-------------
           2
(1 row)
```

array_length将返回一个指定数组维度的长度：

```
SELECT array_length(schedule, 1) FROM sal_emp WHERE name = 'Carol';

 array_length
--------------
            2
(1 row)
```

cardinality返回一个数组中在所有维度上的元素总数。 这实际上是调用unnest将会得到的行数：

```
SELECT cardinality(schedule) FROM sal_emp WHERE name = 'Carol';

 cardinality
-------------
           4
(1 row)
```

5.15.4. 修改数组

一个数组值可以被整个替换：

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
   WHERE name = 'Carol';
```

或者使用ARRAY表达式语法：

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
   WHERE name = 'Carol';
```

一个数组也可以在一个元素上被更新：

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
   WHERE name = 'Bill';
```

或者在一个切片上被更新：

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
   WHERE name = 'Carol';
```

也可以使用省略*lower-bound*或者 *upper-bound*的切片语法，但是只能用于 更新一个不是 NULL 或者零维的数组值（否则无法替换现有的下标界线）。

一个已存储的数组值可以被通过为其还不存在的元素赋值来扩大之。任何位于之前已存在的元素和新元素之间的位置都将被空值填充。例如，如果数组myarray目前有4个元素，在用一个更新对myarray[6]赋值后它将有6个元素，其中myarray[5]为空值。目前，采用这种方式扩大数组只允许使用在一维数组上。

带下标的赋值方式允许创建下标不是从1开始的数组。例如，我们可以为myarray[-2:7]赋值来创建一个下标值从-2到7的数组。

新的数组值也可以通过串接操作符||构建：

```
SELECT ARRAY[1,2] || ARRAY[3,4];
 ?column?
-----------
 {1,2,3,4}
(1 row)

SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
     ?column?
---------------------
 {{5,6},{1,2},{3,4}}
(1 row)
```

串接操作符允许把一个单独的元素加入到一个一维数组的开头或末尾。它也能接受两个*N*维数组，或者一个*N*维数组和一个*N+1*维数组。

当一个单独的元素被加入到一个一维数组的开头或末尾时，其结果是一个和数组操作数具有相同下界下标的新数组。例如：

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
 array_dims
------------
 [0:2]
(1 row)

SELECT array_dims(ARRAY[1,2] || 3);
 array_dims
------------
 [1:3]
```

```
(1 row)
```

当两个具有相同维度数的数组被串接时，其结果保留左操作数的外维度的下界下标。结果将是一个数组，它由左操作数的每一个元素以及紧接着的右操作数的每一个元素。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
 array_dims
------------
 [1:5]
(1 row)

SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
 array_dims
------------
 [1:5][1:2]
(1 row)
```

当一个*N*维数组被放在另一个*N+1*维数组的前面或者后面时，结果和上面的例子相似。每一个*N*维子数组实际上是*N+1*维数组外维度的一个元素。例如：

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
 array_dims
------------
 [1:3][1:2]
(1 row)
```

一个数组也可以通过使用函数array_prepend、array_append或array_cat构建。前两个函数仅支持一维数组，但array_cat支持多维数组。 一些例子：

```
SELECT array_prepend(1, ARRAY[2,3]);
 array_prepend
---------------
 {1,2,3}
(1 row)

SELECT array_append(ARRAY[1,2], 3);
 array_append
--------------
 {1,2,3}
(1 row)

SELECT array_cat(ARRAY[1,2], ARRAY[3,4]);
 array_cat
-----------
 {1,2,3,4}
(1 row)

SELECT array_cat(ARRAY[[1,2],[3,4]], ARRAY[5,6]);
     array_cat
---------------------
 {{1,2},{3,4},{5,6}}
(1 row)
```

```
SELECT array_cat(ARRAY[5,6], ARRAY[[1,2],[3,4]]);
     array_cat
---------------------
 {{5,6},{1,2},{3,4}}
```

在简单的情况中，上面讨论的串接操作符比直接使用这些函数更好。不过，由于 串接操作符需要服务于所有三种情况，所以它的负担比较重，在有些情况下使用 这些函数之一有助于避免混淆。例如：

```
SELECT ARRAY[1, 2] || '{3, 4}';  -- 没有指定类型的文字被当做一个数组
 ?column?
-----------
 {1,2,3,4}

SELECT ARRAY[1, 2] || '7';               -- 这个也是
ERROR:  malformed array literal: "7"

SELECT ARRAY[1, 2] || NULL;              -- 未修饰的 NULL 也是如此
 ?column?
----------
 {1,2}
(1 row)

SELECT array_append(ARRAY[1, 2], NULL);   -- 这可能才是想要的意思
 array_append
--------------
 {1,2,NULL}
```

在上面的例子中，解析器看到在串接操作符的一遍看到了一个整数数组，并且在 另一边看到了一个未确定类型的常量。它用来决定该常量类型的启发式规则是假 定它和该操作符的另一个输入具有相同的类型 — 在这种情况中是整数数 组。因此串接操作符表示array_cat而不是array_append。如果这样做是错误的选择，它可以通过将该常 量造型成数组的元素类型来修复。但是显式地使用array_append 可能是一种最好的方案。

5.15.5. 在数组中搜索

要在一个数组中搜索一个值，每一个值都必须被检查。这可以手动完成，但是我们必须知道数组的尺寸。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter[1] = 10000 OR
                            pay_by_quarter[2] = 10000 OR
                            pay_by_quarter[3] = 10000 OR
                            pay_by_quarter[4] = 10000;
```

但是这对于大型数组来说太过冗长，且在数组尺寸未知时无法使用。一种可选的方法可见第 6.23 节 “行和数组比较”。上面的查询可以被替换为：

```
SELECT * FROM sal_emp WHERE 10000 = ANY (pay_by_quarter);
```

此外，我们还可以查找所有元素值都为10000的数组所在的行：

```
SELECT * FROM sal_emp WHERE 10000 = ALL (pay_by_quarter);
```

另外，generate_subscripts函数也可以用来完成类似的查找。例如：

```
SELECT * FROM
  (SELECT pay_by_quarter,
          generate_subscripts(pay_by_quarter, 1) AS s
     FROM sal_emp) AS foo
 WHERE pay_by_quarter[s] = 10000;
```

该函数的描述见表　6.70　“下标生成函数”。

我们也可以使用&&操作符来搜索一个数组，它会检查左操作数是否与右操作数重叠。例如：

```
SELECT * FROM sal_emp WHERE pay_by_quarter && ARRAY[10000];
```

该操作符和其他数组操作符的进一步描述请见第　6.18　节　“数组函数和操作符”。如第　8.2　节　“索引类型”所述，它可以使用一个合适的索引来提速。

也可以使用array_position和array_positions在一个　数组中搜索特定值。前者返回值在数组中第一次出现的位置的下标。后者返回一个数组，　　其中有该值在数组中的所有出现位置的下标。例如：

```
SELECT array_position(ARRAY['sun','mon','tue','wed','thu','fri','sat'], 'mon');
 array_positions
-----------------
 2

SELECT array_positions(ARRAY[1, 4, 3, 1, 3, 4, 2, 1], 1);
 array_positions
-----------------
 {1,4,8}
```

> **提示**
>
> 数组不是集合，在其中搜索指定数组元素可能是数据设计失误的表现。考虑使用一个独立的表来替代，其中每一行都对应于一个数组元素。这将更有利于搜索，并且对于大量元素的可扩展性更好。

5.15.6.　数组输入和输出语法

一个数组值的外部文本表现由根据数组元素类型的I/O转换规则解释的项构成，并在其上加上修饰用于指示数组结构。修饰包括数组值周围的花括号（{和}）以及相邻项之间的定界字符。定界字符通常是一个逗号（,），但是也可能是别的：它由数组元素类型的typdelim设置决定。在UXDB发行版提供的标准数据类型中，除了box类型使用分号（;）之外，其他都是用逗号。在一个多维数组中，每一个维度（行、平面、方体等）都有其自己的花括号层次，且同层的被花括号限定的相邻实体之间也必须有定界符。

如果元素值是空字符串、包含花括号、包含定界字符、包含双引号、包含反斜线、包含空白或者匹配词NULL，数组输出例程将在元素值周围放上双引号。嵌在元素值中的双引号以及反斜线将

被反斜线转义。对于数字数据类型可以安全地假设双引号绝不会出现，但是对于文本数据类型我们必须准备好处理可能出现亦可能不出现的引号。

默认情况下，一个数组的一个维度的下界索引值被设置为1。要表示具有其他下界的数组，数组下标的范围应在填充数组内容之前被显式地指定好。这种修饰包括在每个数组维度上下界周围的方括号（[]），以及上下界之间的一个冒号（:）定界符。数组维度修饰后面要跟一个等号（=）。例如：

```
SELECT f1[1][-2][3] AS e1, f1[1][-1][5] AS e2
 FROM (SELECT '[1:1][-2:-1][3:5]={{{1,2,3},{4,5,6}}}'::int[] AS f1) AS ss;

 e1 | e2
----+----
  1 |  6
(1 row)
```

只有当数组的维度中有一个或多个的下界不为1时，数组输出例程才会在结果中包括维度。

如果为一个元素给定的值是NULL（或者是任何变体），该元素将被设置为NULL。任何引号或反斜线的存在将阻止这种行为，而允许为元素值输入“NULL”的字面意思。

如前所示，在写一个数组值时我们可以在任何单独数组元素周围使用引号。如果元素值可能混淆数组值分析器时，我们必须　这样做。例如，包含花括号、逗号（或者数据类型的定界符）、双引号、反斜线或首尾有空白的元素必须使用双引号。空字符串和匹配单词NULL的字符串也必须使用双引号。要把一个双引号或反斜线放在一个使用了双引号的数组元素值中，需要在它前面放一个反斜线。作为一种选择，我们可以免去使用引号而使用反斜线转义的方式来保护可能被认为是数组语法的所有数据字符。

我们可以在左括号前面或右括号后面增加空白。我们也可以在任何单独的项之前或之后加上空白。在所有这些情况中空白将被忽略。但是，在被使用了双引号的元素中的空白以及周围有其他非空白字符的空白不会被忽略。

> **提示**
>
> 在SQL命令中写数组值时，ARRAY构造器语法（见第　1.2.12　节　“数组构造器”）常常比数组文字语法要更容易使用。在ARRAY中，单独的元素值可以使用不属于数组成员时的方式来书写。

5.16.　组合类型

一个组合类型表示一行或一个记录的结构，它本质上就是一个域名和它们数据类型的列表。UXDB允许把组合类型用在很多能用简单类型的地方。例如，一个表的一列可以被声明为一种组合类型。

5.16.1.　组合类型的声明

这里有两个定义组合类型的简单例子：

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
```

```
);

CREATE TYPE inventory_item AS (
    name            text,
    supplier_id     integer,
    price           numeric
);
```

该语法堪比CREATE TABLE，不过只能指定域名和类型，当前不能包括约束（例如NOT NULL）。注意AS关键词是必不可少的，如果没有它，系统将认为用户想要的是一种不同类型的CREATE TYPE命令，并且将得到奇怪的语法错误。

定义了类型之后，我们可以用它们来创建表：

```
CREATE TABLE on_hand (
    item      inventory_item,
    count     integer
);

INSERT INTO on_hand VALUES (ROW('fuzzy dice', 42, 1.99), 1000);
```

or functions:

```
CREATE FUNCTION price_extension(inventory_item, integer) RETURNS numeric
AS 'SELECT $1.price * $2' LANGUAGE SQL;

SELECT price_extension(item, 10) FROM on_hand;
```

只要创建了一个表，也会自动创建一个组合类型来表示表的行类型，它具有和表一样的名称。例如，如果我们说：

```
CREATE TABLE inventory_item (
    name            text,
    supplier_id     integer REFERENCES suppliers,
    price           numeric CHECK (price > 0)
);
```

那么和上面所示相同的inventory_item组合类型将成为一种副产品，并且可以按上面所说的进行使用。不过要注意当前实现的一个重要限制：因为没有约束与一个组合类型相关，显示在表定义中的约束不会应用于表外组合类型的值（要解决这个问题，可以在该组合类型上创建一个域，并且把想要的约束应用为这个域上的CHECK约束）。

5.16.2. 构造组合值

要把一个组合值写作一个文字常量，将该域值封闭在圆括号中并且用逗号分隔它们。可以在任何域值周围放上双引号，并且如果该域值包含逗号或圆括号则必须这样做（更多细节见下文）。这样，一个组合常量的一般格式是下面这样的：

```
'( val1 , val2 , ... )'
```

一个例子是：

'("fuzzy dice",42,1.99)'

这将是上文定义的inventory_item类型的一个合法值。要让一个域为 NULL，在列表中它的位置上根本不写字符。例如，这个常量指定其第三个域为 NULL：

'("fuzzy dice",42,)'

如果写一个空字符串而不是 NULL，写上两个引号：

'("",42,)'

这里第一个域是一个非 NULL 空字符串，第三个是 NULL。

（这些常量实际上只是第 1.1.2.7 节 “其他类型的常量”中讨论的一般类型常量的特殊类型。该常量最初被当做一个字符串并且被传递给组合类型输入转换例程。有必要用一次显式类型说明来告知要把该常量转换成何种类型。）。

ROW表达式也能被用来构建组合值。在大部分情况下，比起使用字符串语法，这相当简单易用，因为不必担心多层引用。我们已经在上文用过这种方法：

ROW('fuzzy dice', 42, 1.99)
ROW('', 42, NULL)

只要在表达式中有多于一个域，ROW 关键词实际上就是可选的，因此这些可以被简化成：

('fuzzy dice', 42, 1.99)
('', 42, NULL)

第 1.2.13 节 “行构造器”中更加详细地讨论了ROW表达式语法。

5.16.3. 访问组合类型

要访问一个组合列的一个域，可以写成一个点和域的名称，更像从一个表名中选择一个域。事实上，它太像从一个表名中选择，这样我们不得不使用圆括号来避免让解析器混淆。例如，可能尝试从例子表on_hand中选取一些子域：

SELECT item.name FROM on_hand WHERE item.price > 9.99;

这不会有用，因为名称item会被当成是一个表名，而不是on_hand的一个列名。必须写成这样：

SELECT (item).name FROM on_hand WHERE (item).price > 9.99;

或者还需要使用表名（例如在一个多表查询中），像这样：

SELECT (on_hand.item).name FROM on_hand WHERE (on_hand.item).price > 9.99;

现在加上括号的对象就被正确地解释为对item列的引用，然后可以从中选出子域。

只要从一个组合值中选择一个域，相似的语法问题就适用。例如，要从一个返回组合值的函数的结果中选取一个域，需要这样写：

```
SELECT (my_func(...)).field FROM ...
```

如果没有额外的圆括号，这将生成一个语法错误。

特殊的域名称*表示“所有的域”，第 5.16.5 节 “在查询中使用组合类型”中有进一步的解释。

5.16.4. 修改组合类型

这里有一些插入和更新组合列的正确语法的例子。首先，插入或者更新一整个列：

```
INSERT INTO mytab (complex_col) VALUES((1.1,2.2));

UPDATE mytab SET complex_col = ROW(1.1,2.2) WHERE ...;
```

第一个例子忽略ROW，第二个例子使用它，我们可以用两者之一完成。

我们能够更新一个组合列的单个子域：

```
UPDATE mytab SET complex_col.r = (complex_col).r + 1 WHERE ...;
```

注意这里我们不需要（事实上也不能）把圆括号放在正好出现在SET之后的列名周围，但是当在等号右边的表达式中引用同一列时确实需要圆括号。

并且我们也可以指定子域作为INSERT的目标：

```
INSERT INTO mytab (complex_col.r, complex_col.i) VALUES(1.1, 2.2);
```

如果我们没有为该列的所有子域提供值，剩下的子域将用空值填充。

5.16.5. 在查询中使用组合类型

对于查询中的组合类型有各种特殊的语法规则和行为。这些规则提供了有用的捷径，但是如果不懂背后的逻辑就会被此困扰。

在UXDB中，查询中对一个表名（或别名）的引用实际上是对该表的当前行的组合值的引用。例如，如果我们有一个如上所示的表inventory_item，我们可以写：

```
SELECT c FROM inventory_item c;
```

这个查询产生一个单一组合值列，所以我们会得到这样的输出：

```
          c
------------------------
 ("fuzzy dice",42,1.99)
(1 row)
```

不过要注意简单的名称会在表名之前先匹配到列名，因此这个例子可行的原因仅仅是因为在该查询的表中没有名为*c*的列。

普通的限定列名语法*table_name.column_name*可以理解为把字段选择应用在该表的当前行的组合值上（由于效率的原因，实际上不是以这种方式实现）。

当我们写

```
SELECT c.* FROM inventory_item c;
```

时，根据SQL标准，我们应该得到该表展开成列的内容：

```
   name    | supplier_id | price
-----------+-------------+-------
 fuzzy dice |          42 |  1.99
(1 row)
```

就好像查询是

```
SELECT c.name, c.supplier_id, c.price FROM inventory_item c;
```

尽管如上所示，UXDB将对任何组合值表达式应用这种展开行为，但只要.*所应用的值不是一个简单的表名，就需要把该值写在圆括号内。例如，如果myfunc()是一个返回组合类型的函数，该组合类型由列*a*、*b*和*c*组成，那么这两个查询有相同的结果：

```
SELECT (myfunc(x)).* FROM some_table;
SELECT (myfunc(x)).a, (myfunc(x)).b, (myfunc(x)).c FROM some_table;
```

> **提示**
>
> UXDB实际上通过将第一种形式转换为第二种来处理列展开。因此，在这个例子中，用两种语法时对每行都会调用myfunc()三次。如果它是一个开销很大的函数，可能希望避免这样做，所以可以用一个这样的查询：
>
> ```
> SELECT m.* FROM some_table, LATERAL myfunc(x) AS m;
> ```
>
> 把该函数放在一个LATERAL FROM项中会防止它对每一行被调用超过一次。m.*仍然会被展开为m.a, m.b, m.c，但现在那些变量只是对这个FROM项的输出的引用（这里关键词LATERAL是可选的，但我们在这里写上它是为了说明该函数从some_table中得到*x*）。

当*composite_value*.*出现在一个SELECT输出列表的顶层中、INSERT/UPDATE/DELETE中的一个RETURNING列表中、一个VALUES子句中或者一个行构造器中时，该语法会导致这种类型的列展开。在所有其他上下文（包括被嵌入在那些结构之一中时）中，把.*附加到一个组合值不会改变该值，因为它表示“所有的列”并且因此同一个组合值会被再次产生。例如，如果somefunc()接受一个组合值参数，这些查询是相同的：

```
SELECT somefunc(c.*) FROM inventory_item c;
SELECT somefunc(c) FROM inventory_item c;
```

在两种情况中，inventory_item的当前行被传递给该函数作为一个单一的组合值参数。即使.*在这类情况中什么也不做，使用它也是一种好的风格，因为它说清了一个组合值的目的是什么。特别地，解析器将会认为c.*中的c是引用一个表名或别名，而不是一个列名，这样就不会出现混淆。而如果没有.*，就弄不清楚c到底是表示一个表名还是一个列名，并且在有一个名为c的列时会优先选择按列名来解释。

另一个演示这些概念的例子是下面这些查询，它们表示相同的东西：

```
SELECT * FROM inventory_item c ORDER BY c;
SELECT * FROM inventory_item c ORDER BY c.*;
SELECT * FROM inventory_item c ORDER BY ROW(c.*);
```

所有这些ORDER BY子句指定该行的组合值，导致根据第 6.23.6 节 “组合类型比较”中介绍的规则对行进行排序。不过，如果inventory_item包含一个名为*c*的列，第一种情况会不同于其他情况，因为它表示仅按那一列排序。给定之前所示的列名，下面这些查询也等效于上面的那些查询：

```
SELECT * FROM inventory_item c ORDER BY ROW(c.name, c.supplier_id, c.price);
SELECT * FROM inventory_item c ORDER BY (c.name, c.supplier_id, c.price);
```

（最后一种情况使用了一个省略关键字ROW的行构造器）。

另一种与组合值相关的特殊语法行为是，我们可以使用*函数记法*来抽取一个组合值的字段。解释这种行为的简单方式是记法*field*(*table*)和*table*.*field*是可以互换的。例如，这些查询是等效的：

```
SELECT c.name FROM inventory_item c WHERE c.price > 1000;
SELECT name(c) FROM inventory_item c WHERE price(c) > 1000;
```

此外，如果我们有一个函数接受单一的组合类型参数，我们可以以任意一种记法来调用它。这些查询全都是等效的：

```
SELECT somefunc(c) FROM inventory_item c;
SELECT somefunc(c.*) FROM inventory_item c;
SELECT c.somefunc FROM inventory_item c;
```

这种函数记法和字段记法之间的等效性使得我们可以在组合类型上使用函数来实现“计算字段”。 一个使用上述最后一种查询的应用不会直接意识到somefunc不是一个真实的表列。

> **提示**
>
> 由于这种行为，让一个接受单一组合类型参数的函数与该组合类型的任意字段具有相同的名称是不明智的。出现歧义时，如果使用了字段名语法，则字段名解释将被选择，而如果使用的是函数调用语法则会选择函数解释。

5.16.6. 组合类型输入和输出语法

一个组合值的外部文本表达由根据域类型的 I/O 转换规则解释的项，外加指示组合结构的装饰组成。装饰由整个值周围的圆括号（(和)），外加相邻项之间的逗号（,）组成。圆括号之外的空

格会被忽略，但是在圆括号之内空格会被当成域值的一部分，并且根据域数据类型的输入转换规则可能有意义，也可能没有意义。例如，在

```
'( 42)'
```

中，如果域类型是整数则空格会被忽略，而如果是文本则空格不会被忽略。

如前所示，在写一个组合值时，可以在任意域值周围写上双引号。如果不这样做会让域值迷惑组合值解析器，就必须这么做。特别地，包含圆括号、逗号、双引号或反斜线的域必须用双引号引用。要把一个双引号或者反斜线放在一个被引用的组合域值中，需要在它前面放上一个反斜线（还有，一个双引号引用的域值中的一对双引号被认为是表示一个双引号字符，这和 SQL 字符串中单引号的规则类似）。另一种办法是，可以避免引用以及使用反斜线转义来保护所有可能被当作组合语法的数据字符。

一个全空的域值（在逗号或圆括号之间完全没有字符）表示一个 NULL。要写一个空字符串值而不是 NULL，可以写成""。

如果域值是空串或者包含圆括号、逗号、双引号、反斜线或空格，组合输出例程将在域值周围放上双引号（对空格这样处理并不是不可缺少的，但是可以提高可读性）。嵌入在域值中的双引号及反斜线将被双写。

> **注意**
>
> 记住在一个 SQL 命令中写的东西将首先被解释为一个字符串，然后才会被解释为一个组合。这就让所需要的反斜线数量翻倍（假定使用了转义字符串语法）。例如，要在组合值中插入一个含有一个双引号和一个反斜线的text域，需要写成：
>
> ```
> INSERT ... VALUES ('("\"\\")');
> ```
>
> 字符串处理器会移除一层反斜线，这样在组合值解析器那里看到的就会是("\"\\")。接着，字符串被交给text数据类型的输入例程并且变成"\（如果我们使用的数据类型的输入例程也会特别处理反斜线，例如bytea，在命令中我们可能需要八个反斜线用来在组合域中存储一个反斜线）。美元引用（见第 1.1.2.4 节 “美元引用的字符串常量”）可以被用来避免双写反斜线。

> **提示**
>
> 当在 SQL 命令中书写组合值时，ROW构造器语法通常比组合文字语法更容易使用。在ROW中，单个域值可以按照平时不是组合值成员的写法来写。

5.17. 范围类型

范围类型是表达某种元素类型（称为范围的**subtype**）的一个值的范围的数据类型。例如，timestamp的范围可以被用来表达一个会议室被保留的时间范围。在这种情况下，数据类型是tsrange（“timestamp range”的简写）而timestamp是 subtype。subtype 必须具有一种总体的顺序，这样对于元素值是在一个范围值之内、之前或之后就是界线清楚的。

范围类型非常有用，因为它们可以表达一种单一范围值中的多个元素值，并且可以很清晰地表达诸如范围重叠等概念。用于时间安排的时间和日期范围是最清晰的例子；但是价格范围、一种仪器的量程等等也都有用。

5.17.1. 内建范围类型

UXDB 带有下列内建范围类型：

- int4range — integer的范围
- int8range — bigint的范围
- numrange — numeric的范围
- tsrange — 不带时区的 timestamp的范围
- tstzrange — 带时区的 timestamp的范围
- daterange — date的范围

此外，可以定义自己的范围类型，详见CREATE TYPE(7)。

5.17.2. 例子

```
CREATE TABLE reservation (room int, during tsrange);
INSERT INTO reservation VALUES
  (1108, '[2010-01-01 14:30, 2010-01-01 15:30)');

-- 包含
SELECT int4range(10, 20) @> 3;

-- 重叠
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- 抽取上界
SELECT upper(int8range(15, 25));

-- 计算交集
SELECT int4range(10, 20) * int4range(15, 25);

-- 范围为空吗?
SELECT isempty(numrange(1, 5));
```

范围类型上的操作符和函数的完整列表可见表 6.60 “范围操作符”以及表 6.61 “范围函数”。

5.17.3. 包含和排除边界

每一个非空范围都有两个界限，下界和上界。这些值之间的所有点都被包括在范围内。一个包含界限意味着边界点本身也被包括在范围内，而一个排除边界意味着边界点不被包括在范围内。

在一个范围的文本形式中，一个包含下界被表达为“[”而一个排除下界被表达为“(”。同样，一个包含上界被表达为“]”而一个排除上界被表达为“)”（详见第 5.17.5 节 “范围输入/输出”）。

函数lower_inc和upper_inc分别测试一个范围值的上下界。

5.17.4. 无限（无界）范围

一个范围的下界可以被忽略，意味着所有小于上界的值都被包括在范围中，例如(,3]。 同样，如果范围的上界被忽略，那么所有比上界大的值都被包括在范围中。如果上下界都被忽略，该元素类型的所有值都被认为在该范围中。 规定缺失的包括界限自动转换为排除，例如，[,] 转换为(,)。 可以认为这些缺失值为 +/- 无穷大，但它们是特殊范围类型值，并且被视为超出任何范围元素类型的 +/- 无穷大值。

具有“infinity”概念的元素类型可以用它们作为显式边界值。 例如，在时间戳范围，[today,infinity) 不包括特殊的 timestamp 值 infinity，尽管 [today,infinity] 包括它，就好比 [today,) 和 [today,].

函数lower_inf和upper_inf分别测试一个范围的无限上下界。

5.17.5. 范围输入/输出

一个范围值的输入必须遵循下列模式之一：

(*lower-bound*,*upper-bound*)
(*lower-bound*,*upper-bound*]
[*lower-bound*,*upper-bound*)
[*lower-bound*,*upper-bound*]
empty

圆括号或方括号指示上下界是否为排除的或者包含的。注意最后一个模式是empty，它表示一个空范围（一个不包含点的范围）。

*lower-bound*可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有下界。同样，*upper-bound*可以是作为 subtype 的合法输入的一个字符串，或者是空表示没有上界。

每个界限值可以使用"（双引号）字符引用。如果界限值包含圆括号、方括号、逗号、双引号或反斜线时，这样做是必须的，因为否则那些字符会被认作范围语法的一部分。要把一个双引号或反斜线放在一个被引用的界限值中，就在它前面放一个反斜线（还有，在一个双引号引用的界限值中的一对双引号表示一个双引号字符，这与 SQL 字符串中的单引号规则类似）。此外，可以避免引用并且使用反斜线转义来保护所有数据字符，否则它们会被当做返回语法的一部分。还有，要写一个是空字符串的界限值，则可以写成""，因为什么都不写表示一个无限界限。

范围值前后允许有空格，但是圆括号或方括号之间的任何空格会被当做上下界值的一部分（取决于元素类型，它可能是也可能不是有意义的）。

> **注意**
>
> 这些规则与组合类型文字中书写域值的规则非常相似。更多注解请见第 5.16.6 节 “组合类型输入和输出语法”。

例子：

-- 包括 3，不包括 7，并且包括 3 和 7 之间的所有点

```
SELECT '[3,7)'::int4range;

-- 既不包括 3 也不包括 7，但是包括之间的所有点
SELECT '(3,7)'::int4range;

-- 只包括单独一个点 4
SELECT '[4,4]'::int4range;

-- 不包括点（并且将被标准化为 ' 空 ' ）
SELECT '[4,4)'::int4range;
```

5.17.6. 构造范围

每一种范围类型都有一个与其同名的构造器函数。使用构造器函数常常比写一个范围文字常数更方便，因为它避免了对界限值的额外引用。构造器函数接受两个或三个参数。两个参数的形式以标准的形式构造一个范围（下界是包含的，上界是排除的），而三个参数的形式按照第三个参数指定的界限形式构造一个范围。第三个参数必须是下列字符串之一： “()”、 “(]”、 “[)”或者 “[]”。 例如：

```
-- 完整形式是：下界、上界以及指示界限包含性/排除性的文本参数。
SELECT numrange(1.0, 14.0, '(]');

-- 如果第三个参数被忽略，则假定为 ' [) ' 。
SELECT numrange(1.0, 14.0);

-- 尽管这里指定了 ' (] ' ，显示时该值将被转换成标准形式，因为 int8range 是一种离散范围类型（见下文）。
SELECT int8range(1, 14, '(]');

-- 为一个界限使用 NULL 导致范围在那一边是无界的。
SELECT numrange(NULL, 2.2);
```

5.17.7. 离散范围类型

一种范围的元素类型具有一个良定义的“步长”，例如integer或date。在这些类型中，如果两个元素之间没有合法值，它们可以被说成是相邻。这与连续范围相反，连续范围中总是（或者几乎总是）可以在两个给定值之间标识其他元素值。例如，numeric类型之上的一个范围就是连续的，timestamp上的范围也是（尽管timestamp具有有限的精度，并且在理论上可以被当做离散的，最好认为它是连续的，因为通常并不关心它的步长）。

另一种考虑离散范围类型的方法是对每一个元素值都有一种清晰的“下一个”或“上一个”值。了解了这种思想之后，通过选择原来给定的下一个或上一个元素值来取代它，就可以在一个范围界限的包含和排除表达之间转换。例如，在一个整数范围类型中，[4,8]和(3,9)表示相同的值集合，但是对于 numeric 上的范围就不是这样。

一个离散范围类型应该具有一个正规化函数，它知道元素类型期望的步长。正规化函数负责把范围类型的相等值转换成具有相同的表达，特别是与包含或者排除界限一致。如果没有指定一个正规化函数，那么具有不同格式的范围将总是会被当作不等，即使它们实际上是表达相同的一组值。

内建的范围类型int4range、int8range和daterange都使用一种正规的形式，该形式包括下界并且排除上界，也就是[)。不过，用户定义的范围类型可以使用其他习惯。

5.17.8. 定义新的范围类型

用户可以定义他们自己的范围类型。这样做最常见的原因是为了使用内建范围类型中没有提供的 subtype 上的范围。例如，要创建一个 subtype float8的范围类型：

```
CREATE TYPE floatrange AS RANGE (
   subtype = float8,
   subtype_diff = float8mi
);

SELECT '[1.234, 5.678]'::floatrange;
```

因为float8没有有意义的“步长”，我们在这个例子中没有定义一个正规化函数。

定义自己的范围类型也允许指定使用一个不同的子类型 B-树操作符类或者集合， 以便更改排序顺序来决定哪些值会落入到给定的范围中。

如果 subtype 被认为是具有离散值而不是连续值，CREATE TYPE命令应当指定一个canonical函数。正规化函数接收一个输入的范围值，并且必须返回一个可能具有不同界限和格式的等价的范围值。对于两个表示相同值集合的范围（例如[1, 7]和[1, 8)），正规的输出必须一样。选择哪一种表达作为正规的没有关系，只要两个具有不同格式的等价值总是能被映射到具有相同格式的相同值就行。除了调整包含/排除界限格式外，假使期望的补偿比 subtype 能够存储的要大，一个正规化函数可能会舍入边界值。例如，一个timestamp之上的范围类型可能被定义为具有一个一小时的步长，这样正规化函数可能需要对不是一小时的倍数的界限进行舍入，或者可能直接抛出一个错误。

另外，任何打算要和 GiST 或 SP-GiST 索引一起使用的范围类型应当定一个 subtype 差异或subtype_diff函数（没有subtype_diff时索引仍然能工作，但是可能效率不如提供了差异函数时高）。subtype 差异函数采用两个 subtype 输入值，并且返回表示为一个float8值的差（即*X*减*Y*）。在我们上面的例子中，可以使用常规float8减法操作符之下的函数。但是对于任何其他 subtype，可能需要某种类型转换。还可能需要一些关于如何把差异表达为数字的创新型想法。为了最大的可扩展性，subtype_diff函数应该同意选中的操作符类和排序规则所蕴含的排序顺序，也就是说，只要它的第一个参数根据排序顺序大于第二个参数，它的结果就应该是正值。

subtype_diff函数的一个不那么过度简化的例子：

```
CREATE FUNCTION time_subtype_diff(x time, y time) RETURNS float8 AS
'SELECT EXTRACT(EPOCH FROM (x - y))' LANGUAGE sql STRICT IMMUTABLE;

CREATE TYPE timerange AS RANGE (
   subtype = time,
   subtype_diff = time_subtype_diff
);

SELECT '[11:10, 23:00]'::timerange;
```

更多关于创建范围类型的信息请参考CREATE TYPE(7)。

5.17.9. 索引

可以为范围类型的表列创建 GiST 和 SP-GiST 索引。例如，要创建一个 GiST 索引：

```
CREATE INDEX reservation_idx ON reservation USING GIST (during);
```

一个 GiST 或 SP-GiST 索引可以加速涉及以下范围操作符的查询： =、 &&、 <@、 @>、 <<、 >>、 -|-、 &<以及 &> （详见表 6.60 “范围操作符”）。

此外，B-树和哈希索引可以在范围类型的表列上创建。对于这些索引类型，基本上唯一有用的范围操作就是等值。使用相应的< 和 >操作符，对于范围值定义有一种 B-树排序顺序，但是该顺序相当任意并且在真实世界中通常不怎么有用。范围类型的 B-树和哈希支持主要是为了允许在查询内部进行排序和哈希，而不是创建真正的索引。

5.17.10. 范围上的约束

虽然UNIQUE是标量值的一种自然约束，它通常不适合于范围类型。反而，一种排除约束常常更加适合（见CREATE TABLE ... CONSTRAINT ... EXCLUDE）。排除约束允许在一个范围类型上说明诸如“non-overlapping”的约束。例如：

```
CREATE TABLE reservation (
   during tsrange,
   EXCLUDE USING GIST (during WITH &&)
);
```

该约束将阻止任何重叠值同时存在于表中：

```
INSERT INTO reservation VALUES
   ('[2010-01-01 11:30, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO reservation VALUES
   ('[2010-01-01 14:45, 2010-01-01 15:45)');
ERROR:  conflicting key value violates exclusion constraint "reservation_during_excl"
DETAIL:  Key (during)=(["2010-01-01 14:45:00","2010-01-01 15:45:00")) conflicts
with existing key (during)=(["2010-01-01 11:30:00","2010-01-01 15:00:00")).
```

可以使用btree_gist扩展来在纯标量数据类型上定义排除约束，然后把它和范围排除结合可以得到最大的灵活性。例如，安装btree_gist之后，只有会议室号码相等时，下列约束将拒绝重叠的范围：

```
CREATE EXTENSION btree_gist;
CREATE TABLE room_reservation (
   room text,
   during tsrange,
   EXCLUDE USING GIST (room WITH =, during WITH &&)
);

INSERT INTO room_reservation VALUES
   ('123A', '[2010-01-01 14:00, 2010-01-01 15:00)');
INSERT 0 1

INSERT INTO room_reservation VALUES
   ('123A', '[2010-01-01 14:30, 2010-01-01 15:30)');
ERROR:  conflicting key value violates exclusion constraint "room_reservation_room_during_excl"
```

```
DETAIL:  Key (room, during)=(123A, ["2010-01-01 14:30:00","2010-01-01 15:30:00")) conflicts
with existing key (room, during)=(123A, ["2010-01-01 14:00:00","2010-01-01 15:00:00")).

INSERT INTO room_reservation VALUES
  ('123B', '[2010-01-01 14:30, 2010-01-01 15:30)');
INSERT 0 1
```

5.18. 域类型

域是一种用户定义的数据类型，它基于另一种*底层类型*。根据需要，它可以有约束来限制其有效值为底层类型所允许值的一个子集。如果没有约束，它的行为就和底层类型一样 — 例如，任何适用于底层类型的操作符或函数都对该域类型有效。底层类型可以是任何内建或者用户定义的基础类型、枚举类型、数组类型、组合类型、范围类型或者另一个域。

例如，我们可以在整数之上创建一个域，它只接受正整数：

```
CREATE DOMAIN posint AS integer CHECK (VALUE > 0);
CREATE TABLE mytable (id posint);
INSERT INTO mytable VALUES(1);  -- works
INSERT INTO mytable VALUES(-1);  -- fails
```

当底层类型的一个操作符或函数适用于一个域值时，域会被自动向下造型为底层类型。因此，mytable.id - 1的结果会被认为是类型integer而不是posint。我们可以写成(mytable.id - 1)::posint来把结果转换回posint，这会导致域的约束被重新检查。在这种情况下，如果该表达式被应用于一个值为1的*id*就会错误。把底层类型的值赋给域类型的一个字段或者变量不需要写显式的造型，但是域的约束将会被检查。

更多信息请参考CREATE DOMAIN(7)。

5.19. 对象标识符类型

对象标识符（OID）被UXDB用来在内部作为多个系统表的主键。 类型oid表示一个对象标识符。也有多个oid的别名类型：regproc,regprocedure, regoper, regoperator,regclass, regtype, regrole,regnamespace, regconfig, 和regdictionary。 表 5.27 “对象标识符类型”显示了一个概览。

oid类型目前被实现为一个无符号4字节整数。 因此，在大型数据库中它并不足以提供数据库范围内的唯一性，甚至在一些大型的表中也无法提供表范围内的唯一性。

oid类型本身除了比较之外只有很少的操作。不过，它可以被造型成整数，并且接着可以使用标准的整数操作符进行操纵（这样做时要注意有符号和无符号之间可能出现的混乱）。

OID的别名类型除了特定的输入和输出例程之外没有别的操作。这些例程可以接受并显示系统对象的符号名，而不是类型oid使用的原始数字值。别名类型使查找对象的OID值变得简单。例如，要检查与一个表mytable有关的ux_attribute行，可以写：

```
SELECT * FROM ux_attribute WHERE attrelid = 'mytable'::regclass;
```

而不是：

```
SELECT * FROM ux_attribute
```

```
WHERE attrelid = (SELECT oid FROM ux_class WHERE relname = 'mytable');
```

虽然从它本身看起来并没有那么糟，它仍然被过度简化了。如果有多个名为mytable的表存在于不同的模式中，就可能需要一个更复杂的子选择来选择右边的OID。regclass输入转换器会根据模式路径设置处理表查找，并且因此它会自动地完成这种“右边的事情”。类似地，对于一个数字OID的符号化显示可以很方便地通过将表OID造型成regclass来实现。

表 5.27. 对象标识符类型

名字	引用	描述	值示例
oid	任意	数字形式的对象标识符	564182
regproc	ux_proc	函数名字	sum
regprocedure	ux_proc	带参数类型的函数	sum(int4)
regoper	ux_operator	操作符名字	+
regoperator	ux_operator	带参数类型的操作符	*(integer,integer) or -(NONE,integer)
regclass	ux_class	关系名字	ux_type
regtype	ux_type	数据类型名字	integer
regrole	ux_authid	角色名	smithee
regnamespace	ux_namespace	名字空间名称	ux_catalog
regconfig	ux_ts_config	文本搜索配置	english
regdictionary	ux_ts_dict	文本搜索字典	simple

所有用于由名字空间组织的对象的 OID 别名类型都接受模式限定的名字，如果没有被限定的对象在当前搜索路径中无法找到时，将会在输出时显示模式限定的名字。regproc和regoper别名类型将只接受唯一的（非重载的）输入名字，因此它们的使用是受限的；对于大多数使用，regprocedure或regoperator更合适。对于regoperator，通过使用NONE来替代未使用的操作数可以标识一元操作符。

大部分 OID 别名类型的一个附加性质是依赖性的创建。如果这些类型之一的一个常量出现在一个存储的表达式（如一个列默认值表达式或视图）中，它会在被引用的对象上创建一个依赖。例如，如果一个列有一个默认值表达式nextval('my_seq'::regclass)，UXDB会理解该默认值表达式是依赖于序列my_seq的，在删除该默认值表达式之前系统将不允许删除该序列。regrole是这个性质的唯一例外。这种类型的常量不允许出现在这类表达式中。

> **注意**
>
> OID 别名类型不完全遵循事务隔离规则。规划器也把它们当做简单常量，这可能会导致次优的规划。

另一种系统中使用的标识符类型是xid，或者称为事务（简写为xact）标识符。这是系统列*xmin*和*xmax*使用的数据类型。事务标识符是32位量。

系统使用的第三种标识符类型是cid，或者称为命令标识符。这是系统列*cmin*和*cmax*使用的数据类型。命令标识符也是32位量。

系统使用的最后一种标识符类型是tid，或者称为元组标识符（行标识符）。这是系统列*ctid*使用的数据类型。一个元组ID是一个（块号，块内元组索引）对，它标识了行在它的表中的物理位置。

（这些系统列在第 2.5 节 “系统列”中有进一步的解释）。

5.20. ux_lsn 类型

ux_lsn数据类型可以被用来存储 LSN（日志序列号）数据，LSN 是一个指向WAL中的位置的指针。这个类型是XLogRecPtr的一种表达并且是 UXDB的一种内部系统类型。

在内部，一个 LSN 是一个 64 位整数，表示在预写式日志流中的一个字节位置。它被打印成 两个最高 8 位的十六进制数，中间用斜线分隔，例如16/B374D848。 ux_lsn类型支持标准的比较操作符，如=和 >。两个 LSN 可以用-操作符做减法， 结果将是分隔两个预写式日志位置的字节数。

5.21. 伪类型

UXDB类型系统包含了一些特殊目的的项，它们被统称为伪类型。一个伪类型不能被用作一个列的数据类型，但是它可以被用来定义一个函数的参数或者结果类型。每一种可用的伪类型都有其可以发挥作用的情况，这些情况的特点是一个函数的行为并不能符合于简单使用或者返回一种特定SQL数据类型的值。表 5.28 “伪类型”列出了现有的伪类型。

表 5.28. 伪类型

名字	描述
any	表示一个函数可以接受任意输入数据类型。
anyelement	表示一个函数可以接受任意数据类型。
anyarray	表示一个函数可以接受任意数组数据类型。
anynonarray	表示一个函数可以接受任意非数组数据类型。
anyenum	表示一个函数可以接受任意枚举数据类型（参见第 5.7 节 “枚举类型”）。
anyrange	表示一个函数可以接受任意范围数据类型（参见第 5.17 节 “范围类型”）。
cstring	表示一个函数接受或者返回一个非空结尾的C字符串。
internal	表示一个函数接受或返回一个服务器内部数据类型。
language_handler	一个被声明为返回language_handler的过程语言调用处理器。
fdw_handler	一个被声明为返回fdw_handler的外部数据包装器处理器。
index_am_handler	一个被声明为返回index_am_handler索引访问方法处理器。
tsm_handler	一个被声明为返回tsm_handler的表采样方法处理器。
record	标识一个接收或者返回一个未指定的行类型的函数。
trigger	一个被声明为返回trigger的触发器函数。
event_trigger	一个被声明为返回event_trigger的事件触发器函数。

名字	描述
ux_ddl_command	标识一种对事件触发器可用的 DDL 命令的表达。
void	表示一个函数不返回值。
unknown	标识一种还未被解析的类型，例如一个未修饰的字符文本。
opaque	一种已被废弃的类型名称，以前它用于实现以上的很多种目的。

用C编写的函数（不管是内建的还是动态载入的）可以被声明为接受或返回这些为数据类型的任意一种。函数的作者应当保证当一个伪类型被用作一个参数类型时函数的行为是安全的。

用过程语言编写的函数只有在其实现语言允许的情况下才能使用伪类型。目前大部分过程语言都禁止使用伪类型作为一种参数类型，并且只允许使用void和record作为结果类型（如果函数被用于一个触发器或者事件触发器， trigger或者event_trigger也被允许作为结果类型）。某些过程语言也支持在多态函数中使用类型anyelement、anyarray、anynonarray、anyenum和anyrange。

internal伪类型用于定义只在数据库系统内部调用的函数，这些函数不会被SQL直接调用。如果一个函数拥有至少一个internal类型的参数，则它不能从SQL中被调用。为了保持这种限制的类型安全性，遵循以下编码规则非常重要：不要创建任何被声明要返回internal的函数，除非它有至少一个internal参数。

第 6 章 函数和操作符

UXDB为内建的数据类型提供了大量的函数和操作符。 用户也可以定义它们自己的函数和操作符。uxsql命令\df和\do可以分别被用于显示所有可用的函数和操作符的列表。

如果关心移植性，那么请注意，我们在本章描述的大多数函数和操作符， 除了最琐碎的算术和比较操作符以及一些做了明确标记的函数以外，都没有在SQL标准里声明。某些这种扩展的功能也出现在许多其它SQL数据库管理系统中，并且在很多情况下多个实现的这种功能是相互兼容的和一致的。本章也并没有穷尽一切信息；一些附加的函数在本手册的相关小节里出现。

6.1. 逻辑操作符

常用的逻辑操作符有：

AND
OR
NOT
SQL使用三值的逻辑系统，包括真、假和null，null表示“未知”。观察下面的真值表：

a	*b*	*a* AND *b*	*a* OR *b*
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
TRUE	NULL	NULL	TRUE
FALSE	FALSE	FALSE	FALSE
FALSE	NULL	FALSE	NULL
NULL	NULL	NULL	NULL

a	NOT *a*
TRUE	FALSE
FALSE	TRUE
NULL	NULL

操作符AND和OR是可交换的，也就是说，可以交换左右操作数而不影响结果。但是请参阅第 1.2.14 节 “表达式计算规则”获取有关子表达式计算顺序的更多信息。

6.2. 比较函数和操作符

常见的比较操作符都可用，如表 6.1 “比较操作符”所示。

表 6.1. 比较操作符

操作符	描述
<	小于
>	大于
<=	小于等于
>=	大于等于
=	等于

操作符	描述
<> or !=	不等于

注意

1. !=操作符在分析器阶段被转换成<>。不能把!=和<>操作符实现为做不同的事。
2. 支持部分多符号操作符中间存在空白（空格或者换行或者制表符，一个或者多个），后续处理中数据库会忽略掉中间的空白。

比较操作符可以用于所有可以比较的数据类型。所有比较操作符都是双目操作符，它们返回boolean类型；类似于1 < 2 < 3的表达式是非法的（因为没有<操作符可以比较一个布尔值和3）。

如表 6.2 “比较谓词”所示，也有一些比较谓词。它们的行为和操作符很像，但是具有 SQL 标准所要求的特殊语法。

表 6.2. 比较谓词

谓词	描述
a BETWEEN *x* AND *y*	在*x*和*y*之间
a NOT BETWEEN *x* AND *y*	不在*x*和*y*之间
a BETWEEN SYMMETRIC *x* AND *y*	在对比较值排序后位于*x*和*y*之间
a NOT BETWEEN SYMMETRIC *x* AND *y*	在对比较值排序后不位于*x*和*y*之间
a IS DISTINCT FROM *b*	不等于，空值被当做一个普通值
a IS NOT DISTINCT FROM *b*	等于，空值被当做一个普通值
expression IS NULL	是空值
expression IS NOT NULL	不是空值
expression ISNULL	是空值（非标准语法）
expression NOTNULL	不是空值（非标准语法）
boolean_expression IS TRUE	为真
boolean_expression IS NOT TRUE	为假或未知
boolean_expression IS FALSE	为假
boolean_expression IS NOT FALSE	为真或者未知
boolean_expression IS UNKNOWN	值为未知
boolean_expression IS NOT UNKNOWN	为真或者为假

BETWEEN谓词可以简化范围测试：

a BETWEEN *x* AND *y*

等效于

a >= *x* AND *a* <= *y*

注意BETWEEN认为终点值是包含在范围内的。 NOT BETWEEN可以做相反比较：

a NOT BETWEEN *x* AND *y*

等效于

a < *x* OR *a* > *y*

BETWEEN SYMMETRIC和BETWEEN相似，不过BETWEEN SYMMETRIC不要求AND左边的参数小于或等于右边的参数。如果左参数不是小于等于右参数，这两个参数会自动被交换，这样总是会应用一个非空范围。

当有一个输入为空时，普通的比较操作符会得到空（表示“未知”），而不是真或假。例如，7 = NULL得到空，7 <> NULL也一样。如果这种行为不合适，可以使用IS [NOT] DISTINCT FROM谓词：

a IS DISTINCT FROM *b*
a IS NOT DISTINCT FROM *b*

对于非空输入，IS DISTINCT FROM和<>操作符一样。不过，如果两个输入都为空，它会返回假。而如果只有一个输入为空，它会返回真。类似地，IS NOT DISTINCT FROM对于非空输入的行为与=相同，但是当两个输入都为空时它返回真，并且当只有一个输入为空时返回假。因此，这些谓词实际上把空值当作一种普通数据值而不是“unknown”。

要检查一个值是否为空，使用下面的谓词：

expression IS NULL
expression IS NOT NULL

或者等效，但并不标准的谓词：

expression ISNULL
expression NOTNULL

不要写*expression* = NULL，因为NULL是不“等于”NULL的（控制代表一个未知的值，因此我们无法知道两个未知的数值是否相等）。

> **提示**
>
> 有些应用可能要求表达式*expression* = NULL在*expression*得出空值时返回真。我们强烈建议这样的应用修改成遵循 SQL 标准。但是，如果这样修改不可能完成，那么我们可以使用配置变量transform_null_equals。如果打开它，UXDB将把x = NULL子句转换成x IS NULL。

如果*expression*是行值，那么当行表达式本身为非空值或者行的所有域为非空时IS NULL为真。由于这种行为，IS NULL和IS NOT NULL并不总是为行值表达式返回反转的结果，特别是，一个同时包含 NULL 和非空值的域将会对两种测试都返回假。在某些情况下，写成*row* IS DISTINCT FROM NULL或者*row* IS NOT DISTINCT FROM NULL会更好，它们只会检查整个行值是否为空而不需要在行的域上做额外的测试。

布尔值也可以使用下列谓词进行测试：

boolean_expression IS TRUE
boolean_expression IS NOT TRUE
boolean_expression IS FALSE
boolean_expression IS NOT FALSE
boolean_expression IS UNKNOWN
boolean_expression IS NOT UNKNOWN

这些谓词将总是返回真或假，从来不返回空值，即使操作数是空也如此。空值输入被当做逻辑值“未知”。 请注意实际上IS UNKNOWN和IS NOT UNKNOWN分别与IS NULL和IS NOT NULL相同， 只是输入表达式必须是布尔类型。

如表 6.3 “比较函数”中所示，也有一些比较相关的函数可用。

表 6.3. 比较函数

函数	描述	例子	例子结果
num_nonnulls(VARIADIC "any")	返回非空参数的数量	num_nonnulls(1, NULL, 2)	2
num_nulls(VARIADIC "any")	返回空参数的数量	num_nulls(1, NULL, 2)	1

6.3. 数学函数和操作符

UXDB为很多类型提供了数学操作符。对于那些没有标准数学表达的类型（如日期/时间类型），我们将在后续小节中描述实际的行为。

表 6.4 “数学操作符”展示了所有可用的数学操作符。

表 6.4. 数学操作符

操作符	描述	例子	结果
+	加	2 + 3	5
-	减	2 - 3	-1
*	乘	2 * 3	6
/	除（整数除法截断结果）	4 / 2	2
%	模（取余）	5 % 4	1
^	指数（从左至右结合）	2.0 ^ 3.0	8
|/	平方根	|/ 25.0	5
||/	立方根	||/ 27.0	3
!	阶乘	5 !	120
!!	阶乘（前缀操作符）	!! 5	120
@	绝对值	@ -5.0	5
&	按位与	91 & 15	11
|	按位或	32 | 3	35

操作符	描述	例子	结果
#	按位异或	17 # 5	20
~	按位求反	~1	-2
<<	按位左移	1 << 4	16
>>	按位右移	8 >> 2	2

按位操作操作符只能用于整数数据类型，而其它的操作符可以用于全部数字数据类型。按位操作的操作符还可以用于位串类型bit和bit varying， 如表 6.16 “位串操作符”所示。

表 6.5 “数学函数”显示了可用的数学函数。在该表中，dp表示double precision。这些函数中有许多都有多种不同的形式，区别是参数不同。除非特别指明，任何特定形式的函数都返回和它的参数相同的数据类型。 处理double precision数据的函数大多数是在宿主系统的 C 库基础上实现的；因此，边界情况下的准确度和行为是根据宿主系统而变化的。

表 6.5. 数学函数

函数	返回类型	描述	例子	结果
abs(*x*)	（和输入相同）	绝对值	abs(-17.4)	17.4
cbrt(dp)	dp	立方根	cbrt(27.0)	3
ceil(dp or numeric)	（和输入相同）	不小于参数的最近的整数	ceil(-42.8)	-42
ceiling(dp or numeric)	（和输入相同）	不小于参数的最近的整数（ceil的别名）	ceiling(-95.3)	-95
degrees(dp)	dp	把弧度转为角度	degrees(0.5)	28.6478897565412
div(*y* numeric, *x* numeric)	numeric	*y*/*x*的整数商	div(9,4)	2
exp(dp or numeric)	（和输入相同）	指数	exp(1.0)	2.71828182845905
floor(dp or numeric)	（和输入相同）	不大于参数的最近的整数	floor(-42.8)	-43
ln(dp or numeric)	（和输入相同）	自然对数	ln(2.0)	0.693147180559945
log(dp or numeric)	（和输入相同）	以10为底的对数	log(100.0)	2
log10(dp or numeric)	（和输入相同）	以10为底的对数	log10(100.0)	2
log(*b* numeric, *x* numeric)	numeric	以*b*为底的对数	log(2.0, 64.0)	6.0000000000
mod(*y*, *x*)	（和参数类型相同）	*y*/*x*的余数	mod(9,4)	1
pi()	dp	“π”常数	pi()	3.14159265358979
power(*a* dp, *b* dp)	dp	求*a*的*b*次幂	power(9.0, 3.0)	729
power(*a* numeric, *b* numeric)	numeric	求*a*的*b*次幂	power(9.0, 3.0)	729
radians(dp)	dp	把角度转为弧度	radians(45.0)	0.785398163397448

函数	返回类型	描述	例子	结果
round(dp or numeric)	（和输入相同）	圆整为最接近的整数	round(42.4)	42
round(*v* numeric, *s* int)	numeric	圆整为*s*位小数数字	round(42.4382, 2)	42.44
round(*source* ,[*n*])	numeric	将source四舍五入到小数点右侧第n位。如果n是负数则round到小数点左侧。若忽略n则round到个位，等同于round(source,0)，若n是小数，则舍去小数位。 source可支持以下类型：smallint, integer, bigint, double Precision, numeric, real, float,, text, char, nchar, varchar, varchar2, nvarchar2	select round(12.3456::double, '1.2'::varchar);	12.3
scale(numeric)	integer	参数的精度（小数点后的位数）	scale(8.41)	2
sign(dp or numeric)	（和输入相同）	参数的符号（-1, 0, +1）	sign(-8.4)	-1
sqrt(dp or numeric)	（和输入相同）	平方根	sqrt(2.0)	1.4142135623731
trunc(dp or numeric)	（和输入相同）	截断（向零靠近）	trunc(42.8)	42
trunc(*v* numeric, *s* int)	numeric	截断为*s*位小数位置的数字	trunc(42.4382, 2)	42.43
trunc(text)	text	对函数数值或者时间类型格式的字符串截取，优先选择字符串为数值，其次按照dd截取时间，否则报错。	trunc('2021-02-05 12:12:12');	2021-02-05 00:00:00
width_bucket(*op* dp, *b1* dp, *b2* dp, *count* int)	int	返回一个桶号，这个桶是在一个柱状图中*operand*将被分配的那个桶，该	width_bucket(5.35, 0.024, 10.06, 5)	3

函数	返回类型	描述	例子	结果
		柱状图有*count*个散布在范围*b1*到*b2*上的等宽桶。对于超过该范围的输入，将返回0或者*count*+1		
width_bucket(*op* numeric, *b1* numeric, *b2* numeric, *count* int)	int	返回一个桶号，这个桶是在一个柱状图中*operand*将被分配的那个桶，该柱状图有*count*个散布在范围*b1*到*b2*上的等宽桶。对于超过该范围的输入，将返回0或者*count*+1	width_bucket(5.35, 0.024, 10.06, 5)	3
width_bucket (*operand* anyelement, *thresholds* anyarray)	int	返回一个桶号，这个桶是在给定数组中*operand* 将被分配的桶，该数组列出了桶的下界。对于一个低于第一个下界的输入返回0。*thresholds*数组必须被排好序， 最小的排在最前面，否则将会得到意想不到的结果	width_bucket(now (), array['yesterday', 'today', 'tomorrow'] ::timestamptz[])	2

mod(y, x)函数参数类型支持数字类型及纯数字的字符串类型，不支持空串。

表 6.6 “随机函数”展示了用于产生随机数的函数。

表 6.6. 随机函数

函数	返回类型	描述
random()	dp	范围 0.0 <= x < 1.0 中的随机值
setseed(dp)	void	为后续的random()调用设置种子（值为于 -1.0 和 1.0 之间，包括边界值）

random()函数使用了一个简单的线性共轭算法。 它的速度很快，但不适合于密码学应用。 如果setseed()被调用，那么当前会话中的后续random()调用的结果可以通过使用相同的参数重新发布setseed()来重复。

表 6.7 “三角函数”显示了可用的三角函数。所有这些函数都有类型为double precision的参数和返回类型。每一种三角函数都有两个变体，一个以弧度度量角，另一个以角度度量角。

表 6.7. 三角函数

函数（弧度）	函数（角度）	描述
acos(*x*)	acosd(*x*)	反余弦
asin(*x*)	asind(*x*)	反正弦
atan(*x*)	atand(*x*)	反正切
atan2(*y*, *x*)	atan2d(*y*, *x*)	*y*/*x*的反正切
cos(*x*)	cosd(*x*)	余弦
cot(*x*)	cotd(*x*)	余切
sin(*x*)	sind(*x*)	正弦
tan(*x*)	tand(*x*)	正切

> **注意**
>
> 另一种使用以角度度量的角的方法是使用早前展示的单位转换函数radians()和degrees()。不过，使用基于角度的三角函数更好，因为这类方法能避免sind(30)等特殊情况下的舍入偏差。

表 6.8 “双曲函数”显示的是 可用的双曲函数。 所有这些函数接收参数，并返回类型为double precision的值。

表 6.8. 双曲函数

函数	描述	举例	结果
sinh(*x*)	双曲正弦	sinh(0)	0
cosh(*x*)	双曲余弦	cosh(0)	1
tanh(*x*)	双曲切线	tanh(0)	0
asinh(*x*)	反双曲正弦	asinh(0)	0
acosh(*x*)	反双曲余弦	acosh(1)	0
atanh(*x*)	反双曲切线	atanh(0)	0

6.4. 字符串函数和操作符

本节描述了用于检查和操作字符串值的函数和操作符。在这个环境中的串包括所有类型character、character varying和text的值。除非另外说明，所有下面列出的函数都可以处理这些类型，不过要注意的是，在使用character类型的时候， 它有自动填充空白的潜在影响。有些函数还可以处理位串类型。

SQL定义了一些字符串函数，它们使用关键字，而不是逗号来分隔参数。详情请见表 6.9 “SQL字符串函数和操作符”，UXDB也提供了这些函数使用正常函数调用语法的版本（见表 6.10 “其他字符串函数”）。

> **注意**
>
> 这些强制措施在目前的版本中已经被删除，因为它们常常导致令人惊讶的行为。不过，字符串串接操作符（||）仍然接受非字符串输入，只要至少一个输入是一种字符串类型，如表 6.9 “SQL字符串函数和操作符”所示。对于其他情况，如果需要复制之前的行为，可以为text插入一个显式强制措施。

表 6.9. SQL字符串函数和操作符

函数	返回类型	描述	例子	结果
string || *string*	text	串接	'Post' || 'greSQL'	UXDB
string || *non-string* or *non-string* || *string*	text	使用一个非字符串输入的串接	'Value: ' || 42	Value: 42
bit_length(*string*)	int	串中的位数	bit_length('jose')	32
char_length(*string*) or character_length(*string*)	int	串中字符数	char_length('jose')	4
lower(*string*)	text	将字符串转换为小写形式	lower('TOM')	tom
octet_length(*string*)	int	串中的字节数	octet_length('jose')	4
overlay(*string* placing *string* from int [for int])	text	替换子串	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(*substring* in *string*)	int	定位指定子串	position('om' in 'Thomas')	3
substring(*string* [from int] [for int])	text	提取子串	substring('Thomas' from 2 for 3)	hom
substring(*string* from *pattern*)	text	提取匹配POSIX正则表达式的子串。模式匹配详情见第 6.7 节 “模式匹配”。	substring('Thomas' from '...$')	mas
substring(*string* from *pattern* for *escape*)	text	提取匹配SQL正则表达式的子串。模式匹配详情见第 6.7 节 “模式匹配”。	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading | trailing | both] [*characters*] from *string*)	text	从*string*的开头、结尾或者两端（both是默认值）移除只包含*characters*（默认是一个空格）	trim(both 'xyz' from 'yxTomxx')	Tom

函数	返回类型	描述	例子	结果
		中字符的最长字符串		
trim([leading | trailing | both] [from] *string* [, *characters*])	text	trim()的非标准版本	trim(both from 'xTomxx', 'x')	Tom
upper(*string*)	text	将字符串转换成大写形式	upper('tom')	TOM

还有额外的串操作函数可以用，它们在表 6.10 “其他字符串函数”中列出。它们有些在内部用于实现表 6.9 “SQL字符串函数和操作符”列出的SQL标准字符串函数。

表 6.10. 其他字符串函数

函数	返回类型	描述	例子	结果
ascii(*string*)	int	参数第一个字符的ASCII代码。对于UTF8返回该字符的Unicode代码点。对于其他多字节编码，该参数必须是一个ASCII字符。	ascii('x')	120
btrim(*string* text [, *characters* text])	text	从*string*的开头或结尾删除最长的只包含*characters*（默认是一个空格）的串	btrim('xyxtrimyyx', 'xyz')	trim
btrim(*expr*[, *chr*])	text	去除expr头尾的指定chr。调用时只有expr而不指定chr时，可以去除expr头尾的空格。	btrim('abcda','a')	Text
chr(int)	text	给定代码的字符。对于UTF8该参数被视作一个Unicode代码点。对于其他多字节编码该参数必须指定一个ASCII字符。NULL (0) 字符不被允许，因为文本数据类型不能存储这种字节。	chr(65)	A

函数	返回类型	描述	例子	结果
concat(*str* "any" [, *str* "any" [, ...]])	text	串接所有参数的文本表示。NULL参数被忽略。	concat('abcde', 2, NULL, 22)	abcde222
concat_ws(*sep* text, *str* "any" [, *str* "any" [, ...]])	text	将除了第一个参数外的其他参数用分隔符串接在一起。第一个参数被用作分隔符字符串。NULL参数被忽略。	concat_ws(',', 'abcde', 2, NULL, 22)	abcde,2,22
convert(*string* bytea, *src_encoding* name, *dest_encoding* name)	bytea	将字符串转换为*dest_encoding*。原始编码由*src_encoding*指定。*string*在这个编码中必须可用。转换可以使用CREATE CONVERSION定义。也有一些预定义的转换。可用的转换请见表 6.11 “内建转换”。	convert ('text_in_utf8', 'UTF8', 'LATIN1')	用Latin-1 encoding (ISO 8859-1) 表示的text_in_utf8
convert_from(*string* bytea, *src_encoding* name)	text	将字符串转换为数据库编码。原始编码由*src_encoding*指定。*string*在这个编码中必须可用。	convert_from ('text_in_utf8', 'UTF8')	用当前数据库编码表示的text_in_utf8
convert_to(*string* text, *dest_encoding* name)	bytea	将字符串转换为*dest_encoding*。	convert_to('some text', 'UTF8')	用UTF8编码表达的some text
decode(*string* text, *format* text)	bytea	从*string*中的文本表达解码二进制数据。*format*的选项和encode中的一样。	decode ('MTIzAAE=', 'base64')	\x3132330001
encode(*data* bytea, *format* text)	text	将二进制数据编码成一个文本表达。支持的格式有：base64、hex、escape。escape将零字节和高位组字节转换为八进制序列（*nnn*）和双写的反斜线。	encode('123\000\001', 'base64')	MTIzAAE=

函数	返回类型	描述	例子	结果
find_in_set(*str* text, *strlist* text)	int	找出str在strlist中第一次出现的位置。	find_in_set('hello', 'hi,hello,ai')	2
format(*formatstr* text [, *formatarg* "any" [, ...]])	text	根据一个格式字符串格式化参数。该函数和C函数sprintf相似。见第 6.4.1 节“format”。	format('Hello %s, %1$s', 'World')	Hello World, World
initcap(*string*)	text	将每一个词的第一个字母转换为大写形式并把剩下的字母转换为小写形式。词是由非字母数字字符分隔的字母数字字符的序列。	initcap('hi THOMAS')	Hi Thomas
left(*str* text, *n* int)	text	返回字符串中的前*n*个字符。当*n*为负时，将返回除了最后|*n*|个字符之外的所有字符。	left('abcde', 2)	ab
length(*string*)	int	*string*中的字符数	length('jose')	4
length(*string* bytea, *encoding* name)	int	*string*在给定编码中的字符数。*string*必须在这个编码中有效。	length('jose', 'UTF8')	4
lpad(*string* text, *length* int [, *fill* text])	text	将*string*通过前置字符*fill*（默认是一个空格）填充到长度*length*。如果*string*已经长于*length*，则它被（从右边）截断。	lpad('hi', 5, 'xy')	xyxhi
ltrim(*string* text [, *characters* text])	text	从*string*的开头删除最长的只包含*characters*（默认是一个空格）的串	ltrim('zzzytest', 'xyz')	test
md5(*string*)	text	计算*string*的 MD5 哈希，返回十六进制的结果	md5('abc')	900150983cd24fb0d6963f7d28e17f72

函数	返回类型	描述	例子	结果
parse_ident (*qualified_identifier* text [, *strictmode* boolean DEFAULT true])	text[]	把 *qualified_identifier* 分成一个标识符数组，移除单个标识符上的任何引号。默认情况下，最后一个标识符后面的多余字符会被当做错误。但是如果第二个参数为false，那么这一类多余的字符会被忽略（这种行为对于解析函数之类的对象名称有用）。注意这个函数不会截断超长标识符。如果想要进行截断，可以把结果转换成name[]。	parse_ident('"SomeSchema".someTable')	{SomeSchema, sometable}
ux_client_encoding()	name	当前的客户端编码名字	ux_client_encoding()	SQL_ASCII
quote_ident(*string* text)	text	将给定字符串返回成合适的引用形式，使它可以在一个SQL语句字符串中被用作一个标识符。只有需要时才会加上引号（即，如果字符串包含非标识符字符或可能是大小写折叠的）。嵌入的引号会被正确地双写。	quote_ident('Foo bar')	"Foo bar"
quote_literal(*string* text)	text	将给定字符串返回成合适的引用形式，使它可以在一个SQL语句字符串中被用作一个字符串文字。嵌入的引号会被正确地双写。注意quote_literal对空输入返回空；如果参数可能为	quote_literal(E'O\'Reilly')	'O''Reilly'

函数	返回类型	描述	例子	结果
		空，quote_nullable通常更合适。		
quote_literal(*value* anyelement)	text	强迫给定值为文本并且接着将它用引号包围作为一个文本。嵌入的单引号和反斜线被正确的双写。	quote_literal(42.5)	'42.5'
quote_nullable(*string* text)	text	将给定字符串返回成合适的引用形式，使它可以在一个SQL语句字符串中被用作一个字符串文字；或者，如果参数为空，返回NULL。嵌入的引号会被正确地双写。	quote_nullable(NULL)	NULL
quote_nullable(*value* anyelement)	text	强迫给定值为文本并且接着将它用引号包围作为一个文本；或者，如果参数为空，返回NULL。嵌入的单引号和反斜线被正确的双写。	quote_nullable(42.5)	'42.5'
regexp_match(*string* text, *pattern* text [, *flags* text])	text[]	返回一个POSIX正则表达式与*string*的第一个匹配得到的子串。更多信息请见第 6.7.3 节“POSIX正则表达式”。	regexp_match('foobarbequebaz', '(bar)(beque)')	{bar,beque}
regexp_matches(*string* text, *pattern* text [, *flags* text])	setof text[]	返回一个POSIX正则表达式与*string*匹配得到的子串。更多信息请见第 6.7.3 节“POSIX正则表达式”。	regexp_matches('foobarbequebaz', 'ba.', 'g')	{bar} {baz} (2 rows)
regexp_replace(*string* text, *pattern* text, *replacement* text [, *flags* text])	text	替换匹配一个POSIX正则表达式的子串。详见第 6.7.3 节	regexp_replace('Thomas', '.[mN]a.', 'M')	ThM

函数	返回类型	描述	例子	结果
		“POSIX正则表达式”。		
regexp_split_to_array(*string* text, *pattern* text [, *flags* text])	text[]	使用一个POSIX正则表达式作为分隔符划分*string*。详见第 6.7.3 节“POSIX正则表达式”。	regexp_split_to_array('hello world', '\s+')	{hello,world}
regexp_split_to_table(*string* text, *pattern* text [, *flags* text])	setof text	使用一个POSIX正则表达式作为分隔符划分*string*。详见第 6.7.3 节“POSIX正则表达式”。	regexp_split_to_table('hello world', '\s+')	hello world (2 rows)
repeat(*string* text, *number* int)	text	重复*string*指定的*number*次	repeat('Ux', 4)	UxUxUxUx
replace(*string* text, *from* text, *to* text)	text	将*string*中出现的所有子串*from*替换为子串*to*	replace ('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
reverse(*str*)	text	返回反转的字符串。	reverse('abcde')	edcba
right(*str* text, *n* int)	text	返回字符串中的最后*n*个字符。如果*n*为负，返回除最前面的|*n*|个字符外的所有字符。	right('abcde', 2)	de
rpad(*string* text, *length* int [, *fill* text])	text	将*string*通过增加字符*fill*（默认为一个空格）填充到长度*length*。如果*string*已经长于*length*则它会被截断。	rpad('hi', 5, 'xy')	hixyx
rtrim(*string* text [, *characters* text])	text	从*string*的结尾删除最长的只包含*characters*（默认是一个空格）的串	rtrim('testxxzx', 'xyz')	test
split_part(*string* text, *delimiter* text, *field* int)	text	按*delimiter*划分*string*并返回给定域（从1开始计算）	split_part('abc~@~def~@~ghi', '~@~', 2)	def
strpos(*string*, *substring*)	int	指定子串的位置（和position (*substring* in *string*)相同，但是注意	strpos('high', 'ig')	2

函数	返回类型	描述	例子	结果
		相反的参数顺序）		
substr(*string*, *from* [, *count*])	text	提取子串（与substring (*string* from *from* for *count*)相同）	substr('alphabet', 3, 2)	ph
starts_with(*string*, *prefix*)	bool	如果*string*以*prefix*开始则返回真。	starts_with ('alphabet', 'alph')	t
to_ascii(*string* text [, *encoding* text])	text	将*string*从另一个编码转换到ASCII（只支持从LATIN1、LATIN2、LATIN9和WIN1250编码的转换）	to_ascii('Karel')	Karel
to_hex(*number* int or bigint)	text	将*number*转换到它等效的十六进制表示	to_hex(2147483647)	7fffffff
translate(*string* text, *from* text, *to* text)	text	*string*中任何匹配*from*集合中一个字符的字符会被替换成*to*集合中的相应字符。如果*from*比*to*长，*from*中的额外字符会被删除。	translate('12345', '143', 'ax')	a2x5

concat、concat_ws和format函数是可变的，因此可以把要串接或格式化的值作为一个标记了VARIADIC关键字的数组进行传递。数组的元素被当作函数的独立普通参数一样处理。如果可变数组参数为 NULL，concat和concat_ws返回 NULL，但format把 NULL 当作一个零元素数组。

还可以参阅第 6.20 节 “聚集函数”中的string_agg。

表 6.11. 内建转换

转换名 [a]	源编码	目标编码
ascii_to_mic	SQL_ASCII	MULE_INTERNAL
ascii_to_utf8	SQL_ASCII	UTF8
big5_to_euc_tw	BIG5	EUC_TW
big5_to_mic	BIG5	MULE_INTERNAL
big5_to_utf8	BIG5	UTF8
euc_cn_to_mic	EUC_CN	MULE_INTERNAL
euc_cn_to_utf8	EUC_CN	UTF8
euc_jp_to_mic	EUC_JP	MULE_INTERNAL
euc_jp_to_sjis	EUC_JP	SJIS

转换名 [a]	源编码	目标编码
euc_jp_to_utf8	EUC_JP	UTF8
euc_kr_to_mic	EUC_KR	MULE_INTERNAL
euc_kr_to_utf8	EUC_KR	UTF8
euc_tw_to_big5	EUC_TW	BIG5
euc_tw_to_mic	EUC_TW	MULE_INTERNAL
euc_tw_to_utf8	EUC_TW	UTF8
gb18030_to_utf8	GB18030	UTF8
gbk_to_utf8	GBK	UTF8
iso_8859_10_to_utf8	LATIN6	UTF8
iso_8859_13_to_utf8	LATIN7	UTF8
iso_8859_14_to_utf8	LATIN8	UTF8
iso_8859_15_to_utf8	LATIN9	UTF8
iso_8859_16_to_utf8	LATIN10	UTF8
iso_8859_1_to_mic	LATIN1	MULE_INTERNAL
iso_8859_1_to_utf8	LATIN1	UTF8
iso_8859_2_to_mic	LATIN2	MULE_INTERNAL
iso_8859_2_to_utf8	LATIN2	UTF8
iso_8859_2_to_windows_1250	LATIN2	WIN1250
iso_8859_3_to_mic	LATIN3	MULE_INTERNAL
iso_8859_3_to_utf8	LATIN3	UTF8
iso_8859_4_to_mic	LATIN4	MULE_INTERNAL
iso_8859_4_to_utf8	LATIN4	UTF8
iso_8859_5_to_koi8_r	ISO_8859_5	KOI8R
iso_8859_5_to_mic	ISO_8859_5	MULE_INTERNAL
iso_8859_5_to_utf8	ISO_8859_5	UTF8
iso_8859_5_to_windows_1251	ISO_8859_5	WIN1251
iso_8859_5_to_windows_866	ISO_8859_5	WIN866
iso_8859_6_to_utf8	ISO_8859_6	UTF8
iso_8859_7_to_utf8	ISO_8859_7	UTF8
iso_8859_8_to_utf8	ISO_8859_8	UTF8
iso_8859_9_to_utf8	LATIN5	UTF8
johab_to_utf8	JOHAB	UTF8
koi8_r_to_iso_8859_5	KOI8R	ISO_8859_5
koi8_r_to_mic	KOI8R	MULE_INTERNAL
koi8_r_to_utf8	KOI8R	UTF8
koi8_r_to_windows_1251	KOI8R	WIN1251

转换名 [a]	源编码	目标编码
koi8_r_to_windows_866	KOI8R	WIN866
koi8_u_to_utf8	KOI8U	UTF8
mic_to_ascii	MULE_INTERNAL	SQL_ASCII
mic_to_big5	MULE_INTERNAL	BIG5
mic_to_euc_cn	MULE_INTERNAL	EUC_CN
mic_to_euc_jp	MULE_INTERNAL	EUC_JP
mic_to_euc_kr	MULE_INTERNAL	EUC_KR
mic_to_euc_tw	MULE_INTERNAL	EUC_TW
mic_to_iso_8859_1	MULE_INTERNAL	LATIN1
mic_to_iso_8859_2	MULE_INTERNAL	LATIN2
mic_to_iso_8859_3	MULE_INTERNAL	LATIN3
mic_to_iso_8859_4	MULE_INTERNAL	LATIN4
mic_to_iso_8859_5	MULE_INTERNAL	ISO_8859_5
mic_to_koi8_r	MULE_INTERNAL	KOI8R
mic_to_sjis	MULE_INTERNAL	SJIS
mic_to_windows_1250	MULE_INTERNAL	WIN1250
mic_to_windows_1251	MULE_INTERNAL	WIN1251
mic_to_windows_866	MULE_INTERNAL	WIN866
sjis_to_euc_jp	SJIS	EUC_JP
sjis_to_mic	SJIS	MULE_INTERNAL
sjis_to_utf8	SJIS	UTF8
windows_1258_to_utf8	WIN1258	UTF8
uhc_to_utf8	UHC	UTF8
utf8_to_ascii	UTF8	SQL_ASCII
utf8_to_big5	UTF8	BIG5
utf8_to_euc_cn	UTF8	EUC_CN
utf8_to_euc_jp	UTF8	EUC_JP
utf8_to_euc_kr	UTF8	EUC_KR
utf8_to_euc_tw	UTF8	EUC_TW
utf8_to_gb18030	UTF8	GB18030
utf8_to_gbk	UTF8	GBK
utf8_to_iso_8859_1	UTF8	LATIN1
utf8_to_iso_8859_10	UTF8	LATIN6
utf8_to_iso_8859_13	UTF8	LATIN7
utf8_to_iso_8859_14	UTF8	LATIN8
utf8_to_iso_8859_15	UTF8	LATIN9

转换名 [a]	源编码	目标编码
utf8_to_iso_8859_16	UTF8	LATIN10
utf8_to_iso_8859_2	UTF8	LATIN2
utf8_to_iso_8859_3	UTF8	LATIN3
utf8_to_iso_8859_4	UTF8	LATIN4
utf8_to_iso_8859_5	UTF8	ISO_8859_5
utf8_to_iso_8859_6	UTF8	ISO_8859_6
utf8_to_iso_8859_7	UTF8	ISO_8859_7
utf8_to_iso_8859_8	UTF8	ISO_8859_8
utf8_to_iso_8859_9	UTF8	LATIN5
utf8_to_johab	UTF8	JOHAB
utf8_to_koi8_r	UTF8	KOI8R
utf8_to_koi8_u	UTF8	KOI8U
utf8_to_sjis	UTF8	SJIS
utf8_to_windows_1258	UTF8	WIN1258
utf8_to_uhc	UTF8	UHC
utf8_to_windows_1250	UTF8	WIN1250
utf8_to_windows_1251	UTF8	WIN1251
utf8_to_windows_1252	UTF8	WIN1252
utf8_to_windows_1253	UTF8	WIN1253
utf8_to_windows_1254	UTF8	WIN1254
utf8_to_windows_1255	UTF8	WIN1255
utf8_to_windows_1256	UTF8	WIN1256
utf8_to_windows_1257	UTF8	WIN1257
utf8_to_windows_866	UTF8	WIN866
utf8_to_windows_874	UTF8	WIN874
windows_1250_to_iso_8859_2	WIN1250	LATIN2
windows_1250_to_mic	WIN1250	MULE_INTERNAL
windows_1250_to_utf8	WIN1250	UTF8
windows_1251_to_iso_8859_5	WIN1251	ISO_8859_5
windows_1251_to_koi8_r	WIN1251	KOI8R
windows_1251_to_mic	WIN1251	MULE_INTERNAL
windows_1251_to_utf8	WIN1251	UTF8
windows_1251_to_windows_866	WIN1251	WIN866
windows_1252_to_utf8	WIN1252	UTF8
windows_1256_to_utf8	WIN1256	UTF8
windows_866_to_iso_8859_5	WIN866	ISO_8859_5

转换名 [a]	源编码	目标编码
windows_866_to_koi8_r	WIN866	KOI8R
windows_866_to_mic	WIN866	MULE_INTERNAL
windows_866_to_utf8	WIN866	UTF8
windows_866_to_windows_1251	WIN866	WIN
windows_874_to_utf8	WIN874	UTF8
euc_jis_2004_to_utf8	EUC_JIS_2004	UTF8
utf8_to_euc_jis_2004	UTF8	EUC_JIS_2004
shift_jis_2004_to_utf8	SHIFT_JIS_2004	UTF8
utf8_to_shift_jis_2004	UTF8	SHIFT_JIS_2004
euc_jis_2004_to_shift_jis_2004	EUC_JIS_2004	SHIFT_JIS_2004
shift_jis_2004_to_euc_jis_2004	SHIFT_JIS_2004	EUC_JIS_2004

[a] 转换名遵循一种标准命名模式：将全部非字母数字字符替换为下划线的源编码的官方名称，后面跟上_to_，最后是按照相似方式处理过的目标编码名称。因此，名称可能会不同于习惯的编码名称。

6.4.1. format

函数format根据一个格式字符串产生格式化的输出，其形式类似于 C 函数sprintf。

format(*formatstr* text [, *formatarg* "any" [, ...]])

*formatstr*是一个格式字符串，它指定了结果应该如何被格式化。格式字符串中的文本被直接复制到结果中，除了使用格式说明符的地方。格式说明符在字符串中扮演着占位符的角色，它定义后续的函数参数如何被格式化及插入到结果中。每一个*formatarg*参数会被根据其数据类型的常规输出规则转换为文本，并接着根据格式说明符被格式化和插入到结果字符串中。

格式说明符由一个%字符开始并且有这样的形式

%[*position*][*flags*][*width*]*type*

其中的各组件域是：

position（可选）

一个形式为*n*$的字符串，其中*n*是要打印的参数的索引。索引 1 表示*formatstr*之后的第一个参数。如果*position*被忽略，默认会使用序列中的下一个参数。

flags（可选）

控制格式说明符的输出如何被格式化的附加选项。当前唯一支持的标志是一个负号（-），它将导致格式说明符的输出会被左对齐（left-justified）。除非*width*域也被指定，否者这个域不会产生任何效果。

width（可选）

指定用于显示格式说明符输出的最小字符数。输出将被在左部或右部（取决于-标志）用空格填充以保证充满该宽度。太小的宽度设置不会导致输出被截断，但是会被简单地忽略。宽

度可以使用下列形式之一指定：一个正整数；一个星号（*）表示使用下一个函数参数作为宽度；或者一个形式为*n$的字符串表示使用第*n*个函数参数作为宽度。

如果宽度来自于一个函数参数，则参数在被格式说明符的值使用之前就被消耗掉了。如果宽度参数是负值，结果会在长度为abs(*width*)的域中被左对齐（如果-标志被指定）。

type（必需）

格式转换的类型，用于产生格式说明符的输出。支持下面的类型：

- s将参数值格式化为一个简单字符串。一个控制被视为一个空字符串。
- I将参数值视作 SQL 标识符，并在必要时用双写引号包围它。如果参数为空，将会是一个错误（等效于quote_ident）。
- L将参数值引用为 SQL 文字。一个空值将被显示为不带引号的字符串NULL（等效于quote_nullable）。

除了以上所述的格式说明符之外，要输出一个文字形式的%字符，可以使用特殊序列%%。

下面有一些基本的格式转换的例子：

```
SELECT format('Hello %s', 'World');
结果：Hello World

SELECT format('Testing %s, %s, %s, %%', 'one', 'two', 'three');
结果：Testing one, two, three, %

SELECT format('INSERT INTO %I VALUES(%L)', 'Foo bar', E'O\'Reilly');
结果：INSERT INTO "Foo bar" VALUES('O''Reilly')

SELECT format('INSERT INTO %I VALUES(%L)', 'locations', 'C:\Program Files');
结果：INSERT INTO locations VALUES(E'C:\\Program Files')
```

下面是使用*width*域和-标志的例子：

```
SELECT format('|%10s|', 'foo');
结果：|       foo|

SELECT format('|%-10s|', 'foo');
结果：|foo       |

SELECT format('|%*s|', 10, 'foo');
结果：|       foo|

SELECT format('|%*s|', -10, 'foo');
结果：|foo       |

SELECT format('|%-*s|', 10, 'foo');
结果：|foo       |

SELECT format('|%-*s|', -10, 'foo');
结果：|foo       |
```

这些例子展示了*position*域的例子：

SELECT format('Testing %3$s, %2$s, %1$s', 'one', 'two', 'three');
结果：Testing three, two, one

SELECT format('|%*2$s|', 'foo', 10, 'bar');
结果：|　　bar|

SELECT format('|%1$*2$s|', 'foo', 10, 'bar');
结果：|　　foo|

不同于标准的 C 函数sprintf，UXDB的format函数允许将带有或者不带有*position*域的格式说明符被混在同一个格式字符串中。一个不带有*position*域的格式说明符总是使用最后一个被消耗的参数的下一个参数。另外，format函数不要求所有函数参数都被用在格式字符串中。例如：

SELECT format('Testing %3$s, %2$s, %s', 'one', 'two', 'three');
结果：Testing three, two, three

对于安全地构造动态 SQL 语句，%I和%L格式说明符特别有用。

6.4.2. btrim

- 功能

  btrim函数可以去除expr头尾的指定chr。调用时只有expr而不指定chr时，可以去除expr头尾的空格。

- 函数

  btrim(expr[,chr])

- 参数

  表 6.12. btrim 参数说明

参数	说明
expr	一个表中已经存在的列名。
chr	指定字符。

- 返回值

  Text

- 示例

  select btrim(‘ abcda ’) from dual;
  select btrim(‘ abcda ’,’a’) from dual;

- 注意事项

  此函数需要加载oracle插件。

此函数不支持blob和clob等大对象。

6.4.3. substr

- 功能

  substr截取字符串或字符串表达式。

- 函数

  substr(A, B, C);
  substr(A, B);
  substr(A from B for C);
  substr(A from B);
  substr(A for C);
  substr(A for C from B);

- 参数

  表 6.13. substr 参数说明

参数	说明
A	需要被截取的字符串或字符串表达式。
B	从第B个字符开始截取。
C	截取长度为C的字符串信息。

- 返回值

  Text

- 示例

  select substr('tesabcdefg', 2, 5);
  select substr('tesabcdefg', 2);
  select substr('tesabcdefg' from 2 for 5);
  select substr('tesabcdefg' from 2);
  select substr('tesabcdefg' for 5);
  select substr('tesabcdefg' for 5 from 2);

6.5. 二进制串函数和操作符

本节描述那些检查和操作类型为bytea的值的函数和操作符。

SQL定义了一些使用关键字而不是逗号来分割参数的串函数。详情请见表 6.14 “SQL二进制串函数和操作符”。UXDB也提供了这些函数使用常规函数调用语法的版本（参阅表 6.15 “其他二进制串函数”）。

> **注意**
>
> 本页中显示的示例结果假设服务器参数bytea_output被设置为escape（传统UXDB格式）。

表 6.14. SQL二进制串函数和操作符

函数	返回类型	描述	例子	结果
string || *string*	bytea	串连接	'\\Post'::bytea || '\047gres\ \000'::bytea	\\Post'gres\000
octet_length(*string*)	int	二进制串中的字节数	octet_length('jo \000se'::bytea)	5
overlay(*string* placing *string* from int [for int])	bytea	替换子串	overlay('Th \000omas'::bytea placing '\002\003'::bytea from 2 for 3)	T\\002\\003mas
position(*substring* in *string*)	int	指定子串的位置	position('\000om':: bytea in 'Th \000omas'::bytea)	3
substring(*string* [from int] [for int])	bytea	提取子串	substring('Th \000omas'::bytea from 2 for 3)	h\000o
trim([both] *bytes* from *string*)	bytea	从*string*的开头或结尾删除只包含出现在*bytes*中字节的最长串	trim('\000\001':: bytea from '\000Tom\001' ::bytea)	Tom

还有一些二进制串处理函数可以使用，在表 6.15 “其他二进制串函数”列出。 其中有一些是在内部使用，用于实现表 6.14 “SQL二进制串函数和操作符”列出的 SQL 标准串函数。

表 6.15. 其他二进制串函数

函数	返回类型	描述	例子	结果
btrim(*string* bytea, *bytes* bytea)	bytea	从*string*的开头或结尾删除只由出现在*bytes*中字节组成的最长串	btrim('\000trim \001'::bytea, '\000\001'::bytea)	trim
decode(*string* text, *format* text)	bytea	从*string*中的文本表示解码二进制数据。*format*的参数和在encode中一样。	decode('123\000 456', 'escape')	123\000456
encode(*data* bytea, *format* text)	text	将二进制数据编码为一个文本表示。支持的格式有：base64、hex、escape。escape将零字节和高位组字节转换为八进制序列（*nnn*）和双反斜线。	encode('123\000 456'::bytea, 'escape')	123\000456

函数	返回类型	描述	例子	结果
get_bit(*string*, *offset*)	int	从串中抽取位	get_bit('Th \000omas'::bytea, 45)	1
get_byte(*string*, *offset*)	int	从串中抽取字节	get_byte('Th \000omas'::bytea, 4)	109
length(*string*)	int	二进制串的长度	length('jo \000se'::bytea)	5
md5(*string*)	text	计算*string*的MD5哈希码，以十六进制形式返回结果	md5('Th \000omas'::bytea)	8ab2d3c9689aaf18b4958c334c82d8b1
set_bit(*string*, *offset*, *newvalue*)	bytea	设置串中的位	set_bit('Th \000omas'::bytea, 45, 0)	Th\000omAs
set_byte(*string*, *offset*, *newvalue*)	bytea	设置串中的字节	set_byte('Th \000omas'::bytea, 4, 64)	Th\000o@as
sha224(bytea)	bytea	SHA-224哈希	sha224('abc')	\x23097d223405d8228642a477bda255b32aadbce4bda0b3f7e36c9da7
sha256(bytea)	bytea	SHA-256哈希	sha256('abc')	\xba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
sha384(bytea)	bytea	SHA-384哈希	sha384('abc')	\xcb00753f45a35e8bb5a03d699ac6507272c32ab0eded1631a8b605a43ff5bed8086072ba1e7cc2358baeca134c825a7
sha512(bytea)	bytea	SHA-512哈希	sha512('abc')	\xddaf35a193617abacc417349ae204131112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f

get_byte和set_byte把一个二进制串中的一个字节计数为字节 0。get_bit和set_bit在每一个字节中从右边起计数位；例如位 0 是第一个字节的最低有效位，而位 15 是第二个字节的最高有效位。

注意由于历史原因，函数md5返回的是一个十六进制编码的text值，而SHA-2函数返回类型bytea。可以使用函数encode和decode在两者之间转换，例如encode(sha256('abc'), 'hex')可以得到一个十六进制编码的文本表示。

参见第 6.20 节 “聚集函数”中的聚集函数string_agg以及大对象函数。

6.6. 位串函数和操作符

本节描述用于检查和操作位串的函数和操作符，也就是操作类型为bit和bit varying的值的函数和操作符。除了常用的比较操作符之外，还可以使用表 6.16 “位串操作符”里显示的操作符。&、|和#的位串操作数必须等长。在移位的时候，保留原始的位串的的长度，如例子所示。

表 6.16. 位串操作符

操作符	描述	例子	结果
||	连接	B'10001' || B'011'	10001011
&	按位与	B'10001' & B'01101'	00001
|	按位或	B'10001' | B'01101'	11101
#	按位异或	B'10001' # B'01101'	11100
~	按位求反	~ B'10001'	01110
<<	按位左移	B'10001' << 3	01000
>>	按位右移	B'10001' >> 2	00100

下面的SQL标准函数除了可以用于字符串之外，也可以用于位串： length、 bit_length、 octet_length、 position、 substring、 overlay。

下面的函数除了可以用于二进制串之外，也可以用于位串： get_bit、 set_bit。 当使用于一个位串时，这些函数将串的第一（最左）位计数为位 0。

另外，我们可以在整数和bit之间来回转换。一些例子：

```
44::bit(10)                  0000101100
44::bit(3)                   100
cast(-44 as bit(12))         111111010100
'1110'::bit(4)::integer      14
```

请注意，如果只是转换为“bit”，意思是转换成bit(1)，因此只会转换整数的最低有效位。

> **注意**
>
> 把一个整数转换成bit(n)将拷贝整数的最右边的n位。 把一个整数转换成比整数本身长的位串，就会在最左边扩展符号。

6.7. 模式匹配

UXDB提供了三种独立的实现模式匹配的方法：SQL LIKE操作符、更近一些的SIMILAR TO操作符（SQL:1999 里添加进来的）和POSIX-风格的正则表达式。除了这些基本的“这个串匹配这个模式吗？”操作符外，还有一些函数可用于提取或替换匹配子串并在匹配位置分离一个串。

> **提示**
>
> 如果模式匹配的要求超出了这些，请考虑用 Perl 或 Tcl 写一个用户定义的函数。

> **注意**
>
> 虽然大部分的正则表达式搜索都能被很快地执行，但是正则表达式仍可能被人为地弄成需要任意长的时间和任意量的内存进行处理。要当心从不怀好意的来源接受正则表达式搜索模式。如果必须这样做，建议加上语句超时限制。
>
> 使用SIMILAR TO模式的搜索具有同样的安全性危险，因为SIMILAR TO提供了很多和POSIX-风格正则表达式相同的能力。
>
> LIKE搜索比其他两种选项简单得多，因此在使用不怀好意的模式来源时要更安全些。

这三种类型的模式匹配算子都不支持非确定性拼贴。 如果需要的话，可以在表达式中应用不同的拼贴来绕过这个限制。

6.7.1. LIKE

string LIKE *pattern* [ESCAPE *escape-character*]
string NOT LIKE *pattern* [ESCAPE *escape-character*]

如果该*string*匹配了提供的*pattern*，那么LIKE表达式返回真（和预期的一样，如果LIKE返回真，那么NOT LIKE表达式返回假， 反之亦然。一个等效的表达式是NOT (*string* LIKE *pattern*)）。

如果*pattern*不包含百分号或者下划线，那么该模式只代表它本身的串；这时候LIKE的行为就象等号操作符。在*pattern*里的下划线 （_）代表（匹配）任何单个字符； 而一个百分号（%）匹配任何零或更多个字符的序列。

一些例子：

'abc' LIKE 'abc' *true*
'abc' LIKE 'a%' *true*
'abc' LIKE '_b_' *true*
'abc' LIKE 'c' *false*

LIKE模式匹配总是覆盖整个串。因此，要匹配在串内任何位置的序列，该模式必须以百分号开头和结尾。

要匹配文本的下划线或者百分号，而不是匹配其它字符， 在*pattern*里相应的字符必须 前导逃逸字符。缺省的逃逸字符是反斜线，但是可以用ESCAPE子句指定一个不同的逃逸字符。 要匹配逃逸字符本身，写两个逃逸字符。

> **注意**
>
> 如果关掉了standard_conforming_strings，在文串常量中写的任何反斜线都需要被双写。详见第 1.1.2.1 节 “字符串常量”

请注意反斜线在串文本里已经有特殊含义了，所以如果写一个 包含反斜线的模式常量，那就要在 SQL 语句里写两个反斜线。 因此，写一个匹配单个反斜线的模式实际上要在语句里写四个反斜线。 可以通过用 ESCAPE 选择一个不同的逃逸字符 来避免这样；这样反斜线就不再是 LIKE 的特殊字符了。 但仍然是字符文本分析器的特殊字符，所以还是需要两个反斜线。） 我们也可以通过写ESCAPE ''的方式不选择逃逸字符，这样可以有效地禁用逃逸机制，但是没有办法关闭下划线和百分号在模式中的特殊含义。

关键字ILIKE可以用于替换LIKE， 它令该匹配根据活动区域成为大小写无关。这个不属于SQL标准而是一个UXDB扩展。

操作符~~等效于LIKE， 而~~*对应ILIKE。 还有 !~~和!~~*操作符分别代表NOT LIKE和NOT ILIKE。所有这些操作符都是UXDB特有的。

在仅需要从字符串的开始部分搜索的情况，还有前缀操作符^@和相应的starts_with函数可以使用。

6.7.2. SIMILAR TO正则表达式

string SIMILAR TO *pattern* [ESCAPE *escape-character*]
string NOT SIMILAR TO *pattern* [ESCAPE *escape-character*]

SIMILAR TO操作符根据自己的模式是否匹配给定串而返回真或者假。 它和LIKE非常类似，只不过它使用 SQL 标准定义的正则表达式理解模式。 SQL 正则表达式是在LIKE标记和普通的正则表达式标记的奇怪的杂交。

类似LIKE，SIMILAR TO操作符只有在它的模式匹配整个串的时候才能成功；这一点和普通的正则表达式的行为不同，在普通的正则表达式里，模式匹配串的任意部分。 和LIKE类似的地方还有，SIMILAR TO使用_和%作为分别代表任意单个字符和任意串的通配符（这些可以比得上 POSIX 正则表达式里的.和.*）。

除了这些从LIKE借用的功能之外，SIMILAR TO支持下面这些从 POSIX 正则表达式借用的 模式匹配元字符：

- |表示选择（两个候选之一）。
- *表示重复前面的项零次或更多次。
- +表示重复前面的项一次或更多次。
- ?表示重复前面的项零次或一次。
- {*m*}表示重复前面的项刚好*m*次。
- {*m*,}表示重复前面的项*m*次或更多次。
- {*m*,*n*}表示重复前面的项至少*m*次并且不超过*n*次。
- 可以使用圆括号()把多个项组合成一个逻辑项。
- 一个方括号表达式[...]声明一个字符类，就像 POSIX 正则表达式一样。

注意点号（.）不是SIMILAR TO的一个元字符。

和LIKE一样，反斜线禁用所有这些元字符的特殊含义；当然我们也可以用ESCAPE指定一个不同的逃逸字符。

一些例子：

'abc' SIMILAR TO 'abc' *true*
'abc' SIMILAR TO 'a' *false*
'abc' SIMILAR TO '%(b|d)%' *true*
'abc' SIMILAR TO '(b|c)%' *false*

带有三个参数的substring函数可以提取匹配SQL正则表达式模式的子字符串。 该函数可以按照SQL99语法编写。

substring(*string* from *pattern* for *escape-character*)

或作为一个普通的三参数函数:

substring(*string*, *pattern*, *escape-character*)

与SIMILAR TO一样，指定的模式必须与整个数据字符串匹配，否则函数失败并返回空值。 为了表示匹配的数据子字符串的模式中，模式中应该包含两个转义字符的出现，并在后面加上一个双引号(")。 匹配成功后，将返回与这些分隔符之间的模式部分匹配的文本。

转义-双引号分隔符实际上是 将子字符串的模式分成三个独立的 正则表达式；例如，竖条(|) 三节中的任何一节只影响到该节。 此外，第一节和第三种正则表达式的定义是为了匹配最小的 尽可能多的文字，而不是最大的文字，当有歧义的时候，就不应该是最大的文字。关于有多少数据字符串符合哪种模式。 (在POSIX术语中，第一和第三种正则表达式被强行规定为非贪婪)。

作为对SQL标准的扩展，UXDB只允许有一个转义双引号分隔符，在这种情况下，第三个正则表达式被视为空；或者没有分隔符，在这种情况下，第一个和第三个正则表达式被视为空。

一些例子，使用#"定界返回串：

substring('foobar' from '%#"o_b#"%' for '#') *oob*
substring('foobar' from '#"o_b#"%' for '#') *NULL*

6.7.3. POSIX正则表达式

表 6.17 “正则表达式匹配操作符”列出了所有可用于 POSIX 正则表达式模式匹配的操作符。

表 6.17. 正则表达式匹配操作符

操作符	描述	例子
~	匹配正则表达式，大小写敏感	'thomas' ~ '.*thomas.*'
~*	匹配正则表达式，大小写不敏感	'thomas' ~* '.*Thomas.*'
!~	不匹配正则表达式，大小写敏感	'thomas' !~ '.*Thomas.*'
!~*	不匹配正则表达式，大小写不敏感	'thomas' !~* '.*vadim.*'

POSIX正则表达式提供了比LIKE和SIMILAR TO操作符更强大的含义。许多 Unix 工具，例如egrep、sed或awk使用一种与我们这里描述的类似的模式匹配语言。

正则表达式是一个字符序列，它是定义一个串集合 （一个正则集）的缩写。 如果一个串是正则表达式描述的正则集中的一员时， 我们就说这个串匹配该正则表达式。 和LIKE一样，模式字符准确地匹配串字符， 除非在正则表达式语言里有特殊字符 — 不过正则表达式用的 特殊字符和LIKE用的不同。 和LIKE模式不一样的是，正则表达式允许匹配串里的任何位置，除非该正则表达式显式地挂接在串的开头或者结尾。

一些例子：

'abc' ~ 'abc' *true*
'abc' ~ '^a' *true*
'abc' ~ '(b|d)' *true*
'abc' ~ '^(b|c)' *false*

POSIX模式语言的详细描述见下文。

带两个参数的substring函数，即substring(*string* from *pattern*)，提供了抽取一个匹配 POSIX 正则表达式模式的子串的方法。如果没有匹配它返回空值，否则就是文本中匹配模式的那部分。 但是如果该模式包含任何圆括号，那么将返回匹配第一对子表达式（对应第一个左圆括号的） 的文本。如果想在表达式里使用圆括号而又不想导致这个例外，那么可以在整个表达式外边放上一对圆括号。 如果需要在想抽取的子表达式前有圆括号，参阅后文描述的非捕获性圆括号。

一些例子：

substring('foobar' from 'o.b') *oob*
substring('foobar' from 'o(.)b') *o*

regexp_replace函数提供了将匹配 POSIX 正则表达式模式的子串替换为新文本的功能。 它的语法是 regexp_replace(*source*, *pattern*, *replacement* [, *flags*])。 如果没有匹配*pattern*，那么返回不加修改的*source*串。 如果有匹配，则返回的*source*串里面的匹配子串将被*replacement*串替换掉。*replacement*串可以包含*n*， 其中*n*是 1 到 9， 表明源串里匹配模式里第*n*个圆括号子表达式的子串应该被插入， 并且它可以包含\&表示应该插入匹配整个模式的子串。如果需要放一个文字形式的反斜线在替换文本里，那么写\\。*flags*参数是一个可选的文本串，它包含另个或更多单字母标志，这些标志可以改变函数的行为。标志i指定大小写无关的匹配，而标志g指定替换每一个匹配的子串而不仅仅是第一个。支持的标志（但不是g）在表 6.25 “ARE 嵌入选项字母”中描述。

一些例子：

regexp_replace('foobarbaz', 'b..', 'X')
fooXbaz
regexp_replace('foobarbaz', 'b..', 'X', 'g')
fooXX
regexp_replace('foobarbaz', 'b(..)', 'X\1Y', 'g')
fooXarYXazY

regexp_match返回一个文本数组，它包含一个POSIX正则表达式模式与一个字符串第一个匹配所得到的子串。其语法是regexp_match(*string*, *pattern* [, *flags*])。如果没有匹配，则结果为NULL。如果找到一个匹配并且*pattern*不包含带括号的子表达式，那么结果是一个单一元素的文本数组，其中包含匹配整个模式的子串。如果找到一个匹配并且*pattern*含有带括号的子表达式，那么结果是一个文本数组，其中第*n*个元素是与*pattern*的第*n*个圆括号子表达式匹配的子串（“非捕获”圆括号不计入在内，详见下文）。*flags*参数是一个可选的文本字符串，它包含零个

或者更多个可以改变该函数行为的单字母标志。所支持的标志在表 6.25 “ARE 嵌入选项字母”中介绍。

一些例子：

```
SELECT regexp_match('foobarbequebaz', 'bar.*que');
 regexp_match
--------------
 {barbeque}
(1 row)

SELECT regexp_match('foobarbequebaz', '(bar)(beque)');
 regexp_match
--------------
 {bar,beque}
(1 row)
```

在通常情况下，人们只是想要的大整个匹配的子串或者NULL（没有匹配），可以写成这样

```
SELECT (regexp_match('foobarbequebaz', 'bar.*que'))[1];
 regexp_match
--------------
 barbeque
(1 row)
```

regexp_matches函数返回一个文本数组的集合，其中包含着一个POSIX正则表达式模式与一个字符串匹配得到的子串。它和regexp_match具有相同的语法。如果没有匹配，这个函数不会返回行。如果有一个匹配并且给定了g标志，则返回一行。如果有*N*个匹配并且给定了g标志，则返回*N*行。每一个返回的行都是一个文本数组，其中含有整个匹配的子串或者匹配*pattern*的圆括号子表达式的子串，这和上面对regexp_match的介绍一样。regexp_matches接受表 6.25 “ARE 嵌入选项字母”中展示的所有标志，外加令它返回所有匹配而不仅仅是第一个匹配的g标志。

一些例子：

```
SELECT regexp_matches('foo', 'not there');
 regexp_matches
----------------
(0 rows)

SELECT regexp_matches('foobarbequebazilbarfbonk', '(b[^b]+)(b[^b]+)', 'g');
 regexp_matches
----------------
 {bar,beque}
 {bazil,barf}
(2 rows)
```

> **提示**
>
> 在大部分情况下，regexp_matches()应该与g标志一起使用，因为如果只是想要第一个匹配，使用regexp_match()会更加简单高效。

regexp_split_to_table把一个 POSIX 正则表达式模式当作一个定界符来分离一个串。它的语法形式是regexp_split_to_table(*string*, *pattern* [, *flags*])。如果没有与*pattern*的匹配，该函数返回*string*。如果有至少有一个匹配，对每一个匹配它都返回从上一个匹配的末尾（或者串的开头）到这次匹配开头之间的文本。当没有更多匹配时，它返回从上一次匹配的末尾到串末尾之间的文本。*flags*参数是一个可选的文本串，它包含零个或更多单字母标志，这些标识可以改变该函数的行为。regexp_split_to_table能支持的标志在表 6.25 “ARE 嵌入选项字母”中描述。

regexp_split_to_array函数的行为和regexp_split_to_table相同，不过regexp_split_to_array会把它的结果以一个text数组的形式返回。它的语法是regexp_split_to_array(*string*, *pattern* [, *flags*])。这些参数和regexp_split_to_table的相同。

一些例子：

```
SELECT foo FROM regexp_split_to_table('the quick brown fox jumps over the lazy dog', '\s+') AS foo;
  foo
-------
 the
 quick
 brown
 fox
 jumps
 over
 the
 lazy
 dog
(9 rows)

SELECT regexp_split_to_array('the quick brown fox jumps over the lazy dog', '\s+');
             regexp_split_to_array
-----------------------------------------------
 {the,quick,brown,fox,jumps,over,the,lazy,dog}
(1 row)

SELECT foo FROM regexp_split_to_table('the quick brown fox', '\s*') AS foo;
 foo
-----
 t
 h
 e
 q
 u
 i
 c
 k
 b
 r
 o
 w
 n
 f
 o
 x
```

(16 rows)

正如上一个例子所示，正则表达式分离函数会忽略零长度的匹配，这种匹配发生在串的开头或结尾或者正好发生在前一个匹配之后。这和正则表达式匹配的严格定义是相悖的，后者由regexp_match和regexp_matches实现，但是通常前者是实际中最常用的行为。其他软件系统如Perl也使用相似的定义。

6.7.3.1. 正则表达式细节

UXDB的正则表达式是使用 Henry Spencer 写的一个包来实现的。下面的正则表达式的大部分描述都是从他的手册页中逐字拷贝过来的。

正则表达式（RE），在POSIX 1003.2 中定义， 它有两种形式：扩展的RE或者是ERE（大概地说就是那些在egrep里的）， 基本的RE或者是BRE（大概地说就是那些在ed里的）。UXDB支持两种形式，并且还实现了一些POSIX标准中没有但是在类似 Perl 或者 Tcl 这样的语言中得到广泛应用的一些扩展。使用了那些非POSIX扩展的RE叫高级RE， 或者本文档里说的ARE。ARE 几乎完全是 ERE 的超集，但是 BRE 有几个符号上的不兼容（以及更多的限制）。我们首先描述 ARE 和 ERE 形式， 描述那些只适用于 ARE 的特性，然后描述 BRE 的区别是什么。

> **注意**
>
> UXDB初始时总是推测一个正则表达式遵循 ARE 规则。但是，可以通过为 RE 模式预置一个embedded option来选择限制更多的 ERE 或 BRE 规则，如第 6.7.3.4 节 “正则表达式元语法”中所述。这对为期望准确的POSIX 1003.2 规则的应用提供兼容性很有用。

一个正则表达式被定义为一个或更多分支，它们之间被|分隔。只要能匹配其中一个分支的东西都能匹配正则表达式。

一个分支是一个或多个量化原子或者约束连接而成。一个原子匹配第一个，然后后面的原子匹配第二个， 以此类推；一个空分支匹配空串。

一个量化原子是一个原子， 后面可能跟着一个量词。没有量词的时候，它匹配一个原子， 有量词的时候，它可以匹配若干个原子。一个原子可以是在表 6.18 “正则表达式原子”里面显示的任何可能。 可能的量词和它们的含义在表 6.19 “正则表达式量词”里显示。

一个约束匹配一个空串，但只是在满足特定条件下才匹配。 约束可以在能够使用原子的地方使用，只是它不能跟着量词。简单的约束在表 6.20 “正则表达式约束”里显示； 更多的约束稍后描述。

表 6.18. 正则表达式原子

原子	描述
(*re*)	（其中*re*是任何正则表达式） 匹配一个对*re*的匹配，匹配将为可能的报告被记下
(?:*re*)	同上，但是匹配不会为了报告而被记下 （一个“非捕获”圆括号集） （只对 ARE）
.	匹配任意单个字符
[*chars*]	一个方括号表达式， 匹配*chars*中的任意一个（详见第 6.7.3.2 节 “方括号表达式”）
k	（其中*k*是一个非字母数字字符） 匹配一个被当作普通字符看待的特定字符， 例如，\\匹配一个反斜线字符

原子	描述
\c	其中c是一个字母数字（可能跟着其它字符），它是一个逃逸，参阅第 6.7.3.3 节“正则表达式逃逸”（仅对 ARE；在 ERE 和 BRE 中，它匹配c）
{	如果后面跟着一个字符，而不是数字，那么就匹配左花括弧{；如果跟着一个数字，那么它是range的开始（见下文）
x	其中x是一个没有其它意义的单个字符，则匹配该字符

RE 不能以反斜线（\）结尾。

> **注意**
>
> 如果关掉了standard_conforming_strings，任何写在文字串常量中的反斜线都需要被双写。详见第 1.1.2.1 节 “字符串常量”。

表 6.19. 正则表达式量词

量词	匹配
*	一个由原子的 0 次或更多次匹配组成的序列
+	一个由原子的 1 次或更多次匹配组成的序列
?	一个由原子的 0 次或 1 次匹配组成的序列
{m}	一个由原子的正好m次匹配组成的序列
{m,}	一个由原子的m次或更多次匹配组成的序列
{m,n}	一个由原子的从m次到n次（包括）匹配组成的序列；m不能超过n
*?	*的非贪婪版本
+?	+的非贪婪版本
??	?的非贪婪版本
{m}?	{m}的非贪婪版本
{m,}?	{m,}的非贪婪版本
{m,n}?	{m,n}的非贪婪版本

使用{...}的形式被称作范围。一个范围内的数字m和n都是无符号十进制整数，允许的数值从 0 到 255（包含）。

非贪婪的量词（只在 ARE 中可用）匹配对应的正常（贪婪）模式，区别是它寻找最少的匹配，而不是最多的匹配。详见第 6.7.3.5 节 “正则表达式匹配规则”。

> **注意**
>
> 一个量词不能紧跟在另外一个量词后面，例如**是非法的。量词不能作为表达式或者子表达式的开头，也不能跟在^或者|后面。

表 6.20. 正则表达式约束

约束	描述
^	串开头的匹配
$	串末尾的匹配
(?=*re*)	在匹配*re*的子串开始的任何点的positive lookahead匹配（只对 ARE）
(?!*re*)	在匹配*re*的子串开始的任何点的negative lookahead匹配（只对 ARE）
(?<=*re*)	只要有一个点上有一个子串匹配*re*端，positive lookbehind就在这个点上匹配（只对 ARE）
(?<!*re*)	只要有一个点上没有子串匹配*re*端， negative lookbehind就在这个点上匹配（只对 ARE）

lookahead 和 lookbehind 约束不能包含*后引用*（参阅第 6.7.3.3 节 “正则表达式逃逸”），并且其中的所有圆括号 都被认为是非捕获的。

6.7.3.2. 方括号表达式

*方括号表达式*是一个包围在[]中的字符列表。它通常匹配列表中的任意单个字符（但见下文）。如果列表以^开头，它匹配任意单个不在该列表参与部分中的字符。如果该列表中两个字符用-隔开， 那它就是那两个字符（包括在内）之间的所有字符范围的缩写，例如，在ASCII中[0-9]匹配任何十进制数字。两个范围共享一个端点是非法的，例如，a-c-e。范围与字符集关系密切， 可移植的程序应该避免依靠它们。

想在列表中包含文本]，可以让它做列表的首字符（如果使用了^，需要放在其后）。 想在列表中包含文本-，可以让它做列表的首字符或者尾字符，或者一个范围的第二个端点。 想在列表中把文本-当做范围的起点， 把它用[.和.]包围起来，这样它就成为一个排序元素（见下文）。 除了这些字符本身、一些用[的组合（见下段）以及逃逸（只在 ARE 中有效）以外，所有其它特殊字符 在方括号表达式里都失去它们的特殊含义。特别是，在 ERE 和 BRE 规则下\不是特殊的，但在 ARE 里，它是特殊的（引入一个逃逸）。

在一个方括号表达式里，一个排序元素（一个字符、一个被当做一个单一字符排序的多字符序列或者一个表示上面两种情况的排序序列名称） 包含在[.和.]里面的时候表示该排序元素的字符序列。该序列被当做该方括号列表 的一个单一元素。这允许一个包含多字符排序元素的方括号表达式去匹配多于一个字符，例如，如果排序序列包含一个ch排序元素， 那么 RE [[.ch.]]*c匹配chchcc的头五个字符。

> **注意**
>
> UXDB当前不支持多字符排序元素。这些信息描述了将来可能有的行为。

在方括号表达式里，包围在[=和=]里的排序元素是一个*等价类*， 代表等效于那一个的所有排序元素的字符序列，包括它本身（如果没有其它等效排序元素，那么就好象封装定界符是[.和.]）。例如，如果o和^是一个等价类的成员，那么[[=o=]]、[[=^=]]和[o^]都是同义的。一个等价类不能是一个范围的端点。

在方括号表达式里，在[:和:]里面封装的字符类的名字代表属于该类的所有字符的列表。 字符类不能作为范围的端点使用。POSIX标准定义了这些字符类的名称： alnum （字符和数字）， alpha

(字符), blank (空格和制表符tab), cntrl (控制符), digit (数位数), graph (空格除外可打印字符), lower (小写字母), print (包含空格可打印字符), punct (标点符号), space (空白), upper (大写字母), 和 xdigit (十六进制数). 对于7位ASCII字符集中的字符来说，这些标准字符类的行为在不同平台上一般是一致的。一个给定的非ASCII字符是否被认为属于这些类别中的一个，取决于正则表达式函数或运算符使用的collation，或者默认情况下取决于数据库的LC_CTYPE locale设置。非ASCII字符的分类在不同的平台上会有不同的分类，即使是在类似命名的locale中也是如此。（但C locale从不认为任何非ASCII字符属于上述任何一类）。除了这些标准字符类之外，UXDB定义了ascii字符类，它完全包含7位ASCII字符集。

方括号表达式里有两个特例：方括号表达式[[:<:]]和[[:>:]]是约束，分别匹配一个单词开头和结束的空串。 单词定义为一个单词字符序列，前面和后面都没有其它单词字符。单词字符是一个alnum字符（和如上所述POSIX字符类中定义的一样） 或者一个下划线。这是一个扩展，兼容 POSIX 1003.2， 但那里面并没有说明， 而且在准备移植到其他系统里去的软件里一定要注意使用。通常下文描述的约束逃逸更好些（它们并非更标准，但是更容易键入）。

6.7.3.3. 正则表达式逃逸

逃逸是以\开头，后面跟着一个字母数字字符得特殊序列。 逃逸有好几种变体：字符项、类缩写、约束逃逸以及后引用。在 ARE 里， 如果一个\后面跟着一个字母数字，但是并未组成一个合法的逃逸， 那么它是非法的。在 ERE 中没有逃逸：在方括号表达式之外，一个后面跟着字母数字字符的\只是表示该字符是一个普通的字符，而且在一个方括号表达式里，\是一个普通的字符（后者实际上在 ERE 和 ARE 不兼容）。

字符项逃逸用于便于我们在 RE 中声明那些不可打印的或其他习惯的字符。它们显示在表 6.21 “正则表达式字符项逃逸”中。

类缩写逃逸用来提供一些常用的字符类缩写。它们显示在表 6.22 “正则表达式类缩写逃逸”中。

约束逃逸是一个约束，如果满足特定的条件，它匹配该空串。它们显示在表 6.23 “正则表达式约束逃逸”中。

后引用（*n*）匹配数字*n*指定的被前面的圆括号子表达式匹配的同一个串 （参阅表 6.24 “正则表达式后引用”）。例如， ([bc])\1匹配bb或者cc， 但是不匹配bc或者cb。RE 中子表达式必须完全在后引用前面。子表达式以它们的先导圆括号的顺序编号。非捕获圆括号并不定义子表达式。

表 6.21. 正则表达式字符项逃逸

逃逸	描述
\a	警告（响铃）字符，和 C 中一样
\b	退格，和 C 中一样
\B	反斜线（\）的同义词，用来减少双写反斜线
\c*X*	（其中*X*是任意字符）低序5位和*X*相同的字符，它的其他位都是零
\e	排序序列名为ESC的字符，如果无法做到该字符为八进制值 033
\f	换页，和 C 中一样
\n	新行，和 C 中一样
\r	回车，和 C 中一样

逃逸	描述
\t	水平制表符，和 C 中一样
\u*wxyz*	（其中*wxyz*正好是四个十六进制位）十六进制值为0x*wxyz*的字符
\U*stuvwxyz*	（其中*stuvwxyz*正好是八个十六进制位）十六进制值为0x*stuvwxyz*的字符
\v	垂直制表符，和 C 中一样
\x*hhh*	（其中*hhh*是十六进制位的任意序列）十六进制值为0x*hhh*的字符（一个单一字符，不管用了多少个十六进制位）
\0	值为0（空字节）的字符
xy	（其中*xy*正好是两个八进制位，并且不是一个后引用）八进制值为0*xy*的字符
xyz	（其中*xyz*正好是三个八进制位，并且不是一个后引用）八进制值为0*xyz*的字符

十六进制位是0-9、a-f和A-F。八进制位是0-7。

指定 ASCII 范围（0-127）之外的值的数字字符项转义的含义取决于数据库编码。 当编码是 UTF-8 时，转义值等价于 Unicode 代码点，例如 \u1234表示字符U+1234。对于其他多字节编码， 字符项转义通常只是指定该字符的字节值的串接。如果该转义值不对应数据库编码 中的任何合法字符，将不会发生错误，但是它不会匹配任何数据。

字符项逃逸总是被当作普通字符。例如，\135是 ASCII 中的]， 但\135并不终止一个方括号表达式。

表 6.22. 正则表达式类缩写逃逸

逃逸	描述
\d	[[:digit:]]
\s	[[:space:]]
\w	[[:alnum:]_]（注意下划线是被包括的）
\D	[^[:digit:]]
\S	[^[:space:]]
\W	[^[:alnum:]_] （注意下划线是被包括的）

在方括号表达式里，\d、\s和\w会失去它们的外层方括号，而\D、\S和 \W是非法的（也就是说，例如[a-c\d]等效于[a-c[:digit:]]。同样[a-c\D]等效于 [a-c^[:digit:]]的，也是非法的）。

表 6.23. 正则表达式约束逃逸

逃逸	描述
\A	只在串开头匹配（与^的不同请参见第 6.7.3.5 节 “正则表达式匹配规则”）
\m	只在一个词的开头匹配
\M	只在一个词的末尾匹配

逃逸	描述
\y	只在一个词的开头或末尾匹配
\Y	只在一个词的不是开头或末尾的点上匹配
\Z	只在串的末尾匹配（与$的不同请参见第 6.7.3.5 节 “正则表达式匹配规则”）

一个词被定义成在上面[[:<:]]和[[:>:]]中的声明。在方括号表达式里，约束逃逸是非法的。

表 6.24. 正则表达式后引用

逃逸	描述
m	（其中*m*是一个非零位）一个到第*m*个子表达式的后引用
mnn	（其中*m*是一个非零位，并且*nn*是一些更多的位，并且十六进制值*mnn*不超过目前能看到的封闭捕获圆括号的数目）一个到第*mnn*个子表达式的后引用

> **注意**
>
> 在八进制字符项逃逸和后引用之间有一个历史继承的歧义存在，这个歧义是 通过下面的启发式规则解决的，像上面描述地那样。前导零总是表示这是一个八进制逃逸。 而单个非零数字，如果没有跟着任何其它位，那么总是被认为后引用。 一个多位的非零开头的序列也被认为是后引用，只要它出现在合适的子表达式后面 （也就是说，在后引用的合法范围中的数），否则就被认为是一个八进制。

6.7.3.4. 正则表达式元语法

除了上面描述的主要语法之外，还有几种特殊形式和杂项语法。

如果一个 RE 以***:开头，那么剩下的 RE 都被当作 ARE（这在UXDB中通常是无效的，因为 RE 被假定为 ARE，但是如果 ERE 或 BRE 模式通过*flags*参数被指定为一个正则表达式函数时，它确实能产生效果）。如果一个 RE 以***=开头， 那么剩下的 RE 被当作一个文本串，所有的字符都被认为是一个普通字符。

一个 ARE 可以以嵌入选项开头：一个序列(?*xyz*)（这里的*xyz*是一个或多个字母字符）声明影响剩余 RE 的选项。 这些选项覆盖任何前面判断的选项 — 特别地，它们可以覆盖一个正则表达式操作符隐含的大小写敏感的行为，或者覆盖*flags*参数中的正则表达式函数。可用的选项字母在表 6.25 “ARE 嵌入选项字母”中显示。注意这些同样的选项字母也被用在正则表达式函数的*flags*参数中。

表 6.25. ARE 嵌入选项字母

选项	描述
b	RE的剩余部分是一个BRE
c	大小写敏感的匹配（覆盖操作符类型）
e	RE的剩余部分是一个ERE

选项	描述
i	大小写不敏感的匹配（见第 6.7.3.5 节 “正则表达式匹配规则”）（覆盖操作符类型）
m	n的历史原因的同义词
n	新行敏感的匹配（见第 6.7.3.5 节 “正则表达式匹配规则”）
p	部分新行敏感的匹配（见第 6.7.3.5 节 “正则表达式匹配规则”）
q	RE的剩余部分是一个文字（“quoted”）串，全部是普通字符
s	非新行敏感的匹配（默认）
t	紧语法（默认，见下文）
w	逆部分新行敏感（“怪异”）的匹配（见第 6.7.3.5 节 “正则表达式匹配规则”）
x	扩展语法（见下文）

嵌入选项在)终止序列时发生作用。它们只在 ARE 的开始处起作用 （在任何可能存在的***:控制器后面）。

除了通常的（紧）RE 语法（这种情况下所有字符都有效）， 还有一种扩展语法，可以通过声明嵌入的x选项获得。在扩展语法里，RE 中的空白字符被忽略，就像那些在#和其后的新行（或 RE 的末尾）之间的字符一样。这样就允许我们给一个复杂的 RE 分段和注释。不过这个基本规则有三种例外：

- 空白字符或前置了\的#将被保留
- 方括号表达式里的空白或者#将被保留
- 在多字符符号里面不能出现空白和注释，例如(?:

为了这个目的，空白是空格、制表符、新行和任何属于空白字符类的字符。

最后，在 ARE 里，方括号表达式外面，序列(?#*ttt*)（其中*ttt*是任意不包含一个))的文本）是一个注释， 它被完全忽略。同样，这样的东西是不允许出现在多字符符号的字符中间的，例如 (?:。这种注释更像是一种历史产物而不是一种有用的设施，并且它们的使用已经被废弃；请使用扩展语法来替代。

如果声明了一个初始的***=控制器，那么所有这些元语法扩展都不能使用，因为这样表示把用户输入当作一个文字串而不是 RE 对待。

6.7.3.5. 正则表达式匹配规则

在 RE 可以在给定串中匹配多于一个子串的情况下， RE 匹配串中最靠前的那个子串。如果 RE 可以匹配在那个位置开始 的多个子串，要么是取最长的子串，要么是最短的，具体哪种， 取决于 RE 是贪婪的还是非贪婪的。

一个 RE 是否贪婪取决于下面规则：

- 大多数原子以及所有约束，都没有贪婪属性（因为它们毕竟无法匹配个数变化的文本）。
- 在一个 RE 周围加上圆括号并不会改变其贪婪性。

- 带一个固定重复次数量词 （{*m*}或者{*m*}?） 的量化原子和原子自身具有同样的贪婪性（可能是没有）。
- 一个带其他普通的量词（包括{*m*,*n*}中*m*等于*n*的情况）的量化原子是贪婪的（首选最长匹配）。
- 一个带非贪婪量词（包括{*m*,*n*}?中*m*等于 *n*的情况）的量化原子是非贪婪的（首选最短匹配）。
- 一个分支 — 也就是说，一个没有顶级|操作符的 RE — 和它里面的第一个有贪婪属性的量化原子有着同样的贪婪性。
- 一个由|操作符连接起来的两个或者更多分支组成的 RE 总是贪婪的。

上面的规则所描述的贪婪属性不仅仅适用于独立的量化原子， 而且也适用于包含量化原子的分支和整个 RE。这里的意思是， 匹配是按照分支或者整个 RE 作为一个整体匹配最长或者最短的可能子串。 一旦整个匹配的长度确定，那么匹配任意特定子表达式的部分就基于该子表达式的贪婪属性进行判断，在 RE 里面靠前的子表达式的优先级高于靠后的子表达式。

一个相应的例子：

SELECT SUBSTRING('XY1234Z', 'Y*([0-9]{1,3})');
结果：123
SELECT SUBSTRING('XY1234Z', 'Y*?([0-9]{1,3})');
结果：1

在第一个例子里，RE 作为整体是贪婪的，因为Y*是贪婪的。它可以匹配从Y开始的东西，并且它匹配从这个位置开始的最长的串， 也就是，Y123。输出是这里的圆括号包围的部分，或者说是123。在第二个例子里， RE 总体上是一个非贪婪的 RE，因为Y*?是非贪婪的。它可以匹配从Y开始的最短的子串，也就是说Y1。子表达式[0-9]{1,3}是贪婪的，但是它不能修改总体匹配长度的决定； 因此它被迫只匹配1。

简而言之，如果一个 RE 同时包含贪婪和非贪婪的子表达式，那么总的匹配长度要么是尽可能长，要么是尽可能短，这取决于给整个 RE 赋予的属性。给子表达式赋予的属性只影响在这个匹配里，各个子表达式之间相互允许“吃掉”的多少。

量词{1,1}和{1,1}?可以分别用于在一个子表达式 或者整个 RE 上强制贪婪或者非贪婪。当需要整个 RE 具有不同于从其元素中 推导出的贪婪属性时，这很有用。例如，假设我们尝试将一个包含一些数字的 字符串分隔成数字以及在它们之前和之后的部分，我们可能会尝试这样做：

SELECT regexp_matches('abc01234xyz', '(.*)(\d+)(.*)');
Result: {abc0123,4,xyz}

这不会有用：第一个.*是贪婪的，因此它会“吃掉” 尽可能多的字符而留下\d+去匹配在最后一个可能位置上的最 后一个数字。我们可能会通过让它变成非贪婪来修复：

SELECT regexp_matches('abc01234xyz', '(.*?)(\d+)(.*)');
Result: {abc,0,""}

这也不会有用：因为现在 RE 作为整体来说是非贪婪的，因此它会尽快结束 全部的匹配。我们可以通过强制 RE 整体是贪婪的来得到我们想要的：

SELECT regexp_matches('abc01234xyz', '(?:(.*?)(\d+)(.*)){1,1}');
Result: {abc,01234,xyz}

独立于 RE 的组件的贪婪性之外控制 RE 的整体贪婪性为处理变长模式提供了 很大的灵活性。

在决定更长或者更短的匹配时，匹配长度是以字符衡量的，而不是排序元素。一个空串会被认为比什么都不匹配长。例如：bb*匹配abbbc的中间三个字符；(week|wee)(night|knights)匹配weeknights的所有十个字符； 而(.*).*匹配 abc的时候，圆括号包围的子表达式匹配所有三个字符；当(a*)*被拿来匹配bc时，整个 RE 和圆括号 子表达式都匹配一个空串。

如果声明了大小写无关的匹配，那么效果就好像所有大小写区别在字母表中消失了。如果在多个情况中一个字母以一个普通字符的形式出现在方括号表达式外面，那么它实际上被转换成 一个包含大小写的方括号表达式，也就是说，x 变成 [xX]。 如果它出现在一个方括号表达式里面，那么它的所有大小写的同族都被加入 方括号表达式中，也就是说，x变成[xX]。当它出现在一个方括号表达式内时，它的所有大小写副本都被加入到方括号表达式中，例如， [x]会变成[xX]，而[^x]会变成[^xX]。

如果指定了新行敏感的匹配，.和使用^的方括号表达式 将永远不会匹配新行字符（这样，匹配就绝对不会跨越新行，除非 RE 显式地安排了这样的情况）并且^和$除了分别匹配串开头和结尾之外，还将分别匹配新行后面和前面的空串。但是 ARE 逃逸\A和\Z仍然只匹配串的开头和结尾。

如果指定了部分新行敏感的匹配，那么它影响.和方括号表达式， 这个时候和新行敏感的匹配一样，但是不影响^和$。

如果指定了逆新行敏感匹配，那么它影响^和$，其作用和在新行敏感的匹配里一样，但是不影响.和方括号表达式。这个并不是很有用，只是为了满足对称性而提供的。

6.7.3.6. 限制和兼容性

在这个实现里，对 RE 的长度没有特别的限制。但是，那些希望高移植性的程序应该避免使用长度超过 256 字节的 RE，因为 POSIX 兼容 的实现可以拒绝接受这样的 RE。

ARE 实际上和 POSIX ERE 不兼容的唯一的特性是在方括号表达式里\并不失去它特殊的含义。所有其它 ARE 特性都使用在 POSIX ERE 里面是非法或者是未定义、未声明效果的语法；指示器的***就是在 POSIX 的 BRE 和 ERE 之外的语法。

许多 ARE 扩展都是从 Perl 那里借来的（但是有些被做了修改来清理它们），以及一些 Perl 里没有出现的扩展。要注意的不兼容性包括\b、\B、对结尾的新行缺乏特别的处理、对那些被新行敏感匹配的东西附加的补齐方括号表达式、在 lookahead/lookbehind 约束里对圆括号和后引用的限制以及最长/最短 匹配（而不是第一匹配）的语义。

6.7.3.7. 基本正则表达式

BRE 在几个方面和 ERE 不太一样。在 BRE 中，|、+和?都是普通字符并且没有与它们功能等价的东西。范围的定界符是\{和\}， 因为 {和}本身是普通字符。嵌套的子表达式的圆括号是\(和\)，因为(和)自身是普通字符。除非在 RE 开头或者是圆括号子表达式开头，^都是一个普通字符。除非在 RE 结尾或者是圆括号子表达式的结尾，$是一个普通字符。如果*出现在 RE 开头或者是圆括号封装的子表达式开头 （前面可能有^），那么它是个普通字符。最后，可以用单数字的后引用，\<和\>分别是[[:<:]]和[[:>:]]的同义词；在 BRE 中没有其它可用的逃逸。

6.7.3.8. 与XQuery的区别（LIKE_REGEX）

从SQL:2008开始，SQL标准中包含了一个LIKE_REGEX操作符，它根据XQuery正则表达式标准执行模式匹配。 UXDB还没有实现这个操作符，但是可以使用regexp_match()函数获得非常类似的行为，因为XQuery正则表达式非常接近于上面描述的ARE语法。

与现有的基于POSIX的 正则表达式功能和XQuery正则表达式包括。

- 不支持XQuery字符类减法。 这个功能的一个例子是使用下面的例子，只匹配英文辅音。[a-z-[aeiou]]。
- XQuery字符类速记c。 不支持C、i和I。
- 不支持使用p{UnicodeProperty}或反过来的P{UnicodeProperty}的XQuery字符类元素。
- POSIX根据当前的locale来解释字符类，如/w （见表 6.22 “正则表达式类缩写逃逸”)(可以通过在操作符或函数中附加一个COLLATE子句来控制)。 XQuery通过引用Unicode字符属性来指定这些类，因此只有遵循Unicode规则的locale才能获得等效的行为。
- SQL标准（而不是XQuery本身）试图满足更多的需求。“newline的变体”比POSIX的变体。 上面描述的对新行敏感的匹配选项只考虑ASCII NL (n)是新行，但SQL会让我们把CR (r)、CRLF (r/n)(Windows风格的新行)，以及一些Unicode唯一的字符，如LINE SEPARATOR (U+2028)也视为新行。值得注意的是，.和s应该算作一个字符，而不是按照SQL的规定算作两个字符。
- 在表 6.21 “正则表达式字符项逃逸”中描述的字符输入转义中，XQuery只支持n、r和t。
- XQuery不支持[::*name*:]]语法，不支持括号表达式中的字符类。
- XQuery没有 lookahead 或 lookbehind 约束，也没有任何在 表 6.23 “正则表达式约束逃逸”.
- 第 6.7.3.4 节 “正则表达式元语法”中描述的metasyntax形式在XQuery中不存在。
- 由XQuery定义的正则表达式标志字母与POSIX的选项字母相关，但不一样。 表 6.25 “ARE 嵌入选项字母”）。 虽然i和q选项的行为是一样的，但其他的选项却不一样。
  - XQuery的s（允许点匹配换行）和m（允许^和$在换行处匹配）标志提供了与POSIX的n相同的行为。p和w标志，但它们与POSIX的s和m标志的行为不匹配。特别要注意的是，点匹配-newline是POSIX中的默认行为，但不是XQuery。
  - XQuery的x（忽略模式中的空格）标志与POSIX的扩展模式标志明显不同。 POSIX的x标志也允许#在模式中开始注释，并且POSIX不会忽略反斜线后的空格字符。

6.8. 数据类型格式化函数

UXDB格式化函数提供一套强大的工具用于把各种数据类型 （日期/时间、整数、浮点、数字）转换成格式化的字符串以及反过来从格式化的字符串转换成 指定的数据类型。表 6.26 “格式化函数”列出了这些函数。这些函数都遵循一个公共的调用规范： 第一个参数是待格式化的值，而第二个是一个定义输出或输入格式的模板。

表 6.26. 格式化函数

函数	返回类型	描述	例子
to_char(text)	text	将text/varchar参数的文本格式转化为字符串	to_char('test varchar'::varchar);

函数	返回类型	描述	例子
to_char(varchar)	text	将varchar参数的文本格式转化为字符串	to_char('1.1'::text);
to_char(timestamp, text)	text	把时间戳转成字符串	to_char(current_timestamp, 'HH12:MI:SS')
to_char(interval, text)	text	把间隔转成字符串	to_char(interval '15h 2m 12s', 'HH24:MI:SS')
to_char(int, text)	text	把整数转成字符串	to_char(125, '999')
to_char(double precision, text)	text	把实数或双精度转成字符串	to_char(125.8::real, '999D9')
to_char(numeric, text)	text	把数字转成字符串	to_char(-125.8, '999D99S')
to_date(timestamp without time zone)	date	将不带时区的时间戳转化为date	to_date('2022-08-17 12:15:35'::timestamp)
to_date(timestamp with time zone)	date	将带时区的时间戳转化为date	to_date('2022-08-17 12:15:35'::timestamptz)
to_date(text, text)	date	把字符串转成日期	to_date('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	把字符串转成数字	to_number('12,454.8-', '99G999D9S')
to_number(double precision)	numeric	双精度浮点型转化为数字	to_number(123.456::double precision)
to_number(numeric, numeric)	numeric	数字按照指定数字的格式转化为数字	to_number(1210.73::numeric, 9999.99::numeric)
to_number(text, numeric)	numeric	文本按照指定数字的格式转化为数字	to_number(2222.22::text, 999)
to_number(numeric, text)	numeric	数字按照指定文本格式转化为数字	to_number(1210.73::numeric, 9999.99::text)
to_number(expr1, fmt)	numeric	将输入的expr1按照fmt格式X输出，将十六进制字符和数字转化为十进制数字	to_number('1,234','XXXX')
to_timestamp(text, text)	timestamp with time zone	把字符串转成时间戳	to_timestamp('05 Dec 2000', 'DD Mon YYYY')

> **注意**
>
> 还有一个单一参数的to_timestamp函数，请见表 6.33 “日期/时间函数”。

> **提示**
>
> to_timestamp和to_date存在的目的是为了处理无法用简单造型转换的输入格式。对于大部分标准的日期/时间格式，简单地把源字符串造型成所需的

> 数据类型是可以的，并且简单很多。类似地，对于标准的数字表示形式，to_number也是没有必要的。

在一个to_char输出模板串中，一些特定的模式可以被识别并且被替换成基于给定值的被恰当地格式化的数据。任何不属于模板模式的文本都简单地照字面拷贝。同样，在一个输入　　模板串里（对其他函数），模板模式标识由输入数据串提供的值。如果在模板字符串中有不是模板模式的字符，输入数据字符串中的对应字符会被简单地跳过（不管它们是否等于模板字符串字符）。

表　6.27　“用于日期/时间格式化的模板模式”展示了可以用于格式化日期和时间值的模版。

表　6.27.　用于日期/时间格式化的模板模式

模式	描述
HH	一天中的小时　（01-12）
HH12	一天中的小时　（01-12）
HH24	一天中的小时　（00-23）
MI	分钟　（00-59）minute (00-59)
SS	秒（00-59）
MS	毫秒（000-999）
US	微秒（000000-999999）
SSSS	午夜后的秒（0-86399）
AM，am，PM or pm	正午指示器（不带句号）
A.M.，a.m.，P.M. or p.m.	正午指示器（带句号）
Y,YYY	带逗号的年（4 位或者更多位）
YYYY	年（4 位或者更多位）
YYY	年的后三位
YY	年的后两位
Y	年的最后一位
IYYY	ISO 8601 周编号方式的年（4 位或更多位）
IYY	ISO 8601 周编号方式的年的最后 3 位
IY	ISO 8601 周编号方式的年的最后 2 位
I	ISO 8601 周编号方式的年的最后一位
BC，bc，AD或者ad	纪元指示器（不带句号）
B.C.，b.c.，A.D.或者a.d.	纪元指示器（带句号）
MONTH	全大写形式的月名（空格补齐到 9 字符）
Month	全首字母大写形式的月名（空格补齐到　9　字符）
month	全小写形式的月名（空格补齐到 9 字符）
MON	简写的大写形式的月名（英文 3 字符，本地化长度可变）
Mon	简写的首字母大写形式的月名（英文 3 字符，本地化长度可变）

模式	描述
mon	简写的小写形式的月名（英文 3 字符，本地化长度可变）
MM	月编号（01-12）
DAY	全大写形式的日名（空格补齐到 9 字符）
Day	全首字母大写形式的日名（空格补齐到 9 字符）
day	全小写形式的日名（空格补齐到 9 字符）
DY	简写的大写形式的日名（英语 3 字符，本地化长度可变）
Dy	简写的首字母大写形式的日名（英语 3 字符，本地化长度可变）
dy	简写的小写形式的日名（英语 3 字符，本地化长度可变）
DDD	一年中的日（001-366）
IDDD	ISO 8601 周编号方式的年中的日（001-371，年的第 1 日时第一个 ISO 周的周一）
DD	月中的日（01-31）
D	周中的日，周日（1）到周六（7）
ID	周中的 ISO 8601 日，周一（1）到周日（7）
W	月中的周（1-5）（第一周从该月的第一天开始）
WW	年中的周数（1-53）（第一周从该年的第一天开始）
IW	ISO 8601 周编号方式的年中的周数（01 - 53；新的一年的第一个周四在第一周）
CC	世纪（2 位数）（21 世纪开始于 2001-01-01）
J	儒略日（从午夜 UTC 的公元前 4714 年 11 月 24 日开始的整数日数）
Q	季度（to_date和to_timestamp会忽略）
RM	大写形式的罗马计数法的月（I-XII；I 是 一月）
rm	小写形式的罗马计数法的月（i-xii；i 是 一月）
TZ	大写形式的时区缩写（仅在to_char中支持）
tz	小写形式的时区缩写（仅在to_char中支持）
TZH	时区的小时
TZM	时区的分钟
OF	从UTC开始的时区偏移（仅在to_char中支持）

修饰语可以被应用于模板模式来修改它们的行为。例如，FMMonth就是带着FM修饰语的Month模式。表 6.28 “用于日期/时间格式化的模板模式修饰语”展示了可用于日期/时间格式化的修饰语模式。

表 6.28. 用于日期/时间格式化的模板模式修饰语

修饰语	描述	例子
FM prefix	填充模式（抑制前导零和填充的空格）	FMMonth
TH suffix	大写形式的序数后缀	DDTH, e.g., 12TH
th suffix	小写形式的序数后缀	DDth, e.g., 12th
FX prefix	固定的格式化全局选项（见使用须知）	FX Month DD Day
TM prefix	翻译模式（基于lc_time打印本地化的日和月名）	TMMonth
SP suffix	拼写模式（未实现）	DDSP

日期/时间格式化的使用须知：

- FM抑制前导的零或尾随的空白， 否则会把它们增加到输入从而把一个模式的输出变成固定宽度。在UXDB中，FM只修改下一个声明，而在 Oracle 中，FM影响所有随后的声明，并且重复的FM修饰语将触发填充模式开和关。

- TM不包括结尾空白。to_timestamp和to_date会忽略TM修饰语。

- to_timestamp和to_date跳过了输入字符串开头和日期和时间值周围的多个空格，除非使用了FX选项。 例如，to_to_timestamp(' 2000 JUN', 'YYY-MON')和to_timestamp('2000 - JUN', 'YYY-MON')都能工作，但to_timestamp('2000 JUN', 'FXYYYY-MON')返回一个错误，因为to_timestamp只期望一个空格。FX必须指定为模板中的第一个项目。

- to_timestamp和to_date的模板字符串中的分隔符（一个空格或非字母/非数字字符）与输入字符串中的任何一个分隔符相匹配，或者被跳过，除非使用了FX选项。例如，to_to_timestamp('2000JUN', 'YYY///MON')和to_timestamp('2000/JUN', 'YYY/MON')可以工作，但to_timestamp('2000/JUN', 'YYYY/MON')返回一个错误，因为输入字符串中的分隔符数量超过了模板中的分隔符数量。

  如果指定了FX，模板字符串中的分隔符正好与输入字符串中的一个字符匹配。 但要注意的是，输入字符串中的字符不需要与模板字符串中的分隔符相同。例如，to_timestamp('2000/JUN','FXYYYY MON')可以工作，但是to_timestamp('2000/JUN','FXYYYY MON')返回错误，因为模板字符串中的第二个空格会消耗掉输入字符串中的字母J。

- TZH模板模式可以匹配一个有符号的数字。如果没有FX选项，减号可能是模糊的，可能被解释为分隔符。这种模棱两可的问题可以通过以下方式解决。 如果模板字符串中TZH前的分隔符的数量小于输入字符串中减号前的分隔符数量，则减号被解释为TZH的一部分。否则，减号被认为是值之间的分隔符。例如，to_timestamp("2000 -10', 'YYY TZH')与-10匹配，但to_timestamp('2000 -10', 'YYYY TZH')匹配10到TZH。

- 在to_char模板里可以有普通文本，并且它们会被照字面输出。可以把一个子串放到双引号里强迫它被解释成一个文本，即使它里面包含模板模式也如此。例如，在 '"Hello Year "YYYY'中，YYYY将被年份数据代替，但是Year中单独的Y不会。在to_date、to_number以及to_timestamp中，文本和双引号字符串会导致跳过该字符串中所包含的字符数量，例如"XX"会跳过两个输入字符（不管它们是不是XX）。

> **提示**
>
> 在UXDB 2.1.1.2版本之前，可以使用非字母或非数字字符跳过输入字符串中的任意文本。例如，to_timestamp('2000y6m1d', 'yyyyy-MM-DD')以前是有效的。现在，只能使用字母字符来实现这个目的。例如，to_timestamp("2000y6m1d', 'yyyytMMtDDt')和to_timestamp('2000y6m1d', 'yyyyy"y"MM"m"DD"d"')跳过y、m和d。

- 如果想在输出里有双引号，那么必须在它们前面放反斜线，例如 '\"YYYY Month\"'。不然，在双引号字符串外面的反斜线就不是特殊的。在双引号字符串内，反斜线会导致下一个字符被取其字面形式，不管它是什么字符（但是这没有特殊效果，除非下一个字符是一个双引号或者另一个反斜线）。

- 在to_timestamp和to_date中，如果年份格式声明少于四位（如YYY）并且提供的年份少于四位，年份将被调整为最接近于 2020 年，例如95会变成 1995。

- 在to_timestamp和to_date中，在处理超过4位数的年份时，YYYY转换具有限制。必须在YYYY后面使用一些非数字字符或者模板，否则年份总是被解释为 4 位数字。例如（对于 20000 年）：to_date('200001131', 'YYYYMMDD')将会被解释成一个 4 位数字的年份，而不是在年份后使用一个非数字分隔符，像to_date('20000-1131', 'YYYY-MMDD')或to_date('20000Nov31', 'YYYYMonDD')。

- 在to_timestamp和to_date中，CC（世纪）字段会被接受，但是如果有YYY、YYYY或者Y,YYY字段则会忽略它。如果CC与YY或Y一起使用，则结果被计算为指定世纪中的那一年。如果指定了世纪但是没有指定年，则会假定为该世纪的第一年。

- 在to_timestamp和to_date中，工作日名称或编号（DAY、D以及相关的字段类型）会被接受，但会为了计算结果的目的而忽略。季度（Q）字段也是一样。

- 在to_timestamp和to_date中，一个 ISO 8601 周编号的日期（与一个格里高利日期相区别）可以用两种方法之一被指定为to_timestamp和to_date：

  - 年、周编号和工作日：例如to_date('2006-42-4', 'IYYY-IW-ID')返回日期2006-10-19。如果忽略工作日，它被假定为 1（周一）。

  - 年和一年中的日：例如to_date('2006-291', 'IYYY-IDDD')也返回2006-10-19。

  尝试使用一个混合了 ISO 8601 周编号和格里高利日期的域来输入一个日期是无意义的，并且将导致一个错误。在一个 ISO 周编号的年的环境下，一个“月”或“月中的日”的概念没有意义。在一个格里高利年的环境下，ISO 周没有意义。用户应当避免混合格里高利和 ISO 日期声明。

> **注意**
>
> 虽然to_date将会拒绝混合使用格里高利和 ISO 周编号日期的域，to_char却不会，因为YYYY-MM-DD (IYYY-IDDD) 这种输出格式也会有用。但是避免写类似IYYY-MM-DD的东西，那会得到在 起始年附近令人惊讶的结果（详见第 6.9.1 节 EXTRACT, date_part”）。

- 在to_timestamp中，毫秒（MS）和微秒（US）域都被用作小数点后的秒位。例如to_timestamp('12.3', 'SS.MS')不是 3 毫秒， 而是 300，因为该转换把它看做 12 +

0.3 秒。这意味着对于格式SS.MS而言，输入值12.3、12.30和12.300指定了相同数目的毫秒。要得到三毫秒，必须使用 12.003，转换会把它看做 12 + 0.003 = 12.003 秒。

下面是一个更复杂的例子：to_timestamp('15:12:02.020.001230', 'HH24:MI:SS.MS.US')是 15 小时、12 分钟和 2 秒 + 20 毫秒 + 1230微秒 = 2.021230 秒。

- to_char(..., 'ID')的一周中日的编号匹配extract(isodow from ...)函数，但是to_char(..., 'D')不匹配extract(dow from ...)的日编号。

- to_char(interval)格式化HH和HH12为显示在一个 12 小时的时钟上，即零小时和 36 小时输出为12，而HH24会输出完整的小时值，对于间隔它可以超过 23.

表 6.29 “用于数字格式化的模板模式”展示了可以用于格式化数字值的模版模式。

表 6.29. 用于数字格式化的模板模式

模式	描述
9	数位（如果无意义可以被删除）
0	数位（即便没有意义也不会被删除）
. (period)	小数点
, (comma)	分组（千）分隔符
PR	尖括号内的负值
S	带符号的数字（使用区域）
L	货币符号（使用区域）
D	小数点（使用区域）
G	分组分隔符（使用区域）
MI	在指定位置的负号（如果数字 < 0）
PL	在指定位置的正号（如果数字 > 0）
SG	在指定位置的正/负号
RN	罗马数字（输入在 1 和 3999 之间）
TH or th	序数后缀
V	移动指定位数（参阅注解）
EEEE	科学记数的指数

数字格式化的用法须知：

- 0指定一个总是被打印的数位，即便它包含前导/拖尾的零。9也指定一个数位，但是如果它是前导零则会被空格替换，而如果是拖尾零并且指定了填充模式则它会被删除（对于to_number()来说，这两种模式字符等效）。

- 模式字符S、L、D以及G表示当前locale定义的负号、货币符号、小数点以及数字分隔符字符。不管locale是什么，模式字符句号和逗号就表示小数点和数字分隔符。

- 对于to_char()的模式中的一个负号，如果没有明确的规定，将为该负号保留一列，并且它将被锚接到（出现在左边）那个数字。如果S正好出现在某个9的左边，它也将被锚接到那个数字。

- 使用SG、PL或MI格式化的符号并不挂在数字上面；例如，to_char(-12, 'MI9999')生成'- 12'，而to_char(-12, 'S9999')生成 ' -12'。（Oracle里的实现不允许在9前面使用MI，而是要求9在MI前面。）
- TH不会转换小于零的数值，也不会转换小数。
- PL、SG和TH是UXDB扩展。
- 在to_number中，如果没有使用L或TH之类的非数据模板模式，相应数量的输入字符会被跳过，不管它们是否匹配模板模式，除非它们是数据字符（也就是数位、负号、小数点或者逗号）。例如，TH会跳过两个非数据字符。
- 带有to_char的V会把输入值乘上10^*n*，其中*n*是跟在V后面的位数。带有to_number的V以类似的方式做除法。to_char和to_number不支持使用结合小数点的V（例如，不允许99.9V99）。
- EEEE（科学记数法）不能和任何其他格式化模式或修饰语（数字和小数点模式除外）组合在一起使用，并且必须位于格式化字符串的最后（例如9.99EEEE是一个合法的模式）。

某些修饰语可以被应用到任何模板来改变其行为。例如，FM99.99是带有FM修饰语的99.99模式。表 6.30 “用于数字格式化的模板模式修饰语”中展示了用于数字格式化模式修饰语。

表 6.30. 用于数字格式化的模板模式修饰语

修饰语	描述	例子
FM prefix	填充模式（抑制拖尾零和填充的空白）	FM99.99
TH suffix	大写序数后缀	999TH
th suffix	小写序数后缀	999th

表 6.31 “to_char例子”展示了一些使用to_char函数的例子。

表 6.31. **to_char**例子

表达式	结果
to_char(current_timestamp, 'Day, DD HH12:MI:SS')	'Tuesday , 06 05:39:18'
to_char(current_timestamp, 'FMDay, FMDD HH12:MI:SS')	'Tuesday, 6 05:39:18'
to_char(-0.1, '99.99')	' -.10'
to_char(-0.1, 'FM9.99')	'-.1'
to_char(-0.1, 'FM90.99')	'-0.1'
to_char(0.1, '0.9')	' 0.1'
to_char(12, '9990999.9')	' 0012.0'
to_char(12, 'FM9990999.9')	'0012.'
to_char(485, '999')	' 485'
to_char(-485, '999')	'-485'
to_char(485, '9 9 9')	' 4 8 5'

表达式	结果
to_char(1485, '9,999')	' 1,485'
to_char(1485, '9G999')	' 1 485'
to_char(148.5, '999.999')	' 148.500'
to_char(148.5, 'FM999.999')	'148.5'
to_char(148.5, 'FM999.990')	'148.500'
to_char(148.5, '999D999')	' 148,500'
to_char(3148.5, '9G999D999')	' 3 148,500'
to_char(-485, '999S')	'485-'
to_char(-485, '999MI')	'485-'
to_char(485, '999MI')	'485 '
to_char(485, 'FM999MI')	'485'
to_char(485, 'PL999')	'+485'
to_char(485, 'SG999')	'+485'
to_char(-485, 'SG999')	'-485'
to_char(-485, '9SG99')	'4-85'
to_char(-485, '999PR')	'<485>'
to_char(485, 'L999')	'DM 485'
to_char(485, 'RN')	' CDLXXXV'
to_char(485, 'FMRN')	'CDLXXXV'
to_char(5.2, 'FMRN')	'V'
to_char(482, '999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8, '"Pre:"999" Post:" .999')	'Pre: 485 Post: .800'
to_char(12, '99V999')	' 12000'
to_char(12.4, '99V999')	' 12400'
to_char(12.45, '99V9')	' 125'
to_char(0.0004859, '9.99EEEE')	' 4.86e-04'

6.9. 时间/日期函数和操作符

表 6.33 “日期/时间函数”展示了可用于处理日期/时间值的函数，其细节在随后的小节中描述。表 6.32 “日期/时间操作符”演示了基本算术操作符 （+、*等）的行为。 而与格式化相关的函数，可以参考第 6.8 节 “数据类型格式化函数”。应该很熟悉第 5.5 节 “日期/时间类型”中的日期/时间数据类型的背景知识。

所有下文描述的接受time或timestamp输入的函数和操作符实际上都有两种变体： 一种接收time with time zone或timestamp with time zone， 另外一种接受time without time zone或者timestamp without time zone。为了简化，这些变种没有被独立地展示。此外，+和*操作符都是可交换的操作符对（例如，date + integer 和 integer + date）；我们只显示其中一个。

表 6.32. 日期/时间操作符

操作符	例子	结果
+	date '2001-09-28' + integer '7'	date '2001-10-05'
+	date '2001-09-28' + interval '1 hour'	timestamp '2001-09-28 01:00:00'
+	date '2001-09-28' + time '03:00'	timestamp '2001-09-28 03:00:00'
+	interval '1 day' + interval '1 hour'	interval '1 day 01:00:00'
+	timestamp '2001-09-28 01:00' + interval '23 hours'	timestamp '2001-09-29 00:00:00'
+	time '01:00' + interval '3 hours'	time '04:00:00'
-	- interval '23 hours'	interval '-23:00:00'
-	date '2001-10-01' - date '2001-09-28'	integer '3'（days）
-	date '2001-10-01' - integer '7'	date '2001-09-24'
-	date '2001-09-28' - interval '1 hour'	timestamp '2001-09-27 23:00:00'
-	time '05:00' - time '03:00'	interval '02:00:00'
-	time '05:00' - interval '2 hours'	time '03:00:00'
-	timestamp '2001-09-28 23:00' - interval '23 hours'	timestamp '2001-09-28 00:00:00'
-	interval '1 day' - interval '1 hour'	interval '1 day -01:00:00'
-	timestamp '2001-09-29 03:00' - timestamp '2001-09-27 12:00'	interval '1 day 15:00:00'
*	900 * interval '1 second'	interval '00:15:00'
*	21 * interval '1 day'	interval '21 days'
*	double precision '3.5' * interval '1 hour'	interval '03:30:00'
/	interval '1 hour' / double precision '1.5'	interval '00:40:00'

表 6.33. 日期/时间函数

函数	返回类型	描述	例子	结果
age(timestamp, timestamp)	interval	减去参数，生成一个使用年、月（而不是只用日）的“符号化”的结果	age(timestamp '2001-04-10', timestamp '1957-06-13')	43 年 9 月 27 日
age(timestamp)	interval	从current_date（在午夜）减去	age(timestamp '1957-06-13')	43 years 8 mons 3 days
clock_timestamp()	timestamp with time zone	当前日期和时间（在语句执行期间变化）；		

函数	返回类型	描述	例子	结果
		见第 6.9.4 节“当前日期/时间”		
current_date	date	当前日期；见第 6.9.4 节“当前日期/时间”		
current_time	time with time zone	当前时间（一天中的时间）；见第 6.9.4 节“当前日期/时间”		
current_timestamp	timestamp with time zone	当前日期和时间（当前事务开始时）；见第 6.9.4 节“当前日期/时间”		
date_format(date, format)	text	将日期值格式化为特定格式；见第 6.9.6 节“date_format”	date_format('2022-05-16 14:24:09.024727','%W %D %M %Y')	
date_part(text, timestamp)	double precision	获得子域（等价于extract）；见第 6.9.1 节“EXTRACT, date_part”	date_part('hour', timestamp '2001-02-16 20:38:40')	20
date_part(text, interval)	double precision	获得子域（等价于extract）；见第 6.9.1 节“EXTRACT, date_part”	date_part('month', interval '2 years 3 months')	3
date_trunc(text, timestamp)	timestamp	截断到指定精度；见第 6.9.2 节“date_trunc”	date_trunc('hour', timestamp '2001-02-16 20:38:40')	2001-02-16 20:00:00
date_trunc(text, timestamp with time zone, text)	timestamp with time zone	在指定的时区截断到指定的精度；参见第 6.9.2 节“date_trunc”	date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney')	2001-02-16 13:00:00+00
date_trunc(text, interval)	interval	截断到指定精度；见第 6.9.2 节“date_trunc”	date_trunc('hour', interval '2 days 3 hours 40 minutes')	2 days 03:00:00
extract(*field* from timestamp)	double precision	获得子域；见第 6.9.1 节	extract(hour from timestamp	20

函数	返回类型	描述	例子	结果
		“EXTRACT, date_part”	'2001-02-16 20:38:40')	
extract(*field* from interval)	double precision	获得子域；见第 6.9.1 节“EXTRACT, date_part”	extract(month from interval '2 years 3 months')	3
last_day(date)	date	返回指定日期对应月份的最后一天。date支持以下类型：date, timestamp, timestamptz, text, char, varchar, varchar2, nvarchar2, name, bpchar, clob	select last_day (timestamptz '2004-10-19 05:35:25+02");	2004-10-31 11:35:25
isfinite(date)	boolean	测试有限日期（不是+/-无限）	isfinite(date '2001-02-16')	true
isfinite(timestamp)	boolean	测试有限时间戳（不是+/-无限）	isfinite(timestamp '2001-02-16 21:28:30')	true
isfinite(interval)	boolean	测试有限间隔	isfinite(interval '4 hours')	true
justify_days (interval)	interval	调整间隔这样30天时间周期可以表示为月	justify_days (interval '35 days')	1 mon 5 days
justify_hours (interval)	interval	调整间隔这样24小时时间周期可以表示为日	justify_hours (interval '27 hours')	1 day 03:00:00
justify_interval (interval)	interval	使用justify_days和justify_hours调整间隔，使用额外的符号调整	justify_interval (interval '1 mon -1 hour')	29 days 23:00:00
localtime	time	当前时间（一天中的时间）；见第 6.9.4 节“当前日期/时间”		
localtimestamp	timestamp	当前日期和时间（当前事务的开始）；见第 6.9.4 节“当前日期/时间”		

函数	返回类型	描述	例子	结果
make_date(*year* int, *month* int, *day* int)	date	从年、月、日域创建日期	make_date(2013, 7, 15)	2013-07-15
make_interval(*years* int DEFAULT 0, *months* int DEFAULT 0, *weeks* int DEFAULT 0, *days* int DEFAULT 0, *hours* int DEFAULT 0, *mins* int DEFAULT 0, *secs* double precision DEFAULT 0.0)	interval	从年、月、周、日、时、分、秒域创建 interval	make_interval(days => 10)	10 days
make_time(*hour* int, *min* int, *sec* double precision)	time	从时、分、秒域创建时间	make_time(8, 15, 23.5)	08:15:23.5
make_timestamp(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision)	timestamp	从年、月、日、时、分、秒域创建时间戳	make_timestamp(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5
make_timestamptz(*year* int, *month* int, *day* int, *hour* int, *min* int, *sec* double precision, [*timezone* text])	timestamp with time zone	从年、月、日、时、分、秒域创建带时区的时间戳。如果没有指定*timezone*，则使用当前时区。	make_timestamptz(2013, 7, 15, 8, 15, 23.5)	2013-07-15 08:15:23.5+01
months_between(date1 timestamp,date2 timestamp)	numeric	返回date1与date2之间的月份的差值（即用date1 - date2），结果可以为整数也可以为小数	SELECT months_between ('2008-02-29', '2008-03-31')	-1
now()	timestamp with time zone	当前日期和时间（当前事务的开始）；见第 6.9.4 节“当前日期/时间”		
round(date, fmt)	与date相同类型	将date按照公历规则四舍五入到fmt单位。如果没	select round(timestamptz	2004-10-19 12:00:00

函数	返回类型	描述	例子	结果
		有明确fmt则舍去到最近一天	'2004-10-19 05:35:25+02','hh');	
statement_timestamp()	timestamp with time zone	当前日期和时间（当前事务的开始）；见第 6.9.4 节“当前日期/时间”		
systimestamp	timestamptz	获取到系统的当前时间，包含时区信息，精确到微秒		
timeofday()	text	当前日期和时间（像clock_timestamp，但是作为一个text字符串）；见第 6.9.4 节“当前日期/时间”		
transaction_timestamp()	timestamp with time zone	当前日期和时间（当前事务的开始）；见第 6.9.4 节“当前日期/时间”		
to_timestamp (double precision)	timestamp with time zone	把 Unix 时间（从 1970-01-01 00:00:00+00 开始的秒）转换成 timestamp	to_timestamp (1284352323)	2010-09-13 04:32:03+00

除了这些函数以外，还支持 SQL 操作符OVERLAPS：

(*start1*, *end1*) OVERLAPS (*start2*, *end2*)
(*start1*, *length1*) OVERLAPS (*start2*, *length2*)

这个表达式在两个时间域（用它们的端点定义）重叠的时候得到真，当它们不重叠时得到假。端点可以用一对日期、时间或者时间戳来指定；或者是用一个后面跟着一个间隔的日期、时间或时间戳来指定。当一对值被提供时，起点或终点都可以被写在前面，OVERLAPS会自动地把较早的值作为起点。每一个时间段被认为是表示半开的间隔*start* <= *time* < *end*，除非*start*和*end*相等，这种情况下它表示单个时间实例。例如这表示两个只有一个共同端点的时间段不重叠。

SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
    (DATE '2001-10-30', DATE '2002-10-30');
结果：true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
    (DATE '2001-10-30', DATE '2002-10-30');
结果：false

```
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
    (DATE '2001-10-30', DATE '2001-10-31');
```
结果：false
```
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
    (DATE '2001-10-30', DATE '2001-10-31');
```
结果：true

当把一个interval值添加到timestamp with time zone上（或从中减去）时， days 部分会按照指定的天数增加或减少timestamp with time zone的日期。 对于横跨夏令时的变化（当会话的时区被设置为可识别DST的时区时），这意味着interval '1 day'并 不一定等于interval '24 hours'。例如，当会话的时区设置为CST7CDT时，timestamp with time zone '2005-04-02 12:00-07' + interval '1 day' 的结果是timestamp with time zone '2005-04-03 12:00-06'，而将interval '24 hours'增加到相同的初始timestamp with time zone的结果 则是timestamp with time zone '2005-04-03 13:00-06'， 因为CST7CDT时区在2005-04-03 02:00有一个夏令时变更。

注意age返回的月数域可能有歧义，因为不同的月份有不同的天数。 UXDB的方法是当计算部分月数时，采用两个日期中较早的月。例如：age('2004-06-01', '2004-04-30')使用4月份得到1 mon 1 day，而用5月分时会得到1 mon 2 days，因为5月有31天，而4月只有30天。

日期和时间戳的减法也可能会很复杂。执行减法的一种概念上很简单的方法是，使用EXTRACT(EPOCH FROM ...)把每个值都转换成秒数，然后执行减法， 这样会得到两个值之间的秒数。这种方法将会适应每个月中天数、 时区改变和夏令时调整。使用“-”操作符的日期或时间 戳减法会返回值之间的天数（24小时）以及时/分/秒，也会做同样的调整。 age函数会返回年、月、日以及时/分/秒，执行按域的减法，然后对 负值域进行调整。下面的查询展示了这些方法的不同。例子中的结果由 timezone = 'US/Eastern'产生，这使得两个使用的日期之间存在着夏令 时的变化：

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
    EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
```
Result: 10537200
```
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
     EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
     / 60 / 60 / 24;
```
Result: 121.958333333333
```
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
```
Result: 121 days 23:00:00
```
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');
```
Result: 4 mons

6.9.1. EXTRACT，date_part

EXTRACT(*field* FROM *source*)

extract函数从日期/时间值中抽取子域，例如年或者小时等。*source*必须是一个类型timestamp、time或interval的值表达式（类型为date的表达式将被造型为 timestamp，并且因此也可以被同样使用）。*field*是一个标识符或者字符串，它指定从源值中抽取的域。extract函数返回类型为double precision的值。 下列值是有效的域名字：

century

世纪

SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
结果：20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
结果：21

第一个世纪从 0001-01-01 00:00:00 AD 开始， 尽管那时候人们还不知道这是第一个世纪。这个定义适用于所有使用格里高利历法的国家。其中没有 0 世纪，我们直接从公元前 1 世纪到公元 1 世纪。 如果认为这个不合理，那么请把抱怨发给：罗马圣彼得教堂，梵蒂冈，教皇收。

day

对于timestamp值，是（月份）里的日域（1-31）；对于interval值，是日数

SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
结果：16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
结果：40

decade

年份域除以10

SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
结果：200

dow

一周中的日，从周日（0）到周六（6）

SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
结果：5

请注意，extract的一周中的日和to_char(..., 'D')函数不同。

doy

一年的第几天（1 -365/366）

SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
结果：47

epoch

对于timestamp with time zone值， 是自 1970-01-01 00:00:00 UTC 以来的秒数（结果可能是负数）； 对于date and timestamp值，是自本地时间 1970-01-01 00:00:00 以来的描述；对于interval值，它是时间间隔的总秒数。

SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08');

结果：982384720.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
结果：442800

不能用to_timestamp把一个 epoch 值转换回成时间戳：

SELECT to_timestamp(982384720.12);
Result: 2001-02-17 04:38:40.12+00

hour

小时域（0 - 23）

SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
结果：20

isodow

一周中的日，从周一（1）到周日（7）

SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
结果：7

除了周日，这和dow相同。这符合ISO 8601 中一周中的日的编号。

isoyear

日期所落在的ISO 8601 周编号的年（不适用于间隔）

SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
结果：2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
结果：2006

每一个ISO 8601 周编号的年都开始于包含1月4日的那一周的周一，在早的1月或迟的12月中ISO年可能和格里高利年不同。更多信息见week域。

microseconds

秒域，包括小数部分，乘以 1,000,000。请注意它包括全部的秒

SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
结果：28500000

millennium

千年

SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
结果：3

19xx的年份在第二个千年里。第三个千年从 2001 年 1 月 1 日开始。

milliseconds

秒域，包括小数部分，乘以 1000。请注意它包括完整的秒。

SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
结果：28500

minute

分钟域（0 - 59）

SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
结果：38

month

对于timestamp值，它是一年里的月份数（1 - 12）； 对于interval值，它是月的数目，然后对 12 取模（0 - 11）

SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
结果：2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
结果：3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
结果：1

quarter

该天所在的该年的季度（1 - 4）

SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
结果：1

second

秒域，包括小数部分（0 - 59[1]）

SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
结果：40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
结果：28.5

timezone

与 UTC 的时区偏移，以秒记。正数对应 UTC 东边的时区，负数对应 UTC 西边的时区（从技术上来看，UXDB不使用 UTC，因为其中不处理闰秒）。

[1]如果操作系统实现了闰秒，则为60

timezone_hour

时区偏移的小时部分。

timezone_minute

时区偏移的分钟部分。

week

该天在所在的ISO 8601 周编号的年份里是第几周。根据定义， 一年的第一周包含该年的 1 月 4 日并且 ISO 周从星期一开始。换句话说，一年的第一个星期四在第一周。

在 ISO 周编号系统中，早的 1 月的日期可能位于前一年的第五十二或者第五十三周，而迟的 12 月的日期可能位于下一年的第一周。例如， 2005-01-01位于 2004 年的第五十三周，并且2006-01-01位于 2005 年的第五十二周，而2012-12-31位于 2013 年的第一周。我们推荐把isoyear域和week一起使用来得到一致的结果。

SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
结果：7

year

年份域。要记住这里没有0 AD，所以从AD年里抽取BC年应该注意处理。

SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
结果：2001

> **注意**
>
> 当输入值为 +/-Infinity 时，extract对于单调增的域（epoch、julian、year、isoyear、decade、century以及millennium）返回 +/-Infinity。对于其他域返回 NULL。

extract函数主要的用途是做计算性处理。对于用于显示的日期/时间值格式化，参阅第 6.8 节“数据类型格式化函数”。

在传统的Ingres上建模的date_part函数等价于SQL标准函数extract：

date_part('*field*', *source*)

请注意这里的*field*参数必须是一个串值，而不是一个名字。有效的date_part域名 和extract相同。

SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
结果：16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
结果：4

6.9.2. date_trunc

date_trunc函数在概念上和用于数字的trunc函数类似。

date_trunc(*field*, *source* [, *time_zone*])

*source*是类型timestamp或interval的值表达式（类型date和 time的值都分别被自动转换成timestamp, timestamp with time zone,或者interval）。*field*选择对输入值选用什么样的精度进行截断。返回的值是timestamp, timestamp with time zone,类型或者所有小于选定的 精度的域都设置为零（或者一，对于日期和月份）的interval。

*field*的有效值是：

microseconds
milliseconds
second
minute
hour
day
week
month
quarter
year
decade
century
millennium

当输入值的类型为timestamp with time zone时。截断是针对特定时区进行的。 例如，截断为day，产生的值是 是该区域的午夜。 默认情况下，截断是在以下方面进行的 到当前的TimeZone设置，但在当前的 可以提供可选的*time_zone*参数。以指定不同的时区。 可以指定时区名称 第 5.5.3 节 “ 时区”中描述的任何一种方式。

当处理timestamp without time zone 或interval输入时，不能指定时区。 这些总是按表面值来处理。

示例（假设当地时区为 America/New_York）：

SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
结果：2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
结果：2001-01-01 00:00:00

SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00');
Result: 2001-02-16 00:00:00-05

SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00', 'Australia/Sydney');
Result: 2001-02-16 08:00:00-05

SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Result: 3 days 02:00:00

6.9.3. AT TIME ZONE

AT TIME ZONE把时间戳without time zone转换成时间戳with time zone或者反过来，并且把time值转换成不同的时区。表 6.34 “AT TIME ZONE变体”展示了它的变体。

表 6.34. **AT TIME ZONE**变体

表达式	返回类型	描述
timestamp without time zone AT TIME ZONE *zone*	timestamp with time zone	把给定的不带时区的时间戳当作位于指定时区的时间对待
timestamp with time zone AT TIME ZONE *zone*	timestamp without time zone	把给定的带时区的时间戳转换到新的时区，不带时区指定
time with time zone AT TIME ZONE *zone*	time with time zone	把给定的带时区的时间转换到新时区

在这些表达式里，我们需要的时区*zone*可以指定为文本串（例如，'America/Los_Angeles'）或者一个间隔 （例如，INTERVAL '-08:00'）。 在文本情况下，可用的时区名字可以用第 5.5.3 节“ 时区”中描述的任何方式指定。

例子（假设本地时区是America/Los_Angeles）：

SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40-05' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'America/Chicago';
Result: 2001-02-16 05:38:40

第一个例子给缺少时区的值加上了时区，并且显示了使用当前TimeZone设置的值。第二个例子把带有时区值的时间戳移动到指定的时区，并且返回不带时区的值。这允许存储和显示不同于当前TimeZone设置的值。第三个例子把东京时间转换成芝加哥时间。把time值转换成其他时区会使用当前活跃的时区规则，因为没有提供日期。

函数timezone(*zone*, *timestamp*)等效于 SQL 兼容的结构*timestamp* AT TIME ZONE *zone*。

6.9.4. 当前日期/时间

UXDB提供了许多返回当前日期和时间的函数。这些 SQL 标准的函数全部都按照当前事务的开始时刻返回值：

CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(*precision*)
CURRENT_TIMESTAMP(*precision*)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(*precision*)
LOCALTIMESTAMP(*precision*)

CURRENT_TIME和CURRENT_TIMESTAMP传递带有时区的值；LOCALTIME和LOCALTIMESTAMP传递的值不带时区。

CURRENT_TIME、CURRENT_TIMESTAMP、LOCALTIME和 LOCALTIMESTAMP可以有选择地接受一个精度参数， 该精度导致结果的秒域被园整为指定小数位。如果没有精度参数，结果将被给予所能得到的全部精度。

一些例子：

SELECT CURRENT_TIME;
结果：14:39:53.662522-05

SELECT CURRENT_DATE;
结果：2001-12-23

SELECT CURRENT_TIMESTAMP;
结果：2001-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
结果：2001-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
结果：2001-12-23 14:39:53.662522

因为这些函数全部都按照当前事务的开始时刻返回结果，所以它们的值在事务运行的整个期间内都不改变。 我们认为这是一个特性：目的是为了允许一个事务在“当前”时间上有一致的概念， 这样在同一个事务里的多个修改可以保持同样的时间戳。

> **注意**
>
> 许多其它数据库系统可能会更频繁地推进这些值。

UXDB同样也提供了返回当前语句开始时间的函数， 它们会返回函数被调用时的真实当前时间。这些非 SQL 标准的函数列表如下：

transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()

transaction_timestamp()等价于CURRENT_TIMESTAMP，但是其命名清楚地反映了它的返回值。statement_timestamp()返回当前语句的开始时刻（更准确的说是收到 客户端最后一条命令的时间）。statement_timestamp()和transaction_timestamp()在一个事务的第一条命令期间返回值相同，但是在随后的命令中却不一定相同。 clock_timestamp()返回真正的当前时间，因此它的值甚至在同一条 SQL 命令中都会变化。timeofday()是一个有历史原因的UXDB函数。和clock_timestamp()相似，timeofday()也返回真实的当前时间，但是它的结果是一个格式化的text串，而不是timestamp with time zone值。now()是UXDB的一个传统，等效于transaction_timestamp()。

所有日期/时间类型还接受特殊的文字值now，用于指定当前的日期和时间（重申，被解释为当前事务的开始时刻）。 因此，下面三个都返回相同的结果：

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- 对于和 DEFAULT 一起使用是不正确的
```

> **提示**
>
> 在创建表期间指定一个DEFAULT子句时，不会希望使用第三种形式。系统将在分析这个常量的时候把now转换为一个timestamp， 这样需要默认值时就会得到创建表的时间！而前两种形式要到实际使用缺省值的时候才被计算， 因为它们是函数调用。因此它们可以给出每次插入行的时刻。

6.9.5. 延时执行

下面的这些函数可以用于让服务器进程延时执行：

```
ux_sleep(seconds)
ux_sleep_for(interval)
ux_sleep_until(timestamp with time zone)
```

ux_sleep让当前的会话进程休眠*seconds* 秒以后再执行。*seconds*是一个double precision 类型的值，所以可以指定带小数的秒数。ux_sleep_for是针对用 interval指定的较长休眠时间的函数。ux_sleep_until 则可以用来休眠到一个指定的时刻唤醒。例如：

```
SELECT ux_sleep(1.5);
SELECT ux_sleep_for('5 minutes');
SELECT ux_sleep_until('tomorrow 03:00');
```

> **注意**
>
> 有效的休眠时间间隔精度是平台相关的，通常 0.01 秒是通用值。休眠延迟将至少持续指 定的时长， 也有可能由于服务器负荷而比指定的时间长。特别地， ux_sleep_until并不保证能刚好在指定的时刻被唤醒，但它不会 在比指定时刻早的时候醒来。

> **警告**
>
> 请确保在调用ux_sleep或者其变体时，会话没有持有不必要的锁。否则其它会话可能必须等待休眠会话，因而减慢整个系统速度。

6.9.6. **date_format**

- 功能

  date_format将日期值格式化为特定格式，将参数1(日期值)按照参数2(特定格式语法结构)进行格式化。

- 函数

date_format(date,format);

- 参数

表 6.35. date_format参数说明

参数	说明
date	要格式化的有效日期值，如2022-05-16 14:24:09.024727。
format	由预定义的说明符组成的格式字符串，每个说明符前面都有一个百分比字符(%)，如%Y、%a。支持的说明符如下表所示。

表 6.36. 说明符

限定符	含义
%a	三个字符缩写的工作日名称，例如：Mon、Tue、Wed等
%b	三个字符缩写的月份名称，例如：Jan、Feb、Mar等
%c	以数字表示的月份值，例如：1、2、3...12
%D	英文后缀，例如：0th、1st、2nd等的一个月之中的第几天
%d	如果是1个数字（小于10），那么一个月之中的第几天表示为加前导加0，例如：00、01、02...31
%e	无前导0的月份的日子，例如：1、2...31
%f	微秒，范围在000000...999999
%H	24小时格式的小时，前导加0，例如：00、01...23
%h	12小时格式的小时，前导加0，例如：01、02...12
%I	与%h相同
%i	分数为零，例如：00、01...59
%j	一年中的第几天，前导加0，例如：001、002...366
%k	24小时格式的小时，无前导0，例如：0、1、2...23
%l	12小时格式的小时，无前导0，例如：0、1、2...12
%M	月份全名称，例如：January、February...December
%m	前导加0的月份名称，例如：00、01、02...12
%p	AM或PM，取决于其他时间说明符
%r	表示时间，12小时格式hh:mm:ss AM或PM
%S	表示秒，前导加0，例如：00、01...59
%s	与%S相同
%T	表示时间，24小时格式hh:mm:ss
%U	周的第一天是星期日，前导加0的周数，例如：00、01、02...53
%u	周的第一天是星期一，前导加0的周数，例如：00、01、02...53
%V	与%U相同，与%X一起使用
%v	与%u相同，与%x一起使用
%W	工作日的全程，例如：Sunday、Monday...Saturday
%w	以数字表示工作日（0=星期日，1=星期一等）

限定符	含义
%X	周的四位数表示年份，第一天是星期日，常与%V一起使用
%x	周的四位数表示年份，第一天是星期日，常与%v一起使用
%Y	表示年份，四位数，例如：2000、2001等
%y	表示年份，两位数，例如：00、01等
%%	将百分比（%）字符添加到输出

- 返回值

  返回text类型。

- 示例

```
select date_format('2022-05-16 14:24:09.024727','%W %D %M %Y');
select date_format(now(),'%W %D %M %Y');
select DATE_FORMAT(now(),'%a %b %d %f %H %h %I %i %j %M %m %p %S %s %W
 %Y %y %c %D %e %k %l %r %T %U %u %V %v %w %X %x');
select date_format(now(),'%W - %D - %M - %Y test 123');
```

6.9.7. round

- 功能

  将date按照公历规则四舍五入到fmt单位。如果没有明确fmt则舍去到最近一天。

- 函数

  round(date, fmt)

- 参数

表 6.37. round参数说明

参数	说明
date	有效日期值，如2022-05-16 14:24:09.024727。支持以下类型：timestamp (timestamp without time zone), timestamptz (timestamp with time zone), date, timetz (time with time zone), time (time without time zone)
fmt	指定处理格式，支持格式参考表 6.38 “fmt格式”

表 6.38. fmt格式

格式	舍入或截断单位
CC SCC	一个大于四位数年份的前两位
SYYYY	年份（7月1日四舍五入）

格式	舍入或截断单位
YYYY YEAR SYEAR YYY YY Y	
IYYY IYY IY I	ISO 8601标准定义的包含日历周的年份
Q	季度（在本季度第二个月的第16天进行四舍五入）
MONTH MON MM RM	月份 （第16天四舍五入）
WW	按每年的1月1日的作为第一周第一天
IW	与ISO 8601标准定义的日历周的第一天相同的一周中的一天，即星期一
W	按月份1日的第一天作为每周第一天
DDD DD J	天
DAY DY D	一周的开始日期
HH HH12 HH24	小时
MI	分钟

- 示例

```
select round(timestamptz '2004-10-19 05:35:25+02','hh');
round
-----------
2004-10-19  12:00:00

select round('2004-10-19'::timestamp,'Year');
round
-----------
2005-01-01 00:00:00
```

6.10. 枚举支持函数

对于枚举类型(在第 5.7 节 “枚举类型”中描述)， 有一些函数允许更清洁的编码，而不需要为一个枚举类型硬写特定的值。它们被列在表 6.39 “枚举支持函数”中。本例假定一个枚举类型被创建为：

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green', 'blue', 'purple');
```

表 6.39. 枚举支持函数

函数	描述	例子	例子结果
enum_first(anyenum)	返回输入枚举类型的第一个值	enum_first(null::rainbow)	red
enum_last(anyenum)	返回输入枚举类型的最后一个值	enum_last(null::rainbow)	purple
enum_range(anyenum)	将输入枚举类型的所有值作为一个有序的数组返回	enum_range(null::rainbow)	{red,orange,yellow,green,blue,purple}
enum_range(anyenum, anyenum)	以一个数组返回在给定两个枚举值之间的范围。值必须来自相同的枚举类型。 如果第一个参数为空，其结果将从枚举类型的第一个值开始。如果第二参数为空，其结果将以枚举类型的最后一个值结束。	enum_range('orange'::rainbow, 'green'::rainbow)	{orange,yellow,green}
		enum_range(NULL, 'green'::rainbow)	{red,orange,yellow,green}
		enum_range('orange'::rainbow, NULL)	{orange,yellow,green,blue,purple}

请注意，除了双参数形式的enum_range外， 这些函数忽略传递给它们的具体值，它们只关心声明的数据类型。 空值或类型的一个特定值可以通过，并得到相同的结果。这些函数更多地被用于一个表列或函数参数，而不是一个硬写的类型名，如例子中所建议。

6.11. 几何函数和操作符

几何类型point、box、 lseg、line、path、 polygon和circle有一大堆本地支持函数和操作符，如表 6.40 “几何操作符”、表 6.41 “几何函数”和表 6.42 “几何类型转换函数”中所示。

> **注意**
>
> 请注意“same as”操作符（~=），表示point、box、polygon和circle类型的一般相等概念。这些类型中的某些还有一个=操作符，但是=只比较相同的面积。其它的标量比较操作符 （<=等等）也是为这些类型比较面积。

表 6.40. 几何操作符

操作符	描述	例子
+	平移	box '((0,0),(1,1))' + point '(2.0,0)'
-	平移	box '((0,0),(1,1))' - point '(2.0,0)'
*	缩放/旋转	box '((0,0),(1,1))' * point '(2.0,0)'
/	缩放/旋转	box '((0,0),(2,2))' / point '(2.0,0)'
#	相交的点或方框	box '((1,-1),(-1,1))' # box '((1,1),(-2,-2))'
#	路径或多边形中的点数	# path '((1,0),(0,1),(-1,0))'
@-@	长度或周长	@-@ path '((0,0),(1,0))'
@@	中心	@@ circle '((0,0),10)'
##	第二个操作数上最接近第一个操作数的点	point '(0,0)' ## lseg '((2,0),(0,2))'
<->	距离	circle '((0,0),1)' <-> circle '((5,0),1)'
&&	是否重叠？（只要有一个公共点这就为真）	box '((0,0),(1,1))' && box '((0,0),(2,2))'
<<	是否严格地在左侧？	circle '((0,0),1)' << circle '((5,0),1)'
>>	是否严格地在右侧？	circle '((5,0),1)' >> circle '((0,0),1)'
&<	没有延展到右边？	box '((0,0),(1,1))' &< box '((0,0),(2,2))'
&>	没有延展到左边？	box '((0,0),(3,3))' &> box '((0,0),(2,2))'
<<|	严格在下？	box '((0,0),(3,3))' <<| box '((3,4),(5,5))'
|>>	严格在上？	box '((3,4),(5,5))' |>> box '((0,0),(3,3))'
&<|	没有延展到上面？	box '((0,0),(1,1))' &<| box '((0,0),(2,2))'
|&>	没有延展到下面？	box '((0,0),(3,3))' |&> box '((0,0),(2,2))'
<^	在下面（允许相切）？	circle '((0,0),1)' <^ circle '((0,5),1)'
>^	在上面（允许相切）？	circle '((0,5),1)' >^ circle '((0,0),1)'
?#	相交？	lseg '((-1,0),(1,0))' ?# box '((-2,-2),(2,2))'
?-	水平？	?- lseg '((-1,0),(1,0))'
?-	水平对齐？	point '(1,0)' ?- point '(0,0)'
?|	垂直？	?| lseg '((-1,0),(1,0))'
?|	垂直对齐？	point '(0,1)' ?| point '(0,0)'
?-|	相互垂直？	lseg '((0,0),(0,1))' ?-| lseg '((0,0),(1,0))'
?||	平行？	lseg '((-1,0),(1,0))' ?|| lseg '((-1,2),(1,2))'

操作符	描述	例子
@>	包含？	circle '((0,0),2)' @> point '(1,1)'
<@	包含在内或在上？	point '(1,1)' <@ circle '((0,0),2)'
~=	相同？	polygon '((0,0),(1,1))' ~= polygon '((1,1),(0,0))'

> **注意**
>
> 在UXDB之前，包含操作符@>和<@被分别称为~和@。 这些名字仍然可以使用，但是已被废除并且最终将被移除。

表 6.41. 几何函数

函数	返回类型	描述	例子
area(*object*)	double precision	面积	area(box '((0,0),(1,1))')
center(*object*)	point	中心	center(box '((0,0),(1,2))')
diameter(circle)	double precision	圆的直径	diameter(circle '((0,0),2.0)')
height(box)	double precision	方框的垂直尺寸	height(box '((0,0),(1,1))')
isclosed(path)	boolean	一个封闭路径？	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	boolean	一个开放路径？	isopen(path '[(0,0),(1,1),(2,0)]')
length(*object*)	double precision	长度	length(path '((-1,0),(1,0))')
npoints(path)	int	点数	npoints(path '[(0,0),(1,1),(2,0)]')
npoints(polygon)	int	点数	npoints(polygon '((1,1),(0,0))')
pclose(path)	path	将路径转换成封闭的	pclose(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	将路径转换成开放	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	double precision	圆的半径	radius(circle '((0,0),2.0)')
width(box)	double precision	方框的水平尺寸	width(box '((0,0),(1,1))')

表 6.42. 几何类型转换函数

函数	返回类型	描述	例子
box(circle)	box	圆到方框	box(circle '((0,0),2.0)')
box(point)	box	点到空方框	box(point '(0,0)')
box(point, point)	box	点到方框	box(point '(0,0)', point '(1,1)')
box(polygon)	box	多边形到方框	box(polygon '((0,0),(1,1),(2,0))')

函数	返回类型	描述	例子
bound_box(box, box)	box	方框到外包框	bound_box(box '((0,0),(1,1))', box '((3,3),(4,4))')
circle(box)	circle	方框到圆	circle(box '((0,0),(1,1))')
circle(point, double precision)	circle	中心和半径到圆	circle(point '(0,0)', 2.0)
circle(polygon)	circle	多边形到圆	circle(polygon '((0,0),(1,1),(2,0))')
line(point, point)	line	点到线	line(point '(-1,0)', point '(1,0)')
lseg(box)	lseg	方框对角线到线段	lseg(box '((-1,0),(1,0))')
lseg(point, point)	lseg	点到线段	lseg(point '(-1,0)', point '(1,0)')
path(polygon)	path	多边形到路径	path(polygon '((0,0),(1,1),(2,0))')
point(double precision, double precision)	point	构造点	point(23.4, -44.5)
point(box)	point	方框的中心	point(box '((-1,0),(1,0))')
point(circle)	point	圆的中心	point(circle '((0,0),2.0)')
point(lseg)	point	线段的中心	point(lseg '((-1,0),(1,0))')
point(polygon)	point	多边形的中心	point(polygon '((0,0),(1,1),(2,0))')
polygon(box)	polygon	方框到4点多边形	polygon(box '((0,0),(1,1))')
polygon(circle)	polygon	圆到12点多边形	polygon(circle '((0,0),2.0)')
polygon(*npts*, circle)	polygon	点到*npts*点多边形	polygon(12, circle '((0,0),2.0)')
polygon(path)	polygon	路径到多边形	polygon(path '((0,0),(1,1),(2,0))')

我们可以把一个point的两个组成数字当作具有索引 0 和 1 的数组访问。例如，如果t.p是一个point列，那么SELECT p[0] FROM t检索 X 座标而 UPDATE t SET p[1] = ...改变 Y 座标。同样，box或者lseg类型的值可以当作两个point值的数组值看待。

函数area可以用于类型box、circle和path。area函数操作path数据类型的时候， 只有在path的点没有交叉的情况下才可用。例如，path '((0,0),(0,1),(2,1),(2,2),(1,2),(1,0),(0,0))'::PATH是不行的， 而下面的视觉上相同的 path '((0,0),(0,1),(1,1),(1,2),(2,2),(2,1),(1,1),(1,0),(0,0))'::PATH就可以。 如果交叉和不交叉的path概念让疑惑，那么把上面两个path都画在一张图纸上，就明白了。

6.12. 网络地址函数和操作符

表 6.43 “cidr和inet操作符”展示了可以用于cidr和 inet类型的操作符。 操作符<<、<<=、>>、>>=和 &&测试用于子网包含。它们只考虑两个地址的网 络部分（忽略任何主机部分），然后判断其中一个网络部分是等于另外一个或者是 另外一个的子网。

表 6.43. cidr和inet操作符

操作符	描述	例子
<	小于	inet '192.168.1.5' < inet '192.168.1.6'
<=	小于等于	inet '192.168.1.5' <= inet '192.168.1.5'
=	等于	inet '192.168.1.5' = inet '192.168.1.5'
>=	大于等于	inet '192.168.1.5' >= inet '192.168.1.5'
>	大于	inet '192.168.1.5' > inet '192.168.1.4'
<>	不等于	inet '192.168.1.5' <> inet '192.168.1.4'
<<	被包含在内	inet '192.168.1.5' << inet '192.168.1/24'
<<=	被包含在内或等于	inet '192.168.1/24' <<= inet '192.168.1/24'
>>	包含	inet '192.168.1/24' >> inet '192.168.1.5'
>>=	包含或等于	inet '192.168.1/24' >>= inet '192.168.1/24'
&&	包含或者被包含contains or is contained by	inet '192.168.1/24' && inet '192.168.1.80/28'
~	按位 NOT	~ inet '192.168.1.6'
&	按位 AND	inet '192.168.1.6' & inet '0.0.0.255'
|	按位 OR	inet '192.168.1.6' | inet '0.0.0.255'
+	加	inet '192.168.1.6' + 25
-	减	inet '192.168.1.43' - 36
-	减	inet '192.168.1.43' - inet '192.168.1.19'

表 6.44 “cidr和inet函数”展示了所有可以用于cidr和inet类型的函数。函数abbrev、host和text主要是为了提供可选的显示格式用的。

表 6.44. cidr和inet函数

函数	返回类型	描述	例子	结果
abbrev(inet)	text	缩写显示格式文本	abbrev(inet '10.1.0.0/16')	10.1.0.0/16
abbrev(cidr)	text	缩写显示格式文本	abbrev(cidr '10.1.0.0/16')	10.1/16
broadcast(inet)	inet	网络广播地址	broadcast('192.168.1.5/24')	192.168.1.255/24
family(inet)	int	抽取地址族；4为IPv4， 6为 IPv6	family('::1')	6
host(inet)	text	抽取 IP 地址为文本	host('192.168.1.5/24')	192.168.1.5
hostmask(inet)	inet	为网络构造主机掩码	hostmask('192.168.23.20/30')	0.0.0.3
masklen(inet)	int	抽取网络掩码长度	masklen('192.168.1.5/24')	24
netmask(inet)	inet	为网络构造网络掩码	netmask('192.168.1.5/24')	255.255.255.0
network(inet)	cidr	抽取地址的网络部分	network('192.168.1.5/24')	192.168.1.0/24
set_masklen(inet, int)	inet	为inet值设置网络掩码长度	set_masklen('192.168.1.5/24', 16)	192.168.1.5/16
set_masklen(cidr, int)	cidr	为cidr值设置网络掩码长度	set_masklen('192.168.1.0/24'::cidr, 16)	192.168.0.0/16
text(inet)	text	抽取 IP 地址和网络掩码长度为文本	text(inet '192.168.1.5')	192.168.1.5/32
inet_same_family(inet, inet)	boolean	地址是来自于同一个家族吗？	inet_same_family('192.168.1.5/24', '::1')	false
inet_merge(inet, inet)	cidr	包括给定网络的最小网络	inet_merge('192.168.1.5/24', '192.168.2.5/24')	192.168.0.0/22

任何cidr值都能够被隐式或显式地转换为inet值， 因此上述能够操作inet值的函数也同样能够操作cidr值（也有独立的用于inet和cidr的函数，因为它的行为应当和这两种情况不同）。inet值也可以转换为cidr值。完成时，该网络掩码右侧的任何位都将无声地转换为零以获得一个有效的cidr值。另外，还可以使用常规的造型语法将一个文本字符串转换为inet或cidr值：例如，inet(*expression*)或*colname*::cidr。

表 6.45 “macaddr函数”展示了可以用于macaddr类型的函数。 函数trunc(macaddr)返回一个 MAC 地址，该地址的最后三个字节设置为零。这样可以把剩下的前缀与一个制造商相关联。

表 6.45. macaddr函数

函数	返回类型	描述	例子	结果
trunc(macaddr)	macaddr	设置最后3个字节为零	trunc(macaddr '12:34:56:78:90:ab')	12:34:56:00:00:00

macaddr类型还支持标准关系操作符（>、<=等）用于编辑次序，并且按位算术操作符（~、&和|）用于 NOT、AND 和 OR。

表 6.46 “macaddr8函数”中展示了可以用于macaddr8类型的函数。函数trunc(macaddr8)返回一个后五个字节设置为零的MAC地址。这可以被用来为一个制造商关联一个前缀。

表 6.46. macaddr8函数

函数	返回类型	描述	例子	结果
trunc(macaddr8)	macaddr8	设置最后五个字节为零	trunc(macaddr8 '12:34:56:78:90:ab:cd:ef')	12:34:56:00:00:00:00:00
macaddr8_set7bit (macaddr8)	macaddr8	设置第7位为一，也被称为修改版的EUI-64，用于内含在IPv6地址中	macaddr8_set7bit (macaddr8 '00:34:56:ab:cd:ef')	02:34:56:ff:fe:ab:cd:ef

macaddr8类型也支持用于排序的标准关系操作符（>、<=等）以及用于NOT、AND和OR的位运算操作符（~、&和|）。

6.13. 文本搜索函数和操作符

表 6.47 “文本搜索操作符”、 表 6.48 “文本搜索函数”和 表 6.49 “文本搜索调试函数”总结了为全文搜索提供的函数和操作符。UXDB的文本搜索功能的详细解释可参考第 9 章 全文搜索。

表 6.47. 文本搜索操作符

操作符	返回类型	描述	例子	结果
@@	boolean	tsvector匹配tsquery吗？	to_tsvector('fat cats ate rats') @@ to_tsquery('cat & rat')	t
@@@	boolean	@@的已废弃同义词	to_tsvector('fat cats ate rats') @@@ to_tsquery('cat & rat')	t
||	tsvector	连接tsvector	'a:1 b:2'::tsvector || 'c:1 d:2 b:3'::tsvector	'a':1 'b':2,5 'c':3 'd':4
&&	tsquery	将tsquery用 AND 连接起来	'fat | rat'::tsquery && 'cat'::tsquery	('fat' | 'rat') & 'cat'

操作符	返回类型	描述	例子	结果
||	tsquery	将tsquery用 OR 连接起来	'fat | rat'::tsquery || 'cat'::tsquery	('fat' | 'rat') | 'cat'
!!	tsquery	对一个tsquery取反	!! 'cat'::tsquery	!'cat'
<->	tsquery	tsquery后面跟着tsquery	to_tsquery('fat') <-> to_tsquery('rat')	'fat' <-> 'rat'
@>	boolean	tsquery包含另一个？	'cat'::tsquery @> 'cat & rat'::tsquery	f
<@	boolean	tsquery被包含？	'cat'::tsquery <@ 'cat & rat'::tsquery	t

> **注意**
>
> tsquery的包含操作符只考虑两个查询中的词位，而忽略组合操作符。

除了显示在表中的操作符，还定义了tsvector和tsquery类型的普通B-tree比较操作符（=、<等）。它们对于文本搜索不是很有用，但是允许使用。例如，建在这些类型列上的唯一索引。

表 6.48. 文本搜索函数

函数	返回类型	描述	例子	结果
array_to_tsvector (text[])	tsvector	把词位数组转换成tsvector	array_to_tsvector ('{fat,cat,rat}' ::text[])	'cat' 'fat' 'rat'
get_current_ts_ config()	regconfig	获得默认文本搜索配置	get_current_ts_ config()	english
length(tsvector)	integer	tsvector中的词位数	length('fat:2,4 cat:3 rat:5A'::tsvector)	3
numnode(tsquery)	integer	tsquery中词位外加操作符的数目	numnode('(fat & rat) | cat'::tsquery)	5
plainto_tsquery([*config* regconfig ,] *query* text)	tsquery	产生tsquery但忽略标点符号	plainto_tsquery ('english', 'The Fat Rats')	'fat' & 'rat'
phraseto_tsquery([*config* regconfig ,] *query* text)	tsquery	产生忽略标点搜索短语的tsquery	phraseto_tsquery ('english', 'The Fat Rats')	'fat' <-> 'rat'
websearch_to_ tsquery([*config* regconfig ,] *query* text)	tsquery	从一个Web搜索风格的查询产生tsquery	websearch_to_ tsquery('english', '"fat rat" or rat')	'fat' <-> 'rat' | 'rat'
querytree(*query* tsquery)	text	获得一个tsquery的可索引部分	querytree('foo & ! bar'::tsquery)	'foo'

函数	返回类型	描述	例子	结果
setweight(*vector* tsvector, *weight* "char")	tsvector	为*vector*的每一个元素分配权重	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A')	'cat':3A 'fat':2A,4A 'rat':5A
setweight(*vector* tsvector, *weight* "char", *lexemes* text[])	tsvector	为*lexemes*中列出的*vector*的元素分配权重	setweight('fat:2,4 cat:3 rat:5B'::tsvector, 'A', '{cat,rat}')	'cat':3A 'fat':2,4 'rat':5A
strip(tsvector)	tsvector	从tsvector中移除位置和权重	strip('fat:2,4 cat:3 rat:5A'::tsvector)	'cat' 'fat' 'rat'
to_tsquery([*config* regconfig ,] *query* text)	tsquery	规范化词并转换成tsquery	to_tsquery('english', 'The & Fat & Rats')	'fat' & 'rat'
to_tsvector([*config* regconfig ,] *document* text)	tsvector	缩减文档文本成tsvector	to_tsvector('english', 'The Fat Rats')	'fat':2 'rat':3
to_tsvector([*config* regconfig ,] *document* json(b))	tsvector	把该文档中的每个字符串值缩减成一个tsvector，然后将它们按在文档中的顺序串接起来形成一个tsvector	to_tsvector('english', '{"a": "The Fat Rats"}'::json)	'fat':2 'rat':3
json(b)_to_tsvector([*config* regconfig,] *document* json(b), *filter* json(b))	tsvector	把*filter*指定的文档中的每个值缩减为一个tsvector，然后把它们按照文档中的顺序串接起来形成一个tsvector。*filter*是一个jsonb数组，它列举哪些种类的元素需要被包括在结果tsvector中。*filter*的可能值是"string"（包括所有字符串值）、"numeric"（包括所有字符串格式的数字值）、"boolean"（包括所有字符串格式"true"/"false"的布尔	json_to_tsvector ('english', '{"a": "The Fat Rats", "b": 123}'::json, '["string", "numeric"]')	'123':5 'fat':2 'rat':3

函数	返回类型	描述	例子	结果
		值）、"key"（包括所有键）或者"all"（包括上述所有）。这些值可以被组合在一起，例如用来包括所有的字符串和数字值。		
ts_delete(*vector* tsvector, *lexeme* text)	tsvector	从*vector*中移除给定的*lexeme*	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, 'fat')	'cat':3 'rat':5A
ts_delete(*vector* tsvector, *lexemes* text[])	tsvector	从*vector*中移除*lexemes*中词位的任何出现	ts_delete('fat:2,4 cat:3 rat:5A'::tsvector, ARRAY['fat','rat'])	'cat':3
ts_filter(*vector* tsvector, *weights* "char"[])	tsvector	从*vector*中只选择带有给定权重的元素	ts_filter('fat:2,4 cat:3b rat:5A'::tsvector, '{a,b}')	'cat':3B 'rat':5A
ts_headline([*config* regconfig,] *document* text, *query* tsquery [, *options* text])	text	显示一个查询匹配	ts_headline('x y z', 'z'::tsquery)	x y z
ts_headline([*config* regconfig,] *document* json(b), *query* tsquery [, *options* text])	text	显示一个查询匹配	ts_headline('{"a":"x y z"}'::json, 'z'::tsquery)	{"a":"x y z"}
ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])	float4	为查询排名文档	ts_rank(textsearch, query)	0.818
ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer])	float4	使用覆盖密度为查询排名文档	ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', textsearch, query)	2.01317
ts_rewrite(*query* tsquery, *target* tsquery, *substitute* tsquery)	tsquery	在查询内用*substitute*替换*target*	ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'foo|bar'::tsquery)	'b' & ('foo' | 'bar')

函数	返回类型	描述	例子	结果
ts_rewrite(*query* tsquery, *select* text)	tsquery	使用来自一个SELECT的目标和替换者进行替换	SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases')	'b' & ('foo' | 'bar')
tsquery_phrase(*query1* tsquery, *query2* tsquery)	tsquery	制造搜索后面跟着*query2*的*query1*的查询（和<->操作符相同）	tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'))	'fat' <-> 'cat'
tsquery_phrase(*query1* tsquery, *query2* tsquery, *distance* integer)	tsquery	制造查询来搜索在*query1*后面最大距离*distance*上跟着*query2* 的情况	tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10)	'fat' <10> 'cat'
tsvector_to_array(tsvector)	text[]	把tsvector转换为词位数组	tsvector_to_array('fat:2,4 cat:3 rat:5A'::tsvector)	{cat,fat,rat}
tsvector_update_trigger()	trigger	用于自动tsvector列更新的触发器函数	CREATE TRIGGER ... tsvector_update_trigger(tsvcol, 'ux_catalog.swedish', title, body)	
tsvector_update_trigger_column()	trigger	用于自动tsvector列更新的触发器函数	CREATE TRIGGER ... tsvector_update_trigger_column (tsvcol, configcol, title, body)	
unnest(tsvector, OUT *lexeme* text, OUT *positions* smallint[], OUT *weights* text)	setof record	把一个 tsvector扩展成一组行	unnest('fat:2,4 cat:3 rat:5A'::tsvector)	(cat,{3},{D}) ...

> **注意**
>
> 所有接受一个可选的regconfig参数的文本搜索函数在该参数被忽略时，使用由default_text_search_config指定的配置。

表 6.49 “文本搜索调试函数”中的函数被单独列出，因为它们通常不被用于日常的文本搜索操作。 它们有助于开发和调试新的文本搜索配置。

表 6.49. 文本搜索调试函数

函数	返回类型	描述	例子	结果
ts_debug([*config* regconfig,] *document* text, OUT *alias* text, OUT *description* text, OUT *token* text, OUT *dictionaries* regdictionary[], OUT *dictionary* regdictionary, OUT *lexemes* text[])	setof record	测试一个配置	ts_debug('english', 'The Brightest supernovaes')	(asciiword,"Word, all ASCII",The, {english_stem}, english_stem,{}) ...
ts_lexize(*dict* regdictionary, *token* text)	text[]	测试一个字典	ts_lexize('english_stem', 'stars')	{star}
ts_parse (*parser_name* text, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	测试一个解析器	ts_parse('default', 'foo - bar')	(1,foo) ...
ts_parse(*parser_oid* oid, *document* text, OUT *tokid* integer, OUT *token* text)	setof record	测试一个解析器	ts_parse(3722, 'foo - bar')	(1,foo) ...
ts_token_type (*parser_name* text, OUT *tokid* integer, OUT *alias* text, OUT *description* text)	setof record	获得解析器定义的记号类型	ts_token_type ('default')	(1,asciiword,"Word, all ASCII") ...
ts_token_type (*parser_oid* oid, OUT *tokid* integer, OUT *alias* text, OUT *description* text)	setof record	获得解析器定义的记号类型	ts_token_type (3722)	(1,asciiword,"Word, all ASCII") ...
ts_stat(*sqlquery* text, [*weights* text,] OUT *word* text, OUT *ndoc* integer, OUT *nentry* integer)	setof record	获得一个tsvector列的统计	ts_stat('SELECT vector from apod')	(foo,10,15) ...

6.14. XML 函数

本节中描述的函数以及类函数的表达式都在类型xml的值上操作。类型xml的详细信息请参见第 5.13 节 “XML类型”。用于在值和类型xml之间转换的类函数的表达式xmlparse和xmlserialize记录在这里，而不是在本节中。

使用大部分这些函数要求UXDB使用了configure --with-libxml进行编译。

6.14.1. 产生 XML 内容

有一组函数和类函数的表达式可以用来从 SQL 数据产生 XML 内容。它们特别适合于将查询结果格式化成 XML 文档以便于在客户端应用中处理。

6.14.1.1. xmlcomment

xmlcomment(*text*)

函数xmlcomment创建了一个 XML 值，它包含一个使用指定文本作为内容的 XML 注释。该文本不包含“--”或者也不会以一个“-”结尾，这样结果的结构是一个合法的 XML 注释。如果参数为空，结果也为空。

例子：

```
SELECT xmlcomment('hello');

  xmlcomment
--------------
 <!--hello-->
```

6.14.1.2. xmlconcat

xmlconcat(*xml*[, ...])

函数xmlconcat将由单个 XML 值组成的列表串接成一个单独的值，这个值包含一个 XML 内容片断。空值会被忽略，只有当没有参数为非空时结果才为空。

例子：

```
SELECT xmlconcat('<abc/>', '<bar>foo</bar>');

    xmlconcat
----------------------
 <abc/><bar>foo</bar>
```

如果 XML 声明存在，它们会按照下面的方式被组合。如果所有的参数值都有相同的 XML 版本声明，该版本将被用在结果中，否则将不使用版本。如果所有参数值有独立声明值“yes”，那么该值将被用在结果中。如果所有参数值都有一个独立声明值并且至少有一个为“no”，则“no”被用在结果中。否则结果中将没有独立声明。如果结果被决定要要求一个独立声明但是没有版本声明，将会使用一个版本 1.0 的版本声明，因为 XML 要求一个 XML 声明要包含一个版本声明。编码声明会被忽略并且在所有情况中都会被移除。

例子：

```
SELECT xmlconcat('<?xml version="1.1"?><foo/>', '<?xml version="1.1" standalone="no"?><bar/>');

            xmlconcat
-----------------------------------
 <?xml version="1.1"?><foo/><bar/>
```

6.14.1.3. xmlelement

xmlelement(name *name* [, xmlattributes(*value* [AS *attname*] [, ...])] [, *content, ...*])

表达式xmlelement使用给定名称、属性和内容产生一个 XML 元素。

例子：

```
SELECT xmlelement(name foo);

 xmlelement
------------
 <foo/>

SELECT xmlelement(name foo, xmlattributes('xyz' as bar));

   xmlelement
------------------
 <foo bar="xyz"/>

SELECT xmlelement(name foo, xmlattributes(current_date as bar), 'cont', 'ent');

            xmlelement
---------------------------------------
 <foo bar="2007-01-26">content</foo>
```

不是合法 XML 名字的元素名和属性名将被逃逸，逃逸的方法是将违反的字符用序列_x*HHHH*_替换，其中*HHHH*是被替换字符的 Unicode 代码点的十六进制表示。例如：

```
SELECT xmlelement(name "foo$bar", xmlattributes('xyz' as "a&b"));

          xmlelement
-----------------------------------
 <foo_x0024_bar a_x0026_b="xyz"/>
```

如果属性值是一个列引用，则不需要指定一个显式的属性名，在这种情况下列的名字将被默认用于属性的名字。在其他情况下，属性必须被给定一个显式名称。因此这个例子是合法的：

```
CREATE TABLE test (a xml, b xml);
SELECT xmlelement(name test, xmlattributes(a, b)) FROM test;
```

但是下面这些不合法：

```
SELECT xmlelement(name test, xmlattributes('constant'), a, b) FROM test;
SELECT xmlelement(name test, xmlattributes(func(a, b))) FROM test;
```

如果指定了元素内容，它们将被根据其数据类型格式化。如果内容本身也是类型xml，就可以构建复杂的 XML 文档。例如：

```
SELECT xmlelement(name foo, xmlattributes('xyz' as bar),
                    xmlelement(name abc),
                    xmlcomment('test'),
                    xmlelement(name xyz));

              xmlelement
----------------------------------------------
 <foo bar="xyz"><abc/><!--test--><xyz/></foo>
```

其他类型的内容将被格式化为合法的 XML 字符数据。这意味着字符 <, >, 和 & 将被转换为实体。二进制数据（数据类型bytea）将被表示成base64 或十六进制编码，具体取决于配置参数xmlbinary的设置。为了使UXDB的映射与SQL:2006及以后的SQL:2006中指定的映射保持一致，个别数据类型的特殊行为将不断发展。

6.14.1.4. xmlforest

xmlforest(*content* [AS *name*] [, ...])

表达式xmlforest使用给定名称和内容产生一个元素的 XML 森林（序列）。

例子：

```
SELECT xmlforest('abc' AS foo, 123 AS bar);

        xmlforest
------------------------------
 <foo>abc</foo><bar>123</bar>


SELECT xmlforest(table_name, column_name)
FROM information_schema.columns
WHERE table_schema = 'ux_catalog';

                                xmlforest
-------------------------------------------------------------------------------------------
 <table_name>ux_authid</table_name><column_name>rolname</column_name>
 <table_name>ux_authid</table_name><column_name>rolsuper</column_name>
 ...
```

如我们在第二个例子中所见，如果内容值是一个列引用，元素名称可以被忽略，这种情况下默认使用列名。否则，必须指定一个名字。

如上文xmlelement所示，非法 XML 名字的元素名会被逃逸。相似地，内容数据也会被逃逸来产生合法的 XML 内容，除非它已经是一个xml类型。

注意如果 XML 森林由多于一个元素组成，那么它不是合法的 XML 文档，因此在xmlelement中包装xmlforest表达式会有用处。

6.14.1.5. xmlpi

xmlpi(name *target* [, *content*])

表达式xmlpi创建一个 XML 处理指令。如果存在内容，内容不能包含字符序列?>。

例子：

```
SELECT xmlpi(name php, 'echo "hello world";');

            xmlpi
-----------------------------
 <?php echo "hello world";?>
```

6.14.1.6. xmlroot

xmlroot(*xml*, version *text* | no value [, standalone yes|no|no value])

表达式xmlroot修改一个 XML 值的根结点的属性。如果指定了一个版本，它会替换根节点的版本声明中的值；如果指定了一个独立设置，它会替换根节点的独立声明中的值。

```
SELECT xmlroot(xmlparse(document '<?xml version="1.1"?><content>abc</content>'),
               version '1.0', standalone yes);

                xmlroot
----------------------------------------
 <?xml version="1.0" standalone="yes"?>
 <content>abc</content>
```

6.14.1.7. xmlagg

xmlagg(*xml*)

和这里描述的其他函数不同，函数xmlagg是一个聚集函数。它将聚集函数调用的输入值串接起来，非常像xmlconcat所做的事情，除了串接是跨行发生的而不是在单一行的多个表达式上发生。聚集表达式的更多信息请见第 6.20 节 “聚集函数”。

例子：

```
CREATE TABLE test (y int, x xml);
INSERT INTO test VALUES (1, '<foo>abc</foo>');
INSERT INTO test VALUES (2, '<bar/>');
SELECT xmlagg(x) FROM test;
      xmlagg
----------------------
 <foo>abc</foo><bar/>
```

为了决定串接的顺序，可以为聚集调用增加一个ORDER BY子句，如第 1.2.7 节 “聚集表达式”中所述。例如：

```
SELECT xmlagg(x ORDER BY y DESC) FROM test;
```

```
      xmlagg
----------------------
 <bar/><foo>abc</foo>
```

我们推荐在以前的版本中使用下列非标准方法，并且它们在特定情况下仍然有用：

```
SELECT xmlagg(x) FROM (SELECT * FROM test ORDER BY y DESC) AS tab;
      xmlagg
----------------------
 <bar/><foo>abc</foo>
```

6.14.2. XML 谓词

这一节描述的表达式检查xml值的属性。

6.14.2.1. IS DOCUMENT

xml IS DOCUMENT

如果参数 XML 值是一个正确的 XML 文档，则IS DOCUMENT返回真，如果不是则返回假（即它是一个内容片断），或者是参数为空时返回空。文档和内容片断之间的区别请见第 5.13 节“XML类型”。

6.14.2.2. IS NOT DOCUMENT

xml IS NOT DOCUMENT

如果参数中的XML值是一个正确的XML文档，那么表达式IS NOT DOCUMENT返回假，否则返回真（也就是说它是一个内容片段），如果参数为空则返回空。

6.14.2.3. XMLEXISTS

XMLEXISTS(*text* PASSING [BY { REF | VALUE }] *xml* [BY { REF | VALUE }])

函数xmlexists评价一个XPath 1.0表达式(第一个参数)，以传递的XML值作为其上下文项。 如果评价的结果产生一个空节点集，该函数返回false，如果产生任何其他值，则返回true。 如果任何参数为空，则函数返回null。 作为上下文项传递的非空值必须是一个XML文档，而不是内容片段或任何非XML值。

例子：

```
SELECT xmlexists('//town[text() = "Toronto"]' PASSING BY VALUE '<towns><town>Toronto</
town><town>Ottawa</town></towns>');

 xmlexists
------------
 t
(1 row)
```

BY REF和BY VALUE子句在UXDB中被接受。在SQL标准中，xmlexists函数评估XML查询语言中的表达式，但UXDB只允许使用XPath 1.0表达式。

6.14.2.4. xml_is_well_formed

xml_is_well_formed(*text*)
xml_is_well_formed_document(*text*)
xml_is_well_formed_content(*text*)

这些函数检查一个text串是不是一个良构的 XML，返回一个布尔结果。xml_is_well_formed_document检查一个良构的文档，而xml_is_well_formed_content检查良构的内容。如果xmloption配置参数被设置为DOCUMENT，xml_is_well_formed会做第一个函数的工作；如果配置参数被设置为CONTENT，xml_is_well_formed会做第二个函数的工作。这意味着xml_is_well_formed对于检查一个到类型xml的简单造型是否会成功非常有用，而其他两个函数对于检查XMLPARSE的对应变体是否会成功有用。

例子：

```
SET xmloption TO DOCUMENT;
SELECT xml_is_well_formed('<>');
 xml_is_well_formed
--------------------
 f
(1 row)

SELECT xml_is_well_formed('<abc/>');
 xml_is_well_formed
--------------------
 t
(1 row)

SET xmloption TO CONTENT;
SELECT xml_is_well_formed('abc');
 xml_is_well_formed
--------------------
 t
(1 row)
```

最后一个例子显示了这些检查也包括名字空间是否正确地匹配。

6.14.3. 处理 XML

要处理数据类型xml的值， UXDB 提供了函数xpath和xpath_exists，它们计算 XPath 1.0 表达式以及XMLTABLE表函数。

6.14.3.1. xpath

xpath(*xpath*, *xml* [, *nsarray*])

函数xpath在 XML 值 *xml*上计算 XPath 1.0 表达式*xpath* (a text value)。它返回一个 XML 值的数组，该数组对应于该 XPath 表达式产生的结点集合。如果该 XPath 表达式返回一个标量值而不是一个结点集合，将会返回一个单一元素的数组。

第二个参数必须是一个良构的 XML 文档。特殊地，它必须有一个单一根结点元素。

该函数可选的第三个参数是一个名字空间映射的数组。这个数组应该是一个二维text数组，其第二轴长度等于2（即它应该是一个数组的数组，其中每一个都由刚好 2 个元素组成）。每个数组项的第一个元素是名字空间的名称（别名），第二个元素是名字空间的 URI。并不要求在这个数组中提供的别名和在 XML 文档本身中使用的那些名字空间相同（换句话说，在 XML 文档中和在xpath函数环境中，别名都是本地的）。

例子：

```
SELECT xpath('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
        ARRAY[ARRAY['my', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

要处理默认（匿名）命名空间，做这样的事情：

```
SELECT xpath('//mydefns:b/text()', '<a xmlns="http://example.com"><b>test</b></a>',
        ARRAY[ARRAY['mydefns', 'http://example.com']]);

 xpath
--------
 {test}
(1 row)
```

6.14.3.2. xpath_exists

xpath_exists(*xpath*, *xml* [, *nsarray*])

函数xpath_exists是xpath函数的一种特殊形式。这个函数不是返回满足 XPath 1.0 表达式的单一 XML 值，它返回一个布尔值表示查询是否被满足(具体来说，它是否产生了空节点集以外的任何值)。这个函数等价于标准的XMLEXISTS谓词，不过它还提供了对一个名字空间映射参数的支持。

例子：

```
SELECT xpath_exists('/my:a/text()', '<my:a xmlns:my="http://example.com">test</my:a>',
            ARRAY[ARRAY['my', 'http://example.com']]);

 xpath_exists
--------------
 t
(1 row)
```

6.14.3.3. xmltable

```
xmltable( [XMLNAMESPACES(namespace uri AS namespace name[, ...]), ]
        row_expression PASSING [BY { REF | VALUE }] document_expression [BY { REF | VALUE }]
        COLUMNS name { type [PATH column_expression] [DEFAULT default_expression] [NOT
 NULL | NULL]
```

```
            | FOR ORDINALITY }
        [, ...]
)
```

xmltable函数基于给定的XML值产生一个表、一个抽取行的XPath过滤器以及一个列定义集合。

可选的XMLNAMESPACES子句是一个逗号分隔的名字空间列表。它指定文档中使用的XML名字空间极其别名。当前不支持默认的名字空间说明。

所需的*row_expression*参数是一个XPath 1.0表达式，通过传递*document_expression*作为其上下文项，得到一组XML节点。这些节点就是xmltable转换为输出行的内容。如果*document_expression*为空，或者*row_expression*产生空节点集或节点集以外的任何值，则不会产生行。

*document_expression*提供了上下文。*row_expression*的项。它必须是一个格式良好的XML文档；不接受片段/森林。BY REF和BY VALUE子句 如上文所讨论的那样，被接受但被忽略了。在SQL标准中，xmltable函数 评估XML查询语言中的表达式。但UXDB只允许使用XPath 1.0的 表达式。

强制需要的COLUMNS子句指定输出表中的列列表。如果COLUMNS子句被省略，每一项描述一个列。格式请见上面的语法综述。列名和类型是必需的，路径、默认值以及为空性子句是可选的。

被标记为FOR ORDINALITY的列将按照从*row_expression*的结果节点集中检索到的节点的顺序，从1开始，填充行号。最多只能有一个列被标记为FOR ORDINALITY。

> **注意**
>
> XPath 1.0 并没有为节点集中的节点指定顺序，因此依赖特定结果顺序的代码将取决于实现。

列的*column_expression*是一个XPath 1.0表达式，它对每一行都要进行求值，并以*row_expression*结果中的当前节点作为其上下文项，以找到列的值。 如果没有给出*column_expression*，那么列名被用作隐式路径。

如果一个列的XPath表达式返回一个非XML值（在XPath 1.0中仅限于string、boolean或double），而该列的UXDB类型不是xml，那么该列将被设置为将值的字符串表示法分配给UXDB类型。 （如果值是布尔值，如果输出列的类型类别是数字，那么它的字符串表示方式将被认为是1或0，否则true或 false。）

如果一个列的XPath表达式返回一个非空的XML节点集，并且该列的UXDB类型是xml，那么如果该列是文档或内容形式的，那么该列将被精确地分配表达式结果。 [2]

分配给xml输出列的非XML结果会产生内容，一个带有结果字符串值的单个文本节点。分配给任何其他类型的列的XML结果不能有一个以上的节点，否则会产生错误。如果正好有一个节点，则该列将被设置为将该节点的字符串值（如XPath 1.0 string函数定义的那样）分配给UXDB类型。

一个XML元素的字符串值是字符串值的协整，按文档的顺序。该元素中包含的所有文本节点及其子节点。字符串 元素的值是一个没有下级文本节点的元素的值是一个 空字符串（不是NULL）。任何xsi:nil属性都会被忽略。请注意，两个非文本之间的text()节点只用空格，而两个非文本 元素，并且保留了text()上的前导白格。节点不被扁平化。XPath 1.0中的string函数可以参考XPath 1.0中的 定义其他XML节点类型和非XML值的字符串值的规则。

这里介绍的转换规则并不完全是SQL标准中的转换规则。

[2] 在顶层包含一个以上的元素节点的结果，或者在元素之外的非空格文本，就是内容形式的一个例子。一个XPath结果可以是这两种形式的，例如，如果它返回的是一个从包含它的元素中选择的属性节点。这样的结果将被放到内容形式中，每个不允许的节点都会被替换为它的字符串值，就像XPath 1.0string 函数定义的那样。

如果路径表达式为给定行返回一个空节点集（通常情况下，当它不匹配时），该列将被设置为NULL，除非指定了*default_expression*；然后使用评价该表达式产生的值。

列可能会被标记为NOT NULL。如果一个NOT NULL列的*column_expression*不匹配任何东西并且没有DEFAULT或者*default_expression*也计算为空，则会报告一个错误。

default_expression，而不是在调用xmltable时立即被评价，而是在每次需要列的默认值时，都会被评价。 如果表达式符合稳定或不可更改的条件，则可以跳过重复评价。 这意味着，可以在*default_expression*中使用像nextval这样的不稳定函数。

例子：

```
CREATE TABLE xmldata AS SELECT
xml $$
<ROWS>
 <ROW xml:id="1">
  <COUNTRY_ID>AU</COUNTRY_ID>
  <COUNTRY_NAME>Australia</COUNTRY_NAME>
 </ROW>
 <ROW xml:id="5">
  <COUNTRY_ID>JP</COUNTRY_ID>
  <COUNTRY_NAME>Japan</COUNTRY_NAME>
  <PREMIER_NAME>Shinzo Abe</PREMIER_NAME>
  <SIZE unit="sq_mi">145935</SIZE>
 </ROW>
 <ROW xml:id="6">
  <COUNTRY_ID>SG</COUNTRY_ID>
  <COUNTRY_NAME>Singapore</COUNTRY_NAME>
  <SIZE unit="sq_km">697</SIZE>
 </ROW>
</ROWS>
$$ AS data;

SELECT xmltable.*
 FROM xmldata,
     XMLTABLE('//ROWS/ROW'
           PASSING data
           COLUMNS id int PATH '@id',
                 ordinality FOR ORDINALITY,
                 "COUNTRY_NAME" text,
                 country_id text PATH 'COUNTRY_ID',
                 size_sq_km float PATH 'SIZE[@unit = "sq_km"]',
                 size_other text PATH
                     'concat(SIZE[@unit!="sq_km"], " ", SIZE[@unit!="sq_km"]/@unit)',
                 premier_name text PATH 'PREMIER_NAME' DEFAULT 'not specified') ;

 id | ordinality | COUNTRY_NAME | country_id | size_sq_km |  size_other  | premier_name
----+------------+--------------+------------+------------+--------------+---------------
  1 |          1 | Australia    | AU         |            |              | not specified
  5 |          2 | Japan        | JP         |          | 145935 sq_mi | Shinzo Abe
  6 |          3 | Singapore    | SG         |        697 |              | not specified
```

接下来的例子展示了多个text()节点的串接、列名用作XPath过滤器的用法以及对空格、XML注释和处理指令的处理：

```
CREATE TABLE xmlelements AS SELECT
xml $$
 <root>
  <element>  Hello<!-- xyxxz -->2a2<?aaaaa?> <!--x-->  bbb<x>xxx</x>CC  </element>
 </root>
$$ AS data;

SELECT xmltable.*
 FROM xmlelements, XMLTABLE('/root' PASSING data COLUMNS element text);
     element
---------------------
  Hello2a2   bbbxxxCC
```

下面的例子展示了如何使用XMLNAMESPACES子句指定用在XML文档以及XPath表达式中的名字空间列表：

```
WITH xmldata(data) AS (VALUES ('
<example xmlns="http://example.com/myns" xmlns:B="http://example.com/b">
 <item foo="1" B:bar="2"/>
 <item foo="3" B:bar="4"/>
 <item foo="4" B:bar="5"/>
</example>'::xml)
)
SELECT xmltable.*
 FROM XMLTABLE(XMLNAMESPACES('http://example.com/myns' AS x,
                        'http://example.com/b' AS "B"),
           '/x:example/x:item'
              PASSING (SELECT data FROM xmldata)
              COLUMNS foo int PATH '@foo',
                bar int PATH '@B:bar');
 foo | bar
-----+-----
   1 |   2
   3 |   4
   4 |   5
(3 rows)
```

6.14.4. 将表映射到 XML

下面的函数将会把关系表的内容映射成 XML 值。它们可以被看成是 XML 导出功能：

```
table_to_xml(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xml(cursor refcursor, count int, nulls boolean,
              tableforest boolean, targetns text)
```

每一个函数的返回值都是xml。

table_to_xml映射由参数*tbl*传递的命名表的内容。regclass类型接受使用常见标记标识表的字符串，包括可选的模式限定和双引号。query_to_xml执行由参数*query*传递的查询并且映射结果集。cursor_to_xml从*cursor*指定的游标中取出指定数量的行。如果需要映射一个大型的表，我们推荐这种变体，因为每一个函数都是在内存中构建结果值的。

如果*tableforest*为假，则结果的 XML 文档看起来像这样：

```
<tablename>
 <row>
  <columnname1>data</columnname1>
  <columnname2>data</columnname2>
 </row>

 <row>
  ...
 </row>

 ...
</tablename>
```

如果*tableforest*为真，结果是一个看起来像这样的 XML 内容片断：

```
<tablename>
 <columnname1>data</columnname1>
 <columnname2>data</columnname2>
</tablename>

<tablename>
 ...
</tablename>

...
```

如果没有表名可用，在映射一个查询或一个游标时，在第一种格式中使用串table，在第二种格式中使用row。

这几种格式的选择由用户决定。第一种格式是一个正确的 XML 文档，它在很多应用中都很重要。如果结果值要被重组为一个文档，第二种格式在cursor_to_xml函数中更有用。前文讨论的产生 XML 内容的函数（特别是xmlelement）可以被用来把结果修改成符合用户的要求。

数据值会被以前文的函数xmlelement中描述的相同方法映射。

参数*nulls*决定空值是否会被包含在输出中。如果为真，列中的空值被表示为：

```
<columnname xsi:nil="true"/>
```

其中xsi是 XML 模式实例的 XML 名字空间前缀。一个合适的名字空间声明将被加入到结果值中。如果为假，包含空值的列将被从输出中忽略掉。

参数*targetns*指定想要的结果的 XML 名字空间。如果没有想要的特定名字空间，将会传递一个空串。

下面的函数返回 XML 模式文档，这些文档描述上述对应函数所执行的映射：

```
table_to_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)
cursor_to_xmlschema(cursor refcursor, nulls boolean, tableforest boolean, targetns text)
```

最重要的是相同的参数被传递来获得匹配的 XML 数据映射和 XML 模式文档。

下面的函数产生 XML 数据映射和对应的 XML 模式，并把产生的结果链接在一起放在一个文档（或森林）中。在要求自包含和自描述的结果是它们非常有用：

table_to_xml_and_xmlschema(tbl regclass, nulls boolean, tableforest boolean, targetns text)
query_to_xml_and_xmlschema(query text, nulls boolean, tableforest boolean, targetns text)

另外，下面的函数可用于产生相似的整个模式或整个当前数据库的映射：

schema_to_xml(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)
schema_to_xml_and_xmlschema(schema name, nulls boolean, tableforest boolean, targetns text)

database_to_xml(nulls boolean, tableforest boolean, targetns text)
database_to_xmlschema(nulls boolean, tableforest boolean, targetns text)
database_to_xml_and_xmlschema(nulls boolean, tableforest boolean, targetns text)

注意这些函数可能产生很多数据，它们都需要在内存中被构建。在请求大型模式或数据库的内容映射时，可以考虑分别映射每一个表，甚至通过一个游标来映射。

一个模式内容映射的结果看起来像这样：

<schemaname>

table1-mapping

table2-mapping

...

</schemaname>

其中一个表映射的格式取决于上文解释的*tableforest*参数。

一个数据库内容映射的结果看起来像这样：

<dbname>

<schema1name>
 ...
</schema1name>

<schema2name>
 ...
</schema2name>

...

</dbname>

其中的模式映射如上所述。

作为一个使用这些函数产生的输出的例子，图 6.1 “转换 SQL/XML 输出到 HTML 的 XSLT 样式表”展示了一个 XSLT 样式表，它将table_to_xml_and_xmlschema的输出转换为一个包含表数据的扁平转印的 HTML 文档。以一种相似的方式，这些函数的结果可以被转换成其他基于 XML 的格式。

图 6.1. 转换 SQL/XML 输出到 HTML 的 XSLT 样式表

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns="http://www.w3.org/1999/xhtml"
>

 <xsl:output method="xml"
    doctype-system="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
    doctype-public="-//W3C/DTD XHTML 1.0 Strict//EN"
    indent="yes"/>

 <xsl:template match="/*">
  <xsl:variable name="schema" select="//xsd:schema"/>
  <xsl:variable name="tabletypename"
            select="$schema/xsd:element[@name=name(current())]/@type"/>
  <xsl:variable name="rowtypename"
            select="$schema/xsd:complexType[@name=$tabletypename]/xsd:sequence/
xsd:element[@name='row']/@type"/>

   <html>
    <head>
     <title><xsl:value-of select="name(current())"/></title>
    </head>
    <body>
     <table>
      <tr>
       <xsl:for-each select="$schema/xsd:complexType[@name=$rowtypename]/xsd:sequence/
xsd:element/@name">
        <th><xsl:value-of select="."/></th>
       </xsl:for-each>
      </tr>

      <xsl:for-each select="row">
       <tr>
        <xsl:for-each select="*">
         <td><xsl:value-of select="."/></td>
        </xsl:for-each>
       </tr>
      </xsl:for-each>
     </table>
    </body>
   </html>
 </xsl:template>

</xsl:stylesheet>
```

6.15. JSON 函数和操作符

本节描述:

- 用于处理和创建JSON数据的函数和运算器
- SQL/JSON路径语言

有关UXDB中支持的JSON类型的详细信息，见 第 5.14 节 “JSON 类型”。

6.15.1. 处理和创建JSON数据

表 6.50 “json和jsonb 操作符”展示了可以用于 JSON 数据类型（见第 5.14 节 “JSON 类型”）的操作符。

表 6.50. json和jsonb 操作符

操作符	右操作数类型	返回类型	描述	例子	例子结果
->	int	json or jsonb	获得 JSON 数组元素（索引从 0 开始，负整数从末尾开始计）	'[{"a":"foo"}, {"b":"bar"}, {"c":"baz"}]'::json->2	{"c":"baz"}
->	text	json or jsonb	通过键获得 JSON 对象域	'{"a": {"b":"foo"}}'::json->'a'	{"b":"foo"}
->>	int	text	以text形式获得 JSON 数组元素	'[1,2,3]'::json->>2	3
->>	text	text	以text形式获得 JSON 对象域	'{"a":1,"b":2}'::json->>'b'	2
#>	text[]	json or jsonb	获取在指定路径的 JSON 对象	'{"a": {"b": {"c": "foo"}}}'::json#>'{a,b}'	{"c": "foo"}
#>>	text[]	text	以text形式获取在指定路径的 JSON 对象	'{"a": [1,2,3],"b": [4,5,6]}'::json#>>'{a,2}'	3

> **注意**
>
> 对json和jsonb类型，这些操作符都有其并行变体。 域/元素/路径抽取操作符返回与其左手输入（json或jsonb） 相同的类型，不过那些被指定为返回text的除外，它们的返回值会被强制 为文本。如果该 JSON 输入没有匹配请求的正确结构（例如那样的元素不存在），这些 域/元素/路径抽取操作符会返回 NULL 而不是失败。 接受整数 JSON 数组下标的 域/元素/路径抽取操作符都支持表示从数组末尾开始的负值下标形式。

表 6.1 “比较操作符”中展示的标准比较操作符只对 jsonb有效，而不适合json。它们遵循在第 5.14.4 节 “jsonb 索引”中给出的 B 树操作规则。

如表 6.51 “额外的jsonb操作符”中所示，还存在一些只适合 jsonb的操作符。这些操作符中的很多可以用jsonb 操作符类索引。jsonb包含和存在语义的完整描述可参见第 5.14.3 节 “jsonb 包含和存在”。第 5.14.4 节 “jsonb 索引”描述了如何 用这些操作符来有效地索引jsonb数据。

表 6.51. 额外的jsonb操作符

操作符	右操作数类型	描述	例子
@>	jsonb	左边的 JSON 值是否在顶层包含右边的 JSON路径/值项？	'{"a":1, "b":2}'::jsonb @> '{"b":2}'::jsonb
<@	jsonb	左边的 JSON 路径/值项是否被包含在右边的JSON 值的顶层？	'{"b":2}'::jsonb <@ '{"a":1, "b":2}'::jsonb
?	text	键/元素字符串是否存在于 JSON 值的顶层？	'{"a":1, "b":2}'::jsonb ? 'b'
?|	text[]	这些数组字符串中的任何一个是否做为顶层键存在？	'{"a":1, "b":2, "c":3}'::jsonb ?| array['b', 'c']
?&	text[]	是否所有这些数组字符串都作为顶层键存在？	'["a", "b"]'::jsonb ?& array['a', 'b']
||	jsonb	把两个jsonb值串接成一个新的jsonb值	'["a", "b"]'::jsonb || '["c", "d"]'::jsonb
-	text	从左操作数删除键/值对或者string 元素。键/值对基于它们的键值来匹配。	'{"a": "b"}'::jsonb - 'a'
-	text[]	从左操作数中删除多个键/值对或者string元素。键/值对基于它们的键值来匹配。	'{"a": "b", "c": "d"}'::jsonb - '{a,c}'::text[]
-	integer	删除具有指定索引（负值表示倒数）的数组元素。如果 顶层容器不是数组则抛出一个错误。	'["a", "b"]'::jsonb - 1
#-	text[]	删除具有指定路径的域或者元素（对于 JSON数组，负值 表示倒数）	'["a", {"b":1}]'::jsonb #- '{1,b}'
@?	jsonpath	JSON路径是否返回指定的JSON值的任何项目？	'{"a":[1,2,3,4,5]}'::jsonb @? '$.a[*] ? (@ > 2)'
@@	jsonpath	返回指定的JSON路径谓词检查结果。只考虑结果的第一项。 如果结	'{"a":[1,2,3,4,5]}'::jsonb @@ '$.a[*] > 2'

操作符	右操作数类型	描述	例子
		果不是布尔值，那么返回 null 。	

> **注意**
>
> ||操作符将其每一个操作数的顶层的元素串接起来。它不会递归 操作。例如，如果两个操作数都是具有公共域名称的对象，结果中的域值将 只是来自右手操作数的值。

> **注意**
>
> @?和@@@操作符会抑制以下错误：缺乏对象字段或数组元素、意外的JSON项类型和数字错误。当搜索不同结构的JSON文档集合时，这种行为可能会有帮助。

表 6.52 “JSON 创建函数”展示了可用于创建 json 和 jsonb值的函数（没有用于 jsonb的与row_to_json和 array_to_json等价的函数。不过，to_jsonb函数 提供了这些函数的很大一部分相同的功能）。

表 6.52. JSON 创建函数

函数	描述	例子	例子结果
to_json(anyelement) to_jsonb(anyelement)	把该值返回为json或者jsonb。数组和组合 会被（递归）转换成数组和对象；对于不是数组和组合的值，如果有从该类型到json的造型，造型函数将被用来执行该 转换；否则将产生一个标量值。对于任何不是数字、布尔、空值的标 量类型，将使用文本表达，在这种风格下它是一个合法的json或者jsonb值。	to_json('Fred said "Hi."'::text)	"Fred said \"Hi.\""
array_to_json(anyarray [, pretty_bool])	把数组作为一个 JSON 数组返回。一个 UXDB 多维数组会成为一个数组 的 JSON 数组。如果*pretty_bool*为真，将在 第 1 维度的元素之间增加换行。	array_to_json('{{1,5},{99,100}}'::int[])	[[1,5],[99,100]]
row_to_json(record [, pretty_bool])	把行作为一个 JSON 对象返回。如果*pretty_bool*为真，将在第1层元素之间增加换行。	row_to_json(row(1,'foo'))	{"f1":1,"f2":"foo"}

函数	描述	例子	例子结果
json_build_array (VARIADIC "any") jsonb_build_array (VARIADIC "any")	从一个可变参数列表构造一个可能包含异质类型的 JSON 数组。	json_build_array(1,2,'3',4,5)	[1, 2, "3", 4, 5]
json_build_object (VARIADIC "any") jsonb_build_object (VARIADIC "any")	从一个可变参数列表构造一个 JSON 对象。通过转换，该参数列表由交替 出现的键和值构成。	json_build_object('foo',1,'bar',2)	{"foo": 1, "bar": 2}
json_object(text[]) jsonb_object(text[])	从一个文本数组构造一个 JSON 对象。该数组必须可以是具有偶数个成员的 一维数组（成员被当做交替出现的键/值对），或者是一个二维数组（每一个内部数组刚好有 2 个元素，可以被看做是键/值对）。	json_object('{a, 1, b, "def", c, 3.5}') json_object('{{a, 1},{b, "def"},{c, 3.5}}')	{"a": "1", "b": "def", "c": "3.5"}
json_object(keys text[], values text[]) jsonb_object(keys text[], values text[])	json_object的这种形式从两个独立的数组得到键/值对。在其 他方面和一个参数的形式相同。	json_object('{a, b}', '{1,2}')	{"a": "1", "b": "2"}

> **注意**
>
> array_to_json和row_to_json与to_json 具有相同的行为，不过它们提供了更好的打印选项。针对to_json所描述 的行为同样也适用于由其他 JSON 创建函数转换的每个值。

> **注意**
>
> hstore扩展有一个从hstore到json 的造型，因此通过 JSON 创建函数转换的hstore值将被表示为 JSON 对象，而不是原始字符串值。

表 6.53 “JSON 处理”展示了可用来处理json 和jsonb值的函数。

表 6.53. JSON 处理

函数	返回值	描述	例子	例子结果
json_array_length (json) jsonb_array_length (jsonb)	int	返回最外层 JSON 数组中的元素数量。	json_array_length ('[1,2,3,{"f1":1,"f2":[5,6]},4]')	5

函数	返回值	描述	例子	例子结果
json_each(json) jsonb_each(jsonb)	setof key text, value json setof key text, value jsonb	扩展最外层的 JSON 对象成为一组键/值对。	select * from json_each('{"a": "foo", "b":"bar"}')	key | value -----+------- a | "foo" b | "bar"
json_each_text (json) jsonb_each_text (jsonb)	setof key text, value text	扩展最外层的 JSON 对象成为一组键/值对。返回值将是text类型。	select * from json_each_text('{ "a":"foo", "b":"bar"}')	key | value -----+------- a | foo b | bar
json_extract_path (from_json json, VARIADIC path_elems text[]) jsonb_extract_path (from_json jsonb, VARIADIC path_elems text[])	json jsonb	返回由*path_elems*指向的 JSON 值（等效于#>操作符）。	json_extract_path('{ "f2":{"f3":1}, "f4":{"f5":99, "f6":"foo"}}','f4')	{"f5":99,"f6":"foo"}
json_extract_path_text (from_json json, VARIADIC path_elems text[]) jsonb_extract_path_text (from_json jsonb, VARIADIC path_elems text[])	text	以text返回由*path_elems*指向的 JSON 值（等效于#>>操作符）。	json_extract_path_ text('{"f2": {"f3":1}, "f4":{"f5":99,"f6": "foo"}}','f4', 'f6')	foo
json_object_keys (json) jsonb_object_keys (jsonb)	setof text	返回最外层 JSON 对象中的键集合。	json_object_keys ('{"f1":"abc","f2": {"f3":"a", "f4":"b"}}')	json_object_keys ------------------ f1 f2
json_populate_ record(base anyelement, from_json json) jsonb_populate_ record(base anyelement, from_json jsonb)	anyelement	扩展*from_json*中的对象成一个行，它的列匹配由*base*定义的记录类型（见下文的注释）。	select * from json_populate_ record(null::myrowtype, '{"a": 1, "b": ["2", "a b"], "c": {"d": 4, "e": "a b c"}}')	a | b | c ---+-----------+------------- 1 | {2,"a b"} | (4,"a b c")
json_populate_ recordset(base anyelement, from_json json)	setof anyelement	扩展*from_json*中最外的对象数组为一个集合，该集合的列匹配	select * from json_populate_ recordset(null:: myrowtype, '[{"a"	a | b ---+--- 1 | 2 3 | 4

函数	返回值	描述	例子	例子结果
jsonb_populate_recordset(base anyelement, from_json jsonb)		由*base*定义的记录类型。	:1,"b":2},{"a":3,"b":4}]')	
json_array_elements(json) jsonb_array_elements(jsonb)	setof json setof jsonb	把一个 JSON 数组扩展成一个 JSON 值的集合。	select * from json_array_elements('[1,true, [2,false]]')	value ---------- 1 true [2,false]
json_array_elements_text(json) jsonb_array_elements_text(jsonb)	setof text	把一个 JSON 数组扩展成一个text值集合。	select * from json_array_elements_text('["foo", "bar"]')	value ---------- foo bar
json_typeof(json) jsonb_typeof(jsonb)	text	把最外层的 JSON 值的类型作为一个文本字符串返回。可能的类型是： object、array、string、number、boolean以及null。	json_typeof('-123.4')	number
json_to_record(json) jsonb_to_record(jsonb)	record	从一个 JSON 对象（见下文的注解）构建一个任意的记录。正如所有返回record的函数一样，调用者必须用一个AS子句显式地定义记录的结构。	select * from json_to_record('{"a":1,"b":[1,2,3],"c":[1,2,3],"e":"bar","r": {"a": 123, "b": "a b c"}}') as x(a int, b text, c int[], d text, r myrowtype)	a | b | c | d | r ---+---------+---+--------------- 1 | [1,2,3] | {1,2,3} | | (123,"a b c")
json_to_recordset(json) jsonb_to_recordset(jsonb)	setof record	从一个 JSON 对象数组（见下文的注解）构建一个任意的记录集合。正如所有返回record 的函数一样，调用者必须用一个AS子句显式地定义记录的结构。	select * from json_to_recordset('[{"a":1,"b":"foo"},{"a":"2","c":"bar"}]') as x(a int, b text);	a | b ---+----- 1 | foo 2 |
json_strip_nulls(from_json json) jsonb_strip_nulls(from_json jsonb)	json jsonb	返回*from_json*，其中所有具有空值的 对象域都被省略。其他空值不动。	json_strip_nulls('[{"f1":1,"f2":null},2,null,3]')	[{"f1":1},2,null,3]

函数	返回值	描述	例子	例子结果
jsonb_set(target jsonb, path text[], new_value jsonb[,create_ missing boolean])	jsonb	返回*target*，其中由 *path*指定的节用 *new_value*替换，如果 *path*指定的项不存在并且 *create_missing*为真（默认为 true）则加上 *new_value*。正如面向路径的 操作符一样，出现在*path*中的 负整数表示从 JSON 数组的末尾开始数。	jsonb_set('[{"f1":1,"f2":null},2,null,3]', '{0,f1}','[2,3,4]', false) jsonb_set('[{"f1":1,"f2":null},2]', '{0,f3}','[2,3,4]')	[{"f1": [2,3,4],"f2":null},2, null,3] [{"f1": 1, "f2": null, "f3": [2, 3, 4]}, 2]
sonb_insert(target jsonb, path text[], new_value jsonb [, insert_after boolean])	jsonb	返回被插入了*new_value*的*target*。如果*path*指定的*target*节在一个 JSONB 数组中，*new_value*将被插入到目标之前（*insert_after*为 false，默认情况）或者之后（*insert_after*为真）。如果*path*指定的*target*节在一个 JSONB 对象内，则只有当*target*不存在时才插入*new_value*。对于面向路径的操作符来说，出现在*path*中的负整数表示从 JSON 数组的末尾开始计数。	jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"') jsonb_insert('{"a": [0,1,2]}', '{a, 1}', '"new_value"', true)	{"a": [0, "new_value", 1, 2]} {"a": [0, 1, "new_value", 2]}
jsonb_pretty (from_json jsonb)	text	把*from_json*返回成一段 缩进后的 JSON 文本。	jsonb_pretty('[{"f1":1,"f2":null}, 2,null,3]')	[{ "f1": 1, "f2": null }, 2, null, 3

函数	返回值	描述	例子	例子结果
				]
jsonb_path_exists (target jsonb, path jsonpath [, vars jsonb [, silent bool]])	boolean	检查JSON路径是否为指定的JSON值返回任何项目。	jsonb_path_exists ('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2,"max":4}')	true
jsonb_path_match (target jsonb, path jsonpath [, vars jsonb [, silent bool]])	boolean	返回指定的JSON路径谓词检查结果。 只考虑结果的第一项。 如果结果不是布尔值，则返回null。	jsonb_path_match ('{"a":[1,2,3,4,5]}', 'exists($.a[*] ? (@ >= $min && @ <= $max))', '{"min":2,"max":4}')	true
jsonb_path_query (target jsonb, path jsonpath [, vars jsonb [, silent bool]])	setof jsonb	获取指定的JSON值的JSON路径返回的所有JSON项。	select * from jsonb_path_query ('{"a":[1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2,"max":4}');	jsonb_path_query ---------- 2 3 4
jsonb_path_query_array(target jsonb, path jsonpath [, vars jsonb [, silent bool]])	jsonb	获取指定JSON路径返回的所有JSON项，并将结果封装成数组。	jsonb_path_query_array ('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2,"max":4}')	[2, 3, 4]
jsonb_path_query_first(target jsonb, path jsonpath [, vars jsonb [, silent bool]])	jsonb	获取指定的JSON值的第一个JSON路径返回的JSON项。 如果没有结果，则返回NULL。	jsonb_path_query_first('{"a": [1,2,3,4,5]}', '$.a[*] ? (@ >= $min && @ <= $max)', '{"min":2,"max":4}')	2

注意

很多这些函数和操作符将把 JSON 字符串中的 Unicode 转义转换成合适的单一字符。如果 输入类型是jsonb，这就没有问题，因为该转换已经完成了。但是对于json 输入，这可能会导致抛出一个错误（如第 5.14 节“JSON 类型”所述）。

注意

函数 json[b]_populate_record, json[b]_populate_recordset, json[b]_to_record and json[b]_to_recordset 对JSON对象或对象数组进行操作，并提取与输出行类型的列名匹配的键相关的值。不对应于任何输出列名的对象字段将被

忽略，不匹配任何对象字段的输出列将被填充为空。要将JSON值转换为输出列的SQL类型，需要依次应用以下规则：

- 在所有情况下，JSON空值都会被转换为SQL空值。
- 如果输出列的类型为json或jsonb，那么JSON值就会被完全复制。
- 如果输出的列是复合（行）类型，而JSON值是JSON对象，则通过递归应用这些规则，将该对象的字段转换为输出行类型的列。
- 同样，如果输出列是一个数组类型，而JSON值是一个JSON数组，则通过递归应用这些规则将JSON数组的元素转换为输出数组的元素。
- 否则，如果JSON值是一个字符串的字段，则该字符串的内容被送入列的数据类型的输入转换函数。
- 否则，将JSON值的普通文本表示方式送入该列的数据类型的输入转换函数。

虽然这些函数的例子使用了常量，但典型的用法是在FROM子句中引用一个表，并使用其中的json或jsonb列作为函数的参数。 然后，提取的键值可以被引用到查询的其他部分，比如WHERE子句和目标列表。 用这种方式提取多个值，比用每键操作符分别提取多个值可以提高性能。

注意

jsonb_set和jsonb_insert的path参数中除最后一项之外的所有项都必须存在于target中。如果create_missing为假，jsonb_set的path参数的所有项都必须存在。如果这些条件不满足，则返回的target不会被改变。

如果最后的路径项是一个对象键，在它不存在且给定了新值的情况下会创建它。如果最后的路径项是一个数组索引，为正值则表示从左边开始计数，为负值表示从右边开始计数 - -1表示最右边的元素，以此类推。如果该项超过范围 -array_length .. array_length -1 并且 create_missing 为真，则该项为负时把新值加载数组的开始处，而该项为正时把新值加在数组的末尾处。

注意

不要把json_typeof函数的null返回值与 SQL 的 NULL 弄混。 虽然调用json_typeof('null'::json)将会返回null，但调用 json_typeof(NULL::json)将会返回一个 SQL 的 NULL。

注意

如果json_strip_nulls的参数在任何对象中包含重复的域名称， 结果在语义上可能有所不同，具体取决于它们发生的顺序。这不是 jsonb_strip_nulls的一个问题，因为jsonb值 不能具有重复的对象域名称。

注意

jsonb_path_exists、jsonb_path_match、jsonb_path_query、jsonb_path_query_array。和 jsonb_path_query_first函数有可选的vars和silent参数。

如果指定了*vars*参数，它提供了一个包含命名的变量的对象，要替换到jsonpath表达式中。

> 如果指定了*silent*参数，并且具有true值，那么这些函数将抑制与@?和@@操作符相同的错误。

也可参见第 6.20 节 “聚集函数”了解聚集函数json_agg，它可以把记录值聚集成 JSON。还有聚集函数json_object_agg，它可以把值对聚集成一个 JSON 对象。还有它们的jsonb等效体，jsonb_agg和jsonb_object_agg.

6.15.2. SQL/JSON路径语言

SQL/JSON路径表达式指定了要从JSON数据中检索的项目，类似于SQL访问XML时使用的XPath表达式。在UXDB中，路径表达式作为jsonpath数据类型实现，可以使用第 5.14.6 节 “jsonpath Type”中描述的任何元素。

JSON查询函数和操作符将提供的路径表达式传递给路径引擎进行评估。如果表达式与被查询的JSON数据匹配，则返回相应的SQL/JSON项。路径表达式是用SQL/JSON路径语言编写的，也可以包括算术表达式和函数。查询函数将提供的表达式视为文本字符串，所以必须用单引号括起来。

一个路径表达式由jsonpath数据类型允许的元素序列组成。路径表达式从左至右进行评估，但可以使用括号来改变操作顺序。如果评价成功，则会产生一个SQL/JSON项的序列（**SQL/JSON序列**），并将评价结果返回给完成指定计算的JSON查询函数。

要引用要查询的JSON数据（**context项**），请在路径表达式中使用$符号。后面可以有一个或多个accessor操作符，它可以逐级递减JSON结构中的上下文项的内容。下面的每一个操作符都涉及到上一个评估步骤的结果。

例如，假设有一些来自GPS追踪器的JSON数据，想对其进行解析，例如：

```
{
 "track": {
  "segments": [
   {
    "location":  [ 47.763, 13.4034 ],
    "start time": "2018-10-14 10:05:14",
    "HR": 73
   },
   {
    "location":  [ 47.706, 13.2635 ],
    "start time": "2018-10-14 10:39:21",
    "HR": 135
   }
  ]
 }
}
```

要检索可用的轨迹段，需要使用.*key*访问器操作符来检索前面所有的JSON对象。

```
'$.track.segments'
```

如果要检索的项目是一个数组的元素，必须使用[*]操作符来取消这个数组的嵌套。例如，下面的路径将返回所有可用的轨道段的位置坐标。

```
'$.track.segments[*].location'
```

要只返回第一段的坐标，可以在[]访问器操作符中指定相应的下标。请注意，SQL/JSON数组是0相关的：

```
'$.track.segments[0].location'
```

每个路径评估步骤的结果可以由一个或多个jsonpath操作符和第 6.15.2.3 节 “SQL/JSON路径操作符和方法”中列出的方法处理。每个方法名称前必须有一个点。例如，可以得到一个数组大小：

```
'$.track.segments.size()'
```

关于在路径表达式中使用jsonpath操作符和方法的更多示例，请参见 第 6.15.2.3 节 “SQL/JSON路径操作符和方法”.

在定义路径时，还可以使用一个或多个过滤表达式，其工作原理类似于SQL中的WHERE子句。一个过滤表达式以问号开头，并在括号中提供一个条件：

```
? (condition)
```

筛选表达式必须在应用于路径评估步骤之后立即指定。这个步骤的结果被过滤，只包括那些满足所提供条件的项。SQL/JSON定义了三值逻辑，所以条件可以是true、false或unknown。unknown值的作用与SQL的NULL相同，可以用is unknown谓词来测试。进一步的路径评估步骤只使用那些过滤表达式返回true的项目。

在表 6.55 “jsonpath 筛选表达式元素”中列出了可以在过滤表达式中使用的函数和操作符。要过滤的路径评价结果由@变量表示。要引用存储在较低嵌套层的JSON元素，可以在@之后添加一个或多个访问符操作符。

假设想检索所有心率值大于130的心率值。可以用下面的表达式来实现：

```
'$.track.segments[*].HR ? (@ > 130)'
```

如果要得到带有这样的值的段的起始时间，需要在返回起始时间之前过滤掉不相关的段，所以过滤表达式应用于上一步，条件中使用的路径不同：

```
'$.track.segments[*] ? (@.HR > 130)."start time"'
```

如果需要，可以在同一个嵌套层上使用多个过滤表达式。例如，下面的表达式选择所有包含有相关坐标和高心率值的位置的位置段，并选择了以下表达式：

```
'$.track.segments[*] ? (@.location[1] < 13.4) ? (@.HR > 130)."start time"'
```

也允许使用不同嵌套级别的过滤表达式。下面的例子首先按位置过滤所有的段，然后返回这些段的高心率值，如果可用的话：

```
'$.track.segments[*] ? (@.location[1] < 13.4).HR ? (@ > 130)'
```

也可以在彼此之间嵌套过滤表达式：

```
'$.track ? (exists(@.segments[*] ? (@.HR > 130))).segments.size()'
```

该表达式返回轨道的大小，如果它包含任何具有高心跳值的片段，则返回轨道的大小，否则返回空序列。

UXDB的SQL/JSON路径语言的实现与SQL/JSON标准有以下偏差。

- .datetime()项方法还没有实现，这主要是因为不可变的jsonpath函数和操作符不能引用会话时区，而这是在一些日期时间操作中使用的。
- 路径表达式可以是一个布尔谓词，尽管SQL/JSON标准只允许在过滤器中使用谓词。 这对于实现@@操作符是必要的。例如，下面的jsonpath表达式在UXDB中有效。

  '$.track.segments[*].HR < 70'

- 第 6.15.2.2 节 “正则表达式”中描述的like_regex过滤器中使用的正则表达式模式的解释有一些小的差异。

6.15.2.1. 严格和宽松模式

当查询JSON数据时，路径表达式可能与实际的JSON数据结构不匹配。试图访问一个对象或数组元素中不存在的成员，会导致结构错误。SQL/JSON路径表达式有两种模式来处理结构性错误。

- lax（默认值）— 路径引擎隐式地将被查询的数据适应指定的路径。任何剩余的结构性错误都会被压制并转换为空的SQL/JSON序列。
- 严格的—如果发生结构性错误，则会提出错误。

宽松模式方便了JSON文档结构和路径的匹配。如果JSON数据不符合预期的模式，那么就会使用JSON表达式。如果一个操作项不符合特定操作的要求。 它可以被自动包装成一个SQL/JSON数组，或者通过 将其元素转换为SQL/JSON序列，然后再执行 这个操作。此外，比较操作者会自动解开他们的 运算子在宽松模式下，所以可以比较SQL/JSON数组中的 开箱即用。一个大小为1的数组被认为等于它的唯一元素。自动解包并不是只有在以下情况下才会执行。

- 路径表达式包含type()或size()方法，这些方法分别返回数组中的类型和元素数量。
- 被查询的JSON数据包含嵌套数组。在这种情况下，只有最外层的数组被解包，而所有内部数组保持不变。因此，在每个路径评估步骤中，隐式解包只能往下走一层。

例如，在查询上面列出的GPS数据时，可以抽象出它在使用宽松模式时存储了一个数组段的事实。

'lax $.track.segments.location'

在严格模式下，指定的路径必须与被查询的JSON文档的结构完全匹配才能返回一个SQL/JSON项，所以使用这个路径表达式会导致错误。为了得到与宽松模式下相同的结果，必须显式地解包segments数组。

'strict $.track.segments[*].location'

6.15.2.2. 正则表达式

SQL/JSON路径表达式允许用like_regex过滤器将文本匹配到正则表达式。 例如，下面的SQL/JSON路径查询将对数组中所有以英文元音开头的字符串进行大小写匹配。

'$[*] ? (@ like_regex "^[aeiou]" flag "i")'

可选的flag字符串可以包括一个或多个字符i，用于不区分大小写的匹配，m允许^和$在换行处匹配，s允许.匹配一个新行，而q引用整个模式（将行为还原为简单的子串匹配）。

SQL/JSON标准借用了正则表达式的定义。LIKE_REGEX操作符，而该操作符又使用了 XQuery标准。 UXDB目前不支持 LIKE_REGEX运算符。 因此。 like_regex 过滤器是用 POSIX正则表达式引擎在 第 6.7.3 节 “POSIX正则表达式”。 这导致了各种小的 与标准SQL/JSON行为的差异，这些差异在 第 6.7.3.8 节 “与XQuery的区别LIKE_REGEX)”。但是，请注意，这里所描述的旗帜-字母不兼容的问题 不适用于SQL/JSON，因为它将XQuery标志字母转换为 匹配POSIX引擎的期望值。

请记住，like_regex的模式参数是一个JSON路径字符串字段，根据第 5.14.6 节 “jsonpath Type”中给出的规则编写。 这特别意味着，想在正则表达式中使用的任何反斜线必须是双倍的。例如，要匹配只包含数字的字符串。

'$? (@ like_regex "^\\d+$")'

6.15.2.3. SQL/JSON路径操作符和方法

表 6.54 “jsonpath 运算符和方法”显示了jsonpath中的操作符和方法。 表 6.55 “jsonpath 筛选表达式元素”显示了可用的过滤器表达式元素。

表 6.54. jsonpath 运算符和方法

运算符和方法	描述	JSON举例	Query举例	结果
+ (unary)	加号运算符，可对SQL/JSON序列进行迭代。	{"x": [2.85, -14.7, -9.4]}	+ $.x.floor()	2, -15, -10
- (unary)	对SQL/JSON序列进行迭代的减法运算器	{"x": [2.85, -14.7, -9.4]}	- $.x.floor()	-2, 15, 10
+ (binary)	加号	[2]	2 + $[0]	4
- (binary)	减号	[2]	4 - $[0]	2
*	乘号	[4]	2 * $[0]	8
/	除号	[8]	$[0] / 2	4
%	取模	[32]	$[0] % 10	2
type()	SQL/JSON项的类型	[1, "2", {}]	$[*].type()	"number", "string", "object"
size()	SQL/JSON项的大小	{"m": [11, 15]}	$.m.size()	2
double()	由SQL/JSON数字或字符串转换而来的近似浮点数	{"len": "1.9"}	$.len.double() * 2	3.8
ceiling()	大于或等于SQL/JSON数的最近整数	{"h": 1.3}	$.h.ceiling()	2
floor()	小于或等于SQL/JSON数的最近整数	{"h": 1.3}	$.h.floor()	1

运算符和方法	描述	JSON举例	Query举例	结果
abs()	SQL/JSON号的绝对值	{"z": -0.3}	$.z.abs()	0.3
keyvalue()	对象的键-值对的序列，用包含三个字段的数组来表示。 ("key", "value", 和 "id"). "id" 是键-值对所属对象的唯一标识符。	{"x": "20", "y": 32}	$.keyvalue()	{"key": "x", "value": "20", "id": 0}, {"key": "y", "value": 32, "id": 0}

表 6.55. jsonpath 筛选表达式元素

值/谓语	描述	JSON举例	Query举例	结果
==	等号运算符	[1, 2, 1, 3]	$[*] ? (@ == 1)	1, 1
!=	不等号运算符	[1, 2, 1, 3]	$[*] ? (@ != 1)	2, 3
<>	不等号运算符 (same as !=)	[1, 2, 1, 3]	$[*] ? (@ <> 1)	2, 3
<	小于运算符	[1, 2, 3]	$[*] ? (@ < 2)	1
<=	小于等于运算符	[1, 2, 3]	$[*] ? (@ <= 2)	1, 2
>	大于号运算符	[1, 2, 3]	$[*] ? (@ > 2)	3
>=	大于等于运算符	[1, 2, 3]	$[*] ? (@ >= 2)	2, 3
true	Value used to perform comparison with JSON true literal	[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]	$[*] ? (@.parent == true)	{"name": "Chris", "parent": true}
false	Value used to perform comparison with JSON false literal	[{"name": "John", "parent": false}, {"name": "Chris", "parent": true}]	$[*] ? (@.parent == false)	{"name": "John", "parent": false}
null	Value used to perform comparison with JSON null value	[{"name": "Mary", "job": null}, {"name": "Michael", "job": "driver"}]	$[*] ? (@.job == null) .name	"Mary"
&&	布尔与	[1, 3, 7]	$[*] ? (@ > 1 && @ < 5)	3
||	布尔或	[1, 3, 7]	$[*] ? (@ < 1 || @ > 5)	7
!	布尔非	[1, 3, 7]	$[*] ? (!(@ < 5))	7
like_regex	测试第一个操作项是否与第二个操作项给出的正则表达式相匹	["abc", "abd", "aBdC", "abdacb", "babc"]	$[*] ? (@ like_regex "^ab.*c" flag "i")	"abc", "aBdC", "abdacb"

值/谓语	描述	JSON举例	Query举例	结果
	配，可选择用flag字符串描述的修改（见第 6.15.2.2 节“正则表达式”）。			
starts with	测试第二个执行项是否是第一个执行项的初始子串。	["John Smith", "Mary Stone", "Bob Johnson"]	$[*]?(@ starts with "John")	"John Smith"
exists	测试一个路径表达式是否至少匹配一个SQL/JSON项。	{"x": [1, 2], "y": [2, 4]}	strict $.* ? (exists (@ ? (@[*] > 2)))	2, 4
is unknown	测试一个布尔条件是否是unknown	[-1, 2, 7, "infinity"]	$[*] ? ((@ > 0) is unknown)	"infinity"

6.16. 序列操作函数

本节描述用于操作序列对象的函数，序列对象也被称为序列生成器或者就是序列。序列对象都是用CREATE SEQUENCE(7)创建的特殊的单行表。序列对象通常用于为表的行生成唯一的标识符。表 6.56 “序列函数”中列出的这些序列函数，可以为我们从序列对象中获取连续的序列值提供了简单的、多用户安全的 方法。

表 6.56. 序列函数

函数	返回类型	描述
currval(regclass)	bigint	返回最近一次用nextval获取的指定序列的值
lastval()	bigint	返回最近一次用nextval获取的任何序列的值
nextval(regclass)	bigint	递增序列并返回新值
setval(regclass, bigint)	bigint	设置序列的当前值
setval(regclass, bigint, boolean)	bigint	设置序列的当前值以及is_called标志

将要由序列函数调用操作的序列是用一个regclass参数声明的， 它只是序列在ux_class系统表里面的 OID。不过，不需要手工查找 OID， 因为regclass数据类型的输入转换器会帮做这件事情。 只要写出用单引号包围的序列名字即可，因此它看上去像文本常量。为了和普通SQL名字处理兼容，这个字串将转换成小写形式， 除非在序列名字周围包含双引号。因此：

```
nextval('foo')     操作序列foo
nextval('FOO')     操作序列foo
nextval('"Foo"')   操作序列Foo
```

必要时序列名可以用模式限定：

nextval('myschema.foo')　操作myschema.foo
nextval('"myschema".foo')　同上
nextval('foo')　　在搜索路径中查找foo

参阅第　5.19　节　“对象标识符类型”获取有关regclass的更多信息。

> **注意**
>
> 序列函数的参数类型是text，　而不是regclass，并且前文所述的从文本串到OID值的转换将在每次调用的时候发生。　为了向后兼容，这个处理仍然存在，但是在内部实际上是通过在函数调用前隐式地将text转换成regclass实现的。
>
> 当把一个序列函数的参数写成一个无修饰的文字串，那么它将变成类型为regclass的常量。因为这只是一个　OID，它将跟踪最初标识的序列，而不管后面是否改名、模式变化等等。　这种“早期绑定”的行为通常是列默认值和视图中引用的序列所需要的。　但是有时候可能想要“延迟绑定”，其中序列的引用是在运行时解析的。要得到延迟绑定的行为，我们可以强制常量被存储为text常量，而不是regclass：
>
> nextval('foo'::text)　　foo在运行时查找
>
> 当然，序列函数的参数也可以是表达式。如果它是一个文本表达式，那么隐式的转换将导致运行时的查找。

可用的序列函数有：

nextval

递增序列对象到它的下一个值并且返回该值。这个动作是自动完成的：　　即使多个会话并发执行nextval，每个进程也会安全地收到一个唯一的序列值。

如果一个序列对象是用默认参数创建的，连续的nextval调用将会返回从 1 开始的连续的值。其他的行为可以通过在CREATE　SEQUENCE(7)命令中使用特殊参数来获得；详见该命令的参考页。

> **重要**
>
> 为了避免阻塞从同一个序列获取序号的并发事务，nextval操作从来不会被回滚。也就是说，一旦一个值被取出就视同被用掉并且不会被再次返回给调用者，即便调用该操作的外层事务后来中止或者调用查询后来没有使用取得的值也是这样。例如一个带有ON　CONFLICT子句的INSERT会计算要被插入的元组，其中可能就包括调用nextval，然后才会检测到导致它转向ON　CONFLICT规则的冲突。这种情况就会在已分配值的序列中留下未被使用的“空洞”。因此，UXDB的序列对象不能被用来得到“无间隙”的序列。

这个函数要求序列上的USAGE或者UPDATE特权。

currval

在当前会话中返回最近一次nextval取到的该序列的值（如果在本会话中从未在该序列上调用过nextval，那么会报告一个错误）。请注意因为此函数返回一个会话本地的值，不论其它会话是否在当前会话之后执行过nextval，它都能给出一个可预测的回答。

这个函数要求序列上的USAGE或者SELECT特权。

lastval

返回当前会话里最近一次nextval返回的值。 这个函数等效于currval，只是它不用序列名作为参数， 它会引用当前会话里面最近一次被应用的序列的nextval。如果当前会话还没有调用过nextval，那么调用lastval会报错。

这个函数要求上一次使用的序列上的USAGE或者SELECT特权。

setval

重置序列对象的计数器值。双参数的形式设置序列的last_value域为指定值并且将其is_called域设置为 true，表示下一次nextval将在返回值之前递增该序列。currval报告的值也被设置为指定的值。在三参数形式里，is_called可以设置为true或false。true具有和双参数形式相同的效果。如果把它设置为false，那么下一次nextval将返回指定的值，而从随后的nextval才开始递增该序列。此外，在这种情况中currval报告的值不会被改变。 例如：

```
SELECT setval('foo', 42);          下一次nextval会返回 43
SELECT setval('foo', 42, true);    同上
SELECT setval('foo', 42, false);   下一次nextval将返回 42
```

setval返回的结果就是它的第二个参数的值。

> **重要**
>
> 因为序列是非事务的，setval造成的改变不会由于事务的回滚而撤销。

这个函数要求序列上的UPDATE特权。

6.17. 条件表达式和函数

本节描述在UXDB中可用的SQL兼容的条件表达式。

> **提示**
>
> 如果需求超过这些条件表达式的能力，可能会希望用一种更富表现力的编程语言写一个服务器端函数。

> **注意**
>
> 尽管COALESCE、GREATEST和LEAST在语法上类似于函数，但它们不是普通的函数，因此不能使用显式VARIADIC数组参数。

6.17.1. CASE

SQL CASE表达式是一种通用的条件表达式，类似于其它编程语言中的 if/else 语句：

```
CASE WHEN condition THEN result
    [WHEN ...]
```

```
    [ELSE result]
END
CASE [col_name]
    WHEN [value] THEN result...
    [ ELSE result ]
END
```

CASE子句可以用于任何表达式可以出现的地方。每一个*condition*是一个返回boolean结果的表达式。如果结果为真，那么CASE表达式的结果就是符合条件的*result*，并且剩下的CASE表达式不会被处理。如果条件的结果不为真，那么以相同方式搜寻任何随后的WHEN子句。如果没有WHEN *condition*为真，那么CASE表达式的值就是在ELSE子句里的*result*。如果省略了ELSE子句而且没有条件为真，结果为空。

例子：

```
SELECT * FROM test;

 a
---
 1
 2
 3


SELECT a,
      CASE WHEN a=1 THEN 'one'
           WHEN a=2 THEN 'two'
           ELSE 'other'
      END
   FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other

select case c1
 when 1 then 'abc' when 2 then '123' else 'a1b2c3'
 end
from t1;

 case
--------
 abc
 123
 a1b2c3
 a1b2c3
(4 rows)
```

所有*result*表达式的数据类型都必须可以转换成单一的输出类型。 参阅第 7.5 节“UNION、CASE和相关结构”获取细节。

下面这个“简单”形式的CASE表达式是上述通用形式的一个变种：

```
CASE expression
   WHEN value THEN result
   [WHEN ...]
   [ELSE result]
END
```

第一个*expression*会被计算，然后与所有在WHEN子句中的每一个*value*对比，直到找到一个相等的。如果没有找到匹配的，则返回在ELSE子句中的*result*（或者控制）。 这类似于 C 里的switch语句。

上面的例子可以用简单CASE语法来写：

```
SELECT a,
      CASE a WHEN 1 THEN 'one'
           WHEN 2 THEN 'two'
           ELSE 'other'
      END
   FROM test;

 a | case
---+-------
 1 | one
 2 | two
 3 | other
```

CASE表达式并不计算任何无助于判断结果的子表达式。例如，下面是一个可以避免被零除错误的方法：

```
SELECT ... WHERE CASE WHEN x <> 0 THEN y/x > 1.5 ELSE false END;
```

> **注意**
>
> 如第 1.2.14 节 “表达式计算规则”中所述，在有几种情况中一个表达式的子表达式 会被计算多次，因此“CASE只计算必要的表达式”这 一原则并非不可打破。例如一个常量子表达式1/0通常将会在规划时导致一次 除零错误，即便它位于一个执行时永远也不会进入的CASE分支时也是 如此。

6.17.2. COALESCE

COALESCE(*value* [, ...])

COALESCE函数返回它的第一个非空参数的值。当且仅当所有参数都为空时才会返回空。它常用于在为显示目的检索数据时用缺省值替换空值。例如：

```
SELECT COALESCE(description, short_description, '(none)') ...
```

如果description不为空，这将会返回它的值，否则如果short_description非空则返回short_description的值，如果前两个都为空则返回(none)。

和CASE表达式一样，COALESCE将不会 计算无助于判断结果的参数；也就是说，在第一个非空参数右边的参数不会被计算。这个 SQL 标准函数提供了类似于NVL和IFNULL的能力，它们被用在某些其他数据库系统中。

6.17.3. NULLIF

NULLIF(*value1*, *value2*)

当*value1*和*value2*相等时，NULLIF返回一个空值。 否则它返回*value1*。 这些可以用于执行前文给出的COALESCE例子的逆操作：

SELECT NULLIF(value, '(none)') ...

在这个例子中，如果value是(none)，将返回空值，否则返回value的值。

6.17.4. GREATEST和LEAST

GREATEST(*value* [, ...])

LEAST(*value* [, ...])

GREATEST和LEAST函数从一个任意的数字表达式列表里选取最大或者最小的数值。 这些表达式必须都可以转换成一个普通的数据类型，它将会是结果类型 （参阅第 7.5 节“UNION、CASE和相关结构”获取细节）。列表中的 NULL 数值将被忽略。只有所有表达式的结果都是 NULL 的时候，结果才会是 NULL。

请注意GREATEST和LEAST都不是 SQL 标准，但却是很常见的扩展。某些其他数据库让它们在任何参数为 NULL 时返回 NULL，而不是在所有参数都为 NULL 时才返回 NULL。

6.17.5. decode

- 功能

  将输入的数值与函数中的参数列表相比较，根据输入值返回一个对应值。第一个参数为目标值或运算表达式，后续参数都是成对出现，成对参数的第一个作为判断条件，如果判断条件符合，则输出第二个参数，若参数列表末尾参数非成对出现，则为default结果，如果不存在与目标值一致的search，则输出default。default值也可以省略，此时如果不存在与目标值一致的search则输出NULL。

- 函数

  select decode(expr, search1, result1, search2, result2, …, [default]);

- 参数

  表 6.57. decode参数说明

参数	说明
expr	目标值或运算表达式，用于与参数列表中的search比较。
search1 … searchn	判断条件，与expr比较。

参数	说明
result1 … resultn	所要输出的结果，根据对应的search是否与expr匹配。
default	默认输出结果，若列表中没有与expr相匹配的search，则输出default，可省略，省略时输出null。

- 返回值

  返回值类型由result类型确定，各result类型不一致且无法转换时会报错。

- 示例

  正常传入多个参数。

  ```
  select decode (3, 1, 'one', 2, 'two', 3, 'three');
   decode
  --------
   three
  (1 row)
  ```

  没有与expr相匹配且携带default。

  ```
  select decode (4, 1, 'one', 2, 'two', 3, 'three', 'default');
   decode
  ---------
   default
  (1 row)
  ```

  没有与expr相匹配未携带default。

  ```
  select decode (4, 1, 'one', 2, 'two', 3, 'three');
   decode
  --------

  (1 row)
  ```

  传入2个参数，数据转码功能。

  ```
  select decode ('MTIzAAE=', 'base64');
     decode
  --------------
   \x3132330001
  (1 row)
  ```

- 注意事项

  decode可接收的参数个数可变，且参数数据类型不确定，若传入各result参数数据类型的基类型相同，则内部可直接转换，最终输出对应数据类型，但存在传入参数的类型基类型也不一致的场景，需开启语法开关：set mysql_grammar to on，另外，若expr与search中数字与字符类型混合出现，为保证可正常比较，需创建隐式转换插件。

6.18. 数组函数和操作符

表 6.58 “数组操作符”显示了可以用于数组类型的操作符。

表 6.58. 数组操作符

操作符	描述	例子	结果
=	等于	ARRAY[1.1,2.1,3.1]:: int[] = ARRAY[1,2,3]	t
<>	不等于	ARRAY[1,2,3] <> ARRAY[1,2,4]	t
<	小于	ARRAY[1,2,3] < ARRAY[1,2,4]	t
>	大于	ARRAY[1,4,3] > ARRAY[1,2,4]	t
<=	小于等于	ARRAY[1,2,3] <= ARRAY[1,2,3]	t
>=	大于等于	ARRAY[1,4,3] >= ARRAY[1,4,3]	t
@>	包含	ARRAY[1,4,3] @> ARRAY[3,1,3]	t
<@	被包含	ARRAY[2,2,7] <@ ARRAY[1,7,4,2,6]	t
&&	重叠（具有公共元素）	ARRAY[1,4,3] && ARRAY[2,1]	t
||	数组和数组串接	ARRAY[1,2,3] || ARRAY[4,5,6]	{1,2,3,4,5,6}
||	数组和数组串接	ARRAY[1,2,3] || ARRAY[[4,5,6],[7,8,9]]	{{1,2,3},{4,5,6}, {7,8,9}}
||	元素到数组串接	3 || ARRAY[4,5,6]	{3,4,5,6}
||	数组到元素串接	ARRAY[4,5,6] || 7	{4,5,6,7}

数组排序操作符（<、>=等）对数组内容进行逐个元素的比较，使用默认的元素数据类型的B-tree比较函数，并根据第一个差值进行排序。 多维数组的元素按照行序进行访问（最后的下标变化最快）。如果两个数组的内容相同但维数不等，那么维度信息中的第一个不同将决定排序顺序。

数组包含操作符（<@和@>）认为，如果一个数组的每一个元素出现在另一个数组中，那么这个数组就被包含在另一个数组中。重复数组不做特殊处理，因此ARRAY[1]和ARRAY[1,1]分别被认为包含了另一个数组。

参阅第 5.15 节 “数组”获取有关数组操作符行为的更多细节。有关哪些操作符支持被索引的操作，请参阅第 8.2 节 “索引类型”。

表 6.59 “数组函数”展示了可以用于数组类型的函数。 参阅第 5.15 节 “数组”获取更多信息以及使用这些函数的例子。

表 6.59. 数组函数

函数	返回类型	描述	例子	结果
array_append (anyarray, anyelement)	anyarray	向一个数组的末端追加一个元素	array_append (ARRAY[1,2], 3)	{1,2,3}
array_cat(anyarray, anyarray)	anyarray	连接两个数组	array_cat(ARRAY [1,2,3], ARRAY[4,5])	{1,2,3,4,5}
array_ndims (anyarray)	int	返回数组的维度数	array_ndims (ARRAY[[1,2,3], [4,5,6]])	2
array_dims (anyarray)	text	返回数组的维度的文本表示	array_dims (ARRAY[[1,2,3], [4,5,6]])	[1:2][1:3]
array_fill (anyelement, int[], [, int[]])	anyarray	返回一个用提供的值和维度初始化好的数组，可以选择下界不为 1	array_fill(7, ARRAY[3], ARRAY[2])	[2:4]={7,7,7}
array_length (anyarray, int)	int	返回被请求的数组维度的长度	array_length(array [1,2,3], 1)	3
array_lower (anyarray, int)	int	返回被请求的数组维度的下界	array_lower('[0:2]= {1,2,3}'::int[], 1)	0
array_position (anyarray, anyelement [, int])	int	返回在该数组中从第三个参数指定的元素开始或者 第一个元素开始（数组必须是一维的）、第二个参数的 第一次出现的下标	array_position (ARRAY['sun','mon', 'tue','wed','thu', 'fri','sat'], 'mon')	2
array_positions (anyarray, anyelement)	int[]	返回在第一个参数给定的数组（数组必须是一维的）中， 第二个参数所有出现位置的下标组成的数组	array_positions (ARRAY['A','A', 'B','A'], 'A')	{1,2,4}
array_prepend (anyelement, anyarray)	anyarray	向一个数组的首部追加一个元素	array_prepend(1, ARRAY[2,3])	{1,2,3}
array_remove (anyarray, anyelement)	anyarray	从数组中移除所有等于给定值的所有元素（数组必须是一维的）	array_remove (ARRAY[1,2,3,2], 2)	{1,3}
array_replace (anyarray,	anyarray	将每一个等于给定值的数组元素替换成一个新值	array_replace (ARRAY[1,2,5,4], 5, 3)	{1,2,3,4}

函数	返回类型	描述	例子	结果
anyelement, anyelement)				
array_to_string (anyarray, text [, text])	text	使用提供的定界符和可选的空串连接数组元素	array_to_string (ARRAY[1, 2, 3, NULL, 5], ',', '*')	1,2,3,*,5
array_upper (anyarray, int)	int	返回被请求的数组维度的上界	array_upper (ARRAY[1,8,3,7], 1)	4
cardinality (anyarray)	int	返回数组中元素的总数，如果数组为空则返回 0	cardinality(ARRAY [[1,2],[3,4]])	4
string_to_array(text, text [, text])	text[]	使用提供的定界符和可选的空串将字符串划分成数组元素	string_to_array ('xx~^~yy~^~zz', '~^~', 'yy')	{xx,NULL,zz}
unnest(anyarray)	setof anyelement	将一个数组扩展成一组行	unnest (ARRAY[1,2])	1 2 (2 rows)
unnest(anyarray, anyarray [, ...])	setof anyelement, anyelement [, ...]	把多维数组（可能是不同类型）扩展成一个行的集合。 这只允许用在 FROM 子句中，见 第 4.2.1.4 节 “表函数”	unnest(ARRAY[1, 2],ARRAY['foo', 'bar','baz'])	1 foo 2 bar NULL baz (3 rows)

在array_position和array_positions中， 每一个数组元素都使用IS NOT DISTINCT FROM 语义与要搜索的值比较。

在array_position中，如果值没有找到则返回 NULL。

在array_positions中，只有当数组为 NULL时才返回NULL，如果该值 没有在该数组中找到则返回一个空数组。

在string_to_array中，如果定界符参数为 NULL，输入字符串中的每一个字符将变成结果数组中的一个独立元素。如果定界符是一个空串，则整个输入字符串被作为一个单一元素的数组返回。否则输入字符串会被在每一个出现定界符字符串的位置分裂。

在string_to_array中，如果空值串参数被忽略或者为 NULL，输入中的子串不会被替换成 NULL。
在array_to_string中，如果空值串参数被忽略或者为 NULL，该数组中的任何空值元素会被简单地跳过并且不会在输出串中被表示。

也可参见第 6.20 节 “聚集函数”了解用于数组的聚集函数array_agg。

6.19. 范围函数和操作符

范围类型的概述请见第 5.17 节 “范围类型”。

表 6.60 “范围操作符”展示了范围类型可用的操作符。

表 6.60. 范围操作符

操作符	描述	例子	结果
=	等于	int4range(1,5) = '[1,4]'::int4range	t
<>	不等于	numrange(1.1,2.2) <> numrange(1.1,2.3)	t
<	小于	int4range(1,10) < int4range(2,3)	t
>	大于	int4range(1,10) > int4range(1,5)	t
<=	小于等于	numrange(1.1,2.2) <= numrange(1.1,2.2)	t
>=	大于等于	numrange(1.1,2.2) >= numrange(1.1,2.0)	t
@>	包含范围	int4range(2,4) @> int4range(2,3)	t
@>	包含元素	'[2011-01-01,2011-03-01)'::tsrange @> '2011-01-10'::timestamp	t
<@	范围被包含	int4range(2,4) <@ int4range(1,7)	t
<@	元素被包含	42 <@ int4range(1,7)	f
&&	重叠（有公共点）	int8range(3,7) && int8range(4,12)	t
<<	严格左部	int8range(1,10) << int8range(100,110)	t
>>	严格右部	int8range(50,60) >> int8range(20,30)	t
&<	不超过右部	int8range(1,20) &< int8range(18,20)	t
&>	不超过左部	int8range(7,20) &> int8range(5,10)	t
-|-	相邻	numrange(1.1,2.2) -|- numrange(2.2,3.3)	t
+	并	numrange(5,15) + numrange(10,20)	[5,20)
*	交	int8range(5,15) * int8range(10,20)	[10,15)
-	差	int8range(5,15) - int8range(10,20)	[5,10)

简单比较操作符<、 >、<=和 >=首先比较下界，并且只有在下界相等时才比较上界。这些比较通常对范围不怎么有用，但是还是提供它们以便能够在范围上构建 B树索引。

当涉及一个空范围时，左部/右部/相邻操作符总是返回假；即一个空范围被认为不在任何其他范围前面或者后面。

如果结果范围可能需要包含两个分离的子范围，并和差操作符将会失败，因为这样的范围无法被表示。

表 6.61 “范围函数”显示可用于范围类型的函数。

表 6.61. 范围函数

函数	返回类型	描述	例子	结果
lower(anyrange)	范围的元素类型	范围的下界	lower(numrange (1.1,2.2))	1.1
upper(anyrange)	范围的元素类型	范围的上界	upper(numrange (1.1,2.2))	2.2
isempty(anyrange)	boolean	范围为空？	isempty(numrange (1.1,2.2))	false
lower_inc (anyrange)	boolean	下界包含在内？	lower_inc (numrange(1.1,2.2))	true
upper_inc (anyrange)	boolean	上界包含在内？	upper_inc (numrange(1.1,2.2))	false
lower_inf (anyrange)	boolean	下界无限？	lower_inf ('(,)'::daterange)	true
upper_inf (anyrange)	boolean	上界无限？	upper_inf ('(,)'::daterange)	true
range_merge (anyrange, anyrange)	anyrange	包含两个给定范围的最小范围	range_merge('[1,2)' ::int4range, '[3,4)' ::int4range)	[1,4)

如果范围为空或者被请求的界是无限的，lower和upper函数返回空值。函数lower_inc、upper_inc、lower_inf和upper_inf对一个空范围全部返回假。

6.20. 聚集函数

聚集函数从一个输入值的集合计算出一个单一值。内建的通用聚集函数在表 6.62 “通用聚集函数”中列出，而统计性聚集在表 6.63 “用于统计的聚集函数”中列出。内建的组内有序集聚集函数在表 6.64 “有序集聚集函数”中列出，而内建的组内假想集聚集在表 6.65 “假想集聚集函数”中列出。与聚集函数紧密相关的分组操作在表 6.66 “分组操作”中列出。第 1.2.7 节“聚集表达式”中会解释针对聚集函数的特殊语法考虑。

表 6.62. 通用聚集函数

函数	参数类型	返回类型	部分模式	描述
array_agg(*expression*)	任何非数组类型	参数类型的数组	No	输入值（包括空）被连接到一个数组
array_agg(*expression*)	任意数组类型	和参数数据类型相同	No	输入数组被串接到一个更高维度的数组中 （输入

函数	参数类型	返回类型	部分模式	描述
				必须都具有相同的维度并且不能为空或者 NULL）
avg(*expression*)	smallint, int, bigint、real、double precision、numeric或interval	对于任何整数类型参数是numeric，对于一个浮点参数是double precision，否则和参数数据类型相同	Yes	所有非空输入值的平均值（算术平均）
bit_and(*expression*)	smallint、int、bigint或bit	与参数数据类型相同	Yes	所有非空输入值的按位与，如果没有非空值则结果是空值
bit_or(*expression*)	smallint, int, bigint, or bit	与参数数据类型相同	Yes	所有非空输入值的按位或，如果没有非空值则结果是空值
bool_and(*expression*)	bool	bool	Yes	如果所有输入值为真则结果为真，否则为假
bool_or(*expression*)	bool	bool	Yes	至少一个输入值为真时结果为真，否则为假
count(*)		bigint	Yes	输入的行数
count(*expression*)	any	bigint	Yes	*expression*值非空的输入行的数目
every(*expression*)	bool	bool	Yes	等价于bool_and
json_agg(*expression*)	any	json	No	将值，包含空值，聚集成一个 JSON 数组
jsonb_agg(*expression*)	any	jsonb	No	把值，包含空值，聚合成一个 JSON 数组
json_object_agg(*name*, *value*)	(any, any)	json	No	将名字/值对聚集成一个 JSON 对象，值可以为空，但不能是名字。
jsonb_object_agg(*name*, *value*)	(any, any)	jsonb	No	把名字/值对聚合成一个 JSON 对象，值可以为空，但不能是名字。

函数	参数类型	返回类型	部分模式	描述
max(*expression*)	任意数组、数字、串、日期/时间、网络或者枚举类型，或者这些类型的数组	与参数数据类型相同	Yes	所有非空输入值中*expression*的最大值
min(*expression*)	任意数组、数字、串、日期/时间、网络或者枚举类型，或者这些类型的数组	与参数数据类型相同	Yes	所有非空输入值中*expression*的最小值
string_agg(*expression*, *delimiter*)	(text, text) 或(bytea, bytea)	与参数数据类型相同	No	非空输入值连接成一个串，用定界符分隔
sum(*expression*)	smallint、int、bigint、real、double precision、numeric、interval或money	对smallint或int参数是bigint，对bigint参数是numeric，否则和参数数据类型相同	Yes	所有非空输入值的*expression*的和
xmlagg(*expression*)	xml	xml	No	连接非空XML值（参见第 6.14.1.7 节“xmlagg”）

请注意，除了count以外，这些函数在没有行被选中时返回控制。尤其是sum函数在没有输入行时返回空值，而不是零，并且array_agg在这种情况返回空值而不是一个空数组。必要时可以用coalesce把空值替换成零或一个空数组。

支持部分模式的聚集函数有资格参与到各种优化中，例如并行聚集。

注意

布尔聚集bool_and和bool_or对应于标准的 SQL 聚集every和any或some。而对于any 和some，似乎在标准语法中有一个歧义：

SELECT b1 = ANY((SELECT b2 FROM t2 ...)) FROM t1 ...;

如果子查询返回一行有一个布尔值的结果，这里的ANY可以被认为是引入一个子查询，或者是作为一个聚集函数。因而标准的名称不能指定给这些聚集。

注意

在把count聚集应用到整个表上时，习惯于使用其他 SQL 数据管理系统的用户可能会对它的性能感到失望。一个如下的查询：

SELECT count(*) FROM sometable;

将会要求与整个表大小成比例的工作：UXDB将需要扫描整个表或者整个包含表中所有行的索引。

与相似的用户定义的聚集函数一样，聚集函数array_agg、json_agg、jsonb_agg、json_object_agg、jsonb_object_agg、string_agg和xmlagg会依赖输入值的顺序产生有意义的不同结果值。这个顺序默认是不用指定的，但是可以在聚集调用时使用ORDER BY子句进行控制，如第 1.2.7 节 “聚集表达式”中所示。作为一种选择，从一个排序号的子查询来提供输入值通常会有帮助。例如：

SELECT xmlagg(x) FROM (SELECT x FROM test ORDER BY y DESC) AS tab;

注意如果外面的查询层次包含额外的处理（例如连接），这种方法可能会失败，因为这可能导致子查询的输出在计算聚集之前被重新排序。

表 6.63 “用于统计的聚集函数”展示了通常被用在统计分析中的聚集函数（这些被隔离出来是为了不和常用聚集混淆）。其中描述提到了*N*，它表示对应于所有非空输入表达式的输入行的数目。在所有情况中，如果计算是无意义的，将会返回空值，例如当*N*为零。

表 6.63. 用于统计的聚集函数

函数	参数类型	返回类型	部分模式	描述
corr(*Y*, *X*)	double precision	double precision	Yes	相关系数
covar_pop(*Y*, *X*)	double precision	double precision	Yes	总体协方差
covar_samp(*Y*, *X*)	double precision	double precision	Yes	样本协方差
regr_avgx(*Y*, *X*)	double precision	double precision	Yes	自变量的平均值（sum(*X*)/*N*）
regr_avgy(*Y*, *X*)	double precision	double precision	Yes	因变量的平均值（sum(*Y*)/*N*）
regr_count(*Y*, *X*)	double precision	bigint	Yes	两个表达式都不为空的输入行的数目
regr_intercept(*Y*, *X*)	double precision	double precision	Yes	由（*X*, *Y*）对决定的最小二乘拟合的线性方程的 y 截距
regr_r2(*Y*, *X*)	double precision	double precision	Yes	相关系数的平方
regr_slope(*Y*, *X*)	double precision	double precision	Yes	由（*X*, *Y*）对决定的最小二乘拟合的线性方程的斜率
regr_sxx(*Y*, *X*)	double precision	double precision	Yes	sum(*X*^2) - sum(*X*)^2/*N*（自变量的“平方和”）
regr_sxy(*Y*, *X*)	double precision	double precision	Yes	sum(*X***Y*) - sum(*X*) * sum(*Y*)/*N*（自变

函数	参数类型	返回类型	部分模式	描述
				量乘以因变量的“积之合”）
regr_syy(*Y*, *X*)	double precision	double precision	Yes	sum(*Y*^2) - sum(*Y*)^2/*N*（因变量的“平方和”）
stddev(*expression*)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	stddev_samp的历史别名
stddev_pop(*expression*)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的总体标准偏差
stddev_samp(*expression*)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的样本标准偏差
variance(*expression*)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	var_samp的历史别名
var_pop(*expression*)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的总体方差（总体标准偏差的平方）
var_samp(*expression*)	smallint、int、bigint、real、double precision或numeric	浮点参数为double precision，否则为numeric	Yes	输入值的样本方差（样本标准偏差的平方）

表 6.64 “有序集聚集函数”展示了一些使用 有序集聚集语法的聚集函数。这些函数有时也被称为 “逆分布”函数。

表 6.64. 有序集聚集函数

函数	直接参数类型	聚集参数类型	返回类型	部分模式	描述
LISTAGG(expr[,delimiter]) WITHIN GROUP (ORDER BY expr[,expr.....])	可强制转换为text的类型	任何可排序类型	text	YES	可以根据排序参数（ORDER BY后面的列）对表中数据进行排序，然后根据聚集参数对查询列进行聚合，若存在分隔符，则会对需要聚集的

函数	直接参数类型	聚集参数类型	返回类型	部分模式	描述
					列用分隔符进行分隔
mode() WITHIN GROUP (ORDER BY *sort_expression*)		任何可排序类型	与排序表达式相同	No	返回最频繁的输入值（如果有多个频度相同的值就选第一个）
percentile_cont (*fraction*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision	double precision或者interval	与排序表达式相同	No	连续百分率：返回一个对应于排序中指定分数的值，如有必要就在相邻的输入项之间插值
percentile_cont (*fractions*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision[]	double precision或者interval	排序表达式的类型的数组	No	多重连续百分率：返回一个匹配*fractions*参数形状的结果数组， 其中每一个非空元素都用对应于那个百分率的值替换
percentile_disc (*fraction*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision	一种可排序类型	与排序表达式相同	No	离散百分率：返回第一个在排序中位置等于或者超过指定分数的输入值
percentile_disc (*fractions*) WITHIN GROUP (ORDER BY *sort_expression*)	double precision[]	任何可排序类型	排序表达式的类型的数组	No	多重离散百分率：返回一个匹配*fractions*参数形状的结果数组， 其中每一个非空元素都用对应于那个百分率的输入值替换

所有列在表 6.64 “有序集聚集函数”中的聚集会忽略它们的已 排序输入中的空值。对那些有一个*fraction*参数的聚 集来说，该分数值必须位于 0 和 1 之间，否则会抛出错误。不过，一个空分数值会 产生一个空结果。

每个列在表 6.65 “假想集聚集函数”中的聚集都与一个定义在 第 6.21 节 “窗口函数” 中的同名窗口函数相关联。在每种情况中，聚集 结果的计算方法是：假设根据*args*构建的“假想”行已 经被增加到从*sorted_args*计算得到的已排序行分组中， 然后用相关联的窗口函数针对该行返回的值就是聚集的结果。

表 6.65. 假想集聚集函数

函数	直接参数类型	聚集参数类型	返回类型	部分模式	描述
rank(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	bigint	No	假想行的排名，为重复的行留下间隔
dense_rank (*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	bigint	No	假想行的排名，不留间隔
percent_rank (*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	double precision	No	假想行的相对排名，范围从 0 到 1
cume_dist(*args*) WITHIN GROUP (ORDER BY *sorted_args*)	VARIADIC "any"	VARIADIC "any"	double precision	No	假想行的相对排名，范围从 1/*N* 到 1

对于这些假想集聚集的每一个，*args*中给定的直接参数 列表必须匹配*sorted_args*中给定的聚集参数的 数量和类型。与大部分的内建聚集不同，这些聚集并不严格，即它们不会丢弃包含空值 的输入行。空值的排序根据ORDER BY子句中指定的规则进行。

表 6.66. 分组操作

函数	返回类型	描述
GROUPING(*args...*)	integer	整数位掩码指示哪些参数不被包括在当前分组集合中
GROUPING_ID(expr [, expr].....)	Numeric	将超聚合行与常规分组行区分开来

GROUP BY扩展，例如ROLLUP和CUBE生成超聚合行，其中所有值的集合由 null表示。GROUPING_ID函数，可以将表示超聚合行中所有值的集合的空值与常规行中的空值区分开来。

> **注意**
>
> GROUPING_ID函数有如下使用限制。
>
> 1. 必须带有group by子句。
> 2. 不能用作where条件。
> 3. 可以使用having子句筛选。

```
create table liml_test(
id numeric(6),
no int,
first_name VARCHAR(20),
last_name VARCHAR(30),
email     VARCHAR(30)
);
insert into liml_test values(1,1,'james','lebron','13478653gd@a63.com');
insert into liml_test values(2,2,'pual','cris','13378653gd@a63.com');
select id, no, grouping_id(id,no) from liml_test group by cube(id,no);
 id | no | grouping_id
----+----+-------------
  |  |           3
  2 |  2 |           0
  1 |  1 |           0
  1 |  |           1
  2 |  |           1
  | 2 |           2
  | 1 |           2
(7 rows)
```

分组操作用来与分组集合（见第　4.2.4　节　“GROUPING SETS、CUBE和ROLLUP”）共同来区分结果行。GROUPING操作的参数并不会被实际计算，但是它们必　须准确地匹配在相关查询层次的GROUP BY子句中给定的表达式。　最右边参数指派的位是最低有效位，如果对应的表达式被包括在产生结果行的分组 集合的分组条件中则每一位是 0，否则是 1。例如：

```
=> SELECT * FROM items_sold;
 make  | model | sales
-------+-------+-------
 Foo   | GT    |  10
 Foo   | Tour  |  20
 Bar   | City  |  15
 Bar   | Sport |  5
(4 rows)

=> SELECT make, model, GROUPING(make,model), sum(sales) FROM items_sold GROUP BY
 ROLLUP(make,model);
 make  | model | grouping | sum
-------+-------+----------+-----
 Foo   | GT    |        0 | 10
 Foo   | Tour  |        0 | 20
 Bar   | City  |        0 | 15
 Bar   | Sport |        0 | 5
 Foo   |       |        1 | 30
 Bar   |       |        1 | 20
       |       |        3 | 50
(7 rows)
```

6.20.1. LISTAGG

- 功能

LISTAGG列转行的有序聚集函数。可以根据排序参数（ORDER BY后面的列）对表中数据进行排序，然后根据聚集参数对查询列进行聚合，若存在分隔符，则会对需要聚集的列用分隔符进行分隔。

- 函数

LISTAGG(*expr* [,*delimiter*])
WITHIN GROUP (ORDER BY *expr*[,*expr*.....])
[over(partition by *expr*)]

- 参数

表 6.67. LISTAGG参数说明

参数	说明
expr	一个表中已经存在的列名。
delimiter	一个任意的分隔符。

- 示例

将表中的数据按照firest_name分组排序后进行聚合。

select listagg(first_name,',') within group (order by first_name) from liml_test group by first_name;

- 注意

listagg函数用作分析函数时，不能带within group (order by ...)子句。

6.21. 窗口函数

窗口函数提供在与当前查询行相关的行集合上执行计算的能力。 语法细节则请见第 1.2.8 节 “窗口函数调用”。

表 6.68 “通用窗口函数”列出了内建的窗口函数。注意必须使用窗口函数的语法调用这些函数； 一个OVER子句是必需的。

在这些函数之外，任何内建的或者用户定义的通用或者统计性聚集（即非有序集和假想集聚集）都可以被用作一个窗口函数，内建聚集的列表请见第 6.20 节 “聚集函数”。仅当聚集函数调用后面跟着一个OVER子句时，聚集函数才会像窗口函数那样工作，否则它们会按非窗口聚集的方式运行并且为整个集合返回一个单一行。

表 6.68. 通用窗口函数

函数	返回类型	描述
row_number()	bigint	当前行在其分区中的行号，从1计
rank()	bigint	带间隙的当前行排名； 与该行的第一个同等行的row_number相同

函数	返回类型	描述
dense_rank()	bigint	不带间隙的当前行排名； 这个函数计数同等组
percent_rank()	double precision	当前行的相对排名： （rank - 1） / （总行数 - 1）
cume_dist()	double precision	累积分布：（在当前行之前或者平级的分区行数） / 分区行总数
ntile(*num_buckets* integer)	integer	从1到参数值的整数范围，尽可能等分分区
lag(*value* anyelement [, *offset* integer [, *default* anyelement]])	和*value*的类型相同	返回*value*，它在分区内当前行的之前*offset*个位置的行上计算；如果没有这样的行，返回*default*替代（必须和*value*类型相同）。*offset*和*default*都是根据当前行计算的结果。如果忽略它们，则*offset*默认是1，*default*默认是空值
lead(*value* anyelement [, *offset* integer [, *default* anyelement]])	和*value*类型相同	返回*value*，它在分区内当前行的之后*offset*个位置的行上计算；如果没有这样的行，返回*default*替代（必须和*value*类型相同）。*offset*和*default*都是根据当前行计算的结果。如果忽略它们，则*offset*默认是1，*default*默认是空值
first_value(*value* any)	same type as *value*	返回在窗口帧中第一行上计算的*value*
last_value(*value* any)	和*value*类型相同	返回在窗口帧中最后一行上计算的*value*
nth_value(*value* any, *nth* integer)	和*value*类型相同	返回在窗口帧中第*nth*行（行从1计数）上计算的*value*；没有这样的行则返回空值

在表 6.68 “通用窗口函数”中列出的所有函数都依赖于相关窗口定义的ORDER BY子句指定的排序顺序。仅考虑ORDER BY列时不能区分的行被称为是同等行。定义的这四个排名函数（包括cume_dist），对于任何两个同等行的答案相同。

注意first_value、last_value和nth_value只考虑“窗口帧”内的行，它默认情况下包含从分区的开始行直到当前行的最后一个同等行。这对last_value可能不会给出有用的结果，有时对nth_value也一样。可以通过向OVER子句增加一个合适的帧声明（RANGE或GROUPS）来重定义帧。关于帧声明的更多信息请参考第 1.2.8 节 “窗口函数调用”。

当一个聚集函数被用作窗口函数时，它将在当前行的窗口帧内的行上聚集。 一个使用ORDER BY和默认窗口帧定义的聚集产生一种“运行时求和”类型的行为，这可能是或者不是想要的结果。为了获取在整个分区上的聚集，忽略ORDER BY或者使用ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING。 其它窗口帧声明可以用来获得其它的效果。

> **注意**
>
> SQL 标准为lead、lag、first_value、last_value和nth_value定义了一个RESPECT NULLS或IGNORE NULLS选项。 这在UXDB中没有实现：行为总是与标准的默认相同，即RESPECT NULLS。 同样，标准中用于nth_value的FROM FIRST或FROM LAST选项没有实现： 只有支持默认的FROM FIRST行为（可以通过反转ORDER BY的排序达到FROM LAST的结果）。

cume_dist计算小于等于当前行及其平级行的分区行所占的分数，而percent_rank计算小于当前行的分区行所占的分数，假定当前行不存在于该分区中。

6.22. 子查询表达式

本节描述UXDB中可用的SQL兼容的子查询表达式。所有本节中成文的表达式都返回布尔值（真/假）结果。

6.22.1. EXISTS

EXISTS (*subquery*)

EXISTS的参数是一个任意的SELECT语句， 或者说子查询。系统对子查询进行运算以判断它是否返回行。如果它至少返回一行，那么EXISTS的结果就为“真”； 如果子查询没有返回行，那么EXISTS的结果是“假”。

子查询可以引用来自周围的查询的变量，这些变量在该子查询的任何一次计算中都起常量的作用。

这个子查询通常只是运行到能判断它是否可以返回至少一行为止， 而不是等到全部结束。在这里写任何有副作用的子查询都是不明智的（例如调用序列函数）；这些副作用是否发生是很难判断的。

因为结果只取决于是否会返回行，而不取决于这些行的内容， 所以这个子查询的输出列表通常是无关紧要的。一个常用的编码习惯是用EXISTS(SELECT 1 WHERE ...)的形式写所有的EXISTS测试。不过这条规则有例外，例如那些使用INTERSECT的子查询。

下面这个简单的例子类似在col2上的一次内联接，但是它为每个 tab1的行生成最多一个输出，即使存在多个匹配tab2的行也如此：

```
SELECT col1
FROM tab1
WHERE EXISTS (SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

6.22.2. IN

expression IN (*subquery*)

右手边是一个圆括弧括起来的子查询， 它必须正好只返回一个列。左手边表达式将被计算并与子查询结果逐行进行比较。 如果找到任何等于子查询行的情况，那么IN的结果就是“真”。 如果没有找到相等行，那么结果是“假”（包括子查询没有返回任何行的情况）。

请注意如果左手边表达式得到空值，或者没有相等的右手边值， 并且至少有一个右手边行得到空值，那么IN结构的结果将是空值，而不是假。这个行为是遵照 SQL 处理空值的一般规则的。

和EXISTS一样，假设子查询将被完成运行完全是不明智的。

row_constructor IN (*subquery*)

这种形式的IN的左手边是一个行构造器， 如第 1.2.13 节 “行构造器”中所述。 右手边是一个圆括弧子查询，它必须返回和左手边返回的行中表达式所构成的完全一样多的列。 左手边表达式将被计算并与子查询结果逐行进行比较。如果找到任意相等的子查询行，则IN的结果为“真”。如果没有找到相等行， 那么结果为“假”（包括子查询不返回行的情况）。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。 如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等； 否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等， 要么是空值，并且至少有一个空值，那么IN的结果是空值。

6.22.3. NOT IN

expression NOT IN (*subquery*)

右手边是一个用圆括弧包围的子查询，它必须返回正好一个列。左手边表达式将被计算并与子查询结果逐行进行比较。 如果只找到不相等的子查询行（包括子查询不返回行的情况），那么NOT IN的结果是“真”。 如果找到任何相等行，则结果为“假”。

请注意如果左手边表达式得到空值，或者没有相等的右手边值， 并且至少有一个右手边行得到空值，那么NOT IN结构的结果将是空值，而不是真。这个行为是遵照 SQL 处理空值的一般规则的。

和EXISTS一样，假设子查询会完全结束是不明智的。

row_constructor NOT IN (*subquery*)

这种形式的NOT IN的左手边是一个行构造器， 如第 1.2.13 节 “行构造器”中所述。 右手边是一个圆括弧子查询，它必须返回和左手边返回的行中表达式所构成的完全一样多的列。 左手边表达式将被计算并与子查询结果逐行进行比较。如果找到不等于子查询行的行，则NOT IN的结果为“真”。如果找到相等行， 那么结果为“假”（包括子查询不返回行的情况）。

通常，表达式或者子查询行里的空值是按照 SQL 布尔表达式的一般规则进行组合的。 如果两个行对应的成员都非空并且相等，那么认为这两行相等；如果任意对应成员为非空且不等，那么这两行不等； 否则这样的行比较的结果是未知（空值）。如果所有行的结果要么是不等， 要么是空值，并且至少有一个空值，那么NOT IN的结果是空值。

6.22.4. ANY/SOME

expression operator ANY (*subquery*)
expression operator SOME (*subquery*)

这种形式的右手边是一个圆括弧括起来的子查询， 它必须返回正好一个列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。如果获得任何真值结果，那么ANY的结果

就是“真”。 如果没有找到真值结果，那么结果是“假”（包括子查询没有返回任何行的情况）。

SOME是ANY的同义词。IN等价于= ANY。

请注意如果没有任何成功并且至少有一个右手边行为该操作符结果生成空值， 那么ANY结构的结果将是空值，而不是假。 这个行为是遵照 SQL 处理空值布尔组合的一般规则制定的。

和EXISTS一样，假设子查询将被完全运行是不明智的。

row_constructor operator ANY (*subquery*)
row_constructor operator SOME (*subquery*)

这种形式的左手边是一个行构造器，如第 1.2.13 节 “行构造器”所述。右手边是一个圆括弧括起来的子查询， 它必须返回和左手边列表给出的表达式一样多的列。左手边表达式将被计算并使用给出的操作符对子查询结果逐行进行比较。如果比较为任何子查询行返回真，则ANY的结果为“真”。如果比较对每一个子查询行都返回假，则结果为“假”（包括子查询不返回行的情况）。如果比较不对任何行返回真并且至少对一行返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见第 6.23.5 节 “行构造器比较”。

6.22.5. ALL

expression operator ALL (*subquery*)

ALL 的这种形式的右手边是一个圆括弧括起来的子查询， 它必须只返回一列。左手边表达式将被计算并使用给出的 操作符对子查询结果逐行进行比较。该操作符必须生成布尔结果。 如果所有行得到真（包括子查询没有返回任何行的情况），ALL的结果就是“真”。如果没有存在任何假值结果，那么结果是“假”。如果比较为任何行都不返回假并且对至少一行返回 NULL，则结果为 NULL。

NOT IN等价于<> ALL。

和EXISTS一样，假设子查询将被完全运行是不明智的。

row_constructor operator ALL (*subquery*)

ALL的这种形式的左手边是一个行构造器，如第 1.2.13 节 “行构造器”所述。 右手边是一个圆括弧括起来的子查询，它必须返回和左手边行中表达式一样多的列。 左手边表达式将被计算并使用给出的 操作符对子查询结果逐行进行比较。如果对所有子查询行该比较都返回真，那么ALL的结果就是“真”（包括子查询没有返回任何行的情况）。如果对任何子查询行比较返回假，则结果为“假”。如果比较对任何子查询行都不返回假并且对至少一行返回 NULL，则结果为 NULL。

关于行构造器比较的详细含义请见第 6.23.5 节 “行构造器比较”。

6.22.6. 单一行比较

row_constructor operator (*subquery*)

左手边是一个行构造器，如第 1.2.13 节 “行构造器”所述。 右手边是一个圆括弧括起来的子查询，该查询必须返回和左手边行中表达式数目完全一样的列。 另外，该子查询不能返回超过

一行的数量（如果它返回零行，那么结果就是空值）。 左手边被计算并逐行与右手边的子查询结果行比较。

关于行构造器比较的详细含义请见第 6.23.5 节 “行构造器比较”。

6.23. 行和数组比较

本节描述几个特殊的结构，用于在值的组之间进行多重比较。这些形式语法上和前面一节的子查询形式相关，但是不涉及子查询。 这种形式涉及的数组子表达式是UXDB的扩展； 其它的是SQL兼容的。所有本节记录的表达式形式都返回布尔（Boolean）结果（真/假）。

6.23.1. IN

expression IN (*value* [, ...])

右手边是一个圆括弧包围的标量列表。如果左手边表达式的结果等于任何右手边表达式中的一个，结果为“真”。它是下面形式的缩写

expression = *value1*
OR
expression = *value2*
OR
...

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么IN结构的结果将为空值，而不是假。这符合 SQL 处理空值的布尔组合的一般规则。

6.23.2. NOT IN

expression NOT IN (*value* [, ...])

右手边是一个圆括弧包围的标量列表。如果左手边表达式的结果不等于所有右手边表达式，结果为“真”。它是下面形式的缩写

expression <> *value1*
AND
expression <> *value2*
AND
...

请注意如果左手边表达式得到空值，或者没有相等的右手边值并且至少有一个右手边的表达式得到空值，那么NOT IN结构的结果将为空值， 而不是我们可能天真地认为的真值。这符合 SQL 处理空值的布尔组合的一般规则。

> **提示**
>
> x NOT IN y在所有情况下都等效于NOT (x IN y)。但是，在处理空值的时候，用NOT IN比用IN更可能迷惑新手。最好尽可能用正逻辑来表达条件。

6.23.3. ANY/SOME (array)

expression operator ANY (*array expression*)
expression operator SOME (*array expression*)

右手边是一个圆括弧包围的表达式，它必须得到一个数组值。左手边表达式被计算并且使用给出的操作符对数组的每个元素进行比较，这个操作符必须得到布尔结果。如果得到了任何真值结果，那么ANY的结果是“真”。 如果没有找到真值结果（包括数组只有零个元素的情况），那么结果是“假”。

如果数组表达式得到一个空数组，ANY的结果将为空值。如果左手边的表达式得到空值，ANY通常是空值（尽管一个非严格比较操作符可能得到一个不同的结果）。另外，如果右手边的数组包含任何空值元素或者没有得到真值比较结果，ANY的结果将是空值而不是假（再次，假设是一个严格的比较操作符）。这符合 SQL 对空值的布尔组合的一般规则。

SOME是ANY的同义词。

6.23.4. ALL (array)

expression operator ALL (*array expression*)

右手边是一个圆括弧包围的表达式，它必须得到一个数组值。左手边表达式将被计算并使用给出的操作符与数组的每个元素进行比较，这个操作符必须得到一个布尔结果。如果所有比较都得到真值结果，那么ALL的结果是 “真”（包括数组只有零个元素的情况）。如果有任何假值结果，那么结果是“假”。

如果数组表达式得到一个空数组，ALL的结果将为空值。如果左手边的表达式得到空值，ALL通常是空值（尽管一个非严格比较操作符可能得到一个不同的结果）。另外，如果右手边的数组包含任何空值元素或者没有得到假值比较结果，ALL的结果将是空值而不是真（再次，假设是一个严格的比较操作符）。这符合 SQL 对空值的布尔组合的一般规则。

6.23.5. 行构造器比较

row_constructor operator row_constructor

每一边都是一个行构造器，如第 1.2.13 节 “行构造器”所述。两个行值必须具有相同数量的域。每一边被计算并且被逐行比较。当操作符是 =、 <>、 < <=、 >、 >=时，允许进行行构造器比较。每一个行元素必须是具有一个默认 B 树操作符类的类型，否则尝试比较会产生一个错误。

> **注意**
>
> 如果使用早期列解析比较，则可能不会发生与元素数量或类型相关的错误。

=和<>情况略有不同。如果两行的所有对应成员都是非空且相等则这两行被认为相等；如果任何对应成员是非空但是不相等则这两行不相等；否则行比较的结果为未知（空值）。

对于<、<=、>和>=情况，行元素被从左至右比较，在找到一处不等的或为空的元素对就立刻停下来。如果这一对元素都为空值，则行比较的结果为未知（空值）；否则这一对元素的比较结果

决定行比较的结果。例如，ROW(1,2,NULL) < ROW(1,3,0)得到真，而不是空值，因为第三对元素并没有被考虑。

> **注意**
>
> <、<=、>和>=情况不是按照每个 SQL 声明来处理的。一个像ROW(a,b) < ROW(c,d)的比较会被实现为a < c AND b < d，而结果行为等价于a < c OR (a = c AND b < d)。

row_constructor IS DISTINCT FROM *row_constructor*

这个结构与<>行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

row_constructor IS NOT DISTINCT FROM *row_constructor*

这个结构与=行比较相似，但是它对于空值输入不会得到空值。任何空值被认为和任何非空值不相等（有区别），并且任意两个空值被认为相等（无区别）。因此结果将总是为真或为假，永远不会是空值。

6.23.6. 组合类型比较

record operator record

SQL 规范要求在结果依赖于比较两个 NULL 值或者一个 NULL 与一个非 NULL 时逐行比较返回 NULL。UXDB只有在比较两个行构造器（如第 6.23.5 节 “行构造器比较”的结果或者比较一个行构造器与一个子查询的输出时才这样做（如第 6.22 节 “子查询表达式”中所述）。在其他比较两个组合类型值的环境中，两个 NULL 域值被认为相等，并且一个 NULL 被认为大于一个非 NULL。为了得到组合类型的一致的排序和索引行为，这样做是必要的。

每一边都会被计算并且它们会被逐行比较。当操作符是 =、 <>、 <、 <=、 >或者 >=时或者具有与这些类似的语义时，允许组合类型的比较（更准确地说，如果一个操作符是一个 B 树操作符类的成员，或者是一个 B 树操作符类的=成员的否定词，它就可以是一个行比较操作符）。上述操作符的行为与用于行构造器（见第 6.23.5 节 “行构造器比较”的IS [NOT] DISTINCT FROM相同。

为了支持包含无默认 B 树操作符类的元素的行匹配，为组合类型比较定义了下列操作符： *=、 *<>、 *<、 *<=、 *>以及 *>=。 这些操作符比较两行的内部二进制表达。即使两行用相等操作符的比较为真，两行也可能 具有不同的二进制表达。行在这些比较操作符之下的排序是决定性的，其他倒没什么意义。 这些操作符在内部被用于物化视图并且可能对其他如复制之类的特殊功能有用，但是它们 并不打算用在书写查询这类普通用途中。

6.24. 集合返回函数

本节描述那些可能返回多于一行的函数。目前这个类中被使用最广泛的是级数生成函数，如表 6.69 “级数生成函数”和表 6.70 “下标生成函数”所述。其他更特殊的集合返回函数在本手册的其他地方描述。 组合多集合返回函数的方法可见第 4.2.1.4 节 “表函数”。

表 6.69. 级数生成函数

函数	参数类型	返回类型	描述
generate_series(*start*, *stop*)	int、bigint或者numeric	setof int、setof bigint或者setof numeric（与参数类型相同）	产生一系列值，从*start*到*stop*，步长为1
generate_series(*start*, *stop*, *step*)	int、bigint或者numeric	setof int、setof bigint或者setof numeric（与参数类型相同）	产生一系列值，从*start*到*stop*，步长为*step*
generate_series(*start*, *stop*, *step* interval)	timestamp或timestamp with time zone	setof timestamp或setof timestamp with time zone（和参数类型相同）	产生一系列值，从*start*到*stop*，步长为*step*

当*step*为正时，如果*start*大于*stop*则返回零行。相反，当*step*为负时，如果*start*小于*stop*则返回零行。对于NULL输入也会返回零行。*step*为零是一个错误。下面是一些例子：

```
SELECT * FROM generate_series(2,4);
 generate_series
-----------------
               2
               3
               4
(3 rows)

SELECT * FROM generate_series(5,1,-2);
 generate_series
-----------------
               5
               3
               1
(3 rows)

SELECT * FROM generate_series(4,3);
 generate_series
-----------------
(0 rows)

SELECT generate_series(1.1, 4, 1.3);
 generate_series
-----------------
             1.1
             2.4
             3.7
(3 rows)

-- 这个例子依赖于日期+整数操作符
SELECT current_date + s.a AS dates FROM generate_series(0,14,7) AS s(a);
   dates
------------
```

```
 2004-02-05
 2004-02-12
 2004-02-19
(3 rows)

SELECT * FROM generate_series('2008-03-01 00:00'::timestamp,
                              '2008-03-04 12:00', '10 hours');
   generate_series
---------------------
 2008-03-01 00:00:00
 2008-03-01 10:00:00
 2008-03-01 20:00:00
 2008-03-02 06:00:00
 2008-03-02 16:00:00
 2008-03-03 02:00:00
 2008-03-03 12:00:00
 2008-03-03 22:00:00
 2008-03-04 08:00:00
(9 rows)
```

表 6.70. 下标生成函数

函数	返回类型	描述
generate_subscripts(*array anyarray*, *dim int*)	setof int	生成一个级数组成给定数组的下标。
generate_subscripts(*array anyarray*, *dim int*, *reverse boolean*)	setof int	生成一个级数组成给定数组的下标。当*reverse*为真，级数以逆序返回。

generate_subscripts是一个快捷函数，它为给定数组的指定维度生成一组合法的下标。对于不具有请求维度的数组返回零行，对于 NULL 数组也返回零行（但是会对 NULL 数组元素返回合法的下标）。下面是一些例子：

```
-- 基本使用
SELECT generate_subscripts('{NULL,1,NULL,2}'::int[], 1) AS s;
 s
---
 1
 2
 3
 4
(4 rows)

-- 表示一个数组，下标和被下标的值需要一个子查询
SELECT * FROM arrays;
        a
--------------------
 {-1,-2}
 {100,200,300}
(2 rows)

SELECT a AS array, s AS subscript, a[s] AS value
```

```
FROM (SELECT generate_subscripts(a, 1) AS s, a FROM arrays) foo;
    array     | subscript | value
---------------+-----------+-------
 {-1,-2}      |         1 |    -1
 {-1,-2}      |         2 |    -2
 {100,200,300} |         1 |   100
 {100,200,300} |         2 |   200
 {100,200,300} |         3 |   300
(5 rows)

-- 平面化一个 2D 数组
CREATE OR REPLACE FUNCTION unnest2(anyarray)
RETURNS SETOF anyelement AS $$
select $1[i][j]
   from generate_subscripts($1,1) g1(i),
        generate_subscripts($1,2) g2(j);
$$ LANGUAGE sql IMMUTABLE;
CREATE FUNCTION
SELECT * FROM unnest2(ARRAY[[1,2],[3,4]]);
 unnest2
---------
       1
       2
       3
       4
(4 rows)
```

当FROM子句中的一个函数后面有WITH ORDINALITY时，输出中会追加一个bigint列，它的值从1开始并且该函数输出的每一行加1。这在unnest()之类的集合返回函数中最有用。

```
-- set returning function WITH ORDINALITY
SELECT * FROM ux_ls_dir('.') WITH ORDINALITY AS t(ls,n);
       ls       | n
-----------------+----
 ux_serial       |  1
 ux_twophase     |  2
 uxmaster.opts |  3
 ux_notify       |  4
 uxsinodb.conf |  5
 ux_tblspc       |  6
 logfile         |  7
 base            |  8
 uxmaster.pid  |  9
 ux_ident.conf   | 10
 global          | 11
 ux_xact         | 12
 ux_snapshots    | 13
 ux_multixact    | 14
 UX_VERSION      | 15
 ux_wal          | 16
 ux_hba.conf     | 17
 ux_stat_tmp     | 18
```

```
 ux_subtrans    | 19
(19 rows)
```

6.25. 系统信息函数和运算符

表 6.71 “会话信息函数”展示了多个可以抽取会话和系统信息的函数。

除了本节列出的函数，还有一些与统计系统相关的函数也提供系统信息。

表 6.71. 会话信息函数

名称	返回类型	描述
current_catalog	name	当前数据库名（SQL 标准中称作“目录”）
current_database()	name	当前数据库名
current_query()	text	当前正在执行的查询的文本，和客户端提交的一样（可能包含多于一个语句）
current_role	name	等效于current_user
current_schema[()]	name	当前模式名
current_schemas(boolean)	name[]	搜索路径中的模式名，可以选择是否包含隐式模式
current_user	name	当前执行上下文的用户名
inet_client_addr()	inet	远程连接的地址
inet_client_port()	int	远程连接的端口
inet_server_addr()	inet	本地连接的地址
inet_server_port()	int	本地连接的端口
sys_context('context','parameter'[, length])	text	用于获取环境上下文信息或判断当前会话角色
ux_backend_pid()	int	与当前会话关联的服务器进程的进程 ID
ux_blocking_pids(int)	int[]	阻塞指定服务器进程ID获得锁的进程 ID
ux_conf_load_time()	timestamp with time zone	配置载入时间
ux_current_logfile([text])	text	当前日志收集器在使用的主日志文件名或者所要求格式的日志的文件名
ux_my_temp_schema()	oid	会话的临时模式的 OID，如果没有则为 0
ux_is_other_temp_schema(oid)	boolean	模式是另一个会话的临时模式吗？
ux_jit_available()	boolean	这个会话中JIT编译是否可用？如果jit被设置为假，则返回false。
ux_listening_channels()	setof text	会话当前正在监听的频道名称

名称	返回类型	描述
ux_notification_queue_usage()	double	异步通知队列当前被占用的分数（0-1）
ux_uxmaster_start_time()	timestamp with time zone	服务器启动时间
ux_safe_snapshot_blocking_pids (int)	int[]	阻止指定服务器进程ID获取安全快照的进程ID
ux_trigger_depth()	int	UXDB触发器的当前嵌套层次（如果没有调用则为 0，直接或间接，从一个触发器内部开始）
session_user	name	会话用户名
user	name	等价于current_user
version()	text	UXDB版本信息。

> **注意**
>
> current_catalog、current_role、current_schema、current_user、session_user和user在SQL里有特殊的语意状态： 它们被调用时结尾不要跟着园括号（在 UXDB 中，圆括号可以有选择性地被用于current_schema，但是不能和其他的一起用）。

session_user通常是发起当前数据库连接的用户，不过超级用户可以用SET SESSION AUTHORIZATION(7)修改这个设置。current_user是用于权限检查的用户标识。通常， 它总是等于会话用户，但是可以被SET ROLE(7)改变。它也会在函数执行的过程中随着属性SECURITY DEFINER的改变而改变。在 Unix 的说法里，那么会话用户是“真实用户”，而当前用户是“有效用户”。current_role以及user是current_user的同义词（SQL标准在current_role和current_user之间做了区分，但UXDB不区分，因为它把用户和角色统一成了一种实体）。

current_schema返回在搜索路径中的第一个模式名（如果搜索路径是空则返回空值）。 如果创建表或者其它命名对象时没有声明目标模式，那么它将是被用于这些对象的模式。current_schemas(boolean)返回一个在搜索路径中出现的所有模式名的数组。布尔选项决定ux_catalog这样的隐式包含的系统模式是否包含在返回的搜索路径中。

> **注意**
>
> 搜索路径可以在运行时修改。命令是：
>
> SET search_path TO *schema* [, *schema*, ...]

inet_client_addr返回当前客户端的 IP 地址，inet_client_port返回它的端口号。 inet_server_addr返回接受当前连接的服务器的 IP 地址，而inet_server_port返回对应的端口号。如果连接是通过 Unix 域套接字进行的，那么所有这些函数都返回 NULL。

ux_blocking_pids返回一个进程 ID 的数组，数组中的进程中的会话阻塞了指定进程 ID 所代表的服务器进程，如果指定的服务器进程不存在或者没有被阻塞则返回空数组。如果一个进程持有与另一个进程加锁请求冲突的锁（硬锁），或者前者正在等待一个与后者加锁请求冲突的锁并且前者在该锁的等待队列中位于后者的前面（软锁），则前者会阻塞后者。在使用并行查询时，这个函数的结果总是会列出客户端可见的进程 ID（即ux_backend_pid的结果），即便实际的锁是由工

作者进程所持有或者等待也是如此。这样造成的后果是，结果中可能会有很多重复的 PID。还要注意当一个预备事务持有一个冲突锁时，这个函数的结果中它将被表示为一个为零的进程 ID。对这个函数的频繁调用可能对数据库性能有一些影响，因为它需要短时间地独占访问锁管理器的共享状态。

ux_conf_load_time返回服务器配置文件最近被载入的timestamp with time zone（如果当前会话在那时就已经存在，这个值将是该会话自己重新读取配置文件的时间，因此在不同的会话中这个读数会有一点变化。如果不是这样，这个值就是 uxmaster 进程重读配置文件的时间）。

ux_current_logfile以text类型返回当前被日志收集器使用的日志文件的路径。该路径包括log_directory目录和日志文件名。日志收集必须被启用，否则返回值为NULL。当多个日志文件存在并且每一个都有不同的格式时，不带参数调用ux_current_logfile会返回这样的文件的路径：在所有的文件中，没有任何文件的格式在列表stderr、csvlog中排在这个文件的格式前面。如果没有任何日志文件有上述格式，则返回NULL。要请求一种特定的文件格式，可以以text将csvlog或者stderr作为可选参数的值。当所请求的日志格式不是已配置的log_destination时，会返回NULL。ux_current_logfile反映了current_logfiles文件的内容。

ux_my_temp_schema返回当前会话临时模式的 OID，如果没有使用临时模式（因为它没有创建任何临时表）则返回零。如果给定的 OID 是另一个会话的临时模式的 OID，则ux_is_other_temp_schema返回真（这是有用的，例如，要将其他会话的临时表从一个目录显示中排除）。

ux_listening_channels返回当前会话正在监听的异步通知频道的名称的集合。ux_notification_queue_usage返回等待处理的通知占可用的通知空间的比例，它是一个 0-1 范围内的double值。详见LISTEN(7)和NOTIFY(7)。

ux_uxmaster_start_time返回服务器启动的timestamp with time zone。

ux_safe_snapshot_blocking_pids一个进程ID的数组，它们代表阻止指定进程ID对应的服务器进程获取安全快照的会话，如果没有这类服务器进程或者它没有被阻塞，则会返回一个空数组。一个运行着SERIALIZABLE事务的会话会阻止SERIALIZABLE READ ONLY DEFERRABLE事务获取快照，直到后者确定避免拿到任何谓词锁是安全的。更多有关可序列化以及可延迟事务的信息请参考第 10.2.3 节 “可序列化隔离级别”。频繁调用这个函数可能会对数据库性能产生一些影响，因为它需要短时间访问谓词锁管理器的共享状态。

version返回一个描述UXDB服务器版本的字符串。也可以从server_version或者一个机器可读的版本server_version_num得到这个信息。软件开发者应该使用server_version_num或者UXSQLserverVersion ，而不必解析文本形式的版本。

表 6.72 “访问权限查询函数”列出那些允许用户编程查询对象访问权限的函数。参阅第 2.7 节 “权限”获取更多有关权限的信息。

表 6.72. 访问权限查询函数

名称	返回类型	描述
has_any_column_privilege(*user*, *table*, *privilege*)	boolean	用户有没有表中任意列上的权限
has_any_column_privilege(*table*, *privilege*)	boolean	当前用户有没有表中任意列上的权限
has_column_privilege(*user*, *table*, *column*, *privilege*)	boolean	用户有没有列的权限
has_column_privilege(*table*, *column*, *privilege*)	boolean	当前用户有没有列的权限
has_database_privilege(*user*, *database*, *privilege*)	boolean	用户有没有数据库的权限

名称	返回类型	描述
has_database_privilege(*database*, *privilege*)	boolean	当前用户有没有数据库的权限
has_foreign_data_wrapper_privilege(*user*, *fdw*, *privilege*)	boolean	用户有没有外部数据包装器上的权限
has_foreign_data_wrapper_privilege(*fdw*, *privilege*)	boolean	当前用户有没有外部数据包装器上的权限
has_function_privilege(*user*, *function*, *privilege*)	boolean	用户有没有函数上的权限
has_function_privilege(*function*, *privilege*)	boolean	当前用户有没有函数上的权限
has_language_privilege(*user*, *language*, *privilege*)	boolean	用户有没有语言上的权限
has_language_privilege(*language*, *privilege*)	boolean	当前用户有没有语言上的权限
has_schema_privilege(*user*, *schema*, *privilege*)	boolean	用户有没有模式上的权限
has_schema_privilege(*schema*, *privilege*)	boolean	当前用户有没有模式上的权限
has_sequence_privilege(*user*, *sequence*, *privilege*)	boolean	用户有没有序列上的权限
has_sequence_privilege(*sequence*, *privilege*)	boolean	当前用户有没有序列上的权限
has_server_privilege(*user*, *server*, *privilege*)	boolean	用户有没有外部服务器上的权限
has_server_privilege(*server*, *privilege*)	boolean	当前用户有没有外部服务器上的权限
has_table_privilege(*user*, *table*, *privilege*)	boolean	用户有没有表上的权限
has_table_privilege(*table*, *privilege*)	boolean	当前用户有没有表上的权限
has_tablespace_privilege(*user*, *tablespace*, *privilege*)	boolean	用户有没有表空间上的权限
has_tablespace_privilege(*tablespace*, *privilege*)	boolean	当前用户有没有表空间上的权限
has_type_privilege(*user*, *type*, *privilege*)	boolean	用户有没有类型的特权
has_type_privilege(*type*, *privilege*)	boolean	当前用户有没有类型的特权
ux_has_role(*user*, *role*, *privilege*)	boolean	用户有没有角色上的权限
ux_has_role(*role*, *privilege*)	boolean	当前用户有没有角色上的权限
row_security_active(*table*)	boolean	当前用户是否在表上开启了行级安全性

has_table_privilege判断一个用户是否可以用某种特定的方式访问一个表。该用户可以通过名字或者OID（ux_authid.oid）来指定，也可以用public表示PUBLIC伪角色。如果省略该参数，则使用current_user。该表可以通过名字或者OID指定（因此，实际上有六种has_table_privilege的变体，我们可以通过它们的参数数目和类型来区分它们）。如果用名字指定，那么在必要时该名字可以是模式限定的。所希望的权限类型是用一个文本串来指定的，它必须是下面的几个值之一：SELECT、INSERT、UPDATE、DELETE、TRUNCATE、REFERENCES或TRIGGER。WITH GRANT OPTION可以被选择增加到一个权限类型来测试是否该权限是使用转授选项得到。另外，可以使用逗号分隔来列出多个权限类型，在这种情况下只要具有其中之一的权限则结果为真（权限字符串的大小写并不重要，可以在权限名称之间出现额外的空白，但是在权限名内部不能有空白）。一些例子：

SELECT has_table_privilege('myschema.mytable', 'select');

SELECT has_table_privilege('joe', 'mytable', 'INSERT, SELECT WITH GRANT OPTION');

has_sequence_privilege检查一个用户是否能以某种特定方式访问一个序列。它的参数可能性和has_table_privilege相似。所希望测试的访问权限类型必须是下列之一：USAGE、SELECT或UPDATE。

has_any_column_privilege检查一个用户是否能以特定方式访问一个表的任意列。其参数可能性和has_table_privilege类似，除了所希望的访问权限类型必须是下面值的某种组合：SELECT、INSERT、UPDATE或REFERENCES。注意在表层面上具有这些权限的任意一个都会隐式地把它授权给表中的每一列，因此如果has_table_privilege对同样的参数返回真则has_any_column_privilege将总是返回真。但是如果在至少一列上有一个该权限的列级授权，has_any_column_privilege也会成功。

has_column_privilege检查一个用户是否能以特定方式访问一个列。它的参数可能性与has_table_privilege类似，并且列还可以使用名字或者属性号来指定。希望的访问权限类型必须是下列值的某种组合：SELECT、INSERT、UPDATE或REFERENCES。注意在表级别上具有这些权限中的任意一种将会隐式地把它授予给表上的每一列。

has_database_privilege检查一个用户是否能以特定方式访问一个数据库。它的参数可能性类似has_table_privilege。希望的访问权限类型必须是以下值的某种组合：CREATE、CONNECT、TEMPORARY或TEMP（等价于TEMPORARY）。

has_function_privilege检查一个用户是否能以特定方式访问一个函数。其参数可能性类似has_table_privilege。在用一个文本串而不是 OID 指定一个函数时，允许的输入和regprocedure数据类型一样（参阅 第 5.19 节 “对象标识符类型”）。希望的访问权限类型必须是EXECUTE。一个例子：

SELECT has_function_privilege('joeuser', 'myfunc(int, text)', 'execute');

has_foreign_data_wrapper_privilege检查一个用户是否能以特定方式访问一个外部数据包装器。它的参数可能性类似于has_table_privilege。希望的访问权限类型必须是USAGE。

has_language_privilege检查一个用户是否可以以某种特定的方式访问一个过程语言。 其参数可能性类似 has_table_privilege。希望的访问权限类型必须是USAGE。

has_schema_privilege检查一个用户是否可以以某种特定的方式访问一个模式。 其参数可能性类似 has_table_privilege。希望的访问权限类型必须是CREATE或USAGE。

has_server_privilege检查一个用户是否可以以某种特定的方式访问一个外部服务器。 其参数可能性类似 has_table_privilege。希望的访问权限类型必须是USAGE。

has_tablespace_privilege检查一个用户是否可以以某种特定的方式访问一个表空间。其参数可能性类似 has_table_privilege。希望的访问权限类型必须是CREATE。

has_type_privilege检查一个用户是否能以特定的方式访问一种类型。其参数的可能性类同于has_table_privilege。在用字符串而不是 OID 指定类型时，允许的输入和regtype数据类型相同（见第 5.19 节 “对象标识符类型”）。期望的访问特权类型必须等于USAGE。

ux_has_role检查一个用户是否可以以某种特定的方式访问一个角色。其参数可能性类似has_table_privilege，除了public不能被允许作为一个用户名。希望的访问权限类型必须是下列值的某种组合：MEMBER或USAGE。MEMBER表示该角色中的直接或间接成员关系（即使用SET ROLE的权力），而USAGE表示不做SET ROLE的情况下该角色的权限是否立即可用。

row_security_active检查在 current_user的上下文和环境中是否为指定的 表激活了行级安全性。表可以用名称或者 OID 指定。

表 6.74 “aclitem Functions” 显示了aclitem类型的可用操作符，它是访问权限的目录表示。有关如何读取访问权限值的信息，请参阅 第 2.7 节 “权限”。

表 6.73. aclitem Operators

操作符	描述	例子	结果
=	equal	'calvin=r*w/hobbes'::aclitem = 'calvin=r*w*/hobbes'::aclitem	f
@>	包含元素	'{calvin=r*w/hobbes,hobbes=r*w*/uxdb}'::aclitem[] @> 'calvin=r*w/hobbes'::aclitem	t
~	包含元素	'{calvin=r*w/hobbes,hobbes=r*w*/uxdb}'::aclitem[] ~ 'calvin=r*w/hobbes'::aclitem	t

表 6.74 “aclitem Functions” 显示了一些额外的函数来管理aclitem类型。

表 6.74. aclitem Functions

名称	返回类型	描述
acldefault(*type*, *ownerId*)	aclitem[]	获取属于*ownerId*的对象的默认访问权限。
aclexplode(*aclitem[]*)	setof record	获取 aclitem 数组为元组
makeaclitem(*grantee*, *grantor*, *privilege*, *grantable*)	aclitem	从输入中建立一个aclitem。

acldefault返回属于角色*ownerId*的*type*类型的对象的内置默认访问权限。 这些代表了当对象的ACL条目为空时将被假定的访问权限。 (默认的访问权限在第 2.7 节 “权限”中描述了)。 *type* 参数是一个 CHAR: 'c' for COLUMN, 'r' for TABLE 和类表对象, 's' for SEQUENCE, 'd' for DATABASE, 'f' for FUNCTION 或者 PROCEDURE, 'l' for LANGUAGE, 'L' for LARGE OBJECT, 'n' for SCHEMA, 't' for TABLESPACE, 'F' for FOREIGN DATA WRAPPER, 'S' for FOREIGN SERVER, 'T' for TYPE 或者 DOMAIN.

aclexplode返回一个aclitem数组作为行集。输出的列是grantor oid, grantee oid (0 for PUBLIC), 被授权为 text (SELECT, ...) 以及权限是否可以被授予boolean。makeaclitem执行反向操作。

表 6.75 “模式可见性查询函数”展示了决定是否一个特定对象在当前模式搜索路径中可见的函数。例如，如果一个表所在的模式在当前搜索路径中并且在它之前没有出现过相同的名字，这个表就被说是可见的。这等价于在语句中表可以被用名称引用但不加显式的模式限定。要列出所有可见表的名字：

```
SELECT relname FROM ux_class WHERE ux_table_is_visible(oid);
```

表 6.75. 模式可见性查询函数

名称	返回类型	描述
ux_collation_is_visible(*collation_oid*)	boolean	排序规则在搜索路径中可见吗？
ux_conversion_is_visible(*conversion_oid*)	boolean	转换在搜索路径中可见吗？
ux_function_is_visible(*function_oid*)	boolean	函数在搜索路径中可见吗？
ux_opclass_is_visible(*opclass_oid*)	boolean	操作符类在搜索路径中可见吗？
ux_operator_is_visible(*operator_oid*)	boolean	操作符在搜索路径中可见吗？
ux_opfamily_is_visible(*opclass_oid*)	boolean	操作符族在搜索路径中可见吗？
ux_statistics_obj_is_visible(*stat_oid*)	boolean	是搜索路径中的统计信息对象
ux_table_is_visible(*table_oid*)	boolean	表在搜索路径中可见吗？
ux_ts_config_is_visible(*config_oid*)	boolean	文本搜索配置在搜索路径中可见吗？
ux_ts_dict_is_visible(*dict_oid*)	boolean	文本搜索字典在搜索路径中可见吗？
ux_ts_parser_is_visible(*parser_oid*)	boolean	文本搜索解析器在搜索路径中可见吗？
ux_ts_template_is_visible(*template_oid*)	boolean	文本搜索模板在搜索路径中可见吗？
ux_type_is_visible(*type_oid*)	boolean	类型（或域）在搜索路径中可见吗？

每一个函数对一种数据库对象执行可见性检查。注意ux_table_is_visible也可被用于视图、物化视图、索引、序列和外部表，ux_function_is_visible也能被用于过程和聚集，ux_type_is_visible也可以被用于域。对于函数和操作符，如果在路径中更早的地方没有出现具有相同名称和参数数据类型的对象，该对象在搜索路径中是可见的。对于操作符类，名称和相关的索引访问方法都要考虑。

所有这些函数都要求用对象 OID 来标识将被检查的对象。如果想用名称来测试一个对象，使用 OID 别名类型（regclass、regtype、regprocedure、regoperator、regconfig或regdictionary）将会很方便。例如：

SELECT ux_type_is_visible('myschema.widget'::regtype);

注意以这种方式测试一个非模式限定的类型名没什么意义 — 如果该名称完全能被识别，它必须是可见的。

表 6.76 “系统目录信息函数”列出了从系统目录抽取信息的函数。

表 6.76. 系统目录信息函数

名称	返回类型	描述
format_type(*type_oid*, *typemod*)	text	获得一个数据类型的 SQL 名字
ux_get_constraintdef(*constraint_oid*)	text	获得一个约束的定义
ux_get_constraintdef(*constraint_oid*, *pretty_bool*)	text	获得一个约束的定义
ux_get_expr(*ux_node_tree*, *relation_oid*)	text	反编译一个表达式的内部形式，假定其中的任何 Var 指向由第二个参数指示的关系

名称	返回类型	描述
ux_get_expr(*ux_node_tree*, *relation_oid*, *pretty_bool*)	text	反编译一个表达式的内部形式，假定其中的任何 Var 指向由第二个参数指示的关系
ux_get_functiondef(*func_oid*)	text	获得一个函数或过程的定义
ux_get_function_arguments(*func_oid*)	text	获得一个函数或过程定义的参数列表（带有默认值）
ux_get_function_identity_arguments(*func_oid*)	text	获得标识一个函数或过程的参数列表（不带默认值）
ux_get_function_result(*func_oid*)	text	获得函数的RETURNS子句（对过程返回空）
ux_get_indexdef(*index_oid*)	text	获得索引的CREATE INDEX命令
ux_get_indexdef(*index_oid*, *column_no*, *pretty_bool*)	text	获得索引的CREATE INDEX命令，或者当*column_no*为非零时只得到一个索引列的定义
ux_get_keywords()	setof record	获得 SQL 关键字的列表及其分类
ux_get_ruledef(*rule_oid*)	text	获得规则的CREATE RULE命令
ux_get_ruledef(*rule_oid*, *pretty_bool*)	text	获得规则的CREATE RULE命令
ux_get_serial_sequence(*table_name*, *column_name*)	text	获得一个序列列或标识列使用的序列的名称
ux_get_statisticsobjdef(*statobj_oid*)	text	为扩展的统计信息对象得到CREATE STATISTICS命令
ux_get_triggerdef(*trigger_oid*)	text	获得触发器的CREATE [CONSTRAINT] TRIGGER命令
ux_get_triggerdef(*trigger_oid*, *pretty_bool*)	text	获得触发器的CREATE [CONSTRAINT] TRIGGER命令
ux_get_userbyid(*role_oid*)	name	获得给定 OID 指定的角色名
ux_get_viewdef(*view_name*)	text	获得视图或物化视图的底层SELECT命令（已废弃）
ux_get_viewdef(*view_name*, *pretty_bool*)	text	获得视图或物化视图的底层SELECT命令（已废弃）
ux_get_viewdef(*view_oid*)	text	获得视图或物化视图的底层SELECT命令
ux_get_viewdef(*view_oid*, *pretty_bool*)	text	获得视图或物化视图的底层SELECT命令
ux_get_viewdef(*view_oid*, *wrap_column_int*)	text	获得视图或物化视图的底层SELECT命令；带域的行被包装成指定的列数，并隐含了优质打印
ux_index_column_has_property(*index_oid*, *column_no*, *prop_name*)	boolean	测试一个索引列是否有指定的性质
ux_index_has_property(*index_oid*, *prop_name*)	boolean	测试一个索引是否有指定的性质

名称	返回类型	描述
ux_indexam_has_property(*am_oid*, *prop_name*)	boolean	测试一个索引访问方法是否有指定的性质
ux_options_to_table(*reloptions*)	setof record	获得存储选项的名称/值对的集合
ux_tablespace_databases(*tablespace_oid*)	setof oid	获得在该表空间中有对象的数据库的 OID 的集合
ux_tablespace_location(*tablespace_oid*)	text	获得这个表空间所在的文件系统的路径
ux_typeof(*any*)	regtype	获得任意值的数据类型
collation for (*any*)	text	获得该参数的排序规则
to_regclass(*rel_name*)	regclass	得到指定关系的 OID
to_regproc(*func_name*)	regproc	得到指定函数的 OID
to_regprocedure(*func_name*)	regprocedure	得到指定函数的 OID
to_regoper(*operator_name*)	regoper	得到指定操作符的 OID
to_regoperator(*operator_name*)	regoperator	得到指定操作符的 OID
to_regtype(*type_name*)	regtype	得到指定类型的 OID
to_regnamespace(*schema_name*)	regnamespace	得到指定模式的 OID
to_regrole(*role_name*)	regrole	得到指定角色的 OID

format_type返回一个数据类型的 SQL 名称，它由它的类型 OID 标识并且可能是一个类型修饰符。如果不知道相关的修饰符，则为类型修饰符传递 NULL。

ux_get_keywords返回一组记录描述服务器识别的 SQL 关键字。*word*列包含关键字。*catcode*列包含一个分类码：U为未被预定，C 为列名，T类型或函数名，R为预留。*catdesc*列包含一个可能本地化的描述分类的字符串。

ux_get_constraintdef、ux_get_indexdef、ux_get_ruledef、ux_get_statisticsobjdef和ux_get_triggerdef分别重建一个约束、索引、规则、扩展统计对象或触发器的创建命令（注意这是一个反编译的重构，而不是命令的原始文本）。ux_get_expr反编译一个表达式的内部形式，例如一个列的默认值。在检查系统目录内容时有用。如果表达式可能包含 Var，在第二个参数中指定它们引用的关系的 OID；如果不会出现 Var，第二个参数设置为 0 即可。ux_get_viewdef重构定义一个视图的SELECT查询。这些函数的大部分都有两种变体，一种可以可选地“优质打印”结果。在转出目的中避免使用优质打印输出。为优质打印参数传递假将得到和不带该参数的变体相同的结果。

ux_get_functiondef为一个函数返回一个完整的CREATE OR REPLACE FUNCTION语句。ux_get_function_arguments返回一个函数的参数列表，形式按照它们出现在CREATE FUNCTION中的那样。ux_get_function_result类似地返回函数的合适的RETURNS子句。ux_get_function_identity_arguments返回标识一个函数必要的参数列表，形式和它们出现在ALTER FUNCTION中的一样。这种形式忽略默认值。

ux_get_serial_sequence返回与一个列相关联的序列的名称，如果与列相关联的序列则返回 NULL。如果该列是一个标识列，相关联的序列是为该标识列内部创建的序列。对于使用序列类型之一（serial、smallserial、bigserial）创建的列，它是为那个序列列定义创建的序列。在后一种情况中，这种关联可以用ALTER SEQUENCE OWNED BY修改或者移除（该函数可能应该已经被ux_get_owned_sequence调用，它当前的名称反映了它通常被serial或bigserial列使用）。第一个输入参数是一个带可选模式的表名，第二个参数是一个列名。因为第一个参数可能是一个模

式和表，它不能按照一个双引号包围的标识符来对待，意味着它默认情况下是小写的。而第二个参数只是一个列名，将被当作一个双引号包围的来处理并且会保留其大小写。函数返回的值会被适当地格式化以便传递给序列函数（参见第 6.16 节 “序列操作函数”）。一种典型的用法是为标识列或者序列列读取当前值，例如：

SELECT currval(ux_get_serial_sequence('sometable', 'id'));

ux_get_userbyid抽取给定 OID 的角色的名称。

ux_index_column_has_property、ux_index_has_property和ux_indexam_has_property返回指定的索引列、索引或者索引访问方法是否具有指定性质。如果性质的名称找不到或者不适用于特定的对象，亦或者 OID 或者列号不表示合法的对象，则返回NULL。列的性质可参见表 6.77 “索引列属性”，索引的性质可参见表 6.78 “索引性质”，访问方法的性质可参见表 6.79 “索引访问方法性质”（注意扩展访问方法可以为其索引定义额外的性质）。

表 6.77. 索引列属性

名称	描述
asc	在向前扫描时列是按照升序排列吗？
desc	在向前扫描时列是按照降序排列吗？
nulls_first	在向前扫描时列排序会把空值排在前面吗？
nulls_last	在向前扫描时列排序会把空值排在最后吗？
orderable	列具有已定义的排序顺序吗？
distance_orderable	列能否通过一个“distance”操作符（例如ORDER BY col <-> constant）有序地扫描？
returnable	列值是否可以通过一次只用索引扫描返回？
search_array	列是否天然支持col = ANY(array)搜索？
search_nulls	列是否支持IS NULL和IS NOT NULL搜索？

表 6.78. 索引性质

名称	描述
clusterable	索引是否可以用于CLUSTER命令？
index_scan	索引是否支持普通扫描（非位图）？
bitmap_scan	索引是否支持位图扫描？
backward_scan	在扫描中扫描方向能否被更改（为了支持游标上无需物化的FETCH BACKWARD）？

表 6.79. 索引访问方法性质

名称	描述
can_order	访问方法是否支持ASC、DESC以及CREATE INDEX中的有关关键词？
can_unique	访问方法是否支持唯一索引？
can_multi_col	访问方法是否支持多列索引？

名称	描述
can_exclude	访问方法是否支持排除约束？
can_include	访问方法是否支持CREATE INDEX的INCLUDE子句？

当传入ux_class.*reloptions*或ux_attribute.*attoptions*时，ux_options_to_table返回存储选项名称/值对（*option_name*/*option_value*）的集合。

ux_tablespace_databases允许一个表空间被检查。它返回一组数据库的 OID，这些数据库都有对象存储在该表空间中。如果这个函数返回任何行，则该表空间为非空并且不能被删除。为了显示该表空间中的指定对象，将需要连接到ux_tablespace_databases标识的数据库并且查询它们的ux_class目录。

ux_typeof返回传递给它的值的数据类型的 OID。这在检修或者动态构建 SQL 查询时有用。函数被声明为返回regtype，它是一个 OID 别名类型（见第 5.19 节 “对象标识符类型”；这表明它和一个用于比较目的的 OID 相同，但是作为一个类型名称显示。例如：

```
SELECT ux_typeof(33);

 ux_typeof
-----------
 integer
(1 row)

SELECT typlen FROM ux_type WHERE oid = ux_typeof(33);
 typlen
--------
      4
(1 row)
```

表达式collation for返回传递给它的值的排序规则。例子：

```
SELECT collation for (description) FROM ux_description LIMIT 1;
 ux_collation_for
------------------
 "default"
(1 row)

SELECT collation for ('foo' COLLATE "de_DE");
 ux_collation_for
------------------
 "de_DE"
(1 row)
```

值可能被加上引号并且变成模式限定的。如果从参数表达式得不到排序规则，则返回一个空值。如果参数不是一个可排序的数据类型，则抛出一个错误。

to_regclass、to_regproc、to_regprocedure,
to_regoper、to_regoperator、to_regtype、to_regnamespace和to_regrole函数把关系、函数、操作符、类型、模式和角色的名称（以text给出）分别转换
成、regclass、regproc、regprocedure、regoper、regoperator、regtype、regnamespace和

regrole对象。这些函数与 text 转换的不同在于它们不接受数字 OID，并且在名称无法找到时不会抛出错误而是返回空。对于to_regproc和to_regoper，如果给定名称匹配多个对象时返回空。

表 6.80 “对象信息和定位函数”列出了与数据库对象 标识和定位有关的函数。

表 6.80. 对象信息和定位函数

名称	返回类型	描述
ux_describe_object(*catalog_id*, *object_id*, *object_sub_id*)	text	得到一个数据库对象的描述
ux_identify_object(*catalog_id* oid, *object_id* oid, *object_sub_id* integer)	*type* text, *schema* text, *name* text, *identity* text	得到一个数据库对象的标识
ux_identify_object_as_address (*catalog_id* oid, *object_id* oid, *object_sub_id* integer)	*type* text, *name* text[], *args* text[]	得到一个数据库对象的地址的外部表示
ux_get_object_address(*type* text, *name* text[], *args* text[])	*class_id* oid, *object_id* oid, *object_sub_id* int32	从一个数据库对象的内部表示得到它的地址

ux_describe_object返回由目录OID、对象OID以及子对象ID（例如表中的一个列号，当子对象引用了一整个对象时其ID为零）指定的数据库对象的文本描述。这种描述是为 了人类可读的，并且可能是被翻译过的，具体取决于服务器配置。这有助于确定一 个存储在ux_depend目录中的对象的标识。

ux_identify_object返回一行，其中包含有足以唯一标识 由目录 OID、对象 OID 和一个（可能为零的）子对象 ID 指定的数据库对象的信息。 该信息是共机器读取的，并且不会被翻译。*type*标识数据库对象 的类型；*schema*是该对象所属的模式名，如果对象类型不属于 模式则为NULL；如果名称（加上方案名，如果相关）足以唯一标识对象，则*name*就是对象的名称（必要时会被加上引号），否则为NULL；*identity*是完整的对象标识， 它会表现为与对象类型相关的精确格式，并且如有必要，该格式中的每个部分都会 被模式限定。

ux_identify_object_as_address返回一行，其中包含有 足以唯一标识由目录 OID、对象 OID 和一个（可能为零的）子对象 ID 指定的数据 库对象的信息。返回的信息独立于当前服务器，也就是说，它可以被用来在另一个 服务器中标识一个具有相同命名的对象。*type*标识数据库对象 的类型；*object_names*和*object_args*是文本数组，它们一起 构成了对对象的引用。这三个值可以被传递给 ux_get_object_address以获得该对象的内部地址。这个函数是 ux_get_object_address的逆函数。

ux_get_object_address返回一行，其中包含有足以唯一 标识由类型、对象名和参数数组指定的数据库对象的信息。返回值可以被用在诸如 ux_depend等系统目录中并且可以被传递给ux_identify_object或ux_describe_object等其他 系统函数。*class_id*是包含该对象的系统目录 OID；*objid*是对象本身的 OID，而 *objsubid*是子对象 ID，如果没有则为零。这个函数是ux_identify_object_as_address的逆函数。

表 6.81 “注释信息函数”中展示的函数抽取注释，注释是由COMMENT(7)命令在以前存储的。如果对指定参数找不到注释，则返回空值。

表 6.81. 注释信息函数

名称	返回类型	描述
col_description(*table_oid*, *column_number*)	text	为一个表列获得注释

名称	返回类型	描述
obj_description(*object_oid*, *catalog_name*)	text	为一个数据库对象获得注释
obj_description(*object_oid*)	text	为一个数据库对象获得注释（已被废弃）
shobj_description(*object_oid*, *catalog_name*)	text	为一个共享数据库对象获得注释

col_description为一个表列返回注释，该表列由所在表的 OID 和它的列号指定（obj_description不能被用在表列，因为表列没有自己的 OID）。

obj_description的双参数形式返回一个由其 OID 和所在系统目录名称指定的数据库对象的注释。例如，obj_description(123456,'ux_class')将会检索出 OID 为123456的表的注释。obj_description的单参数形式只要求对象 OID。它已经被废弃，因为无法保证 OID 在不同系统目录之间是唯一的；这样可能会返回错误的注释。

shobj_description用起来就像obj_description，但是前者是用于检索共享对象上的注释。某些系统目录对于一个集簇中的所有数据库是全局的，并且其中的对象的描述也是全局存储的。

表 6.82 “事务 ID 和快照”中展示的函数以一种可导出的形式提供了服务器事务信息。这些函数的主要用途是判断在两个快照之间哪些事务被提交。

表 6.82. 事务 ID 和快照

名称	返回类型	描述
txid_current()	bigint	获得当前事务 ID，如果当前事务没有 ID 则分配一个新的 ID
txid_current_if_assigned()	bigint	与txid_current()相同，但是在事务没有分配ID时是返回空值而不是分配一个新的事务ID
txid_current_snapshot()	txid_snapshot	获得当前快照
txid_snapshot_xip(*txid_snapshot*)	setof bigint	获得快照中正在进行的事务 ID
txid_snapshot_xmax(*txid_snapshot*)	bigint	获得快照的xmax
txid_snapshot_xmin(*txid_snapshot*)	bigint	获得快照的xmin
txid_visible_in_snapshot(*bigint*, *txid_snapshot*)	boolean	事务 ID 在快照中可见吗？（不能用于子事务 ID）
txid_status(*bigint*)	text	报告给定事务的状态：committed、aborted、in progress，如果事务ID太老则为空值

内部事务 ID 类型（xid）是 32 位宽并且每 40 亿个事务就会回卷。但是，这些函数导出一种 64 位格式，它被使用一个“世代”计数器，这样在一个安装的生命期内不会回卷。这些函数使用的数据类型txid_snapshot存储了在一个特定时刻有关事务 ID 可见性的信息。它的成分在表 6.83 “快照成分”中描述。

表 6.83. 快照成分

名称	描述
xmin	仍然活动的最早的事务 ID （txid）。所有更早的事务要么已经被提交并且可见，要么已经被回滚并且死亡。
xmax	第一个还未分配的 txid。所有大于等于它的 txid 在快照的时刻还没有开始，并且因此是不可见的。
xip_list	在快照时刻活动的 txid。这个列表只包括那些位于xmin和xmax之间的活动 txid；可能有活动的超过xmax的 txid。一个满足xmin <= txid < xmax并且不在这个列表中的 txid 在快照时刻已经结束，并且因此根据其提交状态要么可见要么死亡。该列表不包括子事务的 txid。

txid_snapshot的文本表示是*xmin*:*xmax*:*xip_list*。例如10:20:10,14,15表示xmin=10, xmax=20, xip_list=10, 14, 15。

txid_status(bigint)报告一个近期事务的提交状态。当一个应用和数据库服务器的连接在COMMIT正在进行时断开，应用可以用它来判断事务是提交了还是中止了。一个事务的状态将被报告为in progress、committed或者aborted，前提是该事务的发生时间足够近，这样系统才会保留它的提交状态。如果事务太老，则系统中不会留下对该事务的引用并且提交状态信息也已经被抛弃，那么这个函数将会返回NULL。注意，预备事务会被报告为in progress，如果应用需要判断该txid是否是一个预备事务，应用必须检查ux_prepared_xacts。

表 6.84 “已提交事务信息”中展示的函数提供了有关于 已经提交事务的信息。这些函数主要提供有关事务何时被提交的信息。只有当 track_commit_timestamp配置选项被启用时它们才能提供有用的数据，并且只对已提交事务提供数据。

表 6.84. 已提交事务信息

名称	返回类型	描述
ux_xact_commit_timestamp(*xid*)	timestamp with time zone	得到一个事务的提交时间戳
ux_last_committed_xact()	*xid* xid, *timestamp* timestamp with time zone	得到最后一个已提交事务的事务 ID 和提交时间戳

表 6.85 “控制数据函数”中所展示的函数能打印initdb期间初始化的信息，例如系统目录版本。它们也能显示有关预写式日志和检查点处理的信息。这些信息是集簇范围内的，不与任何特定的一个数据库相关。对于同一种来源，它们返回和ux_controldata大致相同的信息，不过其形式更适合于SQL函数。

表 6.85. 控制数据函数

名称	返回类型	描述
ux_control_checkpoint()	record	返回有关当前检查点状态的信息。
ux_control_system()	record	返回有关当前控制文件状态的信息。

名称	返回类型	描述
ux_control_init()	record	返回有关集簇初始化状态的信息。
ux_control_recovery()	record	返回有关恢复状态的信息。

ux_control_checkpoint返回一个表 6.86 “ux_control_checkpoint列”中所示的记录

表 6.86. **ux_control_checkpoint**列

列名	数据类型
checkpoint_location	ux_lsn
redo_lsn	ux_lsn
redo_wal_file	text
timeline_id	integer
prev_timeline_id	integer
full_page_writes	boolean
next_xid	text
next_oid	oid
next_multixact_id	xid
next_multi_offset	xid
oldest_xid	xid
oldest_xid_dbid	oid
oldest_active_xid	xid
oldest_multi_xid	xid
oldest_multi_dbid	oid
oldest_commit_ts_xid	xid
newest_commit_ts_xid	xid
checkpoint_time	timestamp with time zone

ux_control_system返回一个表 6.87 “ux_control_system列”中所示的记录

表 6.87. **ux_control_system**列

列名	数据类型
ux_control_version	integer
catalog_version_no	integer
system_identifier	bigint
ux_control_last_modified	timestamp with time zone

ux_control_init返回一个表 6.88 “ux_control_init列”中所示的记录

表　6.88. **ux_control_init**列

列名	数据类型
max_data_alignment	integer
database_block_size	integer
blocks_per_segment	integer
wal_block_size	integer
bytes_per_wal_segment	integer
max_identifier_length	integer
max_index_columns	integer
max_toast_chunk_size	integer
large_object_chunk_size	integer
float4_pass_by_value	boolean
float8_pass_by_value	boolean
data_page_checksum_version	integer

ux_control_recovery返回一个表　6.89　“ux_control_recovery列”中所示的记录

表　6.89. **ux_control_recovery**列

列名	数据类型
min_recovery_end_lsn	ux_lsn
min_recovery_end_timeline	integer
backup_start_lsn	ux_lsn
backup_end_lsn	ux_lsn
end_of_backup_record_required	boolean

6.25.1.　sys_context

- 功能

  sys_context函数用于获取环境上下文信息或判断当前会话角色。

- 函数

  SYS_CONTEXT('context','parameter'[, length])

- 参数

  表　6.90.　SYS_CONTEXT参数说明

参数	说明
context	context名不区分大小写,最大长度128字节;目前支持USERENV和SYS_SESSION_ROLES两种上下文，当值为USERENV时，返回名为parameter的系统参数的值；当值为SYS_SESSION_ROLES

参数	说明
	时，判断parameter是否为在当前会话中启用的角色，是则返回true，否则返回false。
parameter	parameter为context中的属性，最大长度64字节。 • 当context为USERENV时，parameter支持属性参见表 6.91 “parameter属性说明”。 • 当context为SYS_SESSION_ROLES时，parameter为当前会话中启用的角色名。
lenth	返回值的最大长度，默认值为256字节。有效值为1-4000，如果指定的length值不在有效值范围内，则忽略该值，使用默认值256。

表 6.91. parameter属性说明

属性	说明
CURRENT_SCHEMA	当前活动的默认模式的名称。
CURRENT_SCHEMAID	当前活动的默认模式的标识符。
CURRENT_USER	权限处于活动状态的数据库用户的名称。
CURRENT_USERID	权限处于活动状态的数据库用户的标识符。
DB_NAME	初始化参数DB_NAME中指定的数据库名称。
HOST	客户端连接的主机的名称，只在guc参数log_hostname开启时才能返回通过ip地址连接的客户端的主机名。
INSTANCE_NAME	实例的名称。
IP_ADDRESS	连接客户端的机器的 IP 地址。
ISDBA	如果用户已通过操作系统或密码文件验证为具有 DBA 特权，则返回TRUE。
LANG	您的会话当前使用的语言的缩写。
LANGUAGE	您的会话当前使用的语言和地区，以及数据库字符集，格式如下：language_territory.characterset。
NETWORK_PROTOCOL	用于通信的网络协议。
SERVER_HOST	运行实例的机器的主机名。
SESSION_USER	会话用户的名称（登录的用户）。
SESSION_USERID	会话用户（登录的用户）的标识符。
SESSIONID	审计会话标识符，只在安全模式和兼容安全模式下有返回值。
SID	会话 ID。

- 返回值

  返回值类型为text，默认最大长度为256字节。

- 示例

  查询当前数据库的数据库名。

  SELECT sys_context('USERENV', 'DB_NAME');

  查询客户端连接的主机的名称并限制返回值长度为10字节。

  SELECT sys_context('USERENV', 'HOST',10);

  查询用户uxdb是否为在当前会话中启用的角色。

  SELECT sys_context('SYS_SESSION_ROLES', 'uxdb');

6.26. 系统管理函数

这一节描述的函数被用来控制和监视一个UXDB安装。

6.26.1. 配置设定函数

表 6.92 “配置设定函数”展示了那些可以用于查询以及修改运行时配置参数的函数。

表 6.92. 配置设定函数

名称	返回类型	描述
current_setting(*setting_name* [, *missing_ok*])	text	获得设置的当前值
set_config(*setting_name*, *new_value*, *is_local*)	text	设置一个参数并返回新值

current_setting得到*setting_name*设置的当前值。它对应于SQL命令**SHOW**。一个例子：

```
SELECT current_setting('datestyle');

 current_setting
-----------------
 ISO, MDY
(1 row)
```

如果没有名为*setting_name*的设置，除非提供*missing_ok*并且其值为true，current_setting会抛出错误。

set_config将参数*setting_name*设置为*new_value*。如果 *is_local*设置为true，那么新值将只应用于当前事务。 如果希望新值应用于当前会话，那么应该使用false。 它等效于 SQL 命令 **SET**。例如：

```
SELECT set_config('log_statement_stats', 'off', false);

 set_config
------------
```

```
 off
(1 row)
```

6.26.2. 服务器信号函数

在表 6.93 “服务器信号函数”中展示的函数向其它服务器进程发送控制信号。默认情况下这些函数只能被超级用户使用，但是如果需要，可以利用GRANT把访问特权授予给其他用户。

表 6.93. 服务器信号函数

名称	返回类型	描述
ux_cancel_backend(*pid* int)	boolean	取消一个后端的当前查询。如果调用角色是被取消后端的拥有者角色的成员或者调用角色已经被授予ux_signal_backend，这也是允许的，不过只有超级用户才能取消超级用户的后端。
ux_reload_conf()	boolean	导致服务器进程重载它们的配置文件
ux_rotate_logfile()	boolean	切换服务器的日志文件
ux_terminate_backend(*pid* int)	boolean	中止一个后端。如果调用角色是被取消后端的拥有者角色的成员或者调用角色已经被授予ux_signal_backend，这也是允许的，不过只有超级用户才能取消超级用户的后端。

这些函数中的每一个都在成功时返回true，并且在失败时返回false。

ux_cancel_backend和ux_terminate_backend向由进程 ID 标识的后端进程发送信号（分别是SIGINT或SIGTERM）。一个活动后端的进程 ID可以从ux_stat_activity视图的*pid*列中找到，或者通过在服务器上列出uxdb进程（在 Unix 上使用ps或者在Windows上使用任务管理器）得到。一个活动后端的角色可以在ux_stat_activity视图的*usename*列中找到。

ux_reload_conf给服务器发送一个SIGHUP信号， 导致所有服务器进程重载配置文件。

ux_rotate_logfile给日志文件管理器发送信号，告诉它立即切换到一个新的输出文件。这个函数只有在内建日志收集器运行时才能工作，因为否则就不存在日志文件管理器子进程。 subprocess.

6.26.3. 备份控制函数

表 6.94 “备份控制函数”中展示的函数可以辅助制作在线备份。这些函数不能在恢复期间执行（非排他性的ux_start_backup，非排他性的ux_stop_backup，ux_is_in_backup、ux_backup_start_time和ux_wal_lsn_diff除外）。

表 6.94. 备份控制函数

名称	返回类型	描述
ux_create_restore_point(*name* text)	ux_lsn	为执行恢复创建一个命名点（默认只限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）

名称	返回类型	描述
ux_current_wal_flush_lsn()	ux_lsn	得到当前的预写式日志刷写位置
ux_current_wal_insert_lsn()	ux_lsn	获得当前预写式日志插入位置
ux_current_wal_lsn()	ux_lsn	获得当前预写式日志写入位置
ux_start_backup(*label* text [, *fast* boolean [, *exclusive* boolean]])	ux_lsn	准备执行在线备份（默认只限于超级用户或者复制角色，但是可以授予其他用户 EXECUTE 特权来执行该函数）
ux_stop_backup()	ux_lsn	完成执行排他的在线备份（默认只限于超级用户或者复制角色，但是可以授予其他用户 EXECUTE 特权来执行该函数）
ux_stop_backup(*exclusive* boolean)	setof record	结束执行排他或者非排他的在线备份 （默认只限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）
ux_is_in_backup()	bool	如果一个在线排他备份仍在进行中则为真。
ux_backup_start_time()	timestamp with time zone	获得一个进行中的在线排他备份的开始时间。
ux_switch_wal()	ux_lsn	强制切换到一个新的预写式日志文件（默认只限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）
ux_walfile_name(*lsn* text)	ux_lsn	转换预写式日志位置字符串为文件名
ux_walfile_name_offset(*lsn* text)	ux_lsn, integer	转换预写式日志位置字符串为文件名以及文件内的十进制字节偏移
ux_wal_lsn_diff(*lsn* ux_lsn, *lsn* ux_lsn)	numeric	计算两个预写式日志位置间的差别

ux_start_backup接受一个参数，这个参数可以是备份的任意用户定义的标签（通常这是备份转储文件将被存储的名字）。当被用在排他模式中时，该函数向数据库集簇的数据目录写入一个备份标签文件（backup_label）和一个表空间映射文件（tablespace_map，如果在ux_tblspc/目录中有任何链接），执行一个检查点，然后以文本方式返回备份的起始预写式日志位置。用户可以忽略这个结果值，但是为了可能需要的场合我们还是提供该值。 当在非排他模式中使用时，这些文件的内容会转而由ux_stop_backup函数返回，并且应该由调用者写入到备份中去。

```
uxdb=# select ux_start_backup('label_goes_here');
 ux_start_backup
-----------------
 0/D4445B8
(1 row)
```

第二个参数是可选的，其类型为boolean。如果为true，它指定尽快执行ux_start_backup。这会强制一个立即执行的检查点，它会导致 I/O 操作的峰值，拖慢任何并发执行的查询。

在一次排他备份中，ux_stop_backup会移除标签文件以及ux_start_backup创建的tablespace_map文件（如果存在）。在一次非排他备份中，backup_label和tablespace_map的内容会包含在该函数返回的结果中，并且应该被写入到该备份的文件中（这些内容不在数据目录中）。有一个可选的boolean类型的第二参数。如果为假，ux_stop_backup将在备份完成后立即返回而不等待WAL被归档。这种行为仅对独立监控WAL归档的备份软件有用。否则，让备份一致所要求的WAL可能会丢失，进而让备份变得毫无用处。当这个参数被设置为真时，在启用归档的前提下ux_stop_backup将等待WAL被归档，在后备服务器上，这意味只有archive_mode = always时才会等待。如果主服务器上的写活动很低，在主服务器上运行ux_switch_wal以触发一次即刻的段切换会很有用。

当在主服务器上执行时，该函数还在预写式日志归档区里创建一个备份历史文件。这个历史文件包含给予ux_start_backup的标签、备份的起始与终止预写式日志位置以及备份的起始和终止时间。返回值是备份的终止预写式日志位置（同样也可以被忽略）。在记录结束位置之后，当前预写式日志插入点被自动地推进到下一个预写式日志文件，这样结束的预写式日志文件可以立即被归档来结束备份。

ux_switch_wal移动到下一个预写式日志文件，允许当前文件被归档（假定正在使用连续归档）。返回值是在甘冈完成的预写式日志文件中结束预写式日志位置 + 1。如果从上一次预写式日志切换依赖没有预写式日志活动，ux_switch_wal不会做任何事情并且返回当前正在使用的预写式日志文件的开始位置。

ux_create_restore_point创建一个命名预写式日志记录，它可以被用作恢复目标，并且返回相应的预写式日志位置。这个给定的名字可以用于recovery_target_name来指定恢复要进行到的点。避免使用同一个名称创建多个恢复点，因为恢复会停止在第一个匹配名称的恢复目标。

ux_current_wal_lsn以上述函数所使用的相同格式显示当前预写式日志的写位置。类似地，ux_current_wal_insert_lsn显示当前预写式日志插入点，而ux_current_wal_flush_lsn显示当前预写式日志的刷写点。在任何情况下，插入点是预写式日志的“逻辑”终止点，而写入位置是已经实际从服务器内部缓冲区写出的日志的终止点，刷写位置则是被确保写入到持久存储中的日志的终止点。写入位置是可以从服务器外部检查的终止点，对那些关注归档部分完成预写式日志文件的人来说，这就是他们需要的位置。插入和刷写点主要是为了服务器调试目的而存在的。这些都是只读操作并且不需要超级用户权限。

可以使用ux_walfile_name_offset从任何上述函数的结果中抽取相应的预写式日志文件名称以及字节偏移。例如：

```
uxdb=# SELECT * FROM ux_walfile_name_offset(ux_stop_backup());
      file_name        | file_offset
-------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

相似地，ux_walfile_name只抽取预写式日志文件名。当给定的预写式日志位置正好在一个预写式日志文件的边界，这些函数都返回之前的预写式日志文件的名称。这对管理预写式日志归档行为通常是所希望的行为，因为前一个文件是当前需要被归档的最后一个文件。

ux_wal_lsn_diff以字节数计算两个预写式日志位置之间的差别。它可以和ux_stat_replication或表 6.94 “备份控制函数”中其他的函数一起使用来获得复制延迟。

6.26.4. 恢复控制函数

表 6.95 “恢复信息函数”中展示的函数提供有关后备机当前状态的信息。这些函数可以在恢复或普通运行过程中被执行。

表 6.95. 恢复信息函数

名称	返回类型	描述
ux_is_in_recovery()	bool	如果恢复仍在进行中，为真。
ux_last_wal_receive_lsn()	ux_lsn	获得最后一个收到并由流复制同步到磁盘的预写式日志位置。当流复制在进行中时，这将单调增加。如果恢复已经完成，这将保持静止在恢复过程中收到并同步到磁盘的最后一个 WAL 记录。如果流复制被禁用，或者还没有被启动，该函数返回 NULL。
ux_last_wal_replay_lsn()	ux_lsn	获得恢复过程中被重放的最后一个预写式日志位置。当流复制在进行中时，这将单调增加。如果恢复已经完成，这将保持静止在恢复过程中被应用的最后一个 WAL 记录。如果服务器被正常启动而没有恢复，该函数返回 NULL。
ux_last_xact_replay_timestamp()	timestamp with time zone	获得恢复过程中被重放的最后一个事务的时间戳。这是在主机上产生的事务的提交或中止 WAL 记录的时间。如果在恢复过程中没有事务被重放，这个函数返回 NULL。否则，如果恢复仍在进行这将单调增加。如果恢复已经完成，则这个值会保持静止在恢复过程中最后一个被应用的事务。如果服务器被正常启动而没有恢复，该函数返回 NULL。

表 6.96 “恢复控制函数”中展示的函数空值恢复的进程。这些函数只能在恢复过程中被执行。

表 6.96. 恢复控制函数

名称	返回类型	描述
ux_is_wal_replay_paused()	bool	如果恢复被暂停，为真。
ux_promote(*wait* boolean DEFAULT true, *wait_seconds* integer DEFAULT 60)	boolean	促进物理备用服务器。当*wait*设置为true（默认值）时，该函数将等待晋升完成或*wait_seconds*秒，如果晋升成功，则返回true，否则返回false。如果*wait*被设置为false，则会返回true。函数在发送后立即返回true。SIGUSR1给uxmaster触发推广。此功能默认只限于

名称	返回类型	描述
		超级用户，但其他用户可以被授予EXECUTE来运行函数。
ux_wal_replay_pause()	void	立即暂停恢复（默认仅限于超级用户， 但是可以授予其他用户 EXECUTE 特权来执行该函数）。
ux_wal_replay_resume()	void	如果恢复被暂停，重启之（默认仅限于超级用户，但是可以授予其他用户 EXECUTE 特权来执行该函数）。

在恢复被暂停时，不会有进一步的数据库改变被应用。如果在热备模式，所有新的查询将看到数据库的同一个一致快照，并且在恢复被继续之前不会有更多查询冲突会产生。

如果流复制被禁用，暂停状态可以无限制地继续而不出问题。在流复制进行时，WAL 记录将继续被接收，最后将会填满可用的磁盘空间，取决于暂停的持续时间、WAL 的产生率和可用的磁盘空间。

6.26.5. 快照同步函数

UXDB允许数据库会话同步它们的快照。一个快照决定对于正在使用该快照的事务哪些数据是可见的。当两个或者更多个会话需要看到数据库中的相同内容时，就需要同步快照。如果两个会话独立开始其事务，就总是有可能有某个第三事务在两个START TRANSACTION命令的执行之间提交，这样其中一个会话就可以看到该事务的效果而另一个则看不到。

为了解决这个问题，UXDB允许一个事务导出它正在使用的快照。只要导出的事务仍然保持打开，其他事务可以导入它的快照，并且因此可以保证它们可以看到和第一个事务看到的完全一样的数据库视图。但是注意这些事务中的任何一个对数据库所作的更改对其他事务仍然保持不可见，和未提交事务所作的修改一样。因此这些事务是针对以前存在的数据同步，而对由它们自己所作的更改则采取正常的动作。

如表 6.97 “快照同步函数”中所示，快照通过ux_export_snapshot函数导出，并且通过SET TRANSACTION(7)命令导入。

表 6.97. 快照同步函数

名称	返回类型	描述
ux_export_snapshot()	text	保存当前快照并返回它的标识符

函数ux_export_snapshot保存当前的快照并且返回一个text串标识该快照。该字符串必须被传递（到数据库外）给希望导入快照的客户端。直到导出快照的事务的末尾，快照都可以被导入。如果需要，一个事务可以导出多于一个快照。注意这样做只在 READ COMMITTED事务中有用，因为在REPEATABLE READ和更高隔离级别中，事务在它们的生命期中都使用同一个快照。一旦一个事务已经导出了任何快照，它不能使用PREPARE TRANSACTION(7)。

关于如何使用一个已导出快照的细节请见SET TRANSACTION(7).

6.26.6. 复制函数

表 6.98 “复制 SQL 函数”中展示的函数 用于控制以及与复制特性交互。复制原点函数的使用仅限于超级用户。 复制槽的函数只限于超级用户和拥有REPLICATION权限的用户。

很多这些函数在复制协议中都有等价的命令。

第 6.26.3 节 “备份控制函数”、第 6.26.4 节 “恢复控制函数”和第 6.26.5 节 “快照同步函数” 中描述的函数也与复制相关。

表 6.98. 复制 SQL 函数

函数	返回类型	描述
ux_create_physical_replication_slot(*slot_name* name [, *immediately_reserve* boolean])	(*slot_name* name, *xlog_position* ux_lsn)	创建一个新的名为*slot_name*的物理复制槽。第二个参数是可选的，当它为true时，立即为这个物理槽指定要被保留的LSN。否则该LSN会被保留在来自一个流复制客户端的第一个连接上。来自一个物理槽的流改变只可能出现在使用流复制协议时。当可选的第三参数*temporary*被设置为真时，指定那个槽不会被持久地存储在磁盘上并且仅对当前会话的使用有意义。临时槽也会在发生任何错误时被释放。这个函数对应于复制协议命令CREATE_REPLICATION_SLOT ... PHYSICAL。
ux_drop_replication_slot(*slot_name* name)	void	丢弃名为*slot_name*的物理或逻辑复制槽。 和复制协议命令DROP_REPLICATION_SLOT相同。对于逻辑槽，在连接到在其中创建该槽的同一个数据库时，必须调用这个函数。
ux_create_logical_replication_slot(*slot_name* name, *plugin* name)	(*slot_name* name, *lsn* ux_lsn)	使用输出插件*plugin*创建一个名为 *slot_name*的新逻辑（解码）复制槽。当可选的第三参数*temporary*被设置为真时，指定那个槽不会被持久地存储在磁盘上并且仅对当前会话的使用有意义。临时槽也会在发生任何错误时被释放。对这个函数的调用与复制协议命令CREATE_REPLICATION_SLOT ... LOGICAL具有相同的效果。
ux_copy_physical_replication_slot(*src_slot_name* name, *dst_slot_name* name [, *temporary* boolean])	(*slot_name* name, *lsn* ux_lsn)	将一个名为*src_slot_name*的现有物理复制槽复制到一个名为*dst_slot_name*的物理复制槽。被复制的物理槽开始从与源槽相同的LSN开始保

函数	返回类型	描述
		留WAL。*temporary*是可选的。如果省略了*temporary*，则使用与源槽相同的值。
ux_copy_logical_replication_slot(*src_slot_name* name, *dst_slot_name* name [, *temporary* boolean [, *plugin* name]])	(*slot_name* name, *lsn* ux_lsn)	复制一个名为*src_slot_name*的现有逻辑复制槽。到一个名为*dst_slot_name*的逻辑复制槽。同时改变输出插件和持久性。被复制的逻辑槽开始来自与源逻辑槽相同的LSN。这两个*temporary*和*plugin*是可选的。如果省略了*temporary*或*plugin*。使用与源逻辑槽相同的值。
ux_logical_slot_get_changes(*slot_name* name, *upto_lsn* ux_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* ux_lsn, *xid* xid, *data* text)	返回槽*slot_name*中的改变，从上一次已经被消费的点开始返回。如果*upto_lsn*和*upto_nchanges*为NULL，逻辑解码将一直继续到WAL的末尾。如果*upto_lsn*为非NULL，解码将只包括那些在指定LSN之前提交的事务。如果*upto_nchanges*为非NULL，解码将在其产生的行数超过指定值后停止。不过要注意，被返回的实际行数可能更大，因为对这个限制的检查只会在增加了解码每个新的提交事务产生的行之后进行。
ux_logical_slot_peek_changes(*slot_name* name, *upto_lsn* ux_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* text, *xid* xid, *data* text)	行为就像ux_logical_slot_get_changes()函数，不过改变不会被消费，即在未来的调用中还会返回这些改变。
ux_logical_slot_get_binary_changes(*slot_name* name, *upto_lsn* ux_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* ux_lsn, *xid* xid, *data* bytea)	行为就像ux_logical_slot_get_changes()函数，不过改变会以bytea返回。
ux_logical_slot_peek_binary_changes(*slot_name* name, *upto_lsn* ux_lsn, *upto_nchanges* int, VARIADIC *options* text[])	(*lsn* ux_lsn, *xid* xid, *data* bytea)	行为就像ux_logical_slot_get_changes()函数，不过改变会以bytea返回并且这些改变不会被消费，即在未来的调用中还会返回这些改变。

函数	返回类型	描述
ux_replication_slot_advance(*slot_name* name, *upto_lsn* ux_lsn)	(*slot_name* name, *end_lsn* ux_lsn) bool	将复制槽的当前确认的位置提前到名为*slot_name*的复制槽的当前确认位置。该槽不会向后移动，也不会移动到当前插入位置之外。返回该槽的名称和它被推进到的真实位置。 如果有任何推进，则在后续的检查点中写出更新后的槽的信息。如果发生崩溃，该槽位可能会返回到之前的位置。
ux_replication_origin_create(*node_name* text)	oid	用给定的外部名称创建一个复制源，并且返回分配给它的内部 id。
ux_replication_origin_drop(*node_name* text)	void	删除一个之前创建的复制源，包括任何相关的重放进度。
ux_replication_origin_oid(*node_name* text)	oid	用名称查找复制源并且返回内部 id。如果没有找到则抛出错误。
ux_replication_origin_session_setup(*node_name* text)	void	把当前会话标记为正在从给定的源进行重放，允许重放进度被跟踪。使用ux_replication_origin_session_reset可以取消 标记。只有之前没有源被配置时才能使用。
ux_replication_origin_session_reset()	void	取消ux_replication_origin_session_setup()的效果。
ux_replication_origin_session_is_setup()	bool	当前会话中是否已经配置了一个复制源？
ux_replication_origin_session_progress(*flush* bool)	ux_lsn	返回当前会话中配置的复制源的重放位置。参数*flush*决定对应的本地事务是否被确保 已经刷入磁盘。
ux_replication_origin_xact_setup(*origin_lsn* ux_lsn, *origin_timestamp* timestamptz)	void	标记当前事务为正在重放一个已经在给定的LSN 和时间戳提交的事务。只有当之前已经用ux_replication_origin_session_setup()配置过 一个复制源时才能被调用。
ux_replication_origin_xact_reset()	void	取消ux_replication_origin_xact_setup()的效果。
ux_replication_origin_advance(*node_name* text, *pos* ux_lsn)	void	把给定节点的复制进度设置为给定的位置。这主要用于配置更改或者类似 操作之后设置初始位置或者新位

函数	返回类型	描述
		置。注意这个函数的不当使用可能会导致 不一致的复制数据。
ux_replication_origin_progress(*node_name* text, *flush* bool)	ux_lsn	返回给定复制元的重放位置。参数 *flush*决定对应的本地事务是否被确保 已经刷入磁盘。
ux_logical_emit_message(*transactional* bool, *prefix* text, *content* text)	ux_lsn	发出文本形式的逻辑解码消息。这可以被用来通过 WAL 向逻辑解码插件传递一般消息。参数*transactional*指定该消息是否应该是当前事务的一部分或者当逻辑解码读到该记录时该消息是否应该被立刻写入并且解码。*prefix*是逻辑解码插件用来识别它们感兴趣的消息的文本前缀。*content*是消息的文本。
ux_logical_emit_message(*transactional* bool, *prefix* text, *content* bytea)	ux_lsn	发出二进制逻辑解码消息。这可以被用来通过WAL向逻辑解码插件传递一般性消息。参数*transactional*指定该消息是否应该成为当前事务的一部分或者是否应该在逻辑解码过程读到该记录时立刻进行写入和解码。参数*prefix*是一个逻辑解码插件使用的文本前缀，逻辑解码插件用它来识别感兴趣的消息。参数*content*是消息的二进制内容。

6.26.7. 数据库对象管理函数

表 6.99 “数据库对象尺寸函数”中展示的函数计算数据库对象使用的磁盘空间。

表 6.99. 数据库对象尺寸函数

名称	返回类型	描述
ux_column_size(any)	int	存储一个特定值（可能压缩过）所需的字节数
ux_database_size(oid)	bigint	指定 OID 的数据库使用的磁盘空间
ux_database_size(name)	bigint	指定名称的数据库使用的磁盘空间
ux_indexes_size(regclass)	bigint	附加到指定表的索引所占的总磁盘空间

名称	返回类型	描述
ux_relation_size(*relation* regclass, *fork* text)	bigint	指定表或索引的指定分叉（'main'、'fsm'、'vm'或'init'）使用的磁盘空间
ux_relation_size(*relation* regclass)	bigint	ux_relation_size(..., 'main')的简写
ux_size_bytes(text)	bigint	把人类可读格式的带有单位的尺寸转换成字节数
ux_size_pretty(bigint)	text	将表示成一个 64位整数的字节尺寸转换为带尺寸单位的人类可读格式
ux_size_pretty(numeric)	text	将表示成一个数字值的字节尺寸转换为带尺寸单位的人类可读格式
ux_table_size(regclass)	bigint	被指定表使用的磁盘空间，排除索引（但包括 TOAST、空闲空间映射和可见性映射）
ux_tablespace_size(oid)	bigint	指定 OID 的表空间使用的磁盘空间
ux_tablespace_size(name)	bigint	指定名称的表空间使用的磁盘空间
ux_total_relation_size(regclass)	bigint	指定表所用的总磁盘空间，包括所有的索引和TOAST数据

ux_column_size显示用于存储任意独立数据值的空间。

ux_total_relation_size接受一个表或 TOAST 表的 OID 或名称，并返回该表所使用的总磁盘空间，包括所有相关的索引。这个函数等价于ux_table_size + ux_indexes_size。

ux_table_size接受一个表的 OID 或名称，并返回该表所需的磁盘空间，但是排除索引（TOAST 空间、空闲空间映射和可见性映射包含在内）

ux_indexes_size接受一个表的 OID 或名称，并返回附加到该表的所有索引所使用的全部磁盘空间。

ux_database_size以及ux_tablespace_size接受数据库或者表空间的OID或者名称，并且返回它们使用的磁盘空间。要使用ux_database_size，用户必须具有指定数据库上的CONNECT权限（默认情况下已经被授予）或者是ux_read_all_stats角色的一个成员。要使用ux_tablespace_size，用户必须具有指定表空间上的CREATE权限或者是ux_read_all_stats角色的一个成员，除非该表空间是当前数据库的默认表空间。

ux_relation_size接受一个表、索引或 TOAST 表的 OID 或者名称，并且返回那个关系的一个分叉所占的磁盘空间的字节尺寸（注意 对于大部分目的，使用更高层的函数ux_total_relation_size 或者ux_table_size会更方便，它们会合计所有分叉的尺寸）。 如果只得到一个参数，它会返回该关系的主数据分叉的尺寸。提供第二个参数 可以指定要检查哪个分叉：

- 'main'返回该关系主数据分叉的尺寸。
- 'fsm'返回与该关系相关的空闲空间映射的尺寸。
- 'vm'返回与该关系相关的可见性映射的尺寸。
- 'init'返回与该关系相关的初始化分叉（如 果有）的尺寸。

ux_size_pretty可以用于把其它函数之一的结果格式化成一种人类易读的格式，可以根据情况使用字节、kB、MB、GB 或者 TB。

ux_size_bytes可以被用来从人类可读格式的字符串得到其中所表示的字节数。其输入可能带有的单位包括字节、kB、MB、GB 或者 TB，并且对输入进行解析时是区分大小写的。如果没有指定单位，会假定单位为字节。

> **注意**
>
> 函数ux_size_pretty和ux_size_bytes所使用的单位 kB、MB、GB 和 TB 是用 2 的幂而不是 10 的幂来定义，因此 1kB 是 1024 字节，1MB 是 1024^2 = 1048576 字节，以此类推。

上述操作表和索引的函数接受一个regclass参数，它是该表或索引在ux_class系统目录中的 OID。不必手工去查找该 OID，因为regclass数据类型的输入转换器会为代劳。只写包围在单引号内的表名，这样它看起来像一个文字常量。为了与普通SQL名称的处理相兼容，该字符串将被转换为小写形式，除非其中在表名周围包含双引号。

如果一个 OID 不表示一个已有的对象并且被作为参数传递给了上述函数，将会返回 NULL。

表 6.100 “数据库对象定位函数”中展示的函数帮助标识数据库对象相关的磁盘文件。

表 6.100. 数据库对象定位函数

名称	返回类型	描述
ux_relation_filenode(*relation* regclass)	oid	指定关系的文件结点号
ux_relation_filepath(*relation* regclass)	text	指定关系的文件路径名
ux_filenode_relation(*tablespace* oid, *filenode* oid)	regclass	查找与给定的表空间和文件节点相关的关系

ux_relation_filenode接受一个表、索引、序列或 TOAST 表的 OID 或名称，返回当前分配给它的“filenode”号。文件结点是关系的文件名的基本组件。对于大多数表结果和 ux_class.*relfilenode*相同，但是对于某些系统目录*relfilenode*为零，并且必须使用此函数获取正确的值。 如果传递一个没有存储的关系（如视图），此函数将返回 NULL。

ux_relation_filepath与ux_relation_filenode类似，但是它返回关系的整个文件路径名（相对于数据库集簇的数据目录UXDATA）。

ux_filenode_relation是ux_relation_filenode的反向函数。给定一个“tablespace” OID 以及一个“filenode”，它会返回相关关系的 OID。对于一个在数据库的默认表空间中的表，该表空间可以指定为 0。

表 6.101 “排序规则管理函数”列出了用来管理排序规则的函数。

表 6.101. 排序规则管理函数

名称	返回类型	描述
ux_collation_actual_version(oid)	text	返回来自操作系统的排序规则的实际版本
ux_import_system_collations(*schema* regnamespace)	integer	导入操作系统排序规则

ux_collation_actual_version返回当前安装在操作系统中的该排序规则对象的实际版本。如果这个版本与ux_collation.collversion中的值不同，则依赖于该排序规则的对象可能需要被重建。还可以参考ALTER COLLATION(7)。

ux_import_system_collations基于在操作系统中找到的所有locale在系统目录ux_collation中加入排序规则。这是initdb会使用的函数。如果后来在操作系统上安装了额外的locale，可以再次运行这个函数加入新locale的排序规则。匹配ux_collation中现有项的locale将被跳过（但是这个函数不会移除以在操作系统中不再存在的locale为基础的排序规则对象）。*schema*参数通常是ux_catalog，但这不是一种要求，排序规则也可以被安装到其他的方案中。该函数返回其创建的新排序规则对象的数量。

表 6.102. 分区信息函数

名称	返回类型	描述
ux_partition_tree(regclass)	setof record	列出一个分区树中的表或索引的相关信息。给定的分区表或分区索引，每张表有一行 分区。 提供的信息包括分区的名称，它的直系父级的名称，一个布尔值，表示该分区是否是一个叶子，以及一个整数，表示它在层次结构中的级别。level的值从0开始，表示输入表或索引作为分区树的根，1表示其分区，2表示其分区，以此类推。
ux_partition_ancestors(regclass)	setof regclass	列出给定分区的祖先关系，包括分区本身。
ux_partition_root(regclass)	regclass	R返回给定关系所属的分区树的最上层父节点。

要检查第 2.11.2.1 节 “例子”中描述的measurement表中的数据的总大小，可以使用下面的查询：

```
=# SELECT ux_size_pretty(sum(ux_relation_size(relid))) AS total_size
    FROM ux_partition_tree('measurement');
 total_size
------------
 24 kB
(1 row)
```

6.26.8. 索引维护函数

表 6.103 “索引维护函数”展示了可用于 索引维护任务的函数。这些函数不能在恢复期间执行。只有超级用户以及给定索引的拥有者才能是用这些函数。

表 6.103. 索引维护函数

名称	返回类型	描述
brin_summarize_new_values(*index* regclass)	integer	对还没有建立概要的页面范围建立概要

名称	返回类型	描述
brin_summarize_range(*index* regclass, *blockNumber* bigint)	integer	如果还没有对覆盖给定块的页面范围建立概要，则对其建立概要
brin_desummarize_range(*index* regclass, *blockNumber* bigint)	integer	如果覆盖给定块的页面范围已经建立有概要，则去掉概要
gin_clean_pending_list(*index* regclass)	bigint	把 GIN 待处理列表项移动到主索引结构中

brin_summarize_new_values接收一个 BRIN 索引的 OID 或者名称作为参数并且检查该索引以找到基表中当前还没有被该索引汇总的页面范围。对任意一个这样的范围，它将通过扫描那些表页面创建一个新的摘要索引元组。它会返回被插入到该索引的新页面范围摘要的数量。brin_summarize_range做同样的事情，不过它只对覆盖给定块号的范围建立概要。

gin_clean_pending_list接受一个 GIN 索引的 OID 或者名字，并且通过把指定索引的待处理列表中的项批量移动到主 GIN 数据结构来清理该索引的待处理列表。它会返回从待处理列表中移除的页数。注意如果其参数是一个禁用fastupdate选项构建的 GIN 索引，那么不会做清理并且返回值为 0，因为该索引根本没有待处理列表。

6.26.9. 通用文件访问函数

表 6.104 “通用文件访问函数”中展示的函数提供了对数据库服务器所在机器上的文件的本地访问。只能访问数据库集簇目录以及log_directory中的文件，除非用户被授予了角色ux_read_server_files。 使用相对路径访问集簇目录里面的文件，以及匹配 log_directory配置设置的路径访问日志文件。

注意向用户授予ux_read_file()或者相关函数上的EXECUTE特权，函数会允许他们读取服务器上该数据库能读取的任何文件并且这些读取动作会绕过所有的数据库内特权检查。这意味着，除了别的之外，具有这种访问的用户能够读取ux_authid表中包含着认证信息的内容，也能读取数据库中的任意文件。因此，授予对这些函数的访问应该要很仔细地考虑。

表 6.104. 通用文件访问函数

名称	返回类型	描述
ux_ls_dir(*dirname* text [, *missing_ok* boolean, *include_dot_dirs* boolean])	setof text	列出目录中的内容。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。
ux_ls_logdir()	setof record	列出日志目录中文件的名称、尺寸以及最后修改时间。访问被授予给ux_monitor角色的成员，并且可以被授予给其他非超级用户角色。
ux_ls_waldir()	setof record	列出WAL目录中文件的名称、尺寸以及最后修改时间。访问被授予给ux_monitor角色的成员，并且可以被授予给其他非超级用户角色。
ux_ls_archive_statusdir()	setof record	列出WAL存档状态目录中文件的名称、大小和最后一次修改时

名称	返回类型	描述
		间。访问权限只授予ux_monitor角色的成员，也可以授予其他非超级用户角色。
ux_ls_tmpdir([*tablespace* oid])	setof record	为*tablespace*列出临时目录中文件的名称、大小和最后一次修改时间。 如果没有提供*tablespace*，则在临时目录中的 ux_default表空间被使用。 ux_monitor角色的成员可以访问，其他非超级用户角色也可以访问。
ux_read_file(*filename* text [, *offset* bigint, *length* bigint [, *missing_ok* boolean]])	text	返回一个文本文件的内容。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。
ux_read_binary_file(*filename* text [, *offset* bigint, *length* bigint [, *missing_ok* boolean]])	bytea	返回一个文件的内容。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。
ux_stat_file(*filename* text[, *missing_ok* boolean])	record	返回关于一个文件的信息。默认仅限于超级用户使用，但是可以给其他用户授予EXECUTE让他们运行这个函数。

这些函数中的某些有一个可选的*missing_ok*参数， 它指定文件或者目录不存在时的行为。如果为true， 函数会返回 NULL （ux_ls_dir除外，它返回一个空 结果集）。如果为false，则发生一个错误。默认是 false。

ux_ls_dir返回指定目录中所有文件（以及目录和其他特殊文件） 的名称。*include_dot_dirs*指示结果集中是否包括 “.”和“..”。默认是排除它们（false），但是 当*missing_ok*为true时把它们包括在内是 有用的，因为可以把一个空目录与一个不存在的目录区分开。

ux_ls_logdir返回日志目录中每个文件的名称、尺寸以及最后的修改时间（mtime）。默认情况下，只有超级用户以及ux_monitor角色的成员能够使用这个函数。可以使用GRANT把访问授予给其他人。

ux_ls_waldir返回预写式日志（WAL）目录中每个文件的名称、尺寸以及最后的修改时间（mtime）。默认情况下，只有超级用户以及ux_monitor角色的成员能够使用这个函数。可以使用GRANT把访问授予给其他人。

ux_ls_archive_statusdir返回WAL归档状态目录ux_wal/archive_status中每个文件的名称、大小和最后一次修改时间（mtime）。默认情况下，只有超级用户和ux_monitor角色的成员才能使用此函数。可使用GRANT授权其他用户访问。

ux_ls_tmpdir返回指定的*tablespace*临时文件目录中每个文件的名称、大小和最后一次修改时间（mtime）。 如果没有提供*tablespace*，则使用ux_default表空间。 默认情况下，只有超级用户和ux_monitor角色的成员才能使用这个函数。 可使用GRANT授权其他用户访问。

ux_read_file返回一个文本文件的一部分，从给定的*offset*开始，返回最多*length*字节（如果先到达文件末尾则会稍短）。如果*offset*为负，它相对于文件的末尾。如果*offset*和*length*被忽略，整个文

件都被返回。从文件中读的字节被使用服务器编码解释成一个字符串；如果它们在编码中不合法则抛出一个错误。

ux_read_binary_file与ux_read_file相似，除了前者的结果是一个bytea值；相应地，不会执行编码检查。通过与convert_from函数结合，这个函数可以用来读取一个指定编码的文件：

```
SELECT convert_from(ux_read_binary_file('file_in_utf8.txt'), 'UTF8');
```

ux_stat_file返回一个记录，其中包含文件尺寸、最后访问时间戳、最后修改时间戳、最后文件状态改变时间戳（只支持 Unix 平台）、文件创建时间戳（只支持 Windows）和一个boolean指示它是否为目录。通常的用法包括：

```
SELECT * FROM ux_stat_file('filename');
SELECT (ux_stat_file('filename')).modification;
```

6.26.10. 咨询锁函数

表 6.105 “咨询锁函数”中展示的函数管理咨询锁。有关正确使用这些函数的细节请参考第 10.3.5 节 “咨询锁”。

表 6.105. 咨询锁函数

名称	返回类型	描述
ux_advisory_lock(*key* bigint)	void	获得排他会话级别咨询锁
ux_advisory_lock(*key1* int, *key2* int)	void	获得排他会话级别咨询锁
ux_advisory_lock_shared(*key* bigint)	void	获得共享会话级别咨询锁
ux_advisory_lock_shared(*key1* int, *key2* int)	void	获得共享会话级别咨询锁
ux_advisory_unlock(*key* bigint)	boolean	释放一个排他会话级别咨询锁
ux_advisory_unlock(*key1* int, *key2* int)	boolean	释放一个排他会话级别咨询锁
ux_advisory_unlock_all()	void	释放当前会话持有的所有会话级别咨询锁
ux_advisory_unlock_shared(*key* bigint)	boolean	释放一个共享会话级别咨询锁
ux_advisory_unlock_shared(*key1* int, *key2* int)	boolean	释放一个共享会话级别咨询锁
ux_advisory_xact_lock(*key* bigint)	void	获得排他事务级别咨询锁
ux_advisory_xact_lock(*key1* int, *key2* int)	void	获得排他事务级别咨询锁
ux_advisory_xact_lock_shared(*key* bigint)	void	获得共享事务级别咨询锁
ux_advisory_xact_lock_shared(*key1* int, *key2* int)	void	获得共享事务级别咨询锁
ux_try_advisory_lock(*key* bigint)	boolean	如果可能，获得排他会话级别咨询锁
ux_try_advisory_lock(*key1* int, *key2* int)	boolean	如果可能，获得排他会话级别咨询锁

名称	返回类型	描述
ux_try_advisory_lock_shared(*key* bigint)	boolean	如果可能，获得共享会话级别咨询锁
ux_try_advisory_lock_shared(*key1* int, *key2* int)	boolean	如果可能，获得共享会话级别咨询锁
ux_try_advisory_xact_lock(*key* bigint)	boolean	如果可能，获得排他事务级别咨询锁
ux_try_advisory_xact_lock(*key1* int, *key2* int)	boolean	如果可能，获得排他事务级别咨询锁
ux_try_advisory_xact_lock_shared(*key* bigint)	boolean	如果可能，获得共享事务级别咨询锁
ux_try_advisory_xact_lock_shared(*key1* int, *key2* int)	boolean	如果可能，获得共享事务级别咨询锁

ux_advisory_lock锁住一个应用定义的资源，可以使用一个单一64位键值或两个32位键值标识（注意这些两个键空间不重叠）。如果另一个会话已经在同一个资源标识符上持有了一个锁，这个函数将等待直到该资源变成可用。该锁是排他的。多个锁请求会入栈，因此如果同一个资源被锁住三次，则它必须被解锁三次来被释放给其他会话使用。

ux_advisory_lock_shared的工作和ux_advisory_lock相同，不过该锁可以与其他请求共享锁的会话共享。只有想要排他的锁请求会被排除。

ux_try_advisory_lock与ux_advisory_lock相似，不过该函数将不会等待锁变为可用。它要么立刻获得锁并返回true，要么不能立即获得锁并返回false。

ux_try_advisory_lock_shared的工作和ux_try_advisory_lock相同，不过它尝试获得一个共享锁而不是一个排他锁。

ux_advisory_unlock将会释放之前获得的排他会话级别咨询锁。如果锁被成功释放，它返回true。如果锁没有被持有，它将返回false并且额外由服务器报告一个 SQL 警告。

ux_advisory_unlock_shared的工作和ux_advisory_unlock相同，除了它释放一个共享的会话级别咨询锁。

ux_advisory_unlock_all将释放当前会话所持有的所有会话级别咨询锁（这个函数隐式地在会话末尾被调用，即使客户端已经不雅地断开）。

ux_advisory_xact_lock的工作和ux_advisory_lock相同，不过锁是在当前事务的末尾被自动释放的并且不能被显式释放。

ux_advisory_xact_lock_shared的工作和ux_advisory_lock_shared相同，除了锁是在当前事务的末尾自动被释放的并且不能被显式释放。

ux_try_advisory_xact_lock的工作和ux_try_advisory_lock相同，不过锁（若果获得）是在当前事务的末尾被自动释放的并且不能被显式释放。

ux_try_advisory_xact_lock_shared的工作和ux_try_advisory_lock_shared相同，不过锁（若果获得）是在当前事务的末尾被自动释放的并且不能被显式释放。

6.27. 触发器函数

当前UXDB提供一个内建的触发器函数suppress_redundant_updates_trigger， 它将阻止任何不会实际更改行中数据的更新发生，这与正常的行为不管数据是否改变始终执行更新相反（这是正常的行为，使得更新运行速度更快，因为不需要检查，并在某些情况下也是有用的）。

理想的情况下，通常应该避免运行实际上并没有改变记录中数据的更新。 冗余更新会花费大量不必要的时间，尤其是如果有大量索引要改变， 并将最终不得不清理被死亡行占用的空间。但是，在客户端代码中检测这种情况并不总是容易的，甚至不可能做到。 而写表达式来检测它们容易产生错误。作为替代，使用suppress_redundant_updates_trigger可以跳过不改变数据的更新。但是，使用它需要注意。触发器需要很短但不能忽略的时间来处理每条记录，所以如果大多数被一个更新影响的记录确实被更改，此触发器的使用将实际上使更新运行得更慢。

suppress_redundant_updates_trigger函数可以像这样被加到一个表：

```
CREATE TRIGGER z_min_update
BEFORE UPDATE ON tablename
FOR EACH ROW EXECUTE FUNCTION suppress_redundant_updates_trigger();
```

在大部分情况下，可能希望在最后为每行触发这个触发器。考虑到触发器是按照名字顺序被触发，需要选择一个位于该表所有其他触发器之后的触发器名字。

有关创建触发器的更多信息请参考CREATE TRIGGER(7)。

6.28. 事件触发器函数

UXDB提供了这些助手函数来从 事件触发器检索信息。

6.28.1. 在命令结束处捕捉更改

当在一个ddl_command_end事件触发器的函数中调用时， ux_event_trigger_ddl_commands返回被每一个用户动作 执行的DDL命令的列表。如果在其他任何环境中 调用这个函数，会发生错误。ux_event_trigger_ddl_commands为每一个被执行的基本 命令返回一行，某些只有一个单一 SQL 句子的命令可能会返回多于一行。 这个函数返回下面的列：

名称	类型	描述
classid	oid	对象所属的目录的 OID
objid	oid	对象本身的 OID
objsubid	integer	对象的子-id（例如列的属性号）
command_tag	text	命令标签
object_type	text	对象的类型
schema_name	text	该对象所属的模式的名称（如果有），如果没有则为NULL。 没有引号。
object_identity	text	对象标识的文本表现形式，用模式限定。如果必要，出现在该标识中的每一个标识符都会被引用。
in_extension	bool	如果该命令是一个扩展脚本的一部分则为真

名称	类型	描述
command	ux_ddl_command	以内部格式表达的该命令的一个完整表现形式。这不能被直接输出， 但是可以把它传递给其他函数来得到有关于该命令不同部分的信息。

6.28.2. 处理被 DDL 命令删除的对象

ux_event_trigger_dropped_objects返回其sql_drop事件中命令所删除的所有对象的列表。如果在任何其他环境中被调用，ux_event_trigger_dropped_objects将抛出一个错误。ux_event_trigger_dropped_objects返回下列列：

名称	类型	描述
classid	oid	对象所属的目录的 OID
objid	oid	对象本身的 OID
objsubid	integer	对象的子ID（如列的属性号）
original	bool	如果这是删除中的一个根对象则为真
normal	bool	指示在依赖图中有一个普通依赖关系指向该对象的标志
is_temporary	bool	如果该对象是一个临时对象则为真
object_type	text	对象的类型
schema_name	text	对象所属模式的名称（如果存在）；否则为NULL。不应用引用。
object_name	text	如果模式和名称的组合能被用于对象的一个唯一标识符，则是对象的名称；否则是NULL。不应用引用，并且名称不是模式限定的。
object_identity	text	对象身份的文本表现，模式限定的。每一个以及所有身份中出现的标识符在必要时加引号。
address_names	text[]	一个数组，它可以和object_type及 address_args 一起通过ux_get_object_address()函数在一台包含有 同类相同名称对象的远程服务器上重建该对象地址。
address_args	text[]	上述address_names的补充。

ux_event_trigger_dropped_objects可以被这样用在一个事件触发器中：

```
CREATE FUNCTION test_event_trigger_for_drops()
      RETURNS event_trigger LANGUAGE pluxsql AS $$
DECLARE
   obj record;
BEGIN
   FOR obj IN SELECT * FROM ux_event_trigger_dropped_objects()
   LOOP
      RAISE NOTICE '% dropped object: % %.% %',
                   tg_tag,
                   obj.object_type,
                   obj.schema_name,
                   obj.object_name,
                   obj.object_identity;
   END LOOP;
END
$$;
CREATE EVENT TRIGGER test_event_trigger_for_drops
  ON sql_drop
  EXECUTE FUNCTION test_event_trigger_for_drops();
```

6.28.3. 处理表重写事件

表 6.106 “表重写信息”中所示的函数提供刚刚被调用过table_rewrite 事件的表的信息。如果在任何其他环境中调用，会发生错误。

表 6.106. 表重写信息

名称	返回类型	描述
ux_event_trigger_table_rewrite_oid()	oid	要被重写的表的 OID。
ux_event_trigger_table_rewrite_reason()	int	解释重写原因的原因代码。这些代码的确切含义在单独的文档中。

可以在一个这样的事件触发器中使用 ux_event_trigger_table_rewrite_oid函数：

```
CREATE FUNCTION test_event_trigger_table_rewrite_oid()
 RETURNS event_trigger
 LANGUAGE pluxsql AS
$$
BEGIN
  RAISE NOTICE 'rewriting table % for reason %',
               ux_event_trigger_table_rewrite_oid()::regclass,
               ux_event_trigger_table_rewrite_reason();
END;
$$;

CREATE EVENT TRIGGER test_table_rewrite_oid
                ON table_rewrite
  EXECUTE FUNCTION test_event_trigger_table_rewrite_oid();
```

6.29. 统计信息函数

UXDB提供了一个函数来检查使用CREATE STATISTICS命令定义的复杂统计。

6.29.1. 检查MCV列表

ux_mcv_list_items返回存储在多列MCV列表中的所有项目的列表，并返回以下列:

名称	类型	描述
index	int	MCV 列表中的项目索引
values	text[]	存储在MCV项目中的值
nulls	boolean[]	标识NULL值的标志
frequency	double precision	MCV项目的频率
base_frequency	double precision	MCV项目的基本频率

ux_mcv_list_items函数可以这样使用:

```
SELECT m.* FROM ux_statistic_ext join ux_statistic_ext_data on (oid = stxoid),
        ux_mcv_list_items(stxdmcv) m WHERE stxname = 'stts';
```

ux_mcv_list的值只能从ux_statistic_ext_data.stxdmcv列中获得。

6.30. 正则表达式函数

正则表达式是一个字符序列，它是定义一个字符串集合(一个正则集合)的缩写。如果一个字符串是正则表达式描述的正则集合中的一员时，就说这个字符串匹配该正则表达式。

对正则表达式的支持可以使应用程序开发人员可以在数据库中实现复杂的模式匹配逻辑，支持的类型为text，兼容varchar、char、int等数据类型。

使用场景如下所示。

1. 对于表中的字段进行正则表达式匹配，实现更精准的查找。
2. 对于已知字符串直接进行正则表达式匹配。

6.30.1. REGEXP_COUNT

- 功能

  REGEXP_COUNT函数统计源字符串中模式匹配的次数。

- 示例

  表中字段的正则匹配，表中字段 empName，从字符串首开始匹配，字段中含有e的次数，大小写敏感。

```
SELECT empName, REGEXP_COUNT(empName, 'e', 1, 'c') "CASE_SENSITIVE_E" From
 regexp_temp;
empname | CASE_SENSITIVE_E
```

```
--------------------------------
John Doe |   1
Jane Doe |   2
(2 行记录)
```

字符串直接进行正则匹配，搜索源字符串中，统计符合子字符串（大写字母开头，后跟2个数字）的个数。

```
SELECT REGEXP_COUNT('ABC123', '[A-Z][0-9]{2}');
regexp_count
--------------
 1
(1 行记录)
```

6.30.2. REGEXP_INSTR

- 功能

  REGEXP_INSTR函数输出符合正则表达式的子字符串的起始位置或紧接模式结束后的位置。

- 示例

  表中字段的正则匹配，针对表中字段 emailID，从字符串首开始匹配，字段中含有@和.的字符串，输出匹配项在源串的起始位置，无匹配项输出0。

```
SELECT emailID, REGEXP_INSTR(emailID, '\w+@\w+(\.\w+)+') "IS_A_VALID_EMAIL"
 FROM regexp_temp;
emailid  |  IS_A_VALID_EMAIL
-----------------------------------------
johndoe@example.com | 1
janedoe  | 0
(2 行记录)
```

字符串直接进行正则匹配，查找源字符串中以s/r/p开头的单词（不分大小写），后跟任意6个字符，从字符串第三个字符开始搜索，并在第二个出现以s/r/p开头的第七个字母的单词之后，返回该字符在字符串中的位置。

```
SELECT REGEXP_INSTR('500 Oracle Parkway, Redwood Shores, CA', '[s|r|p][[:alpha:]]{6}',
 3, 2, 1, 'i');
regexp_instr
--------------
 28
(1 行记录)
```

6.30.3. REGEXP_SUBSTR

- 功能

  REGEXP_SUBSTR函数输出符合正则表达式的子字符串本身。

- 示例

表中字段的正则匹配，针对表中字段 emailID，从字符串首开始匹配，查找其中含有字母/数字+@+或字母/数字+.+字母/数字的串，找到则输出整个字符串，没有则输出为空。

```
SELECT empName, REGEXP_SUBSTR(emailID, '[[:alnum:]]+\@[[:alnum:]]+\.[[:alnum:]]+')
 "Valid Email" FROM regexp_temp;
empname |  Valid Email
---------------------------------
John Doe | johndoe@example.com
Jane Doe |
(2 行记录)
```

字符串直接进行正则匹配，搜索源字符串中，统计符合子字符串（大写字母开头，后跟2个数字）的个数。

```
SELECT REGEXP_SUBSTR('500 Oracle Parkway, Redwood Shores, CA', ',[^,]+,');
regexp_substr
-------------------
, Redwood Shores,
(1 行记录)
```

6.30.4. REGEXP_REPLACE_ORACLE

- 功能

  REGEXP_REPLACE_ORACLE函数找到符合正则表达式的子字符串，并进行替换，输出替换后的字符串。

- 示例

  表中字段的正则匹配，针对表中字段 emailID，从字符串首开始匹配，针对表中字段 empName，从字符串首开始匹配，如果源串中Jane，就替换成John，然后输出替换后的结果，默认替换所有匹配项。

```
SELECT empName, REGEXP_REPLACE_ORACLE(empName, 'Jane', 'John')
 "STRING_REPLACE" FROM regexp_temp;
empname |  STRING_REPLACE
---------------------------------
John Doe |  John Doe
Jane Doe |  John Doe
(2 行记录)
```

字符串直接进行正则匹配，搜索源字符串中，出现两个或两个以上的空格，将其替换成一个空格并输出替换后的字符串。

```
SELECT REGEXP_REPLACE_ORACLE('500 Oracle Parkway, Redwood Shores, CA',
 '( ){2,}', ' ');
regexp_replace_oracle
-----------------------------------------
500 Oracle Parkway, Redwood Shores, CA
(1 行记录)
```

6.30.5. regexp_like

- 功能

  regexp_like函数从string中查找一个匹配并且pattern不包含带括号的子表达式，若匹配，返回true，否则为false。

- 函数

  regexp_like(string, pattern [, flags])

- 参数

  flags参数是一个可选的文本字符串，它包含零个或者更多个可以改变该函数行为的单字母标志。所支持的标志请参见表 6.25 “ARE 嵌入选项字母”。

- 返回值

  Boolean

- 示例

```
SELECT regexp_like('foobarbequebaz', 'bar.*que');
 regexp_like
-------------
 t
(1 row)
SELECT regexp_like('foobarbequebaz', '(bar)(beque)');
 regexp_like
-------------
 t
(1 row)
SELECT regexp_like('foobarbequebaz', '(xxx)');
 regexp_like
-------------
 f
(1 row)
```

6.31. 空值判断函数

6.31.1. isnull

- 功能

  Isnull返回参数列表中第一个非空参数的值，支持一个参数或两个参数，两个参数仅当两个参数都为空时才会返回空。它常用于在为显示目的检索数据时用缺省值替换空值。

- 函数

  isnull (value1)

  isnull (value1,value2)

- 返回值

isnull (value1)参数值为空时返回1，不为空返回0。参数个数大于二时报错。

isnull (value1,value2)第一个参数不为空时返回第一个参数值，第一个参数为空时返回第二个参数值。

- 示例

isnull有一个参数的使用方式，如下所示。

```
select ISNULL(NULL::int);
select ISNULL(NULL::numeric);
select ISNULL(NULL::varchar);
select ISNULL(NULL::text);
select ISNULL(NULL::date);
select ISNULL(1134::int);
select ISNULL(34.56543::numeric);
select ISNULL('uxsinodb'::varchar);
select ISNULL('fasdgerhgaehgwgwtrhwtg'::text);
select ISNULL('2021-09-14'::date);
```

isnull有两个参数的使用方式，如下所示。

```
select isnull(v1,v2) from isnull_test;
select isnull(v4,v1) from isnull_test;
select isnull(v1) from isnull_test;
```

6.31.2. nvl

- 功能

nvl为空值置换函数，返回表达式中第一个参数不为null的表达式，如果表达式都是null，则返回null。它常用于在为显示目的检索数据时用缺省值替换空值。

- 函数

nvl(expr1,expr2)

- 参数

expr1和expr2可以为任意类型，字符串，数值，时间等。

- 返回值

返回值为表达式非空表达式的值。当表达式中的expr1为null时，返回expr2；当expr1 不为null，则返回expr1。

- 示例

```
select nvl(v1,v2) from nvl_test;
 nvl
--------
 1
```

```
 5
 23123
 123
(4 行记录)
```

第 7 章 类型转换

SQL语句可能（有意无意地）要求在同一表达式里混合不同的数据类型。 UXDB在计算混合类型表达式方面有许多功能。

在大多数情况下，用户不需要明白类型转换机制的细节。但是，由UXDB进行的隐式类型转换会对查询的结果产生影响。必要时这些结果可以被使用显式类型转换来调整。

本章介绍UXDB类型转换的机制和习惯。 关于特定的类型和允许的函数及操作符的进一步信息，请参考第 5 章 数据类型和第 6 章 函数和操作符的相关章节。

7.1. 概述

SQL是一种强类型语言。也就是说，每个数据项都有一个相关的数据类型，数据类型决定其行为和允许的用法。 UXDB有一个可扩展的类型系统，该系统比其它SQL实现更具通用和灵活。因而，UXDB中大多数类型转换行为是由通用规则来管理的，而不是ad hoc启发式规则。这种做法允许使用混合类型表达式，即便是其中包含用户定义的类型。

UXDB扫描器/解析器只将词法元素分解成五个基本种类：整数、非整数数字、字符串、标识符、关键字。大多数非数字类型常量首先被分类为字符串。SQL语言定义允许将类型名指定为字符串， 这个机制被UXDB用于保证解析器沿着正确的方向运行。例如，查询：

```
SELECT text 'Origin' AS "label", point '(0,0)' AS "value";

 label  | value
--------+-------
 Origin | (0,0)
(1 row)
```

有两个文字常量，类型分别为text和point。如果一个串文字没有指定类型，初始将被分配一个占位符类型unknown，该类型将在下文描述的后续阶段被解析。

在SQL解析器里，有四种基本的SQL结构要求独立的类型转换规则：

函数调用

UXDB类型系统的大部分建立在一套丰富的函数上。 函数可以有一个或多个参数。由于UXDB允许函数重载， 所以函数名自身并不唯一地标识将要被调用的函数，解析器必须根据提供的参数类型选择正确的函数。

操作符

UXDB允许带有前缀和后缀一元（单目）操作符的表达式，也允许二元（两个参数）操作符。像函数一样，操作符也可以被重载，因此操作符的选择也有同样的问题。

值存储

SQL INSERT和UPDATE语句将表达式的结果放 入表中。语句中的表达式类型必须和目标列的类型一致（或者可以被转换为一致）。

UNION、CASE和相关结构

因为来自一个联合的SELECT语句中的所有查询结果必须在一个列集中显示，所以每个SELECT子句的结果类型必须能相互匹配并被转换成一个统一的集合。类似地，一个 CASE结

构的结果表达式必须被转换成一种公共的类型，这样CASE表达式作为整体才 有一种已知的输出类型。同样的要求也存在于ARRAY结构以及GREATEST和LEAST函数中。

系统目录存储有关哪些数据类型之间存在哪种转换（或造型）以及如何执行这些转换的相关信息。额外的造型可以由用户通过CREATE CAST(7)命令增加（这个通常和定义一种新的数据类型一起完成。 内建的类型转换集已经经过了仔细的雕琢，最好不要去更改它们）。

解析器提供了一种额外的启发式规则，它允许在具有隐式造型的类型组中恰当造型行为的改进决定。 数据类型被分为几个基本的类型分类，包括boolean、numeric、string、bitstring、datetime、timespan、geometric、network和用户自定义（需要注意的是可以创建自定义的类型分类）。在每个分类中，可以有一个或多个首选类型， 当存在类型选择时，这个是更好的选择。利用精心选择的首选类型和可用的隐式造型，我们可以确保有歧义的表达式（那些有多个候选解析方案的表达式）可以用一种有用的方式来处理。

所有类型转换规则都是建立在下面几个基本原则上的：

- 隐式转换决不能有意外的或不可预见的输出。
- 如果一个查询不需要隐式类型转换，解析器或执行器不应该有额外的开销。也就是说，如果一个查询是结构良好的并且类型已经匹配，则查询不应该在解析器里耗费额外的时间执行，也不会在查询中引入不必要的隐式类型转换调用。
- 另外，如果一个查询通常要求为某个函数进行隐式类型转换，而用户定义了一个有正确参数类型的新函数， 解析器应该使用新函数并不再做隐式转换来使用旧函数。

7.2. 操作符

被一个操作符表达式引用的特定操作符由下列过程决定。注意这个过程会被所涉及的操作符的优先级间接地影响，因为这将决定哪些子表达式被用作哪个操作符的输入。详见第 1.1.6 节 “操作符优先级”。

过程 7.1. 操作符类型决定

1. 从系统目录ux_operator中选出要考虑的操作符。如果使用了一个不带模式限定的操作符 名（常见的情况），那么操作符被认为是那些在当前搜索路径中可见并有匹配的名字和参数个数的操作符（参见第 2.9.3 节 “模式搜索路径”）。如果给出一个被限定的操作符名，那么只考虑指定模式中的操作符。

   - （可选的） 如果搜索路径找到了多个有相同参数类型的操作符，那么只考虑最早出现在路径中的那一个。 但是不同参数类型的操作符将被平等看待，而不管它们在路径中的位置如何。

2. 查找一个正好接受输入参数类型的操作符。如果找到一个（在一组被考虑的操作符中，可能只存在一个正好匹配的），则使用之。在通过限定名称（非典型）调用在一个允许不可信用户创建对象的方案中找到的任意操作符时，精确匹配的缺失会导致安全性危害[1]。在这样的情况下，应该造型参数以便强制一次精确匹配。

   a. （可选的） 如果一个二元操作符调用中的一个参数是unknown类型，则在本次检查中假设它与另一个参数类型相同。 对于涉及两个unknown输入的调用或者带有一个unknown输入的一元操作符，在这一步将永远找不到一个匹配。

[1] 对非方案限定的名称，不会出现这种危害，因为包含允许不可信用户创建对象的方案的搜索路径不是一种安全的方案使用模式。

b. (可选的) 如果一个二元操作符调用的其中一个参数是unknown类型 而另一个是一种域类型，下一次检查会看看是否有一个操作符正好在两边都 接受该域的基类型，如果有就使用它。

3. 寻找最优匹配。

a. 抛弃那些输入类型不匹配并且也不能被转换成匹配的候选操作符。 unknown文字被假定为可以为这个目的被转换为 任何东西。如果只剩下一个候选操作符，则使用之，否则继续下一 步。

b. 如果任何输入参数是一种域类型，对所有后续步骤都把它当做是该 域的基类型。这确保在做有歧义的操作符解析时，域的举止像它们 的基类型。

c. 遍历所有候选操作符，保留那些在输入类型上的匹配最准确的。如果没有一个操作符能准确匹配，则保留所有候选。如果只剩下一个候选操作符，则使用之，否则继续下一步。

d. 遍历所有候选操作符，保留那些在最多个需要类型转换的位置上接受首选类型（属于输入数据类型的类型分类）的操作符。如果没有接受首选类型的操作符，则保留所有候选。如果只剩下一个候选操作符，则使用之， 否则继续下一步。

e. 如果有任何输入参数是unknown类型，检查被剩余候选操作符在那些参数位置上接受的类型分类。 在每一个位置，如果任何候选接受该分类，则选择string分类（这种对字符串的偏爱是合适的， 因为未知类型的文本确实像字符串）。否则，如果所有剩下的候选操作符都接受相同的类型 分类，则选择该分类；否则抛出一个错误（因为在没有更多线索的条件下无法作出正确 的推断）。现在抛弃不接受选定的类型分类的候选操作符。然后，如果任意候选操作符接受那个分类中的首选类型， 则抛弃那些在该参数位置接受非首选类型的候选操作符。如果没有候选操作符能通过这些测试则保留全部候选者。如果只剩下一个候选者，则使用之；否则继续下一步。

f. 如果既有unknown参数也有已知类型的参数，并且所有已知类型参数具有相同的类型，则假定该unknown参数也是那种类型的，并且检查哪些候选操作符可以在该unknown参数的位置上接受那个类型。如果正好有一个候选者通过了这个测试，则使用之；否则失败。

下面是一些例子。

例 7.1. 阶乘操作符类型决定

在标准目录中只有一个被定义的阶乘操作符（后缀!），它接受一个类型为bigint的参数。在下面这个查询表达式中，扫描器会为该参数分配一个初始类型integer：

```
SELECT 40 ! AS "40 factorial";

                40 factorial
-----------------------------------------------
 815915283247897734345611269596115894272000000000
(1 row)
```

因此，解析器在操作数上做了一个类型转换，该查询等价于：

```
SELECT CAST(40 AS bigint) ! AS "40 factorial";
```

例 7.2. 字符串连接操作符类型决定

一个类字符串的语法被用来处理字符串类型和处理复杂的扩展类型。未指定类型的字符串与可能的候选操作符匹配。

一个未指定参数的例子：

```
SELECT text 'abc' || 'def' AS "text and unknown";

 text and unknown
------------------
 abcdef
(1 row)
```

在这种情况下，解析器查看是否有一个操作符的两个参数都使用text。既然有，那么它假设第二个参数应被解释为text类型。

下面是两个未指定类型的值的连接：

```
SELECT 'abc' || 'def' AS "unspecified";

 unspecified
-------------
 abcdef
(1 row)
```

在这种情况下，没有对于使用哪种类型的初始提示，因为在查询中没有指定类型。 因此，解析器查找所有的候选操作符并找到候选者同时接受字符串分类和位串分类的输入。 因为字符串分类在可用时是首选的，该分类会被选中，并且接下来字符串的首选类型（text）会被用作解决未知类型文字的指定类型。

例 7.3. 绝对值与否定操作符类型决定

UXDB操作符目录中有几个对于前缀操作符@的条目， 这些都现实了针对各种数字数据类型的绝对值操作。其中之一用于float8类型，它是在数字分类中的首选类型。 因此，UXDB将在遇到一个unknown输入时使用它：

```
SELECT @ '-4.5' AS "abs";
 abs
-----
 4.5
(1 row)
```

在这里，系统在应用所选操作符之前已经隐式地解决了将未知类型文字作为float8类型。 我们可以验证我们使用的是float8而不是别的类型：

```
SELECT @ '-4.5e500' AS "abs";

ERROR:  "-4.5e500" is out of range for type double precision
```

另一方面，前缀符~（按位取反）只为整数数据类型定义，而没有为float8定义。因此，如果我们尝试一个与使用~类似的情况，我们会得到：

```
SELECT ~ '20' AS "negation";

ERROR:  operator is not unique: ~ "unknown"
HINT:  Could not choose a best candidate operator. You might need to add
explicit type casts.
```

这是因为系统不能决定在几个可能的~符号中应该选择哪一个。我们可以用一个显式造型来帮助它：

```
SELECT ~ CAST('20' AS int8) AS "negation";

 negation
----------
      -21
(1 row)
```

例 7.4. 数组包含操作符类型决定

这里是另一个决定带有一个已知和一个未知输入的操作符的例子：

```
SELECT array[1,2] <@ '{1,2,3}' as "is subset";

 is subset
-----------
 t
(1 row)
```

UXDB操作符目录有一些条目用于中缀操作符<@，但是仅有的两个可以在左手边接受一个整数数组的是数组包含（anyarray <@ anyarray）和范围包含（anyelement <@ anyrange）。因为这些多态伪类型（见第 5.21 节 “伪类型”）中没有一个被认为是首选的，解析器不能以此为基础来解决歧义。不过，步骤 3.f告诉它假定位置类型的文字和其他输入的类型相同，即整数数组。现在这两个操作符中只有一个可以匹配，因此数组包含被选择（如果选择范围包含，我们将得到一个错误，因为该字符串没有成为一个范围文字的正确格式）。

例 7.5. 域类型上的自定义操作符

用户有时会尝试声明只适用于一种域类型的操作符。这是可能的， 但是远非它看起来那么有用，因为操作符解析规则被设计为选择 适用于域的基类型的操作符。考虑这个例子：

```
CREATE DOMAIN mytext AS text CHECK(...);
CREATE FUNCTION mytext_eq_text (mytext, text) RETURNS boolean AS ...;
CREATE OPERATOR = (procedure=mytext_eq_text, leftarg=mytext, rightarg=text);
CREATE TABLE mytable (val mytext);

SELECT * FROM mytable WHERE val = 'foo';
```

这个查询将不会使用自定义操作符。解析器将首先看看是否有一个 mytext = mytext操作符（步骤 2.a），当然这里没有； 然后它将会考虑该域的基类型text，并且看看是否有一 个text = text操作符（步骤 2.b），这里也没有；因 此它会把unknown-类型文字解析为text 并使用text = text操作符。 让自定义操作符能被使用的唯一方法是显式地转换改文字：

SELECT * FROM mytable WHERE val = text 'foo';

这样根据准确匹配规则会立即找到 mytext = text操作符。如果 到达最佳匹配规则，它们会积极地排斥域类型上的操作符。如果它 们没有，这样一个操作符将创建太多歧义操作符失败，因为转换规 则总是认为一个域可以和它的基类型相互转换，并且因此该域操作 符在所有与该基类型上的一个类似命名的操作符相同的情况中都被 认为可用。

7.3. 函数

被一个函数调用引用的特定函数使用下面的过程来决定。

过程 7.2. 函数类型决定

1. 从ux_proc系统目录中选择要被考虑的函数。 如果使用一个非模式限定的函数名称，那么函数被认为是那些在当前搜索路径中可见并有匹配的名字和参数个数的函数（参见第 2.9.3 节 “模式搜索路径”）。如果给出一个被限定的函数名，那么只考虑指定模式中的函数。

   a. （可选的） 如果搜索路径发现多个参数类型相同的函数，那么只考虑最早在搜索路径中出现的那个。 不同参数类型的函数被平等对待，不受在搜索路径中位置的影响。

   b. （可选的） 如果使用一个VARIADIC数组参数声明一个函数，并且调用不使用关键字VARIADIC， 那么该函数就好像其数组参数被它的元素类型的一次或多次出现所替换，根据需要去匹配调用。 这样的扩展之后，函数可能会有和非可变函数相同的参数类型。在这种情况下，在搜索路径中出现比较早的函数将被使用，或者如果两个函数在相同的模式中时首选非可变的那一个。

      在通过限定名称调用在一个允许不可信用户创建对象的方案中找到的可变函数时，会导致安全性危害[2]。 恶意用户可以拿到控制权并且执行任意SQL函数（就好像在执行它们一样）。将涉及VARIADIC关键词的调用替换掉就可以绕过这种危害。涉及到VARIADIC "any"参数的调用通常没有等效的包含VARIADIC关键词的形式。为了安全地发出那些调用，函数的方案必须只允许可信用户创建对象。

   c. （可选的） 考虑使用有默认参数值的函数来匹配任何省略了零个或者多个可默认参数位置的调用。如果有超出一个的这种函数匹配一个调用，那么使用最早出现在搜索路径中的那个。如果同一个模式中在同一个非默认位置上有两个或者更多这样的函数（如果它们有 不同的默认参数设置，这是可能的），系统将不能确定去选择哪一个，并且如果不能找到该调用更好的匹配，将会导致一个“有歧义的函数调用” 错误。

      在通过限定名称[2]调用在一个允许不可信用户创建对象的方案中找到的任意函数时，会导致可用性危害。恶意用户可以用一个已有函数的名称创建一个函数，复制该函数的参数并且追加新的具有默认值的参数。这会妨碍对原始函数的新调用。为了防止这种危害，应将函数放在仅允许可信用户创建对象的方案中。

2. 检查一个函数正好接受输入参数类型。如果存在一个（在所考虑的一组函数中只能有一个准确匹配），则使用之。在通过限定名称[2]调用在一个允许不可信用户创建对象的方案中找到的函数时，精确匹配的缺失会导致安全性危害。在这样的情况下，应该造型参数以便强制一次精确匹配（在该步骤中，涉及unknown的情况将永远找不到一个匹配）。

3. 如果没有发现准确匹配，那么查看函数调用是否作为一个特定的类型转换请求出现。 如果函数调用仅有一个参数并且函数名和一些数据类型的（内部）名称相同，那么该情况将会发

[2] 对非方案限定的名称，不会出现这种危害，因为包含允许不可信用户创建对象的方案的搜索路径不是一种安全的方案使用模式。

生。 并且，该函数参数必须是一个未知类型的文字，或者是一个可以被二进制强制转换到命名数据类型的类型， 或者是一个可以通过应用其I/O函数被转换为命名数据类型的类型（也就是，转换是转到标准字符串类型或者从标准字符串类型转来）。当满足这些条件的时候，函数调用被当做CAST声明的一种形式来对待。 [3]

4. 查找最佳匹配。

   a. 如果候选函数的输入类型不匹配并且不能通过转换（使用一个隐式转换）达到匹配，则丢弃它。为了这个目的，unknown文字被假定可被转换成任何东西。如果仅有一个候选项，则使用之；否则继续下一步。

   b. 如果任何输入参数是一种域类型，在所有后续步骤中都把它当做 该域的基类型。这确保在做有歧义的操作符解析时，域的举止像它们 的基类型。

   c. 遍历所有候选函数并保留那些最匹配输入类型的。如果没有准确匹配，则保留所有候选项。 如果仅有一个候选项，则使用之；否则继续下一步。

   d. 遍历所有候选函数并保留那些在最多要求类型转换的位置上接受首选类型（属于输入数据类型的类型分类）的候选项。如果没有接受首选类型的候选项，则保留所有候选项。如果仅有一个候选项，则使用之；否则继续下一步。

   e. 如果任何输入参数是unknown，那么检查那些被剩余候选项在那些参数位置上接受的类型分类。在每一个位置上，如果任何候选项接受该分类则选择string分类 （这个偏向于字符串是恰当的，因为一个未知类型文字看起来像字符）。 否则，如果所有剩余的候选项接受相同的类型分类，那么选择那个分类； 否则将失败，因为缺乏更多线索来推断出正确的选择。现在，丢弃不接受被选中类型分类的候选项。此外，如果任何候选项接受那个分类中的一个首选类型，则丢弃对该参数接受非首选类型的候选项。如果没有候选项能通过这些测试，则保留所有候选项。如果只剩下一个候选项，则使用之；否则继续下一步。

   f. 如果既有unknown参数也有已知类型的参数，并且所有已知类型参数具有相同的类型，则假定该unknown参数也是那种类型的，并且检查哪些候选函数可以在该unknown参数的位置上接受那个类型。如果正好有一个候选者通过了这个测试，则使用之；否则失败。

注意，对于操作符和函数类型决定来说“最优匹配”规则是完全相同的。下面是一些例子。

例 7.6. 圆整函数参数类型决定

只有一个带有两个参数的圆整函数； 它采用第一个参数类型为numeric和第二个参数类型为integer。这样下面的查询自动将第一个类型为integer参数转换为numeric：

```
SELECT round(4, 4);

 round
--------
 4.0000
(1 row)
```

该查询实际上被解析器转换为：

```
SELECT round(CAST (4 AS numeric), 4);
```

[3] 这一步的原因是在没有一个实际的造型函数的情况下支持函数风格的造型声明。如果有一个造型函数，它被按惯例以其输出类型命名，并且不需要有特殊情况。更多信息请见CREATE CAST(7)。

因为包含小数点的数字常数初始会被分配类型numeric，下面的查询将不需要类型转换并因此可能会稍稍高效一些：

```
SELECT round(4.0, 4);
```

例 7.7. 可变函数决定

```
CREATE FUNCTION public.variadic_example(VARIADIC numeric[]) RETURNS int
  LANGUAGE sql AS 'SELECT 1';
CREATE FUNCTION
```

这个函数接受（但不要求）VARIADIC关键词。它能同时容忍integer以numeric参数：

```
SELECT public.variadic_example(0),
      public.variadic_example(0.0),
      public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                1 |                1 |                1
(1 row)
```

不过，如果可以，第一个和第二个调用将更喜欢更明确的函数：

```
CREATE FUNCTION public.variadic_example(numeric) RETURNS int
  LANGUAGE sql AS 'SELECT 2';
CREATE FUNCTION

CREATE FUNCTION public.variadic_example(int) RETURNS int
  LANGUAGE sql AS 'SELECT 3';
CREATE FUNCTION

SELECT public.variadic_example(0),
      public.variadic_example(0.0),
      public.variadic_example(VARIADIC array[0.0]);
 variadic_example | variadic_example | variadic_example
------------------+------------------+------------------
                3 |                2 |                1
(1 row)
```

如果给定默认的配置并且只有第一个函数存在，则第一个和第二个调用是不安全的。任何用户都可以通过创建第二个或者第三个函数来截断它们。通过精确匹配参数类型并且使用VARIADIC关键词，第三个调用是安全的。

例 7.8. 子串函数类型决定

有几个substr函数，其中一个用于text和integer类型。如果使用一个未指定类型的字符常量调用，那么系统选择接受一个首选分类string（也就是text类型）的参数的候选函数。

```
SELECT substr('1234', 3);
```

```
 substr
--------
    34
(1 row)
```

如果字符串被声明为类型varchar（如果它来自于一个表就会这样），那么解析器将尝试转换它为text：

```
SELECT substr(varchar '1234', 3);

 substr
--------
    34
(1 row)
```

解析器所作的转换：

```
SELECT substr(CAST (varchar '1234' AS text), 3);
```

> **注意**
>
> 解析器从ux_cast目录中知道text和varchar是二进制可兼容的， 意思是其中一个可以被传递给接受另一种类型的函数而不需要做任何物理转换。因此，在这种情况下不会真正使用类型转换调用。

并且，如果该函数使用一个integer类型的参数调用，那么解析器将尝试将它转换为text：

```
SELECT substr(1234, 3);
ERROR:  function substr(integer, integer) does not exist
HINT:  No function matches the given name and argument types. You might need
to add explicit type casts.
```

由于integer类型没有到text的一个隐式造型，这将不会工作。但是一次显式造型则可以工作：

```
SELECT substr(CAST (1234 AS text), 3);

 substr
--------
    34
(1 row)
```

7.4. 值存储

将被插入到一个表的值会按照下列步骤被转换到目标列的数据类型。

过程 7.3. 值存储类型转换

1. 检查一个与目标的准确匹配。

2. 否则，尝试转换表达式为目标类型。如果在两种类型之间的一个 赋值造型已经被注册在ux_cast 目录（见CREATE CAST(7)）中， 这是可能的。或者，如果该表达式是一个未知类型的文字， 则该文字串的内容将被提供给目标类型的输入转换例程。

3. 检查是否有一个用于目标类型的尺寸调整造型。尺寸调整造型是一个从该类型到其自身的造型。 如果在ux_cast目录中找到一个，那么把表达式存储到目标列中之前把它应用到表达式。这样一个造型的实现函数总是采用一个额外的integer类型的参数，它接收目标列的*atttypmod*值（通常是它被声明的长度，尽管对于不同数据类型*atttypmod*有不同的解释），并且它可能采用第三个boolean参数来说明造型是显式的还是隐式的。该造型函数负责应用任何长度相关的语义，例如尺寸检查或截断。

例 7.9. character存储类型转换

对于一个声明为character(20)的目标列，下面的语句展示了被存储的值如何被正确地调整尺寸：

```
CREATE TABLE vv (v character(20));
INSERT INTO vv SELECT 'abc' || 'def';
SELECT v, octet_length(v) FROM vv;

          v          | octet_length
---------------------+--------------
 abcdef              |           20
(1 row)
```

实际发生的事情是两个未知文字被默认决定为text，允许||操作符被决定为text连接。 然后操作符的text结果被转换成bpchar（“空白填充字符”，character数据类型的内部名称）来匹配目标列类型（由于从text到bpchar的转换是二进制强制的，这个转换不会插入任何实际的函数调用）。最后，尺寸调整函数bpchar(bpchar, integer, boolean)被从系统目录中找到并应用到操作符的结果和存储的列长度上。这个类型相关的函数执行必要的长度检查并增加填充的空间。

7.5. UNION、CASE和相关结构

SQL UNION结构必须使可能不相似的类型匹配成为一个单一的结果集。该决定算法被独立地应用到一个联合查询的每个输出列。INTERSECT和EXCEPT采用和UNION相同的方法来决定不相似的类型。CASE、ARRAY、VALUES、GREATEST和LEAST结构使用相同的算法来使它们的组成表达式匹配并选择一种结果数据类型。

过程 7.4. UNION、CASE和相关结构的类型决定

1. 如果所有的输入为相同类型，并且不是unknown，那么就决定是该类型。

2. 如果任何输入是一种域类型，在所有后续步骤中都把它当做该域的基类型。 [4]

3. 如果所有的输入为unknown类型，则决定为text（字符串分类的首选类型）类型。否则，为了剩余规则，unknown输入会被忽略。

4. 如果非未知输入不全是相同的类型分类，则失败。

5. 如果有的话，选择第一个在其分类中作为首选类型的非未知输入类型。

[4] 多少有些类似于对待用于操作符和函数的域输入的方式，这种行为允许一种域类型能通过一个UNION或相似的结构保留下来，只要用户确保所有的输入都是（显式地或隐式地）准确类型。否则会优先选择该域的基类型。

6. 否则，选择最后的非未知输入类型，它允许所有在前面的非未知输入被隐式地转换为它（总有这样的一种类型，因为至少在列表中的第一个类型必须满足这个条件）。

7. 转换所有的输入为选定的类型。如果没有一个从给定输入到选定类型的转换将会失败。

下面是一些例子。

例 7.10. 联合中未指定类型的类型决定

```
SELECT text 'a' AS "text" UNION SELECT 'b';

 text
------
 a
 b
(2 rows)
```

这里，未知类型文字'b'将被决定为类型text。

例 7.11. 简单联合中的类型决定

```
SELECT 1.2 AS "numeric" UNION SELECT 1;

 numeric
---------
       1
     1.2
(2 rows)
```

文字1.2是numeric类型，且integer值1可以被隐式地造型为numeric，因此使用numeric类型。

例 7.12. 可换位联合中的类型决定

```
SELECT 1 AS "real" UNION SELECT CAST('2.2' AS REAL);

 real
------
    1
  2.2
(2 rows)
```

这里，由于类型real被能被隐式地造型为integer，而integer可以被隐式地造型为real，联合结果类型被决定为real。

例 7.13. 嵌套合并中的类型决定

```
SELECT NULL UNION SELECT NULL UNION SELECT 1;

ERROR:  UNION types text and integer cannot be matched
```

这个失败发生的原因是UXDB把多个UNION当作是成对操作的嵌套，也就是说上面的输入等同于：

(SELECT NULL UNION SELECT NULL) UNION SELECT 1;

根据上面给定的规则，内层的UNION被确定为类型text。然后外层的UNION的输入是类型text和integer，这就导致了上面看到的错误。通过确保最左边的UNION至少有一个输入类型为想要的结果类型，就可以修正这个问题。

INTERSECT和EXCEPT操作也被当作成对操作。不过，这一节中描述的其他结构会在一个决定步骤中考虑所有的输入。

7.6. SELECT的输出列

前面的小节中给出的规则将会导致对SQL查询中的所有表达式分配非unknown数据类型，不过作为SELECT命令的简单输出列出现的未指定类型的文本除外。例如，在

SELECT 'Hello World';

中没有标识该字符串应该取何种类型。在这种情况下，UXDB将会退而求其次将其类型决定为text。

当SELECT处于UNION（或者INTERSECT，或者EXCEPT）结构的一边或者出现在INSERT ... SELECT中时，这条规则就不适用了，因为在前面小节中给出的规则会优先。在第一种情况下未指定类型文本的类型将从UNION的另一边取得，而在第二种情况下未指定类型文本的类型将从目标列取得。

出于这样的目的，RETURNING列表采用和SELECT输出列表同样的方式对待。

第 8 章 索引

索引是提高数据库性能的常用途径。比起没有索引，使用索引可以让数据库服务器更快找到并获取特定行。但是索引同时也会增加数据库系统的日常管理负担，因此我们应该聪明地使用索引。

8.1. 简介

假设我们有一个如下的表：

```
CREATE TABLE test1 (
    id integer,
    content varchar
);
```

而应用发出很多以下形式的查询：

```
SELECT content FROM test1 WHERE id = constant;
```

在没有事前准备的情况下，系统不得不扫描整个test1表，一行一行地去找到所有匹配的项。如果test1中有很多行但是只有一小部分行（可能是0或者1）需要被该查询返回，这显然是一种低效的方式。但是如果系统被指示维护一个在*id*列上的索引，它就能使用一种更有效的方式来定位匹配行。例如，它可能仅仅需要遍历一棵搜索树的几层而已。

类似的方法也被用于大部分非小说书籍中：经常被读者查找的术语和概念被收集在一个字母序索引中放在书籍的末尾。感兴趣的读者可以相对快地扫描索引并跳到合适的页而不需要阅读整本书来寻找感兴趣的材料。正如作者的任务是准备好读者可能会查找的术语一样，数据库程序员也需要预见哪些索引会有用。

正如前面讨论的，下列命令可以用来在*id*列上创建一个索引：

```
CREATE INDEX test1_id_index ON test1 (id);
```

索引的名字test1_id_index可以自由选择，但我们最好选择一个能让我们想起该索引用途的名字。

为了移除一个索引，可以使用DROP INDEX命令。索引可以随时被创建或删除。

一旦一个索引被创建，就不再需要进一步的干预：系统会在表更新时更新索引，而且会在它觉得使用索引比顺序扫描表效率更高时使用索引。但我们可能需要定期地运行ANALYZE命令来更新统计信息以便查询规划器能做出正确的决定。通过第 11 章 性能提示的信息可以了解如何找出一个索引是否被使用以及规划器在何时以及为什么会选择不使用索引。

索引也会使带有搜索条件的UPDATE和DELETE命令受益。此外索引还可以在连接搜索中使用。因此，一个定义在连接条件列上的索引可以显著地提高连接查询的速度。

在一个大表上创建一个索引会耗费很长的时间。默认情况下，UXDB允许在索引创建时并行地进行读（SELECT命令），但写（INSERT、UPDATE和DELETE）则会被阻塞直到索引创建完成。在生产环境中这通常是不可接受的。在创建索引时允许并行的写是可能的，但是有些警告需要注意，更多信息可以参考“并发构建索引”一节。

一个索引被创建后，系统必须保持它与表同步。这增加了数据操作的负担。因此哪些很少或从不在查询中使用的索引应该被移除。

8.2. 索引类型

UXDB提供了多种索引类型： B-tree、Hash、GiST、SP-GiST 、GIN 和 BRIN。每一种索引类型使用了 一种不同的算法来适应不同类型的查询。默认情况下， CREATE INDEX命令创建适合于大部分情况的B-tree 索引。

B-tree可以在可排序数据上的处理等值和范围查询。特别地，UXDB的查询规划器会在任何一种涉及到以下操作符的已索引列上考虑使用B-tree索引：

<
<=
=
>=
>
将这些操作符组合起来，例如BETWEEN和IN，也可以用B-tree索引搜索实现。同样，在索引列上的IS NULL或IS NOT NULL条件也可以在B-tree索引中使用。

优化器也会将B-tree索引用于涉及到模式匹配操作符LIKE和~ 的查询，前提是如果模式是一个常量且被固定在字符串的开头—例如：col LIKE 'foo%'或者col ~ '^foo', 但在col LIKE '%bar'上则不会。但是，如果我们的数据库没有使用C区域设置，我们需要创建一个具有特殊操作符类的索引来支持模式匹配查询，参见下面的第 8.10 节 “操作符类和操作符族”。同样可以将B-tree索引用于ILIKE和~*，但仅当模式以非字母字符开始，即不受大小写转换影响的字符。

B-tree索引也可以用于检索排序数据。这并不会总是比简单扫描和排序更快，但是总是有用的。

Hash索引只能处理简单等值比较。不论何时当一个索引列涉及到一个使用了=操作符的比较时，查询规划器将考虑使用一个Hash索引。下面的命令将创建一个Hash索引：

CREATE INDEX *name* ON *table* USING HASH (*column*);

GiST索引并不是一种单独的索引，而是可以用于实现很多不同索引策略的基础设施。相应地，可以使用一个GiST索引的特定操作符根据索引策略（操作符类）而变化。作为一个例子，UXDB的标准捐献包中包括了用于多种二维几何数据类型的GiST操作符类，它用来支持使用下列操作符的索引化查询：

<<
&<
&>
>>
<<|
&<|
|&>
|>>
@>
<@
~=
&&
（这些操作符的含义见第 6.11 节 “几何函数和操作符”）contrib集合中还包括了很多其他GiST操作符类。

GiST索引也有能力优化“最近邻”搜索，例如：

```
SELECT * FROM places ORDER BY location <-> point '(101,456)' LIMIT 10;
```

它将找到离给定目标点最近的10个位置。能够支持这种查询的能力同样取决于被使用的特定操作符类。

和GiST相似，SP-GiST索引为支持多种搜索提供了一种基础结构。SP-GiST 允许实现众多不同的非平衡的基于磁盘的数据结构，例如四叉树、k-d树和radix树。作为一个例 子，UXDB的标准捐献包中包含了一个用于二维点的SP-GiST操作符类，它用于支持使用下列操作符的索引化查询：

<<
>>
~=
<@
<^
>^
其含义见第 6.11 节 “几何函数和操作符”。

GIN 索引是“倒排索引”，它适合于包含多个组成值的数据值，例如数组。倒排索引中为每一个组成值都包含一个单独的项，它可以高效地处理测试指定组成值是否存在的查询。

与 GiST 和 SP-GiST相似， GIN 可以支持多种不同的用户定义的索引策略，并且可以与一个 GIN 索引配合使用的特定操作符取决于索引策略。作为一个例子，UXDB的标准贡献包中包含了用于数组的GIN操作符类，它用于支持使用下列操作符的索引化查询：

<@
@>
=
&&
这些操作符的含义见第 6.18 节 “数组函数和操作符”。

BRIN 索引（块范围索引的缩写）存储有关存放在一个表的连续物理块范围上的值摘要信息。与 GiST、SP-GiST 和 GIN 相似，BRIN 可以支持很多种不同的索引策略，并且可以与一个 BRIN 索引配合使用的特定操作符取决于索引策略。对于具有线性排序顺序的数据类型，被索引的数据对应于每个块范围的列中值的最小值和最大值，使用这些操作符来支持用到索引的查询：

<
<=
=
>=
>

8.3. 多列索引

一个索引可以定义在表的多个列上。例如，我们有这样一个表：

```
CREATE TABLE test2 (
  major int,
  minor int,
  name varchar
```

);

（即将我们的/dev目录保存在数据库中）而且我们经常会做如下形式的查询：

SELECT name FROM test2 WHERE major = *constant* AND minor = *constant*;

那么我们可以在*major*和*minor*上定义一个索引：

CREATE INDEX test2_mm_idx ON test2 (major, minor);

目前，只有 B-tree、GiST、GIN 和 BRIN 索引类型支持多列索引，最多可以指定32个列（该限制可以在源代码文件ux_config_manual.h中修改，但是修改后需要重新编译UXDB）。

一个B-tree索引可以用于条件中涉及到任意索引列子集的查询，但是当先导列（即最左边的那些列）上有约束条件时索引最为有效。确切的规则是：在先导列上的等值约束，加上第一个无等值约束的列上的不等值约束，将被用于限制索引被扫描的部分。在这些列右边的列上的约束将在索引中被检查，这样它们适当节约了对表的访问，但它们并未减小索引被扫描的部分。例如，在(a, b, c)上有一个索引并且给定一个查询条件WHERE a = 5 AND b >= 42 AND c < 77，对索引的扫描将从第一个具有a = 5和b = 42的项开始向上进行，直到最后一个具有a = 5的项。在扫描过程中，具有c >= 77的索引项将被跳过，但是它们还是会被扫描到。这个索引在原则上可以被用于在b和/或c上有约束而在a上没有约束的查询，但是整个索引都不得不被扫描，因此在大部分情况下规划器宁可使用一个顺序的表扫描来替代索引。

一个多列GiST索引可以用于条件中涉及到任意索引列子集的查询。在其余列上的条件将限制由索引返回的项，但是第一列上的条件是决定索引上扫描量的最重要因素。当第一列中具有很少的可区分值时，一个GiST索引将会相对比较低效，即便在其他列上有很多可区分值。

一个GIN索引可以用于条件中涉及到任意索引列子集的查询。与B-tree和GiST不同，GIN的搜索效率与查询条件中使用哪些索引列无关。

多列 BRIN 索引可以被用于涉及该索引被索引列的任意子集的查询条件。和 GIN 相似且不同于 B-树 或者 GiST，索引搜索效率与查询条件使用哪个索引列无关。在单个表上使用多个 BRIN 索引来取代一个多列 BRIN 索引的唯一原因是为了使用不同的pages_per_range存储参数。

当然，要使索引起作用，查询条件中的列必须要使用适合于索引类型的操作符，使用其他操作符的子句将不会被考虑使用索引。

多列索引应该较少地使用。在绝大多数情况下，单列索引就足够了且能节约时间和空间。具有超过三个列的索引不太有用，除非该表的使用是极端程式化的。第 8.5 节 “组合多个索引”以及第 8.9 节 “只用索引的扫描和覆盖索引”中有对不同索引配置优点的讨论。

8.4. 索引和ORDER BY

除了简单地查找查询要返回的行外，一个索引可能还需要将它们以指定的顺序传递。这使得查询中的ORDER BY不需要独立的排序步骤。在UXDB当前支持的索引类型中，只有B-tree可以产生排序后的输出，其他索引类型会把行以一种没有指定的且与实现相关的顺序返回。

规划器会考虑以两种方式来满足一个ORDER BY说明：扫描一个符合说明的可用索引，或者先以物理顺序扫描表然后再显式排序。对于一个需要扫描表的大部分的查询，一个显式的排序很可能比使用一个索引更快，因为其顺序访问模式使得它所需要的磁盘I/O更少。只有在少数行需要被取出时，索引才会更有用。一种重要的特殊情况是ORDER BY与LIMIT *n*联合使用：一个显式的

排序将会处理所有的数据来确定最前面的*n*行，但如果有一个符合ORDER BY的索引，前*n*行将会被直接获取且根本不需要扫描剩下的数据。

默认情况下，B-tree索引将它的项以升序方式存储，并将空值放在最后(表TID被处理为其它相等条目之间的分线器列)。这意味着对列x上索引的一次前向扫描将产生满足ORDER BY x（或者更长的形式：ORDER BY x ASC NULLS LAST）的结果。索引也可以被后向扫描，产生满足ORDER BY x DESC（ORDER BY x DESC NULLS FIRST， NULLS FIRST是ORDER BY DESC的默认情况）。

我们可以在创建B-tree索引时通过ASC、DESC、NULLS FIRST和NULLS LAST选项来改变索引的排序，例如：

```
CREATE INDEX test2_info_nulls_low ON test2 (info NULLS FIRST);
CREATE INDEX test3_desc_index ON test3 (id DESC NULLS LAST);
```

一个以升序存储且将空值前置的索引可以根据扫描方向来支持ORDER BY x ASC NULLS FIRST或ORDER BY x DESC NULLS LAST。

读者可能会疑惑为什么要麻烦地提供所有四个选项，因为两个选项连同可能的后向扫描可以覆盖所有ORDER BY的变体。在单列索引中这些选项确实有冗余，但是在多列索引中它们却很有用。考虑(x, y)上的一个两列索引：它可以通过前向扫描满足ORDER BY x, y，或者通过后向扫描满足ORDER BY x DESC, y DESC。但是应用可能需要频繁地使用ORDER BY x ASC, y DESC。这样就没有办法从通常的索引中得到这种顺序，但是如果将索引定义为(x ASC, y DESC)或者(x DESC, y ASC)就可以产生这种排序。

显然，具有非默认排序的索引是相当专门的特性，但是有时它们会为特定查询提供巨大的速度提升。是否值得维护这样一个索引取决于我们会多频繁地使用需要特殊排序的查询。

8.5. 组合多个索引

只有查询子句中在索引列上使用了索引操作符类中的操作符并且通过AND连接时才能使用单一索引。例如，给定一个(a, b) 上的索引，查询条件WHERE a = 5 AND b = 6可以使用该索引，而查询WHERE a = 5 OR b = 6不能直接使用该索引。

幸运的是，UXDB具有组合多个索引（包括多次使用同一个索引）的能力来处理那些不 能用单个索引扫描实现的情况。系统能在多个索引扫描之间安排AND和OR条件。例如， WHERE x = 42 OR x = 47 OR x = 53 OR x = 99这样一个查询可以被分解成为四个独立的在x上索引扫描，每一个扫描使用其中一个条件。这些查询的结果将被“或”起来形成最后的结果。另一个例子是如果我们在x和y上都有独立的索引，WHERE x = 5 AND y = 6这样的查询的一种可能的实现方式就是分别使用两个索引配合相应的条件，然后将结果“与”起来得到最后的结果行。

为了组合多个索引，系统扫描每一个所需的索引并在内存中准备一个位图用于指示表中符合索引条件的行的位置。然后这些位图会被根据查询的需要“与”和“或”起来。最后，实际的表行将被访问并返回。表行将被以物理顺序访问，因为位图就是以这种顺序布局的。这意味着原始索引中的任何排序都会被丢失，并且如果存在一个ORDER BY子句就需要一个单独的排序步骤。由于这个原因以及每一个附加的索引都需要额外的时间，即使有额外的索引可用，规划器有时也会选择使用单一索引扫描。

在所有的应用（除了最简单的应用）中，可能会有多种有用的索引组合，数据库开发人员必须做出权衡以决定提供哪些索引。有时候多列索引最好，但是有时更好的选择是创建单独的索引并依赖于索引组合特性。例如，如果我们的查询中有时只涉及到列x，有时候只涉及到列y，还有时候会同时涉及到两列，我们可以选择在x和y上创建两个独立索引然后依赖索引组合来处理同时涉及

到两列的查询。我们当然也可以创建一个(x, y)上的多列索引。当查询同时涉及到两列时，该索引会比组合索引效率更高，但是正如第 8.3 节 “多列索引”中讨论的，它在只涉及到y的查询中几乎完全无用，因此它不能是唯一的一个索引。一个多列索引和一个y上的独立索引的组合将会工作得很好。多列索引可以用于那些只涉及到x的查询，尽管它比x上的独立索引更大且更慢。最后一种选择是创建所有三个索引，但是这种选择最适合表经常被执行所有三种查询但是很少被更新的情况。如果其中一种查询要明显少于其他类型的查询，我们可能需要只为常见类型的查询创建两个索引。

8.6. 唯一索引

索引也可以被用来强制列值的唯一性，或者是多个列组合值的唯一性。

CREATE UNIQUE INDEX *name* ON *table* (*column* [, ...]);

当前，只有B-tree能够被声明为唯一。

当一个索引被声明为唯一时，索引中不允许多个表行具有相同的索引值。空值被视为不相同。一个多列唯一索引将会拒绝在所有索引列上具有相同组合值的表行。

UXDB会自动为定义了一个唯一约束或主键的表创建一个唯一索引。该索引包含组成主键或唯一约束的所有列（可能是一个多列索引），它也是用于强制这些约束的机制。

> **注意**
>
> 不需要手工在唯一列上创建索引，如果那样做也只是重复了自动创建的索引而已。

8.7. 表达式索引

一个索引列并不一定是底层表的一个列，也可以是从表的一列或多列计算而来的一个函数或者标量表达式。这种特性对于根据计算结果快速获取表中内容是有用的。

例如，一种进行大小写不敏感比较的常用方法是使用lower函数：

SELECT * FROM test1 WHERE lower(col1) = 'value';

这种查询可以利用一个建立在lower(col1)函数结果之上的索引：

CREATE INDEX test1_lower_col1_idx ON test1 (lower(col1));

表达式索引还允许控制唯一索引的范围。例如，此唯一索引可防止在double precision类型列中存储重复的整数值:

CREATE UNIQUE INDEX test1_uniq_int ON tests ((floor(double_col)))

如果我们将该索引声明为UNIQUE，它将阻止创建在col1值上只有大小写不同的行。

另外一个例子，如果我们经常进行如下的查询：

SELECT * FROM people WHERE (first_name || ' ' || last_name) = 'John Smith';

那么值得创建一个这样的索引：

```
CREATE INDEX people_names ON people ((first_name || ' ' || last_name));
```

正如第二个例子所示，`CREATE INDEX`命令的语法通常要求在被索引的表达式周围书写圆括号。而如第一个例子所示，当表达式只是一个函数调用时可以省略掉圆括号。

索引表达式的维护代价较为昂贵，因为在每一个行被插入或更新时都得为它重新计算相应的表达式。然而，索引表达式在进行索引搜索时却不需要重新计算，因为它们的结果已经被存储在索引中了。在上面两个例子中，系统将会发现查询的条件是WHERE indexedcolumn = 'constant'，因此查询的速度将等同于其他简单索引查询。因此，表达式索引对于检索速度远比插入和更新速度重要的情况非常有用。

8.8. 部分索引

一个*部分索引*是建立在表的一个子集上，而该子集则由一个条件表达式（被称为部分索引的*谓词*）定义。而索引中只包含那些符合该谓词的表行的项。部分索引是一种专门的特性，但在很多种情况下它们也很有用。

使用部分索引的一个主要原因是避免索引公值。由于搜索一个公值的查询（一个在所有表行中占比超过一定百分比的值）不会使用索引，所以完全没有理由将这些行保留在索引中。这可以减小索引的尺寸，同时也将加速使用索引的查询。它也将加速很多表更新操作，因为这种索引并不需要在所有情况下都被更新。例 8.1 “建立一个部分索引来排除公值”展示了一种可能的应用：

例 8.1. 建立一个部分索引来排除公值

假设我们要在一个数据库中保存网页服务器访问日志。大部分访问都来自于我们组织内的IP地址，但是有些来自于其他地方（如使用拨号连接的员工）。如果我们主要通过IP搜索来自于外部的访问，我们就没有必要索引对应于我们组织内网的IP范围。

假设有这样一个表：

```
CREATE TABLE access_log (
    url varchar,
    client_ip inet,
    ...
);
```

用以下命令可以创建适用于我们的部分索引：

```
CREATE INDEX access_log_client_ip_ix ON access_log (client_ip)
WHERE NOT (client_ip > inet '192.168.100.0' AND
           client_ip < inet '192.168.100.255');
```

一个使用该索引的典型查询是：

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '212.78.10.32';
```

此处查询的IP地址由部分索引覆盖。以下查询无法使用部分索引，因为它使用从索引中排除的IP 地址：

```
SELECT *
FROM access_log
WHERE url = '/index.html' AND client_ip = inet '192.168.100.23';
```

可以看到部分索引查询要求公值能被预知，因此部分索引最适合于数据分布不会改变的情况。这样的索引也可以偶尔被重建来适应新的数据分布，但是这会增加维护负担。

例　8.2　“建立一个部分索引来排除不感兴趣的值”展示了部分索引的另一个可能的用途：从索引中排除那些查询不感兴趣的值。这导致了上述相同的好处，但它防止了通过索引来访问“不感兴趣的”值，即便在这种情况下一个索引扫描是有益的。显然，为这种场景建立部分索引需要很多考虑和实验。

例　8.2.　建立一个部分索引来排除不感兴趣的值

如果我们有一个表包含已上账和未上账的订单，其中未上账的订单在整个表中占据一小部分且它们是最经常被访问的行。我们可以通过只在未上账的行上创建一个索引来提高性能。创建索引的命令如下：

```
CREATE INDEX orders_unbilled_index ON orders (order_nr)
  WHERE billed is not true;
```

使用该索引的一个可能查询是：

```
SELECT * FROM orders WHERE billed is not true AND order_nr < 10000;
```

然而，索引也可以用于完全不涉及*order_nr*的查询，例如：

```
SELECT * FROM orders WHERE billed is not true AND amount > 5000.00;
```

这并不如在*amount*列上部分索引有效，因为系统必须扫描整个索引。然而，如果有相对较少的未上账订单，使用这个部分索引来查找未上账订单将会更好。

注意这个查询将不会使用该索引：

```
SELECT * FROM orders WHERE order_nr = 3501;
```

订单3501可能在已上账订单或未上账订单中。

例　8.2　“建立一个部分索引来排除不感兴趣的值”也显示索引列和谓词中使用的列并不需要匹配。UXDB支持使用任意谓词的部分索引，只要其中涉及的只有被索引表的列。然而，记住谓词必须匹配在将要受益于索引的查询中使用的条件。更准确地，只有当系统能识别查询的WHERE条件从数学上索引的谓词时，一个部分索引才能被用于一个查询。UXDB并不能给出一个精致的定理证明器来识别写成不同形式在数学上等价的表达式（一方面创建这种证明器极端困难，另一方面即便能创建出来对于实用也过慢）。系统可以识别简单的不等蕴含，例如“x　<　1”蕴含“x　<　2”；否则谓词条件必须准确匹配查询的WHERE条件中的部分，或者索引将不会被识别为可用。匹配发生在查询规划期间而不是运行期间。因此，参数化查询子句无法配合一个部分索引工作。例如，对于参数的所有可能值来说，一个具有参数“x　<　?”的预备查询绝不会蕴含“x　<　2”。

部分索引的第三种可能的用途并不要求索引被用于查询。其思想是在一个表的子集上创建一个唯一索引，如例　8.3　“建立一个部分唯一索引”所示。这对那些满足索引谓词的行强制了唯一性，而对那些不满足的行则没有影响。

例　8.3.　建立一个部分唯一索引

假设我们有一个描述测试结果的表。我们希望保证其中对于一个给定的主题和目标组合只有一个“成功”项，但其中可能会有任意多个“不成功”项。实现它的方式是：

```
CREATE TABLE tests (
    subject text,
    target text,
    success boolean,
    ...
);

CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
    WHERE success;
```

当有少数成功测试和很多不成功测试时这是一种特别有效的方法。

通过使用部分索引子句只处理空列值，该索引只允许索引列中有一个空值，并且用一个表达式索引子句来索引 true 来替代 null:

```
CREATE UNIQUE INDEX tests_target_one_null ON tests ((target IS NULL)) WHERE target IS
 NULL;
```

最后，一个部分索引也可以被用来重载系统的查询规划选择。同样，具有特殊分布的数据集可能导致系统在它并不需要索引的时候选择使用索引。在此种情况下可以被建立，这样它将不会被那些无关的查询所用。通常，UXDB会对索引使用做出合理的选择（例如，它会在检索公值时避开索引，这样前面的例子只能节约索引尺寸，它并非是避免索引使用所必需的），非常不正确的规划选择则需要作为故障报告。

记住建立一个部分索引意味着我们知道的至少和查询规划器所知的一样多，尤其是我们知道什么时候一个索引会是有益的。构建这些知识需要经验和对于UXDB中索引工作方式的理解。在大部分情况下，一个部分索引相对于一个普通索引的优势很小。

8.9.　只用索引的扫描和覆盖索引

UXDB中的所有索引是二级索引,这意味着每个索引都是与表的主数据区（在UXDB术语称为表的堆中）分开存储。这意味着在普通索引扫描中，每行检索都需要从索引和堆中取数据。　此外，虽然匹配给定的可索引WHERE条件的索引条目通常在一起靠近存储，但它们引用的表行可能在堆中的任何地方。　因此索引扫描的堆访问部分涉及到对堆的大量随机访问，这可能很慢，特别是在传统旋转媒介上。如 第　8.5　节　“组合多个索引”中所述，位图扫描尝试通过按排序的顺序进行堆访问来减少成本，但这远远不够）。

为了解决这种性能问题，UXDB支持*只用索引的扫描*，这类扫描可以仅用一个索引来回答查询而不产生任何堆访问。其基本思想是直接从每一个索引项中直接返回值，而不是去参考相关的堆项。在使用这种方法时有两个根本的限制：

1. 索引类型必须支持只用索引的扫描。B-树索引总是支持只用索引的扫描。GiST　和　SP-GiST　索引只对某些操作符类支持只用索引的扫描。其他索引类型不支持这种扫描。底层的要求是索引必须在物理上存储或者可以重构出每一个索引项对应的原始数据值。GIN　索引是

一个不支持只用索引的扫描的反例，因为它的每一个索引项通常只包含原始数据值的一部分。

2. 查询必须只引用存储在该索引中的列。例如，给定的索引建立在表的列x和y上，而该表还有一个列z，这些查询可以使用只用索引的扫描：

```
SELECT x, y FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND y < 42;
```

但是这些查询不能使用只用索引的查询：

```
SELECT x, z FROM tab WHERE x = 'key';
SELECT x FROM tab WHERE x = 'key' AND z < 42;
```

（如下面所讨论的，表达式索引和部分索引会让这条规则更加复杂）。

如果符合这两个根本要求，那么该查询所要求的所有数据值都可以从索引得到，因此才可能使用只用索引的扫描。但是对UXDB中的任何表扫描还有一个额外的要求：必须验证每一个检索到的行对该查询的 MVCC 快照是“可见的”，如第 10 章 并发控制讨论的那样。可见性信息并不存储在索引项中，只存储在堆项中。因此，乍一看似乎每一次行检索无论如何都会要求一次堆访问。如果表行最近被修改过，确实是这样。但是，对于很少更改的数据有一种方法可以解决这个问题。UXDB为表堆中的每一个页面跟踪是否其中所有的行的年龄都足够大，以至于对所有当前以及未来的事务都可见。这个信息存储在该表的可见性映射的一个位中。在找到一个候选索引项后，只用索引的扫描会检查对应堆页面的可见性映射位。如果该位被设置，那么这一行就是可见的并且该数据库可以直接被返回。如果该位没有被设置，那么就必须访问堆项以确定这一行是否可见，这种情况下相对于标准索引扫描就没有性能优势。即便是在成功的情况下，这种方法也是把对堆的访问换成了对可见性映射的访问。不过由于可见性映射比它所描述的堆要小四个数量级，所以访问可见性映射所需的物理 I/O 要少很多。在大部分情况下，可见性映射总是会被保留在内存中的缓冲中。

总之，虽然当两个根本要求满足时可以使用只用索引的扫描，但是只有该表的堆页面中有很大一部分的“所有都可见”映射位被设置时这种索引才有优势。不过，有很大一部分行不被更改的表是很常见的，这也让这一类扫描在实际中非常有用。

为了有效利用仅索引扫描功能，您可以选择创建一个*覆盖索引*，它是一个特别设计的索引，包含经常运行的特殊类型查询所需要的列。由于查询通常需要检索的列不仅仅是他们搜索的列，UXDB允许您创建索引,这个索引中有些列只是“负荷”而不是搜索键的一部分。这可以通过添加INCLUDE来完成子句来列出了额外的列。 例如，如果您通常可以运行这样的查询：

```
SELECT y FROM tab WHERE x = 'key';
```

加快此类查询的传统方法是仅在x上的索引。但是，一个索引定义为

```
CREATE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

可以将这些查询作为仅索引扫描处理，因为y可以从索引中获取而不需要访问堆。

因为列y不是搜索键的一部分，它不必是索引可以处理的数据类型；它只存储在索引中，不由索引机解释。另外，如果索引是唯一的索引，则

```
CREATE UNIQUE INDEX tab_x_y ON tab(x) INCLUDE (y);
```

唯一性条件仅适用于x　　列，而不是x和y的组合。（如果使用和在索引中设置的类似语法，一个INCLUDE子句可以写在UNIQUE和PRIMARY KEY约束中。）

保守地将非键负载列添加到索引是明智的，尤其是宽列。　　如果索引元组超过索引类型允许的最大大小，数据插入将失败。在任何情况下，非键列都将复制索引表中的数据并放大了索引的大小，从而有可能减慢搜索速度。请记住，除非一个表足够慢以至于仅索引扫描可能不必访问堆，否则没有什么理由在一个索引中包含负载列。无论如何，如果必须访问堆元组，从堆里获取列的值并不会带来更高的开销。其他限制是表达式不被作为包含的来支持。只有B树和GiST索引当前支持包含的列。

在 UXDB有INCLUDE特性之前，人们有时会通过写负载列作为普通索引列来制作覆盖索引。它这样写：

```
CREATE INDEX tab_x_y ON tab(x, y);
```

即使他们无意将y用作WHER子句的一部分，只要额外的列是尾列就可以很好的工作。 让它们成为前导字段是不明智的，原因在第　8.3　节　“多列索引”中有说明。但是，此方法不支持您希望索引在键列上实施唯一性。

Suffix　truncation总是从B-Tree的上层移除非键列。作为有效负载列，它们从不用于指导索引扫描。　当键列的其余前缀恰好足以描述最低B-Tree级别上的元组时，截断过程还会删除一个或多个尾随的键列。　　实际中，不使用INCLUDE子句覆盖索引通常会避免存储在上层有效负载的列。 然而，显式地将有效负载列定义为非键列reliably使上层元组保持较小。

原则上，只用索引的扫描可以被用于表达式索引。例如，给定一个f(x)上的索引（x是一个表列），可以把

```
SELECT f(x) FROM tab WHERE f(x) < 1;
```

作为只用索引的扫描执行，如果f()是一个计算代价昂贵的函数，这会非常有吸引力。不过，UXDB的规划器当前面对这类情况时并不是很聪明。只有在索引中有查询所需要的所有列时，规划器才会考虑用只用索引的扫描来执行一个查询。在这个例子中，除了在f(x)环境中之外，查询的其他部分不需要x，但是规划器并不能意识到这一点，因此它会得出不能使用只用索引的扫描的结论。如果只用索引的扫描足够有价值，有一种解决方法是把该索引定义在(f(x),　x)上，其中第二个列实际上并不会被使用，它只是用来说服规划器可以使用只用索引的扫描而已。如果目标是避免重复计算f(x)，一个额外的警示是规划器不一定会把不在可索引WHERE子句中对f(x)的使用匹配到索引列。通常在上述那种简单查询中一切正常，但是涉及到连接的查询中就不行了。

部分索引也和只用索引的扫描之间有着有趣的关系。考虑例　8.3　“建立一个部分唯一索引”中所展示的部分索引：

```
CREATE UNIQUE INDEX tests_success_constraint ON tests (subject, target)
   WHERE success;
```

原则上，我们可以在这个索引上使用只用索引的扫描来满足查询

```
SELECT target FROM tests WHERE subject = 'some-subject' AND success;
```

但是有一个问题：WHERE子句引用的是不能作为索引结果列的success。尽管如此，还是可以使用只用索引的扫描，因为在运行时计划不需要重新检查WHERE子句的那个部分：在该索引中找

到的所有项必定具有success = true，因此在计划中检查这个部分的需要并不明显。UXDB会识别这种情况，并且允许生成只用索引的扫描。

8.10. 操作符类和操作符族

一个索引定义可以为索引中的每一列都指定一个操作符类。

CREATE INDEX *name* ON *table* (*column opclass* [*sort options*] [, ...]);

操作符类标识该列上索引要使用的操作符。例如，一个int4类型上的B树索引会使用int4_ops类，这个操作符类包括用于int4类型值的比较函数。实际上列的数据类型的默认操作符类通常就足够了。存在多个操作符类的原因是，对于某些数据类型可能会有多于一种的有意义的索引行为。例如，我们可能想要对一种复数数据类型按照绝对值排序或者按照实数部分排序。我们可以通过为该数据类型定义两个操作符类来实现，并且在创建一个索引时选择合适的类。操作符类会决定基本的排序顺序（可以通过增加排序选项COLLATE、 ASC/DESC和/或 NULLS FIRST/NULLS LAST来修改）。

除了默认的操作符类，还有一些内建的操作符类：

- 操作符类text_pattern_ops、varchar_pattern_ops和 bpchar_pattern_ops分别支持类型text、varchar和 char上的B树索引。它们与默认操作符类的区别是值的比较是严格按照字符进行而不是根据区域相关的排序规则。这使得这些操作符类适合于当一个数据库没有使用标准“C”区域时被使用在涉及模式匹配表达式（LIKE或POSIX正则表达式）的查询中。一个例子是，可以这样索引一个varchar列：

  CREATE INDEX test_index ON test_table (col varchar_pattern_ops);

  注意如果希望涉及到<、<=、>或>=比较的查询使用一个索引，也应该创建一个使用默认操作符类的索引。这些查询不能使用*xxx*_pattern_ops操作符类（但是普通的等值比较可以使用这些操作符类）。可以在同一个列上创建多个使用不同操作符类的索引。如果正在使用C区域，并不需要*xxx*_pattern_ops操作符类，因为在C区域中的模式匹配查询可以用带有默认操作符类的索引。

下面的查询展示了所有已定义的操作符类：

```
SELECT am.amname AS index_method,
    opc.opcname AS opclass_name,
    opc.opcintype::regtype AS indexed_type,
    opc.opcdefault AS is_default
  FROM ux_am am, ux_opclass opc
  WHERE opc.opcmethod = am.oid
  ORDER BY index_method, opclass_name;
```

一个操作符类实际上只是一个更大的被称为操作符族的结构的一个子集。在多种数据类型具有相似行为的情况下，常常会定义跨数据类型的操作符并且允许索引使用它们。为了实现该目的，这些类型的操作符类必须被分组到同一个操作符族中。跨类型的操作符是该族的成员，但是并不与族内任意一个单独的类相关联。

前一个查询的扩展版本展示了每个操作符类所属的操作符族：

```
SELECT am.amname AS index_method,
```

```
    opc.opcname AS opclass_name,
    opf.opfname AS opfamily_name,
    opc.opcintype::regtype AS indexed_type,
    opc.opcdefault AS is_default
  FROM ux_am am, ux_opclass opc, ux_opfamily opf
  WHERE opc.opcmethod = am.oid AND
      opc.opcfamily = opf.oid
  ORDER BY index_method, opclass_name;
```

这个查询展示所有已定义的操作符族和每一个族中包含的所有操作符：

```
SELECT am.amname AS index_method,
    opf.opfname AS opfamily_name,
    amop.amopopr::regoperator AS opfamily_operator
  FROM ux_am am, ux_opfamily opf, ux_amop amop
  WHERE opf.opfmethod = am.oid AND
      amop.amopfamily = opf.oid
  ORDER BY index_method, opfamily_name, opfamily_operator;
```

8.11. 索引和排序规则

一个索引在每一个索引列上只能支持一种排序规则。如果需要多种排序规则，可能需要多个索引。

考虑这些语句：

```
CREATE TABLE test1c (
  id integer,
  content varchar COLLATE "x"
);

CREATE INDEX test1c_content_index ON test1c (content);
```

该索引自动使用下层列的排序规则。因此一个如下形式的查询：

```
SELECT * FROM test1c WHERE content > constant;
```

可以使用该索引，因为比较会默认使用列的排序规则。但是，这个索引无法加速涉及到某些其他排序规则的查询。因此对于下面形式的查询：

```
SELECT * FROM test1c WHERE content > constant COLLATE "y";
```

可以创建一个额外的支持"y"排序规则的索引，例如：

```
CREATE INDEX test1c_content_y_index ON test1c (content COLLATE "y");
```

8.12. 检查索引使用

尽管UXDB中的索引并不需要维护或调优，但是检查真实的查询负载实际使用了哪些索引仍然非常重要。检查一个独立查询的索引使用情况可以使用EXPLAIN(7)命令，它应用于这种目的的内容

在第 11.1 节 “使用EXPLAIN”中有介绍。也可以在一个运行中的服务器上收集有关索引使用的总体统计情况。

很难明确地表达决定创建哪些索引的通用过程。在之前的小节中的例子里有一些典型的情况。通常需要大量的实验才能决定应该创建哪些索引。本小节剩余的部分将给出一些创建索引的提示：

- 总是先运行ANALYZE(7)。这个命令会收集有关表中值分布情况的统计信息。估计一个查询将要返回的行数需要这些信息，而结果行数则被规划器用来为每一个可能的查询计划分配实际的代价。如果没有任何真实的统计信息，将会假定一些默认值，这几乎肯定是不准确的。在没有运行的情况下检查一个应用的索引使用情况是注定要失败的。

- 使用真实数据进行实验。使用测试数据来建立索引将会告诉测试数据需要什么样的索引，但这并不代表真实数据的需要。

  使用非常小的测试数据集是特别致命的。在从100000行中选出1000行时可能会用到索引，但是从100行里选出1行是很难用到索引的，因为100行完全可能放入到一个磁盘页面中，而没有任何计划能够比得上从一个磁盘页面顺序获取的计划。

  在创建测试数据时也要注意，特别是当应用还没有产生时通常是不可避免的。值非常相似、完全随机或以排好序的方式被插入都将使得统计信息倾斜于真实数据中的值分布。

- 如果索引没有被用到，强制使用它们将会对测试非常有用。有一些运行时参数可以关闭多种计划类型。例如，关闭顺序扫描（enable_seqscan）以及嵌套循环连接（enable_nestloop）将强制系统使用一种不同的计划。如果系统仍然选择使用一个顺序扫描或嵌套循环连接，则索引没有被使用的原因可能更加根本，例如查询条件不匹配索引（哪种查询能够使用哪种索引已经在前面的小节中解释过了）。

- 如果强制索引使用确实使用了索引，则有两种可能性：系统是正确的并且索引确实不合适，或者查询计划的代价估计并没有反映真实情况。因此应该对用索引的查询和不用索引的查询计时。此时EXPLAIN ANALYZE命令就能发挥作用了。

- 如果发现代价估计是错误的，也分为两种可能性。总代价是用每个计划节点的每行代价乘以计划节点的选择度估计来计算的。计划节点的代价估计可以通过运行时参数调整。不准确的选择度估计可能是由于缺乏统计信息，可以通过调节统计信息收集参数（见ALTER TABLE(7)）来改进。

  如果不能成功地把代价调整得更合适，那么可能必须依靠显式地强制索引使用。也可能希望联系UXDB开发者来检查该问题。

第 9 章 全文搜索

9.1. 介绍

全文搜索（或者文本搜索）提供了确定满足一个查询的自然语言文档的能力，并可以选择将它们按照与查询的相关度排序。最常用的搜索类型是找到所有包含给定查询词的文档并按照它们与查询的相似性顺序返回它们。查询和相似性的概念非常灵活并且依赖于特定的应用。最简单的搜索认为查询是一组词而相似性是查询词在文档中的频度。

文本搜索操作符已经在数据库中存在很多年了。UXDB对文本数据类型提供了~、~*、LIKE和ILIKE操作符，但是它们缺少现代信息系统所要求的很多基本属性：

- 即使对英语也缺乏语言的支持。正则表达式是不够的，因为它们不能很容易地处理派生词，例如satisfies和satisfy。可能会错过包含satisfies的文档，尽管可能想要在对于satisfy的搜索中找到它们。可以使用OR来搜索多个派生形式，但是这样做太罗嗦也容易出错（有些词可能有数千种派生）。
- 它们不提供对搜索结果的排序（排名），这使它们面对数以千计被找到的文档时变得无效。
- 它们很慢因为没有索引支持，因此它们必须为每次搜索处理所有的文档。

全文索引允许文档被预处理并且保存一个索引用于以后快速的搜索。预处理包括：

将文档解析成记号。标识出多种类型的记号是有所帮助的，例如数字、词、复杂的词、电子邮件地址，这样它们可以被以不同的方式处理。原则上记号分类取决于相关的应用，但是对于大部分目的都可以使用一套预定义的分类。UXDB使用一个解析器来执行这个步骤。其中提供了一个标准的解析器，并且为特定的需要也可以创建定制的解析器。

将记号转换成词位。和一个记号一样，一个词位是一个字符串，但是它已经被正规化，这样同一个词的不同形式被变成一样。例如，正规化几乎总是包括将大写字母转换成小写形式，并且经常涉及移除后缀（例如英语中的s或es）。这允许搜索找到同一个词的变体形式，而不需要冗长地输入所有可能的变体。此外，这个步骤通常会消除停用词，它们是那些太普通的词，它们对于搜索是无用的（简而言之，记号是文档文本的原始片段，而词位是那些被认为对索引和搜索有用的词）。UXDB使用词典来执行这个步骤。已经提供了多种标准词典，并且为特定的需要也可以创建定制的词典。

为搜索优化存储预处理好的文档。例如，每一个文档可以被表示为正规化的词位的一个有序数组。与词位一起，通常还想要存储用于近似排名的位置信息，这样一个包含查询词更“密集”区域的文档要比那些包含分散的查询词的文档有更高的排名。

词典允许对记号如何被正规化进行细粒度的控制。使用合适的词典，可以：

- 定义不应该被索引的停用词。
- 使用Ispell把同义词映射到一个单一词。
- 使用一个分类词典把短语映射到一个单一词。
- 使用一个Ispell词典把一个词的不同变体映射到一种规范的形式。
- 使用Snowball词干分析器规则将一个词的不同变体映射到一种规范的形式。

我们提供了一种数据类型tsvector来存储预处理后的文档，还提供了一种类型tsquery来表示处理过的查询（第 5.11 节 “文本搜索类型”）。有很多函数和操作符可以用于这些数据类型（第 6.13 节 “文本搜索函数和操作符”），其中最重要的是匹配操作符@@，它在第 9.1.2 节 “基本文本匹配”中介绍。全文搜索可以使用索引来加速（第 9.9 节 “GIN 和 GiST 索引类型”）。

9.1.1. 什么是一个文档？

一个document是在一个全文搜索系统中进行搜索的单元，例如，一篇杂志文章或电子邮件消息。文本搜索引擎必须能够解析文档并存储词位（关键词）与它们的父文档之间的关联。随后，这些关联会被用来搜索包含查询词的文档。

对于UXDB中的搜索，一个文档通常是一个数据库表中一行内的一个文本形式的域，或者可能是这类域的一个组合（连接），这些域可能存储在多个表或者是动态获取。换句话说，一个文档可能从用于索引的不同部分构建，并且它可能被作为一个整体存储在某个地方。例如：

```
SELECT title || ' ' ||  author || ' ' ||  abstract || ' ' || body AS document
FROM messages
WHERE mid = 12;

SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body AS document
FROM messages m, docs d
WHERE mid = did AND mid = 12;
```

> **注意**
>
> 实际上在这些例子查询中，coalesce应该被用来防止一个单一NULL属性导致整个文档的一个NULL结果。

另一种存储文档的可能性是作为文件系统中的简单文本文件。在这种情况下，数据库可以被用来存储全文索引并执行搜索，并且某些唯一标识符可以被用来从文件系统检索文档。但是，从数据库的外面检索文件要求超级用户权限或者特殊函数支持，因此这种方法通常不如把所有数据放在UXDB内部方便。另外，把所有东西放在数据库内部允许方便地访问文档元数据来协助索引和现实。

对于文本搜索目的，每一个文档必须被缩减成预处理后的tsvector格式。搜索和排名被整个在一个文档的tsvector表示上执行 — 只有当文档被选择来显示给用户时才需要检索原始文本。我们因此经常把tsvector说成是文档，但是当然它只是完整文档的一种紧凑表示。

9.1.2. 基本文本匹配

UXDB中的全文搜索基于匹配操作符@@，它在一个tsvector（文档）匹配一个tsquery（查询）时返回true。哪种数据类型写在前面没有影响：

```
SELECT 'a fat cat sat on a mat and ate a fat rat'::tsvector @@ 'cat & rat'::tsquery;
 ?column?
----------
 t

SELECT 'fat & cow'::tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;
 ?column?
----------
 f
```

正如以上例子所建议的，一个tsquery并不只是一个未经处理的文本，顶多一个tsvector是这样。一个tsquery包含搜索术语，它们必须是已经正规化的词位，并且可以使用 AND 、OR、NOT 以及 FOLLOWED BY 操作符结合多个术语（语法详见第 5.11.2 节 “tsquery”。有几个函数to_tsquery、plainto_tsquery以及phraseto_tsquery可用于将用户书写的文本转换为正确的tsquery，它们会主要采用正则化出现在文本中的词的方法。相似地，to_tsvector被用来解析和正规化一个文档字符串。因此在实际上一个文本搜索匹配可能看起来更像：

```
SELECT to_tsvector('fat cats ate fat rats') @@ to_tsquery('fat & rat');
 ?column?
----------
 t
```

注意如果这个匹配被写成下面这样它将不会成功：

```
SELECT 'fat cats ate fat rats'::tsvector @@ to_tsquery('fat & rat');
 ?column?
----------
 f
```

因为这里不会发生词rats的正规化。一个tsvector的元素是词位，它被假定为已经正规化好，因此rats不匹配rat。

@@操作符也支持text输出，它允许在简单情况下跳过从文本字符串到tsvector或tsquery的显式转换。可用的变体是：

```
tsvector @@ tsquery
tsquery  @@ tsvector
text @@ tsquery
text @@ text
```

前两种我们已经见过。形式text @@ tsquery等价于to_tsvector(x) @@ y。形式text @@ text等价于to_tsvector(x) @@ plainto_tsquery(y)。

在tsquery中，&（AND）操作符指定它的两个参数都必须出现在文档中才表示匹配。类似地，|（OR）操作符指定至少一个参数必须出现，而!（NOT）操作符指定它的参数不出现才能匹配。例如，查询fat & ! rat匹配包含fat但不包含rat的文档。

在<->（FOLLOWED BY） tsquery操作符的帮助下搜索可能的短语，只有该操作符的参数的匹配是相邻的并且符合给定顺序时，该操作符才算是匹配。例如：

```
SELECT to_tsvector('fatal error') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 t

SELECT to_tsvector('error is not fatal') @@ to_tsquery('fatal <-> error');
 ?column?
----------
 f
```

FOLLOWED BY 操作符还有一种更一般的版本，形式是<*N*>，其中*N*是一个表示匹配词位位置之间的差。<1>和<->相同，而<2>允许刚好一个其他词位出现在匹配之间，以此类推。当有些词是停用词时，phraseto_tsquery函数利用这个操作符来构造一个能够匹配多词短语的tsquery。例如：

```
SELECT phraseto_tsquery('cats ate rats');
      phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <-> 'rat'
```

```
SELECT phraseto_tsquery('the cats ate the rats');
     phraseto_tsquery
-------------------------------
 'cat' <-> 'ate' <2> 'rat'
```

一种有时候有用的特殊情况是，<0>可以被用来要求两个匹配同一个词的模式。

圆括号可以被用来控制tsquery操作符的嵌套。如果没有圆括号，|的计算优先级最低，然后从低到高依次是&、<->、!。

值得注意的是，当AND/OR/NOT操作符在一个FOLLOWED BY操作符的参数中时，它们表示与不在那些参数中时不同的含义，因为在FOLLOWED BY中匹配的准确位置是有意义的。例如，通常!x仅匹配在任何地方都不包含x的文档。但如果y不是紧接在一个x后面，!x <-> y就会匹配那个y，在文档中其他位置出现的x不会阻止匹配。另一个例子是，x & y通常仅要求x和y均出现在文档中的某处，但是(x & y) <-> z要求x和y在紧挨着z之前的同一个位置匹配。因此这个查询的行为会不同于x <-> z & y <-> z，它将匹配一个含有两个单独序列x z以及y z的文档（这个特定的查询一点用都没有，因为x和y不可能在同一个位置匹配，但是对于前缀匹配模式之类的更复杂的情况，这种形式的查询就会有用武之地）。

9.1.3. 配置

前述的都是简单的文本搜索例子。正如前面所提到的，全文搜索功能包括做更多事情的能力：跳过索引特定词（停用词）、处理同义词并使用更高级的解析，例如基于空白之外的解析。这个功能由*文本搜索配置*控制。UXDB中有多种语言的预定义配置，并且可以很容易地创建自己的配置（uxsql的\dF命令显示所有可用的配置）。

在安装期间一个合适的配置将被选择并且default_text_search_config也被相应地设置在uxsinodb.conf中。如果正在对整个集簇使用相同的文本搜索配置，可以使用在uxsinodb.conf中使用该值。要在集簇中使用不同的配置但是在任何一个数据库内部使用同一种配置，使用ALTER DATABASE ... SET。否则，可以在每个会话中设置default_text_search_config。

依赖一个配置的每一个文本搜索函数都有一个可选的regconfig参数，因此要使用的配置可以被显式指定。只有当这个参数被忽略时，default_text_search_config才被使用。

为了让建立自定义文本搜索配置更容易，一个配置可以从更简单的数据库对象来建立。UXDB的文本搜索功能提供了四类配置相关的数据库对象：

- *文本搜索解析器*将文档拆分成记号并分类每个记号（例如，作为词或者数字）。
- *文本搜索词典*将记号转变成正规化的形式并拒绝停用词。
- *文本搜索模板*提供位于词典底层的函数（一个词典简单地指定一个模板和一组用于模板的参数）。
- *文本搜索配置*选择一个解析器和一组用于将解析器产生的记号正规化的词典。

文本搜索解析器和模板是从低层 C 函数构建而来，因此它要求 C 编程能力来开发新的解析器和模板，并且还需要超级用户权限来把它们安装到一个数据库中（在UXDB发布的contrib/区域中有一些附加的解析器和模板的例子）。由于词典和配置只是对底层解析器和模板的参数化和连接，不需要特殊的权限来创建一个新词典或配置。创建定制词典和配置的例子将在本章稍后的部分给出。

9.2. 表和索引

在前一节中的例子演示了使用简单常数字符串进行全文匹配。本节展示如何搜索表数据，以及可选择地使用索引。

9.2.1. 搜索一个表

可以在没有一个索引的情况下做一次全文搜索。一个简单的查询将打印每一个行的title，这些行在其*body*域中包含词friend：

```
SELECT title
FROM uxweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

这将还会找到相关的词例如friends和friendly，因为这些都被约减到同一个正规化的词位。

以上的查询指定要使用english配置来解析和正规化字符串。我们也可以忽略配置参数：

```
SELECT title
FROM uxweb
WHERE to_tsvector(body) @@ to_tsquery('friend');
```

这个查询将使用由default_text_search_config设置的配置。

一个更复杂的例子是选择 10 个最近的文档，要求它们在title或body中包含create和table：

```
SELECT title
FROM uxweb
WHERE to_tsvector(title || ' ' || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

为了清晰，我们忽略coalesce函数调用，它可能需要被用来查找在这两个域之中包含NULL的行。

尽管这些查询可以在没有索引的情况下工作，大部分应用会发现这种方法太慢了，除了偶尔的临时搜索。实际使用文本搜索通常要求创建一个索引。

9.2.2. 创建索引

我们可以创建一个GIN索引（第 9.9 节 “GIN 和 GiST 索引类型”）来加速文本搜索：

```
CREATE INDEX uxweb_idx ON uxweb USING GIN(to_tsvector('english', body));
```

注意这里使用了to_tsvector的双参数版本。只有指定了一个配置名称的文本搜索函数可以被用在表达式索引（第 8.7 节 “表达式索引”）中。这是因为索引内容必须是没有被default_text_search_config影响的。如果它们被影响，索引内容可能会不一致因为不同的项可能包含被使用不同文本搜索配置创建的tsvector，并且没有办法猜测哪个是哪个。也没有可能正确地转储和恢复这样的一个索引。

由于to_tsvector的双参数版本被使用在上述的索引中，只有一个使用了带有相同配置名的双参数版to_tsvector的查询引用才能使用该索引。即，WHERE to_tsvector('english', body) @@ 'a & b' 可以使用该索引，但WHERE to_tsvector(body) @@ 'a & b'不能。这保证一个索引只能和创建索引项时所用的相同配置一起使用。

可以建立更复杂的表达式索引，在其中配置名被另一个列指定，例如：

```
CREATE INDEX uxweb_idx ON uxweb USING GIN(to_tsvector(config_name, body));
```

这里config_name是uxweb表中的一个列。这允许在同一个索引中有混合配置，同时记录哪个配置被用于每一个索引项。例如，如果文档集合包含不同语言的文档，这就可能会有用。同样，要使用索引的查询必须被措辞成匹配，例如WHERE to_tsvector(config_name, body) @@ 'a & b'。

索引甚至可以连接列：

```
CREATE INDEX uxweb_idx ON uxweb USING GIN(to_tsvector('english', title || ' ' || body));
```

另一种方法是创建一个单独的tsvector列来保存to_tsvector的输出。若要使此列与其源数据保持自动更新，用存储生成的列。这个例子是title和body的连接，使用coalesce来保证当其他域为NULL时一个域仍然能留在索引中：

```
ALTER TABLE uxweb
   ADD COLUMN textsearchable_index_col tsvector
          GENERATED ALWAYS AS (to_tsvector('english', coalesce(title, '') || ' ' || coalesce(body, ''))) 
 STORED;
```

然后我们创建一个GIN索引来加速搜索：

```
CREATE INDEX textsearch_idx ON uxweb USING GIN(textsearchable_index_col);
```

现在我们准备好执行一个快速的全文搜索了：

```
SELECT title
FROM uxweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC
LIMIT 10;
```

单独列方法相对于表达式索引的一个优势在于，它不必为了利用索引而在查询中显式地指定文本搜索配置。如上述例子所示，查询可以依赖default_text_search_config。另一个优势是搜索将会更快，因为它不必重做to_tsvector调用来验证索引匹配（在使用 GiST 索引时这一点比使用 GIN 索引时更重要；见第 9.9 节 “GIN 和 GiST 索引类型”）。表达式索引方法更容易建立，但是它要求更少的磁盘空间，因为tsvector表示没有被显式地存储下来。

9.3. 空值文本搜索

要实现全文搜索必须要有一个从文档创建tsvector以及从用户查询创建tsquery的函数。而且我们需要一种有用的顺序返回结果，因此我们需要一个函数能够根据文档与查询的相关性比较文档。还有一点重要的是要能够很好地显示结果。UXDB对所有这些函数都提供了支持。

9.3.1. 解析文档

UXDB提供了函数to_tsvector将一个文档转换成tsvector数据类型。

```
to_tsvector([ config regconfig, ] document text) returns tsvector
```

to_tsvector把一个文本文档解析成记号，把记号缩减成词位，并且返回一个tsvector，它列出了词位以及词位在文档中的位置。文档被根据指定的或默认的文本搜索配置来处理。下面是一个简单例子：

```
SELECT to_tsvector('english', 'a fat  cat sat on a mat - it ate a fat rats');
                  to_tsvector
-----------------------------------------------------
 'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
```

在上面这个例子中我们看到，作为结果的tsvector不包含词a、on或it，词rats变成了rat，并且标点符号-被忽略了。

to_tsvector函数在内部调用了一个解析器，它把文档文本分解成记号并且为每一种记号分配一个类型。对于每一个记号，会去查询一个词典列表（第 9.6 节 “词典”），该列表会根据记号的类型而变化。第一个识别记号的词典产生一个或多个正规化的词位来表示该记号。例如，rats变成rat是因为一个词典识别到该词rats是rat的复数形式。一些词会被识别为停用词（第 9.6.1 节 “停用词”），这将导致它们被忽略，因为它们出现得太频繁以至于在搜索中起不到作用。在我们的例子中有a、on和it是停用词。如果在列表中没有词典能识别该记号，那它将也会被忽略。在这个例子中标点符号-就属于这种情况，因为事实上没有词典会给它分配记号类型（空间符号），即空间记号不会被索引。对于解析器、词典以及要索引哪些记号类型是由所选择的文本搜索配置（第 9.7 节 “配置例子”）决定的。可以在同一个数据库中有多种不同的配置，并且有用于很多种语言的预定义配置。在我们的例子中，我们使用用于英语的默认配置english。

函数setweight可以被用来对tsvector中的项标注一个给定的权重，这里一个权重可以是四个字母之一：A、B、C或D。这通常被用来标记来自文档不同部分的项，例如标题对正文。稍后，这种信息可以被用来排名搜索结果。

因为to_tsvector(NULL) 将返回NULL，不论何时一个域可能为空时，我们推荐使用coalesce。下面是我们推荐的从一个结构化文档创建一个tsvector的方法：

```
UPDATE tt SET ti =
   setweight(to_tsvector(coalesce(title,'')), 'A')    ||
   setweight(to_tsvector(coalesce(keyword,'')), 'B')  ||
   setweight(to_tsvector(coalesce(abstract,'')), 'C') ||
   setweight(to_tsvector(coalesce(body,'')), 'D');
```

这里我们已经使用了setweight在完成的tsvector标注每一个词位的来源，并且接着将标注过的tsvector值用tsvector连接操作符||合并在一起（第 9.4.1 节 “操纵文档”给出了关于这些操作符的细节）。

9.3.2. 解析查询

UXDB提供了函数to_tsquery、plainto_tsquery、phraseto_tsquery以及websearch_to_tsquery用来把一个查询转换成tsquery数据类型。to_tsquery提供了比plainto_tsquery和phraseto_tsquery更多的特性，但是它对其输入要求更加严格。websearch_to_tsquery是to_tsquery的一个简化版本，它使用一种可选择的语法，类似于Web搜索引擎使用的语法。

```
to_tsquery([ config regconfig, ] querytext text) returns tsquery
```

to_tsquery从*querytext*创建一个tsquery值，该值由被tsquery操作符&（AND）、|（OR）、!（NOT）和<->（FOLLOWED BY）分隔的单个记号组成。 这些操作符可以

使用圆括号分组。换句话说，to_tsquery的输入必须已经遵循tsquery输入的一般规则，如第 5.11.2 节 “tsquery所述。区别在于基本的tsquery输入把记号当作表面值，而to_tsquery 会使用指定的或者默认的配置把每一个记号正规化成一个词位，并且丢弃掉任何根据配置是停用词的记号。例如：

```
SELECT to_tsquery('english', 'The & Fat & Rats');
  to_tsquery
---------------
 'fat' & 'rat'
```

和在基本tsquery输入中一样，权重可以被附加到每一个词位来限制它只匹配属于那些权重的tsvector词位。例如：

```
SELECT to_tsquery('english', 'Fat | Rats:AB');
    to_tsquery
------------------
 'fat' | 'rat':AB
```

同样，*可以被附加到一个词位来指定前缀匹配：

```
SELECT to_tsquery('supern:*A & star:A*B');
       to_tsquery
--------------------------
 'supern':*A & 'star':*AB
```

这样一个词位将匹配一个tsvector中的任意以给定字符串开头的词。

to_tsquery也能够接受单引号短语。当配置包括一个会在这种短语上触发的分类词典时就是它的主要用处。在下面的例子中，一个分类词典含规则supernovae stars : sn：

```
SELECT to_tsquery('"supernovae stars" & !crab');
  to_tsquery
---------------
 'sn' & !'crab'
```

在没有引号时，to_tsquery将为那些没有被 AND、OR 或者 FOLLOWED BY 操作符分隔的记号产生一个语法错误。

plainto_tsquery([*config* regconfig,] *querytext* text) returns tsquery

plainto_tsquery将未格式化的文本*querytext*转换成一个tsquery值。该文本被解析并被正规化，很像to_tsvector，然后&（AND）布尔操作符被插入到留下来的词之间。

例子：

```
SELECT plainto_tsquery('english', 'The Fat Rats');
 plainto_tsquery
-----------------
 'fat' & 'rat'
```

注意plainto_tsquery不会识其输入中的tsquery操作符、权重标签或前缀匹配标签：

```
SELECT plainto_tsquery('english', 'The Fat & Rats:C');
  plainto_tsquery
---------------------
 'fat' & 'rat' & 'c'
```

这里，所有输入的标点都被作为空间符号并且丢弃。

phraseto_tsquery([*config* regconfig,] *querytext* text) returns tsquery

phraseto_tsquery的行为很像plainto_tsquery，不过前者会在留下来的词之间插入<->（FOLLOWED BY）操作符而不是&（AND）操作符。还有，停用词也不是简单地丢弃掉，而是通过插入<*N*>操作符（而不是<->操作符）来解释。在搜索准确的词位序列时这个函数很有用，因为 FOLLOWED BY 操作符不只是检查所有词位的存在性，还会检查词位的顺序。

例子：

```
SELECT phraseto_tsquery('english', 'The Fat Rats');
 phraseto_tsquery
------------------
 'fat' <-> 'rat'
```

和plainto_tsquery相似，phraseto_tsquery函数不会识别其输入中的tsquery操作符、权重标签或者前缀匹配标签：

```
SELECT phraseto_tsquery('english', 'The Fat & Rats:C');
     phraseto_tsquery
-----------------------------
 'fat' <-> 'rat' <-> 'c'
```

websearch_to_tsquery([*config* regconfig,] *querytext* text) returns tsquery

websearch_to_tsquery使用一种可供选择的语法从*querytext*创建一个tsquery值，这种语法中简单的未格式化文本是一个有效的查询。和plainto_tsquery以及phraseto_tsquery不同，它还识别特定的操作符。此外，这个函数绝不会报出语法错误，这就可以把原始的用户提供的输入用于搜索。支持下列语法：

- 无引号文本：不在引号中的文本将被转换成由&操作符分隔的词，就像被plainto_tsquery处理过那样。
- ″引号文本″：在引号中的文本将被转换成由<->操作符分隔的词，就像被phraseto_tsquery处理过那样。
- OR：逻辑或将被转换成|操作符。
- -：逻辑非操作符，被转换成!操作符。

示例：

```
SELECT websearch_to_tsquery('english', 'The fat rats');
```

```
 websearch_to_tsquery
----------------------
 'fat' & 'rat'
(1 row)

SELECT websearch_to_tsquery('english', '"supernovae stars" -crab');
      websearch_to_tsquery
---------------------------------
 'supernova' <-> 'star' & !'crab'
(1 row)

SELECT websearch_to_tsquery('english', '"sad cat" or "fat rat"');
      websearch_to_tsquery
-----------------------------------
 'sad' <-> 'cat' | 'fat' <-> 'rat'
(1 row)

SELECT websearch_to_tsquery('english', 'signal -"segmentation fault"');
       websearch_to_tsquery
---------------------------------------
 'signal' & !( 'segment' <-> 'fault' )
(1 row)

SELECT websearch_to_tsquery('english', """ )( dummy \\ query <->');
 websearch_to_tsquery
----------------------
 'dummi' & 'queri'
(1 row)
```

9.3.3. 排名搜索结果

排名处理尝试度量文档和一个特定查询的接近程度，这样当有很多匹配时最相关的那些可以被先显示。UXDB提供了两种预定义的排名函数，它们考虑词法、临近性和结构信息；即，它们考虑查询词在文档中出现得有多频繁，文档中的词有多接近，以及词出现的文档部分有多重要。不过，相关性的概念是模糊的并且与应用非常相关。不同的应用可能要求额外的信息用于排名，例如，文档修改时间。内建的排名函数只是例子。可以编写自己的排名函数和/或把它们的结果与附加因素整合在一起来适应特定需求。

目前可用的两种排名函数是：

ts_rank([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer]) returns float4

基于向量的匹配词位的频率来排名向量。

ts_rank_cd([*weights* float4[],] *vector* tsvector, *query* tsquery [, *normalization* integer]) returns float4

这个函数为给定文档向量和查询计算覆盖密度排名，该方法在 Clarke、Cormack 和 Tudhope 于 1999 年在期刊 "Information Processing and Management" 上的文章 "Relevance Ranking for One to Three Term Queries" 文章中有描述。覆盖密度类似于ts_rank排名，不过它会考虑匹配词位相互之间的接近度。

这个函数要求词位的位置信息来执行其计算。因此它会忽略tsvector中任何“被剥离的”词位。如果在输入中有未被剥离的词位，结果将会是零（strip函数和tsvector中的位置信息的更多内容请见第 9.4.1 节 “操纵文档”。

对这两个函数，可选的权重参数提供了为词实例赋予更多或更少权重的能力，这种能力是依据它们被标注的情况的。权重数组指定每一类词应该得到多重的权重，按照如下的顺序：

{D-权重, C-权重, B-权重, A-权重}

如果没有提供权重，那么将使用这些默认值：

{0.1, 0.2, 0.4, 1.0}

通常权重被用来标记来自文档特别区域的词，如标题或一个初始的摘要，这样它们可以被认为比来自文档正文的词更重要或更不重要。

由于一个较长的文档有更多的机会包含一个查询术语，因此考虑文档的尺寸是合理的，例如一个一百个词的文档中有一个搜索词的五个实例而零一个一千个词的文档中有该搜索词的五个实例，则前者比后者更相关。两种排名函数都采用一个整数正规化选项，它指定文档长度是否影响其排名以及如何影响。该整数选项控制多个行为，因此它是一个位掩码：可以使用|指定一个或多个行为（例如，2|4）。

- 0（默认值）忽略文档长度
- 1 用 1 + 文档长度的对数除排名
- 2 用文档长度除排名
- 4 用长度之间的平均调和距离除排名（只被ts_rank_cd实现）
- 8 用文档中唯一词的数量除排名
- 16 用 1 + 文档中唯一词数量的对数除排名
- 32 用排名 + 1 除排名

如果多于一个标志位被指定，转换将根据列出的顺序被应用。

值得注意的是排名函数并不使用任何全局信息，因此它不可能按照某些时候期望地产生一个公平的正规化，从 1% 或 100%。正规化选项 32 （rank/(rank+1)）可以被应用来缩放所有的排名到范围零到一，但是当然这只是一个外观上的改变；它不会影响搜索结果的顺序。

这里是一个例子，它只选择十个最高排名的匹配：

```
SELECT title, ts_rank_cd(textsearch, query) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                 title                 |   rank
-----------------------------------------------+----------
 Neutrinos in the Sun                  |      3.1
 The Sudbury Neutrino Detector         |      2.4
 A MACHO View of Galactic Dark Matter  |  2.01317
 Hot Gas and Dark Matter               |  1.91171
 The Virgo Cluster: Hot Plasma and Dark Matter |  1.90953
 Rafting for Solar Neutrinos           |      1.9
 NGC 4650A: Strange Galaxy and Dark Matter     |  1.85774
 Hot Gas and Dark Matter               |   1.6123
 Ice Fishing for Cosmic Neutrinos      |      1.6
 Weak Lensing Distorts the Universe    | 0.818218
```

这是相同的例子使用正规化的排名：

```
SELECT title, ts_rank_cd(textsearch, query, 32 /* rank/(rank+1) */ ) AS rank
FROM apod, to_tsquery('neutrino|(dark & matter)') query
WHERE  query @@ textsearch
ORDER BY rank DESC
LIMIT 10;
                 title                 |        rank
-----------------------------------------------+-------------------
 Neutrinos in the Sun                          | 0.756097569485493
 The Sudbury Neutrino Detector                 | 0.705882361190954
 A MACHO View of Galactic Dark Matter          | 0.668123210574724
 Hot Gas and Dark Matter                       |  0.65655958650282
 The Virgo Cluster: Hot Plasma and Dark Matter | 0.656301290640973
 Rafting for Solar Neutrinos                   | 0.655172410958162
 NGC 4650A: Strange Galaxy and Dark Matter     | 0.650072921219637
 Hot Gas and Dark Matter                       | 0.617195790024749
 Ice Fishing for Cosmic Neutrinos              | 0.615384618911517
 Weak Lensing Distorts the Universe            | 0.450010798361481
```

排名可能会非常昂贵，因为它要求查询每一个匹配文档的tsvector，这可能会涉及很多I/O因而很慢。不幸的是，这几乎不可能避免，因为实际查询常常导致巨大数目的匹配。

9.3.4. 加亮结果

要表示搜索结果，理想的方式是显示每一个文档的一个部分并且显示它是怎样与查询相关的。通常，搜索引擎显示文档片段时会对其中的搜索术语进行标记。UXDB提供了一个函数ts_headline来实现这个功能。

ts_headline([*config* regconfig,] *document* text, *query* tsquery [, *options* text]) returns text

ts_headline接受一个文档和一个查询，并且从该文档返回一个引用，在其中来自查询的术语会被加亮。被用来解析该文档的配置可以用*config*指定；如果*config*被忽略，将会使用default_text_search_config配置。

如果一个*options*字符串被指定，它必须由一个逗号分隔的列表组成，列表中是一个或多个*option*=*value*对。可用的选项是：

- StartSel、StopSel：用来定界文档中出现的查询词的字符串，这用来把它们与其他被引用的词区分开。如果这些字符串包含空格或逗号，必须把它们加上双引号。
- MaxWords、MinWords：这些数字决定要输出的最长和最短 headline。
- ShortWord：长度小于等于这个值的词将被从一个 headline 的开头或结尾处丢掉。默认值三消除普通英语文章。
- HighlightAll：布尔标志，如果为true整个文档将被用作 headline，并忽略前面的三个参数。
- MaxFragments：要显示的文本引用或片段的最大数量。默认值零选择一种非片段倾向的 headline 生成方法。一个大于零的值选择基于片段的 headline 生成。这种方法找到有尽可能多查询词的文本片段并且展开查询词周围的那些片段。结果是查询词会靠近每个片段的中间并且在其两侧都有词。每一个片段将是最多MaxWords并且长度小于等于ShortWord的词被从每个片段的开头或结尾丢弃。如果不是所有的查询词都在该文档中找到，文档中第一个MinWords的单一片段将被显示。
- FragmentDelimiter：当多于一个片段被显示时，片段将被这个字符串所分隔。

这些选项名称不区分大小写。任何未指定的选项将收到这些默认值：

```
StartSel=<b>, StopSel=</b>,
MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE,
MaxFragments=0, FragmentDelimiter=" ... "
```

例如：

```
SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'));
                        ts_headline
------------------------------------------------------------
 containing given <b>query</b> terms
 and return them in order of their <b>similarity</b> to the
 <b>query</b>.

SELECT ts_headline('english',
  'The most common type of search
is to find all documents containing given query terms
and return them in order of their similarity to the
query.',
  to_tsquery('query & similarity'),
  'StartSel = <, StopSel = >');
                      ts_headline
-------------------------------------------------------
 containing given <query> terms
 and return them in order of their <similarity> to the
 <query>.
```

ts_headline使用原始文档，而不是一个tsvector摘要，因此它可能很慢并且注意使用。

9.4. 额外特性

这一节描述在文本搜索中有用的一些额外的函数和操作符。

9.4.1. 操纵文档

第 9.3.1 节 “解析文档展示了未经处理的文本文档如何被转换成tsvector值。UXDB也提供了用于操纵已经为tsvector形式的文档的函数和操作符。

tsvector || tsvector

tsvector连接操作符返回一个向量，它结合了作为参数给出的两个向量的词位和位置信息。位置和权重标签在连接期间被保留。出现在右手向量中的位置被使用左手向量中提到的最大位置进行偏移，这样结果几乎等于在两个原始文档字符串的连接上执行to_tsvector的结果（这种等价不是完全的，因为从左手参数的尾端移除的任何停用词将会影响结果，而如果文本连接被使用它们就影响了右手参数中的词位位置）。

使用向量形式的连接而不是在应用to_tsvector之前连接文本的一个优点是可以使用不同配置来解析文档的不同小节。此外，因为setweight函数按照相同的方式标记给定向量的所有词位，如果想把文档的不同部分标注不同的权重，就有必要解析文本并且在连接之前做setweight。

setweight(*vector* tsvector, *weight* "char") returns tsvector

setweight返回输入向量的一个拷贝，其中每一个位置都被标注为给定的权重：A、B、C或D（D是新向量的默认值并且并不会被显示在输出上）。向量被连接时会保留这些标签，允许来自文档的不同部分的词被排名函数给予不同的权重。

注意权重标签是应用到位置而不是词位。如果输入向量已经被剥离了位置，则setweight什么也不会做。

length(*vector* tsvector) returns integer

返回存储在向量中的词位数。

strip(*vector* tsvector) returns tsvector

返回一个向量，其中列出了和给定向量相同的词位，不过没有任何位置或者权重信息。其结果通常比未被剥离的向量小很多，但是用处也小很多。和未被剥离的向量一样，相关度排名在已剥离的向量上也不起作用。此外，<->（FOLLOWED BY）tsquery操作符不会匹配已剥离的输入，因为它无法确定词位之间的距离。

表 6.48 “文本搜索函数”中有tsvector相关函数的完整列表。

9.4.2. 操纵查询

第 9.3.2 节 “解析查询展示了未经处理的文本形式的查询如何被转换成tsquery值。UXDB也提供了用于操纵已经是tsquery形式的查询的函数和操作符。

tsquery && tsquery

返回用 AND 结合的两个给定查询。

tsquery || tsquery

返回用 OR 结合的两个给定查询。

!! tsquery

返回一个给定查询的反（NOT）。

tsquery <-> tsquery

返回一个查询，它用<->（FOLLOWED BY）tsquery操作符搜索两个紧跟的匹配，第一个匹配符合第一个给定的查询而第二个匹配符合第二个给定的查询。例如：

```
SELECT to_tsquery('fat') <-> to_tsquery('cat | rat');
          ?column?
-----------------------------------
 'fat' <-> 'cat' | 'fat' <-> 'rat'
```

tsquery_phrase(*query1* tsquery, *query2* tsquery [, *distance* integer]) returns tsquery

返回一个查询，它使用<*N*> tsquery操作符搜索两个距离为*distance*个词位的匹配，第一个匹配符合第一个给定的查询而第二个匹配符合第二个给定的查询。例如：

```
SELECT tsquery_phrase(to_tsquery('fat'), to_tsquery('cat'), 10);
  tsquery_phrase
------------------
 'fat' <10> 'cat'
```

numnode(*query* tsquery) returns integer

返回一个tsquery中的结点数（词位外加操作符）。要确定查询是否有意义或者是否只包含停用词时，这个函数有用，在前一种情况它返回 > 0，后一种情况返回 0。例子：

```
SELECT numnode(plainto_tsquery('the any'));
NOTICE:  query contains only stopword(s) or doesn't contain lexeme(s), ignored
 numnode
---------
       0

SELECT numnode('foo & bar'::tsquery);
 numnode
---------
       3
```

querytree(*query* tsquery) returns text

返回一个tsquery中可以被用来搜索一个索引的部分。这个函数可用来检测不可被索引的查询，例如那些只包含停用词或者只有否定术语的查询。例如：

```
SELECT querytree(to_tsquery('!defined'));
 querytree
-----------
```

9.4.2.1. 查询重写

ts_rewrite函数族在一个给定的tsquery中搜索一个目标子查询的出现，并且将每一次出现替换成一个替补子查询。本质上这个操作就是一个tsquery版本的子串替换。一个目标和替补的组合可以被看成是一个查询重写规则。一个这类重写规则的集合可以是一个强大的搜索助手。例如，可以使用同义词扩展搜索（如，new york、big apple、nyc、gotham），或者收缩搜索来将用户导向某些特点主题。在这个特性和分类词典（第 9.6.4 节 “分类词典”）有些功能重叠。但是，可以随时修改一组重写规则而无需重索引，而更新一个分类词典则要求进行重索引才能生效。

ts_rewrite (*query* tsquery, *target* tsquery, *substitute* tsquery) returns tsquery

这种形式的ts_rewrite简单地应用一个单一重写规则：不管*target*出现在*query*中的那个地方，它都被*substitute*替代。例如：

```
SELECT ts_rewrite('a & b'::tsquery, 'a'::tsquery, 'c'::tsquery);
 ts_rewrite
```

```
------------
 'b' & 'c'
```

ts_rewrite (*query* tsquery, *select* text) returns tsquery

这种形式的ts_rewrite接受一个开始*query*和一个 SQL *select*命令，它们以一个文本字符串的形式给出。*select*必须得到tsquery类型的两列。对于*select*结果的每一行，在当前*query*值中出现的第一列值（目标）被第二列值（替补）所替换。例如：

```
CREATE TABLE aliases (t tsquery PRIMARY KEY, s tsquery);
INSERT INTO aliases VALUES('a', 'c');

SELECT ts_rewrite('a & b'::tsquery, 'SELECT t,s FROM aliases');
 ts_rewrite
------------
 'b' & 'c'
```

注意当多个重写规则被以这种方式应用时，应用的顺序很重要；因此在实际中会要求源查询按某些排序键ORDER BY。

让我们考虑一个现实的天文学例子。我们将使用表驱动的重写规则扩展查询supernovae：

```
CREATE TABLE aliases (t tsquery primary key, s tsquery);
INSERT INTO aliases VALUES(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
         ts_rewrite
---------------------------------
 'crab' & ( 'supernova' | 'sn' )
```

我们可以通过只更新表来改变重写规则：

```
UPDATE aliases
SET s = to_tsquery('supernovae|sn & !nebulae')
WHERE t = to_tsquery('supernovae');

SELECT ts_rewrite(to_tsquery('supernovae & crab'), 'SELECT * FROM aliases');
                ts_rewrite
-------------------------------------------
 'crab' & ( 'supernova' | 'sn' & !'nebula' )
```

当有很多重写规则时，重写可能会很慢，因为它要为为每一个可能的匹配检查每一条规则。要过滤掉明显不符合的规则，我们可以为tsquery类型使用包含操作符。在下面的例子中，我们只选择那些可能匹配原始查询的规则：

```
SELECT ts_rewrite('a & b'::tsquery,
                  'SELECT t,s FROM aliases WHERE "a & b"::tsquery @> t');
 ts_rewrite
------------
 'b' & 'c'
```

9.4.3. 用于自动更新的触发器

> **注意**
>
> 本节中描述的方法已被使用存储生成的列所淘汰，如 第 9.2.2 节 “创建索引”中所述。

当使用一个单独的列来存储文档的tsvector表示时，有必要创建一个触发器在文档内容列改变时更新tsvector列。两个内建触发器函数可以用于这个目的，或者可以编写自己的触发器函数。

tsvector_update_trigger(*tsvector_column_name*, *config_name*, *text_column_name* [, ...])
tsvector_update_trigger_column(*tsvector_column_name*, *config_column_name*, *text_column_name* [, ...])

这些触发器函数在CREATE TRIGGER命令中指定的参数控制下，自动从一个或多个文本列计算一个tsvector列。它们使用的一个例子是：

```
CREATE TABLE messages (
    title       text,
    body        text,
    tsv         tsvector
);

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
ON messages FOR EACH ROW EXECUTE FUNCTION
tsvector_update_trigger(tsv, 'ux_catalog.english', title, body);

INSERT INTO messages VALUES('title here', 'the body text is here');

SELECT * FROM messages;
  title   |        body         |            tsv
------------+----------------------+----------------------------
 title here | the body text is here | 'bodi':4 'text':5 'titl':1

SELECT title, body FROM messages WHERE tsv @@ to_tsquery('title & body');
  title   |        body
------------+----------------------
 title here | the body text is here
```

在创建了这个触发器后，在*title*或*body*中的任何修改将自动地被反映到*tsv*中，不需要应用来操心同步的问题。

第一个触发器参数必须是要被更新的tsvector列的名字。第二个参数指定要被用来执行转换的文本搜索配置。对于tsvector_update_trigger，配置名被简单地用第二个触发器参数给出。如上所示，它必须是模式限定的，因此该触发器行为不会因为search_path中的改变而改变。对于tsvector_update_trigger_column，第二个触发器参数是另一个表列的名称，它必须是类型regconfig。这允许做一种逐行的配置选择。剩下的参数是文本列的名称（类型为text、varchar或char）。它们将按给定的顺序被包括在文档中。NULL 值将被跳过（但是其他列仍将被索引）。

这些内建触发器的一个限制是它们将所有输入列同样对待。要对列进行不同的处理 — 例如，使标题的权重和正文的不同 — 就需要编写一个自定义触发器。下面是用PL/uxSQL作为触发器语言的一个例子：

```
CREATE FUNCTION messages_trigger() RETURNS trigger AS $$
begin
  new.tsv :=
     setweight(to_tsvector('ux_catalog.english', coalesce(new.title,'')), 'A') ||
     setweight(to_tsvector('ux_catalog.english', coalesce(new.body,'')), 'D');
  return new;
end
$$ LANGUAGE pluxsql;

CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
    ON messages FOR EACH ROW EXECUTE FUNCTION messages_trigger();
```

记住当在触发器内创建tsvector值时，显式地指定配置名非常重要，这样列的内容才不会被default_text_search_config的改变所影响。如果不这样做很可能导致问题，例如在转储并重新载入之后搜索结果改变。

9.4.4. 收集文档统计数据

ts_stat被用于检查配置以及寻找候选的停用词。

```
ts_stat(sqlquery text, [ weights text, ]
        OUT word text, OUT ndoc integer,
        OUT nentry integer) returns setof record
```

*sqlquery*是一个文本值，它包含一个必须返回单一tsvector列的 SQL 查询。ts_stat执行该查询并返回有关包含在该tsvector数据中的每一个可区分词位（词）的统计数据。返回的列是：

- *word* text — 一个词位的值
- *ndoc* integer — 词出现过的文档（tsvector）的数量
- *nentry* integer — 词出现的总次数

如果提供了权重，只有具有其中之一权重的出现才被计算在内。

例如，要在一个文档集合中查找十个最频繁的词：

```
SELECT * FROM ts_stat('SELECT vector FROM apod')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

同样的要求，但是只计算以权重A或B出现的次数：

```
SELECT * FROM ts_stat('SELECT vector FROM apod', 'ab')
ORDER BY nentry DESC, ndoc DESC, word
LIMIT 10;
```

9.5. 解析器

文本搜索解析器负责把未处理的文档文本划分成记号并且标识每一个记号的类型，而可能的类型集合由解析器本身定义。注意一个解析器完全不会修改文本 — 它简单地标识看似有理的词边

界。因为这种有限的视野，对于应用相关的自定义解析器的需求就没有自定义字典那么强烈。目前UXDB只提供了一种内建解析器，它已经被证实对很多种应用都适用。

内建解析器被称为ux_catalog.default。它识别 23 种记号类型，如表 9.1 “默认解析器的记号类型”所示。

表 9.1. 默认解析器的记号类型

别名	描述	例子
asciiword	单词，所有 ASCII 字母	elephant
word	单词，所有字母	mañana
numword	单词，字母和数字	beta1
asciihword	带连字符的单词，所有 ASCII	up-to-date
hword	带连字符的单词，所有字母	lógico-matemática
numhword	带连字符的单词，字母和数字	uxsinodb-beta1
hword_asciipart	带连字符的单词部分，所有 ASCII	uxsinodb in the context uxsinodb-beta1
hword_part	带连字符的单词部分，所有字母	lógico or matemática in the context lógico-matemática
hword_numpart	带连字符的单词部分，字母和数字	beta1 in the context uxsinodb-beta1
email	Email 地址	foo@example.com
protocol	协议头部	http://
url	URL	example.com/stuff/index.html
host	主机	example.com
url_path	URL 路径	/stuff/index.html, in the context of a URL
file	文件或路径名	/usr/local/foo.txt, if not within a URL
sfloat	科学记数法	-1.234e56
float	十进制记数法	-1.234
int	有符号整数	-1234
uint	无符号整数	1234
version	版本号	8.3.0
tag	XML 标签	<a href="dictionaries.html">
entity	XML 实体	&
blank	空格符号	（其他不识别的任意空白或标点符号）

注意

解析器的一个“字母”的概念由数据库的区域设置决定，具体是lc_ctype。只包含基本 ASCII 字母的词被报告为一个单独的记号类型，因为有时可以

> 用来区别它们。在大部分欧洲语言中，记号类型word和asciiword应该被同样对待。
>
> email不支持 RFC 5322 定义的所有合法 email 字符。具体来说，对 email 用户名被支持的非字母数字字符只有句点、破折号和下划线。

解析器有可能从同一份文本得出相互覆盖的记号。例如，一个带连字符的词可能会被报告为一整个词或者多个部分：

```
SELECT alias, description, token FROM ts_debug('foo-bar-beta1');
      alias      |              description               |     token
-----------------+----------------------------------------+---------------
 numhword        | Hyphenated word, letters and digits    | foo-bar-beta1
 hword_asciipart | Hyphenated word part, all ASCII        | foo
 blank           | Space symbols                          | -
 hword_asciipart | Hyphenated word part, all ASCII        | bar
 blank           | Space symbols                          | -
 hword_numpart   | Hyphenated word part, letters and digits | beta1
```

这种行为是值得要的，因为它允许对整个复合词和每个部分进行搜索。这里是另一个例子：

```
SELECT alias, description, token FROM ts_debug('http://example.com/stuff/index.html');
  alias   |  description  |            token
----------+---------------+------------------------------
 protocol | Protocol head | http://
 url      | URL           | example.com/stuff/index.html
 host     | Host          | example.com
 url_path | URL path      | /stuff/index.html
```

9.6. 词典

词典被用来消除不被搜索考虑的词（stop words）、并被用来正规化词这样同一个词的不同派生形式将会匹配。一个被成功地正规化的词被称为一个词位。除了提高搜索质量，正规化和移除停用词减小了文档的tsvector表示的尺寸，因而提高了性能。正规化不会总是有语言上的意义并且通常依赖于应用的语义。

一些正规化的例子：

- 语言学的 - Ispell 词典尝试将输入词缩减为一种正规化的形式；词干分析器词典移除词的结尾
- URL位置可以被规范化来得到等效的 URL 匹配：
  - http://www.uxsql.ru/db/mw/index.html
  - http://www.uxsql.ru/db/mw/
  - http://www.uxsql.ru/db/../db/mw/index.html
- 颜色名可以被它们的十六进制值替换，例如red, green, blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF
- 如果索引数字，我们可以移除某些小数位来缩减可能的数字的范围，因此如果只保留小数点后两位，例如3.14159265359、3.1415926、3.14在正规化后会变得相同。

一个词典是一个程序，它接受一个记号作为输入，并返回：

- 如果输入的记号对词典是已知的，则返回一个词位数组（注意一个记号可能产生多于一个词位）

- 一个TSL_FILTER标志被设置的单一词位，用一个新记号来替换要被传递给后续字典的原始记号（做这件事的一个字典被称为一个过滤字典）
- 如果字典知道该记号但它是一个停用词，则返回一个空数组
- 如果字典不识别该输入记号，则返回NULL

UXDB为许多语言提供了预定义的字典。也有多种预定义模板可以被用于创建带自定义参数的新词典。每一种预定义词典模板在下面描述。如果没有合适的现有模板，可以创建新的；例子见UXDB发布的contrib/区域。

一个文本搜索配置把一个解析器和一组处理解析器输出记号的词典绑定在一起。对于每一中解析器能返回的记号类型，配置都指定了一个单独的词典列表。当该类型的一个记号被解析器找到时，每一个词典都被按照顺序查询，知道某个词典将其识别为一个已知词。如果它被标识为一个停用词或者没有一个词典识别它，它将被丢弃并且不会被索引和用于搜索。通常，第一个返回非NULL输出的词典决定结果，并且任何剩下的词典都不会被查找；但是一个过滤词典可以将给定词替换为一个被修改的词，它再被传递给后续的词典。

配置一个词典列表的通用规则是将最狭窄、最特定的词典放在第一位，然后是更加通用的词典，以一个非常通用的词典结尾，像一个Snowball词干分析器或什么都识别的simple。例如，对于一个天文学相关的搜索（astro_en 配置）我们可以把记号类型asciiword（ASCII 词）绑定到一个天文学术语的分类词典、一个通用英语词典和一个Snowball英语词干分析器：

```
ALTER TEXT SEARCH CONFIGURATION astro_en
   ADD MAPPING FOR asciiword WITH astrosyn, english_ispell, english_stem;
```

一个过滤词典可以被放置在列表中的任意位置，除了在最后，因为过滤词典放在最后就等于无用。过滤词典可用于部分正规化词来简化后续词典的工作。例如，一个过滤词典可以被用来从音标字母中移除重音符号，就像unaccent模块所做的。

9.6.1. 停用词

停用词是非常常用、在几乎每一个文档中出现并且没有任何区分度的词。因此，在全文搜索的环境中它们可以被忽略。例如，每一段英语文本都包含a和the等次，因此把它们存储在一个索引中是没有用处的。但是，停用词确实会影响在tsvector中的位置，这进而会影响排名：

```
SELECT to_tsvector('english','in the list of stop words');
       to_tsvector
----------------------------
 'list':3 'stop':5 'word':6
```

缺失的位置 1、2、4 是因为停用词。文档的排名计算在使用和不使用停用词的情况下是很不同的：

```
SELECT ts_rank_cd (to_tsvector('english','in the list of stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
       0.05

SELECT ts_rank_cd (to_tsvector('english','list stop words'), to_tsquery('list & stop'));
 ts_rank_cd
------------
```

0.1

如何对待停用词是由指定词典决定的。例如，ispell词典首先正规化词并且查看停用词列表，而Snowball词干分析器首先检查停用词的列表。这种不同行为的原因是一冲降低噪声的尝试。

9.6.2. 简单词典

simple词典模板的操作是将输入记号转换为小写形式并且根据一个停用词文件检查它。如果该记号在该文件中被找到，则返回一个空数组，导致该记号被丢弃。否则，该词的小写形式被返回作为正规化的词位。作为一种选择，该词典可以被配置为将非停用词报告为未识别，允许它们被传递给列表中的下一个词典。

下面是一个使用simple模板的词典定义的例子：

```
CREATE TEXT SEARCH DICTIONARY public.simple_dict (
   TEMPLATE = ux_catalog.simple,
   STOPWORDS = english
);
```

这里，english是一个停用词文件的基本名称。该文件的全名将是$SHAREDIR/tsearch_data/english.stop，其中$SHAREDIR表示UXDB安装的共享数据目录，通常是/usr/local/share/uxsinodb（如果不确定，使用`ux_config --sharedir`）。该文件格式是一个词的列表，每行一个。空行和尾部的空格都被忽略，并且大写也被折叠成小写，但是没有其他对该文件内容的处理。

现在我们能够测试我们的词典：

```
SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------
 {yes}

SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

如果没有在停用词文件中找到，我们也可以选择返回NULL而不是小写形式的词。这种行为可以通过设置词典的Accept参数为false来选择。继续该例子：

```
ALTER TEXT SEARCH DICTIONARY public.simple_dict ( Accept = false );

SELECT ts_lexize('public.simple_dict','YeS');
 ts_lexize
-----------


SELECT ts_lexize('public.simple_dict','The');
 ts_lexize
-----------
 {}
```

在使用默认值Accept = true，只有把一个simple词典放在词典列表的尾部才有用，因为它将不会传递任何记号给后续的词典。相反，Accept = false只有当至少有一个后续词典的情况下才有用。

> **注意**
>
> 大部分类型的词典依赖于配置文件，例如停用词文件。这些文件必须被存储为 UTF-8 编码。当它们被读入服务器时，如果存在不同，它们将被翻译成真实的数据库编码。

> **注意**
>
> 通常，当一个词典配置文件第一次在数据库会话中使用时，数据库会话将只读取它一次。如果修改了一个配置文件并且想强迫现有的会话取得新内容，可以在该词典上发出一个ALTER TEXT SEARCH DICTIONARY命令。这可以是一次“假”更新，它并不实际修改任何参数值。

9.6.3. 同义词词典

这个词典模板被用来创建用于同义词替换的词典。不支持短语（使用分类词典模板（第 9.6.4 节 “分类词典”可以支持）。一个同义词词典可以被用来解决语言学问题，例如，阻止一个英语词干分析器词典把词“Paris”缩减成“pari”。在同义词词典中有一行Paris paris并把它放在english_stem词典之前就足够了。例如：

```
SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | Paris | {english_stem} | english_stem | {pari}

CREATE TEXT SEARCH DICTIONARY my_synonym (
    TEMPLATE = synonym,
    SYNONYMS = my_synonyms
);

ALTER TEXT SEARCH CONFIGURATION english
    ALTER MAPPING FOR asciiword
    WITH my_synonym, english_stem;

SELECT * FROM ts_debug('english', 'Paris');
   alias   |   description   | token |       dictionaries        | dictionary | lexemes
-----------+-----------------+-------+---------------------------+------------+---------
 asciiword | Word, all ASCII | Paris | {my_synonym,english_stem} | my_synonym | {paris}
```

synonym模板要求的唯一参数是SYNONYMS，它是其配置文件的基本名 — 上例中的my_synonyms。该文件的完整名称将是$SHAREDIR/tsearch_data/my_synonyms.syn（其中$SHAREDIR表示UXDB安装的共享数据目录）。该文件格式是每行一个要被替换的词，后面跟着它的同义词，用空白分隔。空行和结尾的空格会被忽略。

synonym模板还有一个可选的参数CaseSensitive，其默认值为false。当CaseSensitive为false时，同义词文件中的词被折叠成小写，这和输入记号一样。当它为true时，词和记号将不会被折叠成小写，但是比较时就好像被折叠过一样。

一个星号（*）可以被放置在配置文件中一个同义词的末尾。这表示该同义词是一个前缀。当项被用在to_tsvector()中时，星号会被忽略；当它被用在to_tsquery()中时，结果将是一个带有前缀匹配标记器（见第 9.3.2 节 “解析查询”的查询项。例如，假设我们在$SHAREDIR/tsearch_data/synonym_sample.syn中有这些项：

```
uxdb       uxsql
uxsinodb     uxsql
uxdb uxsql
gogle   googl
indices index*
```

那么我们将得到这些结果：

```
mydb=# CREATE TEXT SEARCH DICTIONARY syn (template=synonym,
 synonyms='synonym_sample');
mydb=# SELECT ts_lexize('syn','indices');
 ts_lexize
-----------
 {index}
(1 row)

mydb=# CREATE TEXT SEARCH CONFIGURATION tst (copy=simple);
mydb=# ALTER TEXT SEARCH CONFIGURATION tst ALTER MAPPING FOR asciiword WITH
 syn;
mydb=# SELECT to_tsvector('tst','indices');
 to_tsvector
-------------
 'index':1
(1 row)

mydb=# SELECT to_tsquery('tst','indices');
 to_tsquery
------------
 'index':*
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector;
           tsvector
--------------------------------
 'are' 'indexes' 'useful' 'very'
(1 row)

mydb=# SELECT 'indexes are very useful'::tsvector @@ to_tsquery('tst','indices');
 ?column?
----------
 t
(1 row)
```

9.6.4. 分类词典

一个分类词典（有时被简写成TZ）是一个词的集合，其中包括了词与短语之间的联系，即广义词（BT）、狭义词（NT）、首选词、非首选词、相关词等。

基本上一个分类词典会用一个首选词替换所有非首选词，并且也可选择地保留原始术语用于索引。UXDB的分类词典的当前实现是同义词词典的一个扩展，并增加了短语支持。一个分类词典要求一个下列格式的配置文件：

```
# this is a comment
sample word(s) : indexed word(s)
more sample word(s) : more indexed word(s)
...
```

其中冒号（:）符号扮演了一个短语及其替换之间的定界符。

一个分类词典使用一个*子词典*（在词典的配置中指定）在检查短语匹配之前正规化输入文本。只能选择一个子词典。如果子词典无法识别一个词，将报告一个错误。在这种情况下，应该移除该词的使用或者让子词典学会这个词。可以在一个被索引词的开头放上一个星号（*）来跳过在其上应用子词典，但是所有采样词必须被子词典知道。

如果有多个短语匹配输入，则分类词典选择最长的那一个，并且使用最后的定义打破连结。

由子词典识别的特定停用词不能够被指定；改用?标记任何可以出现停用词的地方。例如，假定根据子词典a和the是停用词：

```
? one ? two : swsw
```

匹配a one the two和the one a two；两者都将被swsw替换。

由于一个分类词典具有识别短语的能力，它必须记住它的状态并与解析器交互。一个分类词典使用这些任务来检查它是否应当处理下一个词或者停止累积。分类词典必须注意配置。例如，如果分类词典被分配只处理asciiword记号，则一个形如one 7的分类词典定义将不会工作，因为记号类型uint没有被分配给该分类词典。

> **注意**
>
> 在索引期间要用到分类词典，因此分类词典参数中的任何变化都要求重索引。对于大多数其他索引类型，例如增加或移除停用词等小改动都不会强制重索引。

9.6.4.1. 分类词典配置

要定义一个新的分类词典，可使用thesaurus模板。例如：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_simple (
    TEMPLATE = thesaurus,
    DictFile = mythesaurus,
    Dictionary = ux_catalog.english_stem
);
```

这里：

- thesaurus_simple是新词典的名称
- mythesaurus是分类词典配置文件的基础名称（它的全名将是$SHAREDIR/tsearch_data/mythesaurus.ths，其中$SHAREDIR表示安装的共享数据目录）。

- ux_catalog.english_stem是要用于分类词典正规化的子词典（这里是一个 Snowball 英语词干分析器）。注意子词典将拥有它自己的配置（例如停用词），但这里没有展示。

现在可以在配置中把分类词典thesaurus_simple绑定到想要的记号类型上，例如：

```
ALTER TEXT SEARCH CONFIGURATION russian
   ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
   WITH thesaurus_simple;
```

9.6.4.2. 分类词典例子

考虑一个简单的天文学分类词典thesaurus_astro，它包含一些天文学词组合：

```
supernovae stars : sn
crab nebulae : crab
```

下面我们创建一个词典并绑定一些记号类型到一个天文学分类词典以及英语词干分析器：

```
CREATE TEXT SEARCH DICTIONARY thesaurus_astro (
   TEMPLATE = thesaurus,
   DictFile = thesaurus_astro,
   Dictionary = english_stem
);

ALTER TEXT SEARCH CONFIGURATION russian
   ALTER MAPPING FOR asciiword, asciihword, hword_asciipart
   WITH thesaurus_astro, english_stem;
```

现在我们可以看看它如何工作。ts_lexize对于测试一个分类词典用处不大，因为它把它的输入看成是一个单一记号。我们可以用plainto_tsquery和to_tsvector，它们将把其输入字符串打断成多个记号：

```
SELECT plainto_tsquery('supernova star');
 plainto_tsquery
-----------------
 'sn'

SELECT to_tsvector('supernova star');
 to_tsvector
-------------
 'sn':1
```

原则上，如果对参数加了引号，可以使用to_tsquery：

```
SELECT to_tsquery('''supernova star''');
 to_tsquery
------------
 'sn'
```

注意在thesaurus_astro中supernova star匹配supernovae stars，因为我们在分类词典定义中指定了english_stem词干分析器。该词干分析器移除了e和s。

要和替补一样也索引原始短语，只要将它包含在定义的右手部分中：

```
supernovae stars : sn supernovae stars

SELECT plainto_tsquery('supernova star');
     plainto_tsquery
-----------------------------
 'sn' & 'supernova' & 'star'
```

9.6.5. Ispell 词典

Ispell词典模板支持*词法词典*，它可以把一个词的很多不同语言学的形式正规化成相同的词位。例如，一个英语Ispell词典可以匹配搜索词bank的词尾变化和词形变化，例如banking、banked、banks、banks'和bank's。

标准的UXDB发布不包括任何Ispell配置文件。用于很多种语言的词典可以从Ispell得到。此外，也支持一些更现代的词典文件格式 — MySpell（OO < 2.0.1）和Hunspell（OO >= 2.0.2）。一个很大的词典列表在OpenOffice Wiki上可以得到。

要创建一个Ispell词典，执行这三步：

- 下载词典配置文件。OpenOffice扩展文件的扩展名是.oxt。有必要抽取.aff和.dic文件，把扩展改为.affix和.dict。对于某些词典文件，还需要使用下面的命令把字符转换成 UTF-8 编码（例如挪威语词典）：

```
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.affix nn_NO.aff
iconv -f ISO_8859-1 -t UTF-8 -o nn_no.dict nn_NO.dic
```

- 拷贝文件到$SHAREDIR/tsearch_data目录
- 用下面的命令把文件载入到 UXDB：

```
CREATE TEXT SEARCH DICTIONARY english_hunspell (
    TEMPLATE = ispell,
    DictFile = en_us,
    AffFile = en_us,
    Stopwords = english);
```

这里，DictFile、AffFile和StopWords指定词典、词缀和停用词文件的基础名称。停用词文件的格式和前面解释的simple词典类型相同。其他文件的格式在这里没有指定，但是也可以从上面提到的网站获得。

Ispell 词典通常识别一个有限集合的词，这样它们后面应该跟着另一个更广义的词典；例如，一个 Snowball 词典，它可以识别所有东西。

Ispell的.affix文件具有下面的结构：

```
prefixes
flag *A:
   .         >   RE      # As in enter > reenter
suffixes
flag T:
   E         >   ST      # As in late > latest
```

```
  [^AEIOU]Y   >   -Y,IEST # As in dirty > dirtiest
  [AEIOU]Y    >   EST    # As in gray > grayest
  [^EY]       >   EST    # As in small > smallest
```

.dict文件具有下面的结构：

```
lapse/ADGRS
lard/DGRS
large/PRTY
lark/MRS
```

.dict文件的格式是：

```
basic_form/affix_class_name
```

在.affix文件中，每一个词缀标志以下面的格式描述：

```
condition > [-stripping_letters,] adding_affix
```

这里的条件具有和正则表达式相似的格式。它可以使用分组[...]和[^...]。例如，[AEIOU]Y表示词的最后一个字母是"y"并且倒数第二个字母是"a"、"e"、"i"、"o"或者"u"。[^EY]表示最后一个字母既不是"e"也不是"y"。

Ispell 词典支持划分复合词，这是一个有用的特性。注意词缀文件应该用compoundwords controlled语句指定一个特殊标志，它标记可以参与到复合格式中的词典词：

```
compoundwords  controlled z
```

下面是挪威语的一些例子：

```
SELECT ts_lexize('norwegian_ispell', 'overbuljongterningpakkmesterassistent');
  {over,buljong,terning,pakk,mester,assistent}
SELECT ts_lexize('norwegian_ispell', 'sjokoladefabrikk');
  {sjokoladefabrikk,sjokolade,fabrikk}
```

MySpell格式是Hunspell格式的一个子集。Hunspell的.affix文件具有下面的结构：

```
PFX A Y 1
PFX A   0    re       .
SFX T N 4
SFX T   0    st       e
SFX T   y    iest     [^aeiou]y
SFX T   0    est      [aeiou]y
SFX T   0    est      [^ey]
```

一个词缀类的第一行是头部。头部后面列出了词缀规则的域：

- 参数名（PFX 或者 SFX）
- 标志（词缀类的名称）
- 从该词的开始（前缀）或者结尾（后缀）剥离字符

- 增加词缀
- 和正则表达式格式类似的条件。

.dict文件看起来和Ispell的.dict文件相似：

```
larder/M
lardy/RT
large/RSPMYT
largehearted
```

> **注意**
>
> MySpell 不支持复合词。Hunspell则对复合词有更好的支持。当前，UXDB只实现了 Hunspell 中基本的复合词操作。

9.6.6. Snowball 词典

Snowball词典模板基于 Martin Porter 的一个项目，他是流行的英语 Porter 词干分析算法的发明者。Snowball 现在对许多语言提供词干分析算法（详见Snowball 站点）。每一个算法懂得按照其语言中的拼写，如何缩减词的常见变体形式为一个基础或词干。一个 Snowball 词典要求一个language参数来标识要用哪种词干分析器，并且可以选择地指定一个stopword文件名来给出一个要被消除的词列表（UXDB的标准停用词列表也是由 Snowball 项目提供的）。例如，有一个内建的定义等效于

```
CREATE TEXT SEARCH DICTIONARY english_stem (
   TEMPLATE = snowball,
   Language = english,
   StopWords = english
);
```

停用词文件格式和已经解释的一样。

一个Snowball词典识别所有的东西，不管它能不能简化该词，因此它应当被放置在词典列表的最后。把它放在任何其他词典前面是没有用处的，因为一个记号永远不会穿过它而进入到下一个词典。

9.7. 配置例子

一个文本搜索配置指定了将一个文档转换成一个tsvector所需的所有选项：用于把文本分解成记号的解析器，以及用于将每一个记号转换成词位的词典。每一次to_tsvector或to_tsquery的调用都需要一个文本搜索配置来执行其处理。配置参数default_text_search_config指定了默认配置的名称，如果忽略了显式的配置参数，文本搜索函数将会使用它。它可以在uxsinodb.conf中设置，或者使用SET命令为一个单独的会话设置。

有一些预定义的文本搜索配置可用，并且可以容易地创建自定义的配置。为了便于管理文本搜索对象，可以使用一组SQL命令，并且有多个uxsql命令可以显示有关文本搜索对象（第 9.10 节“uxsql支持”）的信息。

作为一个例子，我们将创建一个配置ux，从复制内建的english配置开始：

CREATE TEXT SEARCH CONFIGURATION public.ux (COPY = ux_catalog.english);

我们将使用一个 UXDB 相关的同义词列表，并将它存储在$SHAREDIR/tsearch_data/ux_dict.syn中。文件内容看起来像：

```
uxdb    ux
uxsql      ux
uxsinodb  ux
```

我们定义同义词词典如下：

```
CREATE TEXT SEARCH DICTIONARY ux_dict (
   TEMPLATE = synonym,
   SYNONYMS = ux_dict
);
```

接下来我们注册Ispell词典english_ispell，它有其自己的配置文件：

```
CREATE TEXT SEARCH DICTIONARY english_ispell (
   TEMPLATE = ispell,
   DictFile = english,
   AffFile = english,
   StopWords = english
);
```

现在我们可以在配置ux中建立词的映射：

```
ALTER TEXT SEARCH CONFIGURATION ux
   ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
                     word, hword, hword_part
   WITH ux_dict, english_ispell, english_stem;
```

我们选择不索引或搜索某些内建配置确实处理的记号类型：

```
ALTER TEXT SEARCH CONFIGURATION ux
   DROP MAPPING FOR email, url, url_path, sfloat, float;
```

现在我们可以测试我们的配置：

```
SELECT * FROM ts_debug('public.ux', '
UXDB, the highly scalable, SQL compliant, open source object-relational
database management system, is now undergoing beta testing of the next
version of our software.
');
```

下一个步骤是设置会话让它使用新配置，它被创建在public模式中：

```
=> \dF
  List of text search configurations
 Schema  | Name | Description
```

```
---------+------+-------------
 public  | ux   |

SET default_text_search_config = 'public.ux';
SET

SHOW default_text_search_config;
 default_text_search_config
----------------------------
 public.ux
```

9.8. 测试和调试文本搜索

一个自定义文本搜索配置的行为很容易变得混乱。本节中描述的函数对于测试文本搜索对象有用。可以测试一个完整的配置，或者独立测试解析器和词典。

9.8.1. 配置测试

函数ts_debug允许简单地测试一个文本搜索配置。

```
ts_debug([ config regconfig, ] document text,
        OUT alias text,
        OUT description text,
        OUT token text,
        OUT dictionaries regdictionary[],
        OUT dictionary regdictionary,
        OUT lexemes text[])
        returns setof record
```

ts_debug显示*document*的每一个记号的信息，记号由解析器产生并由配置的词典处理过。该函数使用由*config*指定的配置，如果该参数被忽略则使用default_text_search_config指定的配置。

ts_debug为解析器在文本中标识的每一个记号返回一行。被返回的列是：

- *alias* `text` — 记号类型的短名称
- *description* `text` — 记号类型的描述
- *token* `text` — 记号的文本
- *dictionaries* `regdictionary[]` — 配置为这种记号类型选择的词典
- *dictionary* `regdictionary` — 识别该记号的词典，如果没有词典能识别则为NULL
- *lexemes* `text[]` — 识别该记号的词典产生的词位，如果没有词典能识别则为NULL；一个空数组（{}）表示该记号被识别为一个停用词

这里是一个简单的例子：

```
SELECT * FROM ts_debug('english','a fat  cat sat on a mat - it ate a fat rats');
  alias   |   description   | token |  dictionaries  |  dictionary  | lexemes
-----------+-----------------+-------+----------------+--------------+---------
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
```

```
 asciiword | Word, all ASCII | cat   | {english_stem} | english_stem | {cat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | sat   | {english_stem} | english_stem | {sat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | on    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | mat   | {english_stem} | english_stem | {mat}
 blank     | Space symbols   |       | {}             |              |
 blank     | Space symbols   | -     | {}             |              |
 asciiword | Word, all ASCII | it    | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | ate   | {english_stem} | english_stem | {ate}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | a     | {english_stem} | english_stem | {}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | fat   | {english_stem} | english_stem | {fat}
 blank     | Space symbols   |       | {}             |              |
 asciiword | Word, all ASCII | rats  | {english_stem} | english_stem | {rat}
```

为了一个更广泛的示范，我们先为英语语言创建一个public.english配置和 Ispell 词典：

```
CREATE TEXT SEARCH CONFIGURATION public.english ( COPY = ux_catalog.english );

CREATE TEXT SEARCH DICTIONARY english_ispell (
    TEMPLATE = ispell,
    DictFile = english,
    AffFile = english,
    StopWords = english
);

ALTER TEXT SEARCH CONFIGURATION public.english
   ALTER MAPPING FOR asciiword WITH english_ispell, english_stem;


SELECT * FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |   description   |    token    |         dictionaries          |   dictionary   |   lexemes
-----------+-----------------+-------------+-------------------------------+----------------+-------------
 asciiword | Word, all ASCII | The         | {english_ispell,english_stem} | english_ispell | {}
 blank     | Space symbols   |             | {}                            |                |
 asciiword | Word, all ASCII | Brightest   | {english_ispell,english_stem} | english_ispell | {bright}
 blank     | Space symbols   |             | {}                            |                |
 asciiword | Word, all ASCII | supernovaes | {english_ispell,english_stem} | english_stem   | {supernova}
```

在这个例子中，词Brightest被解析器识别为一个ASCII 词（别名asciiword）。对于这种记号类型，词典列表是english_ispell和english_stem。该词被english_ispell识别，这个词典将它缩减为名词bright。词supernovaes对于english_ispell词典是未知的，因此它被传递给下一个词典，并且幸运地是，它被识别了（实际上，english_stem是一个 Snowball 词典，它识别所有的东西；这也是为什么它被放置在词典列表的尾部）。

词The被english_ispell词典识别为一个停用词（第 9.6.1 节 “停用词”）并且将不会被索引。空格也被丢弃，因为该配置没有为它们提供词典。

可以通过显式地指定想看哪些列来缩减输出的宽度：

```
SELECT alias, token, dictionary, lexemes
FROM ts_debug('public.english','The Brightest supernovaes');
   alias   |    token    |   dictionary   |   lexemes
-----------+-------------+----------------+-------------
 asciiword | The         | english_ispell | {}
 blank     |             |                |
 asciiword | Brightest   | english_ispell | {bright}
 blank     |             |                |
 asciiword | supernovaes | english_stem   | {supernova}
```

9.8.2. 解析器测试

下列函数允许直接测试一个文本搜索解析器。

```
ts_parse(parser_name text, document text,
        OUT tokid integer, OUT token text) returns setof record
ts_parse(parser_oid oid, document text,
        OUT tokid integer, OUT token text) returns setof record
```

ts_parse解析给定的*document*并返回一系列记录，每一个记录对应一个由解析产生的记号。每一个记录包括一个tokid展示分配给记号的类型以及一个token展示记号的文本。例如：

```
SELECT * FROM ts_parse('default', '123 - a number');
 tokid | token
-------+--------
    22 | 123
    12 |
    12 | -
     1 | a
    12 |
     1 | number
```

```
ts_token_type(parser_name text, OUT tokid integer,
            OUT alias text, OUT description text) returns setof record
ts_token_type(parser_oid oid, OUT tokid integer,
            OUT alias text, OUT description text) returns setof record
```

ts_token_type返回一个表，该表描述指定解析器能够识别的每一种记号类型。对于每一种记号类型，该表给出了解析器用来标注该类型记号的整数tokid，还给出了在配置命令中命名该记号类型的alias，以及一个简短的description。例如：

```
SELECT * FROM ts_token_type('default');
 tokid |      alias      |                description
-------+-----------------+-------------------------------------------
     1 | asciiword       | Word, all ASCII
     2 | word            | Word, all letters
```

```
 3 | numword         | Word, letters and digits
 4 | email           | Email address
 5 | url             | URL
 6 | host            | Host
 7 | sfloat          | Scientific notation
 8 | version         | Version number
 9 | hword_numpart   | Hyphenated word part, letters and digits
10 | hword_part      | Hyphenated word part, all letters
11 | hword_asciipart | Hyphenated word part, all ASCII
12 | blank           | Space symbols
13 | tag             | XML tag
14 | protocol        | Protocol head
15 | numhword        | Hyphenated word, letters and digits
16 | asciihword      | Hyphenated word, all ASCII
17 | hword           | Hyphenated word, all letters
18 | url_path        | URL path
19 | file            | File or path name
20 | float           | Decimal notation
21 | int             | Signed integer
22 | uint            | Unsigned integer
23 | entity          | XML entity
```

9.8.3. 词典测试

ts_lexize函数帮助词典测试。

ts_lexize(*dict* regdictionary, *token* text) returns text[]

如果输入的*token*是该词典已知的，则ts_lexize返回一个词位数组；如果记号是词典已知的但是它是一个停用词，则返回一个空数组；或者如果它对词典是未知词，则返回NULL。

例子：

```
SELECT ts_lexize('english_stem', 'stars');
 ts_lexize
-----------
 {star}

SELECT ts_lexize('english_stem', 'a');
 ts_lexize
-----------
 {}
```

> **注意**
>
> ts_lexize函数期望一个单一记号而不是文本。下面的情况会让它搞混：
>
> ```
> SELECT ts_lexize('thesaurus_astro','supernovae stars') is null;
>  ?column?
> ----------
>  t
> ```

分类词典thesaurus_astro确实知道短语supernovae stars，但是ts_lexize会失败，因为它无法解析输入文本而把它当做一个单一记号。可以使用plainto_tsquery或to_tsvector来测试分类词典，例如：

```
SELECT plainto_tsquery('supernovae stars');
 plainto_tsquery
-----------------
 'sn'
```

9.9. GIN 和 GiST 索引类型

有两种索引可以被用来加速全文搜索。注意全文搜索并非一定需要索引，但是在一个定期会被搜索的列上，通常需要有一个索引。

CREATE INDEX *name* ON *table* USING GIN(*column*);

创建一个基于 GIN（通用倒排索引）的索引。*column*必须是tsvector类型。

CREATE INDEX *name* ON *table* USING GIST(*column*);

创建一个基于 GiST（通用搜索树）的索引。*column*可以是tsvector或tsquery类型。

GIN 索引是更好的文本搜索索引类型。作为倒排索引，每个词（词位）在 其中都有一个索引项，其中有压缩过的匹配位置的列表。多词搜索可以找到 第一个匹配，然后使用该索引移除缺少额外词的行。GIN 索引只存储 tsvector值的词（词位），并且不存储它们的权重标签。因此， 在使用涉及权重的查询时需要一次在表行上的重新检查。

一个 GiST 索引是有损的，这表示索引可能产生假匹配，并且有必要检查真实的表行来消除这种假匹配（UXDB在需要时会自动做这一步）。GiST 索引之所以是有损的，是因为每一个文档在索引中被表示为一个定长的签名。该签名通过哈希每一个词到一个 n 位串中的一个单一位来产生，通过将所有这些位 OR 在一起产生一个 n 位的文档签名。当两个词哈希到同一个位位置时就会产生假匹配。如果查询中所有词都有匹配（真或假），则必须检索表行查看匹配是否正确。

GiST 索引可以被覆盖，例如使用INCLUDE子句。 包含的列可以具有没有任何 GiST 操作符类的数据类型。 包含的属性将非压缩存储。

有损性导致的性能下降归因于不必要的表记录（即被证实为假匹配的记录）获取。因为表记录的随机访问是较慢的，这限制了 GiST 索引的可用性。假匹配的可能性取决于几个因素，特别是唯一词的数量，因此推荐使用词典来缩减这个数量。

注意GIN索引的构件时间常常可以通过增加maintenance_work_mem来改进，而GiST索引的构建时间则与该参数无关。

对大集合分区并正确使用 GIN 和 GiST 索引允许实现带在线更新的快速搜索。分区可以在数据库层面上使用表继承来完成，或者是通过将文档分布在服务器上并收集外部的搜索结果，例如通过外部数据访问。后者是可能的，因为排名函数只使用本地信息。

9.10. uxsql支持

关于文本搜索配置对象的信息可以在uxsql中使用一组命令获得：

\dF{d,p,t}[+] [PATTERN]

可选的+能产生更多细节。

可选参数*PATTERN*可以是一个文本搜索对象的名称，可以是模式限定的。如果*PATTERN*被忽略，则所有可见对象的信息都将被显示。*PATTERN*可以是一个正则表达式并且可以为模式和对象名称提供独立的模式。下面的例子展示了这些特性：

```
=> \dF *fulltext*
      List of text search configurations
 Schema |  Name        | Description
--------+--------------+-------------
 public | fulltext_cfg |


=> \dF *.fulltext*
      List of text search configurations
 Schema   |  Name        | Description
----------+--------------+----------------------------
 fulltext | fulltext_cfg |
 public   | fulltext_cfg |
```

可用的命令是：

\dF[+] [PATTERN]

列出文本搜索配置（加上+得到更多细节）。

```
=> \dF russian
            List of text search configurations
   Schema   |  Name   |            Description
------------+---------+-----------------------------------
 ux_catalog | russian | configuration for russian language

=> \dF+ russian
Text search configuration "ux_catalog.russian"
Parser: "ux_catalog.default"
      Token      | Dictionaries
-----------------+--------------
 asciihword      | english_stem
 asciiword       | english_stem
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | russian_stem
 hword_asciipart | english_stem
 hword_numpart   | simple
 hword_part      | russian_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
```

```
 uint           | simple
 url            | simple
 url_path       | simple
 version        | simple
 word           | russian_stem
```

\dFd[+] [PATTERN]

列出文本搜索词典（加上+得到更多细节）。

```
=> \dFd
                        List of text search dictionaries
  Schema   |     Name       |                     Description
------------+-----------------+-----------------------------------------------------------
 ux_catalog | arabic_stem     | snowball stemmer for arabic language
 ux_catalog | danish_stem     | snowball stemmer for danish language
 ux_catalog | dutch_stem      | snowball stemmer for dutch language
 ux_catalog | english_stem    | snowball stemmer for english language
 ux_catalog | finnish_stem    | snowball stemmer for finnish language
 ux_catalog | french_stem     | snowball stemmer for french language
 ux_catalog | german_stem     | snowball stemmer for german language
 ux_catalog | hungarian_stem  | snowball stemmer for hungarian language
 ux_catalog | indonesian_stem | snowball stemmer for indonesian language
 ux_catalog | irish_stem      | snowball stemmer for irish language
 ux_catalog | italian_stem    | snowball stemmer for italian language
 ux_catalog | lithuanian_stem | snowball stemmer for lithuanian language
 ux_catalog | nepali_stem     | snowball stemmer for nepali language
 ux_catalog | norwegian_stem  | snowball stemmer for norwegian language
 ux_catalog | portuguese_stem | snowball stemmer for portuguese language
 ux_catalog | romanian_stem   | snowball stemmer for romanian language
 ux_catalog | russian_stem    | snowball stemmer for russian language
 ux_catalog | simple          | simple dictionary: just lower case and check for stopword
 ux_catalog | spanish_stem    | snowball stemmer for spanish language
 ux_catalog | swedish_stem    | snowball stemmer for swedish language
 ux_catalog | tamil_stem      | snowball stemmer for tamil language
 ux_catalog | turkish_stem    | snowball stemmer for turkish language
```

\dFp[+] [PATTERN]

列出文本搜索解析器（加上+得到更多细节）。

```
=> \dFp
       List of text search parsers
  Schema   | Name   |     Description
------------+---------+---------------------
 ux_catalog | default | default word parser
=> \dFp+
    Text search parser "ux_catalog.default"
     Method      |    Function    | Description
-----------------+----------------+-------------
 Start parse     | prsd_start     |
 Get next token  | prsd_nexttoken |
 End parse       | prsd_end       |
```

```
 Get headline    | prsd_headline  |
 Get token types | prsd_lextype   |

       Token types for parser "ux_catalog.default"
   Token name    |              Description
-----------------+-----------------------------------------
 asciihword      | Hyphenated word, all ASCII
 asciiword       | Word, all ASCII
 blank           | Space symbols
 email           | Email address
 entity          | XML entity
 file            | File or path name
 float           | Decimal notation
 host            | Host
 hword           | Hyphenated word, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 int             | Signed integer
 numhword        | Hyphenated word, letters and digits
 numword         | Word, letters and digits
 protocol        | Protocol head
 sfloat          | Scientific notation
 tag             | XML tag
 uint            | Unsigned integer
 url             | URL
 url_path        | URL path
 version         | Version number
 word            | Word, all letters
(23 rows)
```

\dFt[+] [PATTERN]

列出文本搜索模板（加上+得到更多细节）。

```
=> \dFt
                        List of text search templates
   Schema   |   Name    |                        Description
------------+-----------+------------------------------------------------------------
 ux_catalog | ispell    | ispell dictionary
 ux_catalog | simple    | simple dictionary: just lower case and check for stopword
 ux_catalog | snowball  | snowball stemmer
 ux_catalog | synonym   | synonym dictionary: replace word by its synonym
 ux_catalog | thesaurus | thesaurus dictionary: phrase by phrase substitution
```

9.11. 限制

UXDB的文本搜索特性的当前限制是：

- 每一个词位的长度必须小于 2K 字节
- 一个tsvector（词位 + 位置）的长度必须小于 1 兆字节
- 词位的数量必须小于 2^{64}
- tsvector中的位置值必须大于 0 并且小于 16,383

- <*N*>（FOLLOWED BY）tsquery操作符中的匹配距离不能超过 16,384
- 每个词位不超过 256 个位置
- 一个tsquery中结点（词位 + 操作符）的个数必须小于 32,768

为了对比，UXDB的文档包含 10,441 个唯一词，总数 335,420 个词，并且最频繁的词“uxsinodb”在 655 个文档中被提到 6,127 次。

另一个例子 — UXDB的邮件列表归档在 461,020 条消息的 57,491,343 个词位中包含 910,989 个唯一词。

第 10 章 并发控制

本章描述UXDB数据库系统在多个会话试图同时访问同一数据时的行为。 这种情况的目标是为所有会话提供高效的访问，同时还要维护严格的数据完整性。每个数据库应用开发人员都应该熟悉本章讨论的话题。

10.1. 介绍

UXDB为开发者提供了一组丰富的工具来管理对数据的并发访问。在内部，数据一致性通过使用一种多版本模型（多版本并发控制，MVCC）来维护。这就意味着每个 SQL 语句看到的都只是一小段时间之前的数据快照（一个数据库版本），而不管底层数据的当前状态。这样可以保护语句不会看到可能由其他在相同数据行上执行更新的并发事务造成的不一致数据，为每一个数据库会话提供事务隔离。MVCC避免了传统的数据库系统的锁定方法，将锁争夺最小化来允许多用户环境中的合理性能。

使用MVCC并发控制模型而不是锁定的主要优点是在MVCC中，对查询（读）数据的锁请求与写数据的锁请求不冲突，所以读不会阻塞写，而写也从不阻塞读。甚至在通过使用革新的可序列化快照隔离（SSI）级别提供最严格的事务隔离级别时，UXDB也维持这个保证。

在UXDB里也有表和行级别的锁功能，用于那些通常不需要完整事务隔离并且想要显式管理特定冲突点的应用。不过，恰当地使用MVCC通常会提供比锁更好的性能。另外，由应用定义的咨询锁提供了一个获得不依赖于单一事务的锁的机制。

10.2. 事务隔离

SQL标准定义了四种隔离级别。最严格的是可序列化，在标准中用了一整段来定义它，其中说到一组可序列化事务的任意并发执行被保证效果和以某种顺序一个一个执行这些事务一样。其他三种级别使用并发事务之间交互产生的现象来定义，每一个级别中都要求必须不出现一种现象。注意由于可序列化的定义，在该级别上这些现象都不可能发生（这并不令人惊讶--如果事务的效果与每个时刻只运行一个的相同，怎么可能看见由于交互产生的现象？）。

在各个级别上被禁止出现的现象是：

脏读

一个事务读取了另一个并行未提交事务写入的数据。

不可重复读

一个事务重新读取之前读取过的数据，发现该数据已经被另一个事务（在初始读之后提交）修改。

幻读

一个事务重新执行一个返回符合一个搜索条件的行集合的查询， 发现满足条件的行集合因为另一个最近提交的事务而发生了改变。

序列化异常

成功提交一组事务的结果与这些事务所有可能的串行执行结果都不一致。

SQL 标准和 UXDB 实现的事务隔离级别在 表 10.1 “事务隔离级别”中描述。

表 10.1. 事务隔离级别

隔离级别	脏读	不可重复读	幻读	序列化异常
读未提交	允许，但不在 ux 中	可能	可能	可能
读已提交	不可能	可能	可能	可能
可重复读	不可能	不可能	允许，但不在 ux 中	可能
可序列化	不可能	不可能	不可能	不可能

在UXDB中，可以请求四种标准事务隔离级别中的任意一种，但是内部只实现了三种不同的隔离级别，即 UXDB 的读未提交模式的行为和读已提交相同。这是因为把标准隔离级别映射到 UXDB 的多版本并发控制架构的唯一合理的方法。

该表格也显示 UXDB 的可重复读实现不允许幻读。而 SQL 标准允许更严格的行为：四种隔离级别只定义了哪种现像不能发生，但是没有定义哪种现像必须发生。可用的隔离级别的行为在下面的小节中详细描述。

要设置一个事务的事务隔离级别，使用SET TRANSACTION(7)命令。

> **重要**
>
> 某些UXDB数据类型和函数关于事务的行为有特殊的规则。特别是，对一个序列的修改（以及用serial声明的一列的计数器）是立刻对所有其他事务可见的，并且在作出该修改的事务中断时也不会被回滚。见第 6.16 节“序列操作函数”和第 5.1.4 节 “序数类型”。

10.2.1. 读已提交隔离级别

读已提交是UXDB中的默认隔离级别。 当一个事务运行使用这个隔离级别时， 一个查询（没有FOR UPDATE/SHARE子句）只能看到查询开始之前已经被提交的数据， 而无法看到未提交的数据或在查询执行期间其它事务提交的数据。实际上，SELECT查询看到的是一个在查询开始运行的瞬间该数据库的一个快照。不过SELECT可以看见在它自身事务中之前执行的更新的效果，即使它们还没有被提交。还要注意的是，即使在同一个事务里两个相邻的SELECT命令可能看到不同的数据， 因为其它事务可能会在第一个SELECT开始和第二个SELECT开始之间提交。

UPDATE、DELETE、SELECT FOR UPDATE和SELECT FOR SHARE命令在搜索目标行时的行为和SELECT一样： 它们将只找到在命令开始时已经被提交的行。 不过，在被找到时，这样的目标行可能已经被其它并发事务更新（或删除或锁住）。在这种情况下， 即将进行的更新将等待第一个更新事务提交或者回滚（如果它还在进行中）。 如果第一个更新事务回滚，那么它的作用将被忽略并且第二个事务可以继续更新最初发现的行。 如果第一个更新事务提交，若该行被第一个更新者删除，则第二个更新事务将忽略该行，否则第二个更新者将试图在该行的已被更新的版本上应用它的操作。该命令的搜索条件（WHERE子句）将被重新计算来看该行被更新的版本是否仍然符合搜索条件。如果符合，则第二个更新者使用该行的已更新版本继续其操作。在SELECT FOR UPDATE和SELECT FOR SHARE的情况下，这意味着把该行的已更新版本锁住并返回给客户端。

带有ON CONFLICT DO UPDATE子句的 INSERT行为类似。在读已提交模式，要插入的 每一行将被插入或者更新。除非有不相干的错误出现，这两种结果之一是肯定 会出现的。如果在另一个事务中发生冲突，并且其效果对于INSERT 还不可见，则UPDATE子句将会 影响那个行，即便那一行对于该命令来说没有惯常的可见版本。

带有ON CONFLICT DO NOTHING子句的 INSERT有可能因为另一个效果对 INSERT快照不可见的事务的结果无法让插入进行 下去。再一次，这只是读已提交模式中的情况。

因为上面的规则，正在更新的命令可能会看到一个不一致的快照： 它们可以看到并发更新命令在它尝试更新的相同行上的作用，但是却看不到那些命令对数据库里其它行的作用。 这样的行为令读已提交模式不适合用于涉及复杂搜索条件的命令。不过，它对于更简单的情况是正确的。例如，考虑用这样的命令更新银行余额：

```
BEGIN;
UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 12345;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 7534;
COMMIT;
```

如果两个这样的事务同时尝试修改帐号 12345 的余额，那我们很明显希望第二个事务从账户行的已更新版本上开始工作。 因为每个命令只影响一个已经决定了的行，让它看到行的已更新版本不会导致任何麻烦的不一致性。

在读已提交模式中，更复杂的使用可能产生不符合需要的结果。例如： 考虑一个在数据上操作的DELETE命令，它操作的数据正被另一个命令从它的限制条件中移除或者加入，例如，假定website是一个两行的表，两行的website.hits等于9和10：

```
BEGIN;
UPDATE website SET hits = hits + 1;
-- run from another session:  DELETE FROM website WHERE hits = 10;
COMMIT;
```

即便在UPDATE之前有一个website.hits = 10的行，DELETE将不会产生效果。这是因为更新之前的行值9被跳过，并且当UPDATE完成并且DELETE获得一个锁，新行值不再是10而是11，这再也不匹配条件了。

因为在读已提交模式中，每个命令都是从一个新的快照开始的，而这个快照包含在该时刻已提交的事务， 因此同一事务中的后续命令将看到任何已提交的并行事务的效果。以上的焦点在于单个命令是否看到数据库的绝对一致的视图。

读已提交模式提供的部分事务隔离对于许多应用而言是足够的，并且这个模式速度快并且使用简单。 不过，它不是对于所有情况都够用。做复杂查询和更新的应用可能需要比读已提交模式提供的更严格一致的数据库视图。

10.2.2. 可重复读隔离级别

可重复读隔离级别只看到在事务开始之前被提交的数据；它从来看不到未提交的数据或者并行事务在本事务执行期间提交的修改（不过，查询能够看见在它的事务中之前执行的更新，即使它们还没有被提交）。这是比SQL标准对此隔离级别所要求的更强的保证，并且阻止表 10.1 “事务隔离级别”中描述的除了序列化异常之外的所有现象。如上面所提到的，这是标准特别允许的，标准只描述了每种隔离级别必须提供的最小保护。

这个级别与读已提交不同之处在于，一个可重复读事务中的查询可以看见在事务中第一个非事务控制语句开始时的一个快照，而不是事务中当前语句开始时的快照。因此，在一个单一事务中的后续SELECT命令看到的是相同的数据，即它们看不到其他事务在本事务启动后提交的修改。

使用这个级别的应用必须准备好由于序列化失败而重试事务。

UPDATE、DELETE、SELECT FOR UPDATE和SELECT FOR SHARE命令在搜索目标行时的行为和SELECT一样： 它们将只找到在事务开始时已经被提交的行。 不过，在被找到时，这样的目标行可能已经被其它并发事务更新（或删除或锁住）。在这种情况下， 可重复读事务将等待第一个更新事务提交或者回滚（如果它还在进行中）。 如果第一个更新事务回滚，那么它的作用将被忽略并且

可重复读事务可以继续更新最初发现的行。 但是如果第一个更新事务提交（并且实际更新或删除该行，而不是只锁住它），则可重复读事务将回滚并带有如下消息

```
ERROR:  could not serialize access due to concurrent update
```

因为一个可重复读事务无法修改或者锁住被其他在可重复读事务开始之后的事务改变的行。

当一个应用接收到这个错误消息，它应该中断当前事务并且从开头重试整个事务。在第二次执行中，该事务将见到作为其初始数据库视图一部分的之前提交的改变，这样在使用行的新版本作为新事务更新的起点时就不会有逻辑冲突。

注意只有更新事务可能需要被重试；只读事务将永远不会有序列化冲突。

可重复读模式提供了一种严格的保证，在其中每一个事务看到数据库的一个完全稳定的视图。不过，这个视图并不需要总是和同一级别上并发事务的某些序列化（一次一个）执行保持一致。例如，即使这个级别上的一个只读事务可能看到一个控制记录被更新，这显示一个批处理已经被完成但是不能看见作为该批处理的逻辑组成部分的一个细节记录，因为它读取空值记录的一个较早的版本。如果使用显式锁来阻塞冲突事务，尝试用运行在这个隔离级别的事务来强制业务规则不太可能正确地工作。

10.2.3. 可序列化隔离级别

可序列化隔离级别提供了最严格的事务隔离。这个级别为所有已提交事务模拟序列事务执行；就好像事务被按照序列一个接着另一个被执行，而不是并行地被执行。但是，和可重复读级别相似，使用这个级别的应用必须准备好因为序列化失败而重试事务。事实上，这个隔离级别完全像可重复读一样地工作，除了它会监视一些条件，这些条件可能导致一个可序列化事务的并发集合的执行产生的行为与这些事务所有可能的序列化（一次一个）执行不一致。这种监控不会引入超出可重复读之外的阻塞，但是监控会产生一些负荷，并且对那些可能导致序列化异常的条件的检测将触发一次序列化失败。

例如，考虑一个表mytab，它初始时包含：

```
 class | value
-------+-------
     1 |    10
     1 |    20
     2 |   100
     2 |   200
```

假设可序列化事务 A 计算：

```
SELECT SUM(value) FROM mytab WHERE class = 1;
```

并且接着把结果（3）作为一个新行的*value*插入，新行的*class* = 2。同时，可序列化事务 B 计算：

```
SELECT SUM(value) FROM mytab WHERE class = 2;
```

并得到结果 300，它会将其与*class* = 1插入到一个新行中。然后两个事务都尝试提交。如果其中一个事务运行在可重复读隔离级别，两者都被允许提交；但是由于没有执行的序列化顺序能在结果上一致，使用可序列化事务将允许一个事务提交并且将回滚另一个并伴有这个消息：

ERROR: could not serialize access due to read/write dependencies among transactions

这是因为，如果 A 在 B 之前执行，B 将计算得到合计值 330 而不是 300，而且相似地另一种顺序将导致 A 计算出一个不同的合计值。

当依赖可序列化事务来阻止异常时，重要的一点是任何从一个持久化用户表读出数据都不被认为是有效的，直到读它的事务已经成功提交为止。即便是对只读事务也是如此，除了在一个可推迟的只读事务中读取的数据是读出以后立刻有效的，因为这样的一个事务在开始读取任何数据之前会等待，直到它能获得一个快照保证来避免这种问题为止。在所有其他情况下，应用不能依靠在一个后来被中断的事务中读取的结果；相反，它们应当重试事务直到它成功。

要保证真正的可序列化，UXDB使用了谓词锁，这意味着它会保持锁，这些锁让它能够判断在它先运行的情况下，什么时候一个写操作会对一个并发事务中之前读取的结果产生影响。在UXDB中，这些锁并不导致任何阻塞，并且因此不会导致一个死锁。它们被用来标识和标志并发可序列化事务之间的依赖性，这些事务的组合可能导致序列化异常。相反，一个想要保证数据一致性的读已提交或可重复读事务可能需要拿走一个在整个表上的锁，这可能阻塞其他尝试使用该表的用户，或者它可能会使用不仅会阻塞其他事务还会导致磁盘访问的SELECT FOR UPDATE或SELECT FOR SHARE。

像大部分其他数据库系统，UXDB中的谓词锁基于被一个事务真正访问的数据。这些谓词锁将显示在ux_locks系统视图中，它们的mode为SIReadLock。这种在一个查询执行期间获得的特别的锁将依赖于该查询所使用的计划，并且在事务过程中多个细粒度锁（如元组锁）可能和少量粗粒度锁（如页面锁）相结合来防止耗尽用于跟踪锁的内存。如果一个READ ONLY事务检测到不会有导致序列化异常的冲突发生，它可以在完成前释放其 SIRead 锁。事实上，READ ONLY事务将常常可以在启动时确立这一事实并避免拿到任何谓词锁。如果显式地请求一个SERIALIZABLE READ ONLY DEFERRABLE事务，它将阻塞直到它能够确立这一事实（这是唯一一种可序列化事务阻塞但可重复读事务不阻塞的情况）。在另一方面，SIRead 锁常常需要被保持到事务提交之后，直到重叠的读写事务完成。

坚持使用可序列化事务可以简化开发。成功提交的并发可序列化事务的任意集合将得到和一次运行一个相同效果的这种保证意味着，如果能证明一个单一事务在独自运行时能做正确的事情，则可以相信它在任何混合的可序列化事务中也能做正确的事情，即使它不知道那些其他事务做了些什么，否则它将不会成功提交。重要的是使用这种技术的环境有一种普遍的方法来处理序列化失败（总是会返回一个 SQLSTATE 值 '40001'），因为它将很难准确地预计哪些事务可能为读/写依赖性做贡献并且需要被回滚来阻止序列化异常。读/写依赖性的监控会产生开销，如重启被序列化失败中止的事务，但是作为在该开销和显式锁及SELECT FOR UPDATE或SELECT FOR SHARE导致的阻塞之间的一种平衡，可序列化事务是在某些环境中最好性能的选择。

虽然UXDB的可序列化事务隔离级别只允许并发事务在能够证明有一种串行执行能够产生相同效果的前提下提交，但它却不能总是阻止在真正的串行执行中不会发生的错误产生。尤其是可能会看到由于可序列化事务重叠执行导致的唯一约束被违背的情况，这些情况即便在尝试插入键之前就显式地检查过该键不存在也会发生。避免这种问题的方法是，确保所有插入可能会冲突的键的可序列化事务首先显式地检查它们能不能那样做。例如，试想一个要求用户输入新键的应用，它会通过尝试查询用户给出的键来检查键是否已经存在，或者是通过选取现有最大的键并且加一来产生一个新键。如果某些可序列化事务不遵循这种协议而直接插入新键，则也可能会报告唯一约束被违背，即便在并发事务串行执行的情况下不会发生唯一约束被违背也是如此。

当依赖可序列化事务进行并发控制时，为了最佳性能应该考虑一下问题：

- 在可能时声明事务为READ ONLY。
- 控制活动连接的数量，如果需要使用一个连接池。这总是一个重要的性能考虑，但是在一个使用可序列化事务的繁忙系统中这尤为重要。
- 只在一个单一事务中放完整性目的所需要的东西。

- 不要让连接不必要地“闲置在事务中”。配置参数idle_in_transaction_session_timeout可以被用来自动断开拖延会话的连接。
- 在那些由于使用可序列化事务自动提供的保护的地方消除不再需要的显式锁、SELECT FOR UPDATE和SELECT FOR SHARE。
- 当系统因为谓词锁表内存短缺而被强制结合多个页面级谓词锁为一个单一的关系级谓词锁时，序列化失败的比例可能会上升。可以通过增加max_pred_locks_per_transaction、max_pred_locks_per_relation和max_pred_locks_per_page来避免这种情况。
- 一次顺序扫描将总是需要一个关系级谓词锁。这可能导致序列化失败的比例上升。通过缩减random_page_cost或增加cpu_tuple_cost来鼓励使用索引扫描将有助于此。一定要在事务回滚和重启数目的任何减少与查询执行时间的任何全面改变之间进行权衡。

10.3. 显式锁定

UXDB提供了多种锁模式用于控制对表中数据的并发访问。 这些模式可以用于在MVCC无法给出期望行为的情境中由应用控制的锁。 同样，大多数UXDB命令会自动要求恰当的锁以保证被引用的表在命令的执行过程中 不会以一种不兼容的方式删除或修改（例如，TRUNCATE无法安全地与同一表中上的其他操作并发地执行，因此它在表上获得一个排他锁来强制这种行为）。

要检查在一个数据库服务器中当前未解除的锁列表，可以使用ux_locks系统视图。

10.3.1. 表级锁

下面的列表显示了可用的锁模式和UXDB自动使用它们的场合。 也可以用LOCK(7)命令显式获得这些锁。请记住所有这些锁模式都是表级锁，即使它们的名字包含“row”单词（这些名称是历史遗产）。 在一定程度上，这些名字反应了每种锁模式的典型用法 — 但是语意却都是一样的。 两种锁模式之间真正的区别是它们有着不同的冲突锁模式集合（参考表 10.2 “冲突的锁模式”）。 两个事务在同一时刻不能在同一个表上持有属于相互冲突模式的锁（但是，一个事务决不会和自身冲突。例如，它可以在同一个表上获得ACCESS EXCLUSIVE锁然后接着获取ACCESS SHARE锁）。非冲突锁模式可以由许多事务同时持有。 请特别注意有些锁模式是自冲突的（例如，在一个时刻ACCESS EXCLUSIVE锁不能被多于一个事务持有）而其他锁模式不是自冲突的（例如，ACCESS SHARE锁可以被多个事务持有）。

表级锁模式

ACCESS SHARE

只与ACCESS EXCLUSIVE锁模式冲突。

SELECT命令在被引用的表上获得一个这种模式的锁。通常，任何只读取表而不修改它的查询都将获得这种锁模式。

ROW SHARE

与EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。

SELECT FOR UPDATE和SELECT FOR SHARE命令在目标表上取得一个这种模式的锁 （加上在被引用但没有选择FOR UPDATE/FOR SHARE的任何其他表上的ACCESS SHARE锁）。

ROW EXCLUSIVE

与SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。

命令UPDATE、DELETE和INSERT在目标表上取得这种锁模式（加上在任何其他被引用表上的ACCESS SHARE锁）。通常，这种锁模式将被任何修改表中数据的命令取得。

SHARE UPDATE EXCLUSIVE

与SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发模式改变和VACUUM运行的影响。

由VACUUM（不带FULL）、ANALYZE、CREATE INDEX CONCURRENTLY、REINDEX CONCURRENTLY、CREATE STATISTICS以及某些ALTER INDEX和 ALTER TABLE的变体获得(完整的详细请参考ALTER INDEX(7) 和ALTER TABLE(7))。

SHARE

与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发数据改变的影响。

由CREATE INDEX（不带CONCURRENTLY）取得。

SHARE ROW EXCLUSIVE

与ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式保护一个表不受并发数据修改所影响，并且是自排他的，这样在一个时刻只能有一个会话持有它。

由CREATE TRIGGER和某些形式的 ALTER TABLE所获得（见 ALTER TABLE(7)）。

EXCLUSIVE

与ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE锁模式冲突。这种模式只允许并发的ACCESS SHARE锁，即只有来自于表的读操作可以与一个持有该锁模式的事务并行处理。

由REFRESH MATERIALIZED VIEW CONCURRENTLY获得。

ACCESS EXCLUSIVE

与所有模式的锁冲突（ACCESS SHARE、ROW SHARE、ROW EXCLUSIVE、SHARE UPDATE EXCLUSIVE、SHARE、SHARE ROW EXCLUSIVE、EXCLUSIVE和ACCESS EXCLUSIVE）。这种模式保证持有者是访问该表的唯一事务。

由ALTER TABLE、DROP TABLE、TRUNCATE、REINDEX、CLUSTER、VACUUM FULL和REFRESH MATERIALIZED VIEW（不带CONCURRENTLY）命令获取。 很多形式的ALTER INDEX和ALTER TABLE也在这个层面上获得锁（见ALTER TABLE(7)）。这也是未显式指定模式的LOCK TABLE命令的默认锁模式。

> **提示**
>
> 只有一个ACCESS EXCLUSIVE锁阻塞一个SELECT（不带FOR UPDATE/SHARE）语句。

一旦被获取，一个锁通常将被持有直到事务结束。 但是如果在建立保存点之后才获得锁，那么在回滚到这个保存点的时候将立即释放该锁。 这与ROLLBACK取消保存点之后所有的影响的原则保持一致。 同样的原则也适用于在PL/uxSQL异常块中获得的锁：一个跳出块的错误将释放在块中获得的锁。

表 10.2. 冲突的锁模式

请求的锁模式	当前的锁模式							
	ACCESS SHARE	ROW SHARE	ROW EXCLUSIVE	SHARE UPDATE EXCLUSIVE	SHARE	SHARE ROW EXCLUSIVE	EXCLUSIVE	ACCESS EXCLUSIVE
ACCESS SHARE								X
ROW SHARE							X	X
ROW EXCLUSIVE					X	X	X	X
SHARE UPDATE EXCLUSIVE				X	X	X	X	X
SHARE			X	X		X	X	X
SHARE ROW EXCLUSIVE			X	X	X	X	X	X
EXCLUSIVE		X	X	X	X	X	X	X
ACCESS EXCLUSIVE	X	X	X	X	X	X	X	X

10.3.2. 行级锁

除了表级锁以外，还有行级锁，在下文列出了行级锁以及在哪些情境下UXDB会自动使用它们。行级锁的完整冲突表请见表 10.3 “冲突的行级锁”。注意一个事务可能会在相同的行上保持冲突的锁，甚至是在不同的子事务中。但是除此之外，两个事务永远不可能在相同的行上持有冲突的锁。行级锁不影响数据查询，它们只阻塞对同一行的写入者和加锁者。

行级锁模式

FOR UPDATE

FOR UPDATE会导致由SELECT语句检索到的行被锁定，就好像它们要被更新。这可以阻止它们被其他事务锁定、修改或者删除，一直到当前事务结束。也就是说其他尝试UPDATE、DELETE、SELECT FOR UPDATE、SELECT FOR NO KEY UPDATE、SELECT FOR SHARE或者SELECT FOR KEY SHARE这些行的事务将被阻塞，直到当前事务结束。反过来，SELECT FOR UPDATE将等待已经在相同行上运行以上这些命令的并发事务，并且接着锁定并且返回被更新的行（或者没有行，因为行可能已被删除）。不过，在一个REPEATABLE READ或SERIALIZABLE事务中，如果一个要被锁定的行在事务开始后被更改，将会抛出一个错误。进一步的讨论请见第 10.4 节 “应用级别的数据完整性检查”。

任何在一行上的DELETE命令也会获得FOR UPDATE锁模式，在某些列上修改值的UPDATE也会获得该锁模式。当前UPDATE情况中被考虑的列集合是那些具有能用于外键的唯一索引的列（所以部分索引和表达式索引不被考虑），但是这种要求未来有可能会改变。

FOR NO KEY UPDATE

行为与FOR UPDATE类似，不过获得的锁较弱：这种锁将不会阻塞尝试在相同行上获得锁的SELECT FOR KEY SHARE命令。任何不获取FOR UPDATE锁的UPDATE也会获得这种锁模式。

FOR SHARE

行为与FOR NO KEY UPDATE类似，不过它在每个检索到的行上获得一个共享锁而不是排他锁。一个共享锁会阻塞其他事务在这些行上执行UPDATE、DELETE、SELECT FOR UPDATE或者SELECT FOR NO KEY UPDATE，但是它不会阻止它们执行SELECT FOR SHARE或者SELECT FOR KEY SHARE。

FOR KEY SHARE

行为与FOR SHARE类似，不过锁较弱：SELECT FOR UPDATE会被阻塞，但是SELECT FOR NO KEY UPDATE不会被阻塞。一个键共享锁会阻塞其他事务执行修改键值的DELETE或者UPDATE，但不会阻塞其他UPDATE，也不会阻止SELECT FOR NO KEY UPDATE、SELECT FOR SHARE或者SELECT FOR KEY SHARE。

UXDB不会在内存里保存任何关于已修改行的信息，因此对一次锁定的行数没有限制。 不过，锁住一行会导致一次磁盘写，例如， SELECT FOR UPDATE将修改选中的行以标记它们被锁住，并且因此会导致磁盘写入。

表 10.3. 冲突的行级锁

要求的锁模式	当前的锁模式			
	FOR KEY SHARE	FOR SHARE	FOR NO KEY UPDATE	FOR UPDATE
FOR KEY SHARE				X
FOR SHARE			X	X
FOR NO KEY UPDATE		X	X	X
FOR UPDATE	X	X	X	X

10.3.3. 页级锁

除了表级别和行级别的锁以外，页面级别的共享/排他锁被用来控制对共享缓冲池中表页面的读/写。 这些锁在行被抓取或者更新后马上被释放。应用开发者通常不需要关心页级锁，我们在这里提到它们只是为了完整。

10.3.4. 死锁

显式锁定的使用可能会增加*死锁*的可能性，死锁是指两个（或多个）事务相互持有对方想要的锁。例如，如果事务 1 在表 A 上获得一个排他锁，同时试图获取一个在表 B 上的排他锁， 而事务 2 已经持有表 B 的排他锁，同时却正在请求表 A 上的一个排他锁，那么两个事务就都不能进行下去。UXDB能够自动检测到死锁情况并且会通过中断其中一个事务从而允许其它事务完成来解决这个问题（具体哪个事务会被中断是很难预测的，而且也不应该依靠这样的预测）。

要注意死锁也可能会作为行级锁的结果而发生（并且因此，它们即使在没有使用显式锁定的情况下也会发生）。考虑如下情况，两个并发事务在修改一个表。第一个事务执行：

UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 11111;

这样就在指定帐号的行上获得了一个行级锁。然后，第二个事务执行：

UPDATE accounts SET balance = balance + 100.00 WHERE acctnum = 22222;
UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 11111;

第一个UPDATE语句成功地在指定行上获得了一个行级锁，因此它成功更新了该行。 但是第二个UPDATE语句发现它试图更新的行已经被锁住了，因此它等待持有该锁的事务结束。事务二现在就在等待事务一结束，然后再继续执行。现在，事务一执行：

UPDATE accounts SET balance = balance - 100.00 WHERE acctnum = 22222;

事务一试图在指定行上获得一个行级锁，但是它得不到：事务二已经持有了这样的锁。所以它要等待事务二完成。因此，事务一被事务二阻塞，而事务二也被事务一阻塞：一个死锁。 UXDB将检测这样的情况并中断其中一个事务。

防止死锁的最好方法通常是保证所有使用一个数据库的应用都以一致的顺序在多个对象上获得锁。在上面的例子里，如果两个事务以同样的顺序更新那些行，那么就不会发生死锁。 我们也应该保证一个事务中在一个对象上获得的第一个锁是该对象需要的最严格的锁模式。如果我们无法提前验证这些，那么可以通过重试因死锁而中断的事务来即时处理死锁。

只要没有检测到死锁情况，寻求一个表级或行级锁的事务将无限等待冲突锁被释放。这意味着一个应用长时间保持事务开启不是什么好事（例如等待用户输入）。

10.3.5. 咨询锁

UXDB提供了一种方法创建由应用定义其含义的锁。这种锁被称为*咨询锁*，因为系统并不强迫其使用 — 而是由应用来保证其正确的使用。咨询锁可用于 MVCC 模型不适用的锁定策略。例如，咨询锁的一种常用用法是模拟所谓“平面文件”数据管理系统典型的悲观锁策略。虽然一个存储在表中的标志可以被用于相同目的，但咨询锁更快、可以避免表膨胀并且会由服务器在会话结束时自动清理。

有两种方法在UXDB中获取一个咨询锁：在会话级别或在事务级别。一旦在会话级别获得了咨询锁，它将被保持直到被显式释放或会话结束。不同于标准锁请求，会话级咨询锁请求不尊重事务语义：在一个后来被回滚的事务中得到的锁在回滚后仍然被保持，并且同样即使调用它的事务后来失败一个解锁也是有效的。一个锁在它所属的进程中可以被获取多次；对于每一个完成的锁请求必须有一个相应的解锁请求，直至锁被真正释放。在另一方面，事务级锁请求的行为更像普通锁请求：在事务结束时会自动释放它们，并且没有显式的解锁操作。这种行为通常比会话级别的行为更方便，因为它使用一个咨询锁的时间更短。对于同一咨询锁标识符的会话级别和事务级别的锁请求按照期望将彼此阻塞。如果一个会话已经持有了一个给定的咨询锁，由它发出的附加请求将总是成功，即使有其他会话在等待该锁；不管现有的锁和新请求是处在会话级别还是事务级别，这种说法都是真的。

和所有UXDB中的锁一样，当前被任何会话所持有的咨询锁的完整列表可以在ux_locks系统视图中找到。

咨询锁和普通锁都被存储在一个共享内存池中，它的尺寸由max_locks_per_transaction和max_connections配置变量定义。 必须当心不要耗尽这些内存，否则服务器将不能再授予任何锁。这对服务器可以授予的咨询锁数量设置了一个上限，根据服务器的配置不同，这个限制通常是数万到数十万。

在使用咨询锁方法的特定情况下，特别是查询中涉及显式排序和LIMIT子句时，由于 SQL 表达式被计算的顺序，必须控制锁的获取。例如：

```
SELECT ux_advisory_lock(id) FROM foo WHERE id = 12345; -- ok
SELECT ux_advisory_lock(id) FROM foo WHERE id > 12345 LIMIT 100; -- danger!
SELECT ux_advisory_lock(q.id) FROM
(
  SELECT id FROM foo WHERE id > 12345 LIMIT 100
) q; -- ok
```

在上述查询中，第二种形式是危险的，因为不能保证在锁定函数被执行之前应用LIMIT。这可能导致获得某些应用不期望的锁，并因此在会话结束之前无法释放。 从应用的角度来看，这样的锁将被挂起，虽然它们仍然在ux_locks中可见。

提供的操作咨询锁函数在第 6.26.10 节 “咨询锁函数”中描述。

10.4. 应用级别的数据完整性检查

对于使用读已提交事务的数据完整性强制业务规则非常困难，因为对每一个语句数据视图都在变化，并且如果一个写冲突发生即使一个单一语句也不能把它自己限制到该语句的快照。

虽然一个可重复读事务在其执行期间有一个稳定的数据视图，在使用MVCC快照进行数据一致性检查时也有一个小问题，它涉及到被称为读/写冲突的东西。如果一个事务写数据并且一个并发事务尝试读相同的数据（不管是在写之前还是之后），它不能看到其他事务的工作。读取事务看起来是第一个执行的，不管哪个是第一个启动或者哪个是第一个提交。如果就到此为止，则没有问题，但是如果读取者也写入被一个并发事务读取的数据，现在有一个事务好像是已经在前面提到的任何一个事务之前运行。如果看起来最后执行的事务实际上第一个提交，在这些事务的执行顺序图中很容易出现一个环。当这样一个环出现时，完整性检查在没有任何帮助的情况下将不会正确地工作。

正如第 10.2.3 节 “可序列化隔离级别”中提到的，可序列化事务仅仅是可重复读事务增加了对读/写冲突的危险模式的非阻塞监控。当检测到一个可能导致表面的执行顺序中产生环的模式，涉及到的一个事务将被回滚来打破该环。

10.4.1. 用可序列化事务来强制一致性

如果可序列化事务隔离级别被用于所有需要一个一致数据视图的写入和读取，不需要其他的工作来保证一致性。在UXDB中，来自于其他环境的被编写成使用可序列化事务来保证一致性的软件应该“只工作”在这一点上。

当使用这种技术时，如果应用软件通过一个框架来自动重试由于序列化错误而回滚的事务，它将避免为应用程序员带来不必要的负担。把default_transaction_isolation设置为serializable可能是个好主意。通过触发器中的事务隔离级别检查来采取某些动作来保证没有其他事务隔离级别被使用（由于疏忽或者为了破坏完整性检查）也是明智的。

性能建议见第 10.2.3 节 “可序列化隔离级别”。

> **警告**
>
> 这个级别的使用可序列化事务的完整性保护还没有扩展到热备份模式。由于这个原因，那些使用热备份的系统可能想要在主控机上使用可重复读和显式锁定。

10.4.2. 使用显式锁定强制一致性

当可以使用非可序列化写时，要保证一行的当前有效性并保护它不受并发更新的影响，我们必须使用SELECT FOR UPDATE、SELECT FOR SHARE或一个合适的LOCK TABLE 语句（SELECT FOR UPDATE和SELECT FOR SHARE锁只针对并发更新返回行，而LOCK TABLE会锁住整个表）。当从其他环境移植应用到UXDB时需要考虑这些。

关于这些来自其他环境的转换还需要注意的是SELECT FOR UPDATE不保证一个并发事务将不会更新或删除一个被选中的行。要在UXDB中这样做，必须真正地更新该行，即便没有值需要被改变。SELECT FOR UPDATE 临时阻塞其他事务，让它们不能获取该相同的锁或者执行一个会影响被锁定行的UPDATE或DELETE，但是一旦正持有该所锁的事务提交或回滚，一个被阻塞的事务将继续执行冲突操作，除非当锁被持有时一个该行的实际UPDATE被执行。

在非可序列化MVCC环境下，全局有效性检查需要一些额外的考虑。例如，一个银行应用可能会希望检查一个表中的所有扣款总和等于另外一个表中的收款总和，同时两个表还会被更新。比较两个连续的在读已提交模式下不会可靠工作的SELECT sum(...)命令， 因为第二个查询很可能会包含没有被第一个查询考虑的事务提交的结果。在一个单一的可重复读事务里进行两个求和则给出在可串行化事务开始之前提交的所有事务产生的准确结果 — 但有人可能会合理地置疑在结果被递交的时候，它们是否仍然相关。 如果可重复读事务本身在尝试做一致性检查之前应用了某些变更，那么检查的有用性就更加值得讨论了， 因为现在它包含了一些（但不是全部）事务开始后的变化。 在这种情况下，一个细心的人可能希望锁住所有需要检查的表，这样才能获得一个无可置疑的当前现状的图像。 一个SHARE模式（或者更高）的锁保证在被锁定表中除了当前事务所作的更改之外，没有未提交的更改。

还要注意如果某人正在依赖显式锁定来避免并发更改，那么他应该使用读已提交模式， 或者是在可重复读模式里在执行命令之前获取锁。 在可重复读事务里获取的锁保证了不会有其它修改该表的事务正在运行，但是如果事务看到的快照在获取锁之前， 那么它可能早于表中一些现在已经提交的更改。 一个可重复读事务的快照实际上是在它的第一个查询或者数据修改命令（SELECT、INSERT、UPDATE或DELETE）开始的时候冻结的，因此我们可以在快照冻结之前显式地获取锁。

10.5. 提醒

一些 DDL 命令（当前只有TRUNCATE(7)和表重写形式的ALTER TABLE(7)）对于 MVCC 不是安全的。这意味着在截断或者重写提交之后，该表将对并发事务（如果它们使用的快照是在 DDL 命令提交前取得的）呈现出空表的形态。这只对没有在该 DDL 命令开始前访问所讨论的表的事务存在问题 — 任何在 DDL 命令开始前访问过该表的事务将持有至少一个 ACCESS SHARE 表锁，这将阻塞该 DDL 命令直到该事务完成。因此这些命令对于目标表上的连续查询将不会造成任何明显的表内容不一致，但是它们可能导致目标表内容和数据库中其他表内容之间的不一致。

对于可序列化事务隔离级别的支持还没有被加入到热备复制目标中。当前在热备模式中支持的最严格的隔离级别是可重复读。虽然在主控机上用可序列化事务执行所有持久化数据库写入将确保所有后备机将最终达到一个一致的状态，但是运行在后备机上的一个可重复读事务有时可能会看到一个短暂的、与主控机上事务的任何串行执行都不一致的状态。

10.6. 锁定和索引

尽管UXDB提供对表数据访问的非阻塞读/写， 但并非UXDB中实现的每一个索引访问方法当前都能够提供非阻塞读/写访问。 不同的索引类型按照下面方法操作：

B-tree、GiST和SP-GiST索引

短期的页面级共享/排他锁被用于读/写访问。每个索引行被取得或被插入后立即释放锁。这些索引类型提供了无死锁情况的最高并发性。

Hash索引

Hash 桶级别的共享/排他锁被用于读/写访问。锁在整个 Hash 桶处理完成后释放。Hash 桶级锁比索引级的锁提供了更好的并发性但是可能产生死锁，因为锁持有的时间比一次索引操作时间长。

GIN索引

短期的页面级共享/排他锁被用于读/写访问。 锁在索引行被插入/抓取后立即释放。但要注意的是一个 GIN 索引值的插入通常导致对每行产生几个索引键的插入，因此 GIN 可能为了插入一个单一值而做大量的工作。

目前，B-tree 索引为并发应用提供了最好的性能。因为它还有比 Hash 索引更多的特性，在那些需要对标量数据进行索引的并发应用中，我们建议使用 B-tree 索引类型。在处理非标量类型数据的时候，B-tree 就没什么用了，应该使用 GiST、SP-GiST 或 GIN 索引替代。

第 11 章 性能提示

查询性能可能受多种因素影响。其中一些因素可以由用户控制，而其它的则属于系统下层设计的基本原理。 本章我们提供一些有关理解和调节UXDB性能的提示。

11.1. 使用EXPLAIN

UXDB为每个收到查询产生一个查询计划。 选择正确的计划来匹配查询结构和数据的属性对于好的性能来说绝对是最关键的，因此系统包含了一个复杂的规划器来尝试选择好的计划。 可以使用EXPLAIN(7)命令察看规划器为任何查询生成的查询计划。 阅读查询计划是一门艺术，它要求一些经验来掌握，但是本节只试图覆盖一些基础。

本节中的例子都是从回归测试数据库中抽取出来的，并且在此之前做过一次VACUUM ANALYZE。应该能够在自己尝试这些例子时得到相似的结果，但是估计代价和行计数可能会小幅变化，因为ANALYZE的统计信息是随机采样而不是精确值，并且代价也与平台有某种程度的相关性。

这些例子使用EXPLAIN的默认“text”输出格式，这种格式紧凑并且便于人类阅读。如果想把EXPLAIN的输出交给一个程序做进一步分析，应该使用它的某种机器可读的输出格式（XML、JSON 或 YAML）。

11.1.1. EXPLAIN基础

查询计划的结构是一个计划结点的树。最底层的结点是扫描结点：它们从表中返回未经处理的行。 不同的表访问模式有不同的扫描结点类型：顺序扫描、索引扫描、位图索引扫描。 也还有不是表的行来源，例如VALUES子句和FROM中返回集合的函数，它们有自己的结点类型。如果查询需要连接、聚集、排序、或者在未经处理的行上的其它操作，那么就会在扫描结点之上有其它额外的结点来执行这些操作。 并且，做这些操作通常都有多种方法，因此在这些位置也有可能出现不同的结点类型。 EXPLAIN给计划树中每个结点都输出一行，显示基本的结点类型和计划器为该计划结点的执行所做的开销估计。 第一行（最上层的结点）是对该计划的总执行开销的估计；计划器试图最小化的就是这个数字。

这里是一个简单的例子，只是用来显示输出看起来是什么样的：

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

由于这个查询没有WHERE子句，它必须扫描表中的所有行，因此计划器只能选择使用一个简单的顺序扫描计划。被包含在圆括号中的数字是（从左至右）：

- 估计的启动开销。在输出阶段可以开始之前消耗的时间，例如在一个排序结点里执行排序的时间。
- 估计的总开销。这个估计值基于的假设是计划结点会被运行到完成，即所有可用的行都被检索。不过实际上一个结点的父结点可能很快停止读所有可用的行（见下面的LIMIT例子）。
- 这个计划结点输出行数的估计值。同样，也假定该结点能运行到完成。
- 预计这个计划结点输出的行平均宽度（以字节计算）。

开销是用规划器的开销参数所决定的捏造单位来衡量的。传统上以取磁盘页面为单位来度量开销； 也就是seq_page_cost将被按照习惯设为1.0，其它开销参数将相对于它来设置。 本节的例子都假定这些参数使用默认值。

有一点很重要：一个上层结点的开销包括它的所有子结点的开销。还有一点也很重要：这个开销只反映规划器关心的东西。特别是这个开销没有考虑结果行传递给客户端所花费的时间，这个时间可能是实际花费时间中的一个重要因素；但是它被规划器忽略了，因为它无法通过修改计划来改变（我们相信，每个正确的计划都将输出同样的行集）。

行数值有一些小技巧，因为它不是计划结点处理或扫描过的行数，而是该结点发出的行数。这通常比被扫描的行数少一些， 因为有些被扫描的行会被应用于此结点上的任意WHERE子句条件过滤掉。 理想中顶层的行估计会接近于查询实际返回、更新、删除的行数。

回到我们的例子：

```
EXPLAIN SELECT * FROM tenk1;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

这些数字的产生非常直接。如果执行：

```
SELECT relpages, reltuples FROM ux_class WHERE relname = 'tenk1';
```

会发现tenk1有358个磁盘页面和10000行。 开销被计算为 （页面读取数*seq_page_cost）+（扫描的行数*cpu_tuple_cost）。默认情况下，seq_page_cost是1.0，cpu_tuple_cost是0.01， 因此估计的开销是 (358 * 1.0) + (10000 * 0.01) = 458。

现在让我们修改查询并增加一个WHERE条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 7000;

                        QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7001 width=244)
   Filter: (unique1 < 7000)
```

请注意EXPLAIN输出显示WHERE子句被当做一个“过滤器”条件附加到顺序扫描计划结点。 这意味着该计划结点为它扫描的每一行检查该条件，并且只输出通过该条件的行。因为WHERE子句的存在，估计的输出行数降低了。不过，扫描仍将必须访问所有 10000 行，因此开销没有被降低；实际上开销还有所上升（准确来说，上升了 10000 * cpu_operator_cost）以反映检查WHERE条件所花费的额外 CPU 时间。

这条查询实际选择的行数是 7000，但是估计的rows只是个近似值。如果尝试重复这个试验，那么很可能得到略有不同的估计。 此外，这个估计会在每次ANALYZE命令之后改变， 因为ANALYZE生成的统计数据是从该表中随机采样计算的。

现在，让我们把条件变得更严格：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100;

                        QUERY PLAN
```

```
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

这里，规划器决定使用一个两步的计划：子计划结点访问访问一个索引来找出匹配索引条件的行的位置，然后上层计划结点实际地从表中取出那些行。独立地抓取行比顺序地读取它们的开销高很多，但是不是所有的表页面都被访问，这么做实际上仍然比一次顺序扫描开销要少（使用两层计划的原因是因为上层规划结点把索引标识出来的行位置在读取之前按照物理位置排序，这样可以最小化单独抓取的开销。结点名称里面提到的“位图”是执行该排序的机制）。

现在让我们给WHERE子句增加另一个条件：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND stringu1 = 'xxx';

                            QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique1 < 100)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
```

新增的条件stringu1 = 'xxx'减少了估计的输出行计数， 但是没有减少开销，因为我们仍然需要访问相同的行集合。 请注意，stringu1子句不能被应用为一个索引条件，因为这个索引只是在unique1列上。 它被用来过滤从索引中检索出的行。因此开销实际上略微增加了一些以反映这个额外的检查。

在某些情况下规划器将更倾向于一个“simple”索引扫描计划：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 = 42;

                            QUERY PLAN
------------------------------------------------------------------------------
 Index Scan using tenk1_unique1 on tenk1  (cost=0.29..8.30 rows=1 width=244)
   Index Cond: (unique1 = 42)
```

在这类计划中，表行被按照索引顺序取得，这使得读取它们开销更高，但是其中有一些是对行位置排序的额外开销。很多时候将在只取得一个单一行的查询中看到这种计划类型。它也经常被用于拥有匹配索引顺序的ORDER BY子句的查询中，因为那样就不需要额外的排序步骤来满足ORDER BY。

如果在WHERE引用的多个行上有独立的索引，规划器可能会选择使用这些索引的一个 AND 或 OR 组合：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000;

                                QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0)
```

```
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
         Index Cond: (unique1 < 100)
   ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0)
         Index Cond: (unique2 > 9000)
```

但是这要求访问两个索引，所以与只使用一个索引并把其他条件作为过滤器相比，它不一定能胜出。如果变动涉及到的范围，将看到计划也会相应改变。

下面是一个例子，它展示了LIMIT的效果：

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT 2;

                                      QUERY PLAN
-------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.48 rows=2 width=244)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..71.27 rows=10 width=244)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
```

这是和上面相同的查询，但是我们增加了一个LIMIT这样不是所有的行都需要被检索，并且规划器改变了它的决定。注意索引扫描结点的总开销和行计数显示出好像它会被运行到完成。但是，限制结点在检索到这些行的五分之一后就会停止，因此它的总开销只是索引扫描结点的五分之一，并且这是查询的实际估计开销。之所以用这个计划而不是在之前的计划上增加一个限制结点是因为限制无法避免在位图扫描上花费启动开销，因此总开销会是超过那种方法（25个单位）的某个值。

让我们尝试连接两个表，使用我们已经讨论过的列：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

在这个计划中，我们有一个嵌套循环连接结点，它有两个表扫描作为输入或子结点。该结点的摘要行的缩进反映了计划树的结构。连接的第一个（或“outer”）子结点是一个与前面见到的相似的位图扫描。它的开销和行计数与我们从SELECT ... WHERE unique1 < 10得到的相同，因为我们将WHERE子句unique1 < 10用在了那个结点上。t1.unique2 = t2.unique2子句现在还不相关，因此它不影响 outer 扫描的行计数。嵌套循环连接结点将为从 outer 子结点得到的每一行运行它的第二个（或“inner”）子结点。当前 outer 行的列值可以被插入 inner 扫描。这里，来自 outer 行的t1.unique2值是可用的，所以我们得到的计划和开销与前面见到的简单SELECT ... WHERE t2.unique2 = *constant*情况相似（估计的开销实际上比前面看到的略低，是因为在t2上的重复索引扫描会利用到高速缓存）。循环结点的开销则被以 outer 扫描的开销为基础设置，外加对每一个 outer 行都要进行一次 inner 扫描 （10 * 7.87），再加上用于连接处理一点 CPU 时间。

在这个例子里，连接的输出行计数等于两个扫描的行计数的乘积，但通常并不是所有的情况中都如此， 因为可能有同时提及两个表的 额外WHERE子句，并且因此它只能被应用于连接点，而不能影响任何一个输入扫描。这里是一个例子：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t2.unique2 < 10 AND t1.hundred < t2.hundred;

                               QUERY PLAN
-----------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..49.46 rows=33 width=488)
   Join Filter: (t1.hundred < t2.hundred)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0)
               Index Cond: (unique1 < 10)
   ->  Materialize  (cost=0.29..8.51 rows=10 width=244)
         ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..8.46 rows=10 width=244)
               Index Cond: (unique2 < 10)
```

条件t1.hundred < t2.hundred不能在tenk2_unique2索引中被测试，因此它被应用在连接结点。这缩减了连接结点的估计输出行计数，但是没有改变任何输入扫描。

注意这里规划器选择了“物化”连接的 inner 关系，方法是在它的上方放了一个物化计划结点。这意味着t2索引扫描将只被做一次，即使嵌套循环连接结点需要读取其数据十次（每个来自 outer 关系的行都要读一次）。物化结点在读取数据时将它保存在内存中，然后在每一次后续执行时从内存返回数据。

在处理外连接时，可能会看到连接计划结点同时附加有“连接过滤器”和普通“过滤器”条件。连接过滤器条件来自于外连接的ON子句，因此一个无法通过连接过滤器条件的行也能够作为一个空值扩展的行被发出。但是一个普通过滤器条件被应用在外连接条件之后并且因此无条件移除行。在一个内连接中这两种过滤器类型没有语义区别。

如果我们把查询的选择度改变一点，我们可能得到一个非常不同的连接计划：

```
EXPLAIN SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                               QUERY PLAN
-----------------------------------------------------------------------------------------
 Hash Join  (cost=230.47..713.98 rows=101 width=488)
   Hash Cond: (t2.unique2 = t1.unique2)
   ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244)
   ->  Hash  (cost=229.20..229.20 rows=101 width=244)
         ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244)
               Recheck Cond: (unique1 < 100)
               ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0)
                     Index Cond: (unique1 < 100)
```

这里规划器选择了使用一个哈希连接，在其中一个表的行被放入一个内存哈希表，在这之后其他表被扫描并且为每一行查找哈希表来寻找匹配。同样要注意缩进是如何反映计划结构

的：tenk1上的位图扫描是哈希结点的输入，哈希结点会构造哈希表。然后哈希表会返回给哈希连接结点，哈希连接结点将从它的 outer 子计划读取行，并为每一个行搜索哈希表。

另一种可能的连接类型是一个归并连接，如下所示：

```
EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=198.11..268.19 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Sort  (cost=197.83..200.33 rows=1000 width=244)
         Sort Key: t2.unique2
         ->  Seq Scan on onek t2  (cost=0.00..148.00 rows=1000 width=244)
```

归并连接要求它的输入数据被按照连接键排序。在这个计划中，tenk1数据被使用一个索引扫描排序，以便能够按照正确的顺序来访问行。但是对于onek则更倾向于一个顺序扫描和排序，因为在那个表中有更多行需要被访问（对于很多行的排序，顺序扫描加排序常常比一个索引扫描好，因为索引扫描需要非顺序的磁盘访问）。

一种查看变体计划的方法是强制规划器丢弃它认为开销最低的任何策略，这可以使用启用/禁用标志实现（这是一个野蛮的工具，但是很有用。另见第 11.3 节 “用显式JOIN子句控制规划器”）。例如，如果我们并不认同在前面的例子中顺序扫描加排序是处理表onek的最佳方法，我们可以尝试：

```
SET enable_sort = off;

EXPLAIN SELECT *
FROM tenk1 t1, onek t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2;

                                QUERY PLAN
------------------------------------------------------------------------------------------
 Merge Join  (cost=0.56..292.65 rows=10 width=488)
   Merge Cond: (t1.unique2 = t2.unique2)
   ->  Index Scan using tenk1_unique2 on tenk1 t1  (cost=0.29..656.28 rows=101 width=244)
         Filter: (unique1 < 100)
   ->  Index Scan using onek_unique2 on onek t2  (cost=0.28..224.79 rows=1000 width=244)
```

这显示规划器认为用索引扫描来排序onek的开销要比用顺序扫描加排序的方式高大约12%。当然，下一个问题是是否真的是这样。我们可以通过使用EXPLAIN ANALYZE来仔细研究一下，如下文所述。

11.1.2. EXPLAIN ANALYZE

可以通过使用EXPLAIN的ANALYZE选项来检查规划器估计值的准确性。通过使用这个选项，EXPLAIN会实际执行该查询，然后显示真实的行计数和在每个计划结点中累计的真实运行时间，还会有一个普通EXPLAIN显示的估计值。例如，我们可能得到这样一个结果：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 10 AND t1.unique2 = t2.unique2;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=4.65..118.62 rows=10 width=488) (actual time=0.128..0.377 rows=10 loops=1)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.36..39.47 rows=10 width=244) (actual time=0.057..0.121
rows=10 loops=1)
         Recheck Cond: (unique1 < 10)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.36 rows=10 width=0) (actual
time=0.024..0.024 rows=10 loops=1)
               Index Cond: (unique1 < 10)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.29..7.91 rows=1 width=244) (actual
time=0.021..0.022 rows=1 loops=10)
         Index Cond: (unique2 = t1.unique2)
 Planning time: 0.181 ms
 Execution time: 0.501 ms
```

注意“actual time”值是以毫秒计的真实时间，而cost估计值被以捏造的单位表示，因此它们不大可能匹配上。在这里面要查看的最重要的一点是估计的行计数是否合理地接近实际值。在这个例子中，估计值都是完全正确的，但是在实际中非常少见。

在某些查询计划中，可以多次执行一个子计划结点。例如，inner 索引扫描可能会因为上层嵌套循环计划中的每一个 outer 行而被执行一次。在这种情况下，loops值报告了执行该结点的总次数，并且 actual time 和行数值是这些执行的平均值。这是为了让这些数字能够与开销估计被显示的方式有可比性。将这些值乘上loops值可以得到在该结点中实际消耗的总时间。在上面的例子中，我们在执行tenk2的索引扫描上花费了总共 0.220 毫秒。

在某些情况中，EXPLAIN ANALYZE会显示计划结点执行时间和行计数之外的额外执行统计信息。例如，排序和哈希结点提供额外的信息：

```
EXPLAIN ANALYZE SELECT *
FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 100 AND t1.unique2 = t2.unique2 ORDER BY t1.fivethous;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=717.34..717.59 rows=101 width=488) (actual time=7.761..7.774 rows=100 loops=1)
   Sort Key: t1.fivethous
   Sort Method: quicksort  Memory: 77kB
   ->  Hash Join  (cost=230.47..713.98 rows=101 width=488) (actual time=0.711..7.427 rows=100
loops=1)
         Hash Cond: (t2.unique2 = t1.unique2)
         ->  Seq Scan on tenk2 t2  (cost=0.00..445.00 rows=10000 width=244) (actual time=0.007..2.583
rows=10000 loops=1)
         ->  Hash  (cost=229.20..229.20 rows=101 width=244) (actual time=0.659..0.659 rows=100
loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 28kB
               ->  Bitmap Heap Scan on tenk1 t1  (cost=5.07..229.20 rows=101 width=244) (actual
time=0.080..0.526 rows=100 loops=1)
                     Recheck Cond: (unique1 < 100)
```

```
                ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
time=0.049..0.049 rows=100 loops=1)
                      Index Cond: (unique1 < 100)
Planning time: 0.194 ms
Execution time: 8.008 ms
```

排序结点显示使用的排序方法（尤其是，排序是在内存中还是磁盘上进行）和需要的内存或磁盘空间量。哈希结点显示了哈希桶的数量和批数，以及被哈希表所使用的内存量的峰值（如果批数超过一，也将会涉及到磁盘空间使用，但是并没有被显示）。

另一种类型的额外信息是被一个过滤器条件移除的行数：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE ten < 7;

                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=7000 width=244) (actual time=0.016..5.107 rows=7000
loops=1)
   Filter: (ten < 7)
   Rows Removed by Filter: 3000
 Planning time: 0.083 ms
 Execution time: 5.905 ms
```

这些值对于被应用在连接结点上的过滤器条件特别有价值。只有在至少有一个被扫描行或者在连接结点中一个可能的连接对被过滤器条件拒绝时，“Rows Removed”行才会出现。

一个与过滤器条件相似的情况出现在“有损”索引扫描中。例如，考虑这个查询，它搜索包含一个指定点的多边形：

```
EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on polygon_tbl  (cost=0.00..1.05 rows=1 width=32) (actual time=0.044..0.044 rows=0
loops=1)
   Filter: (f1 @> '((0.5,2))'::polygon)
   Rows Removed by Filter: 4
 Planning time: 0.040 ms
 Execution time: 0.083 ms
```

规划器认为（非常正确）这个采样表太小不值得劳烦一次索引扫描，因此我们得到了一个普通的顺序扫描，其中的所有行都被过滤器条件拒绝。但是如果我们强制使得一次索引扫描可以被使用，我们看到：

```
SET enable_seqscan TO off;

EXPLAIN ANALYZE SELECT * FROM polygon_tbl WHERE f1 @> polygon '(0.5,2.0)';

                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Index Scan using gpolygonind on polygon_tbl  (cost=0.13..8.15 rows=1 width=32) (actual
time=0.062..0.062 rows=0 loops=1)
   Index Cond: (f1 @> '((0.5,2))'::polygon)
```

```
  Rows Removed by Index Recheck: 1
Planning time: 0.034 ms
Execution time: 0.144 ms
```

这里我们可以看到索引返回一个候选行，然后它会被索引条件的重新检查拒绝。这是因为一个 GiST 索引对于多边形包含测试是 “有损的”：它确实返回覆盖目标的多边形的行，然后我们必须在那些行上做精确的包含性测试。

EXPLAIN有一个BUFFERS选项可以和ANALYZE一起使用来得到更多运行时统计信息：

```
EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 >
 9000;
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=25.08..60.21 rows=10 width=244) (actual time=0.323..0.342
rows=10 loops=1)
   Recheck Cond: ((unique1 < 100) AND (unique2 > 9000))
   Buffers: shared hit=15
   ->  BitmapAnd  (cost=25.08..25.08 rows=10 width=0) (actual time=0.309..0.309 rows=0 loops=1)
         Buffers: shared hit=7
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
               Buffers: shared hit=2
         ->  Bitmap Index Scan on tenk1_unique2  (cost=0.00..19.78 rows=999 width=0) (actual
time=0.227..0.227 rows=999 loops=1)
               Index Cond: (unique2 > 9000)
               Buffers: shared hit=5
 Planning time: 0.088 ms
 Execution time: 0.423 ms
```

BUFFERS提供的数字帮助我们标识查询的哪些部分是对 I/O 最敏感的。

记住因为EXPLAIN ANALYZE实际运行查询，任何副作用都将照常发生，即使查询可能输出的任何结果被丢弃来支持打印EXPLAIN数据。如果想要分析一个数据修改查询而不想改变表，可以在分析完后回滚命令，例如：

```
BEGIN;

EXPLAIN ANALYZE UPDATE tenk1 SET hundred = hundred + 1 WHERE unique1 < 100;
                                                                 QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------------
 Update on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=14.628..14.628 rows=0
loops=1)
   ->  Bitmap Heap Scan on tenk1  (cost=5.07..229.46 rows=101 width=250) (actual time=0.101..0.439
rows=100 loops=1)
         Recheck Cond: (unique1 < 100)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..5.04 rows=101 width=0) (actual
time=0.043..0.043 rows=100 loops=1)
               Index Cond: (unique1 < 100)
 Planning time: 0.079 ms
```

```
 Execution time: 14.727 ms

ROLLBACK;
```

正如在这个例子中所看到的，当查询是一个INSERT、UPDATE或DELETE命令时，应用表更改的实际工作由顶层插入、更新或删除计划结点完成。这个结点之下的计划结点执行定位旧行以及/或者计算新数据的工作。因此在上面，我们看到我们已经见过的位图表扫描，它的输出被交给一个更新结点，更新结点会存储被更新过的行。还有一点值得注意的是，尽管数据修改结点可能要可观的运行时间（这里，它消耗最大份额的时间），规划器当前并没有对开销估计增加任何东西来说明这些工作。这是因为这些工作对每一个正确的查询计划都得做，所以它不影响计划的选择。

当一个UPDATE或者DELETE命令影响继承层次时， 输出可能像这样：

```
EXPLAIN UPDATE parent SET f2 = f2 + 1 WHERE f1 = 101;
                            QUERY PLAN
-------------------------------------------------------------------------------
 Update on parent  (cost=0.00..24.53 rows=4 width=14)
   Update on parent
   Update on child1
   Update on child2
   Update on child3
   ->  Seq Scan on parent  (cost=0.00..0.00 rows=1 width=14)
         Filter: (f1 = 101)
   ->  Index Scan using child1_f1_key on child1  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child2_f1_key on child2  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
   ->  Index Scan using child3_f1_key on child3  (cost=0.15..8.17 rows=1 width=14)
         Index Cond: (f1 = 101)
```

在这个例子中，更新节点需要考虑三个子表以及最初提到的父表。因此有四个输入 的扫描子计划，每一个对应于一个表。为清楚起见，在更新节点上标注了将被更新 的相关目标表，显示的顺序与相应的子计划相同。

EXPLAIN ANALYZE显示的 Planning time是从一个已解析的查询生成查询计划并进行优化 所花费的时间，其中不包括解析和重写。

EXPLAIN ANALYZE显示的Execution time包括执行器的启动和关闭时间，以及运行被触发的任何触发器的时间，但是它不包括解析、重写或规划的时间。如果有花在执行BEFORE执行器的时间，它将被包括在相关的插入、更新或删除结点的时间内；但是用来执行AFTER 触发器的时间没有被计算，因为AFTER触发器是在整个计划完成后被触发的。在每个触发器（BEFORE或AFTER）也被独立地显示。注意延迟约束触发器将不会被执行，直到事务结束，并且因此根本不会被EXPLAIN ANALYZE考虑。

11.1.3. 警告

在两种有效的方法中EXPLAIN ANALYZE所度量的运行时间可能偏离同一个查询的正常执行。首先，由于不会有输出行被递交给客户端，网络传输开销和 I/O 转换开销没有被包括在内。其次，由EXPLAIN ANALYZE所增加的度量符合可能会很可观，特别是在那些gettimeofday()操作系统调用很慢的机器上。可以使用ux_test_timing工具来度量在系统上的计时开销。

EXPLAIN结果不应该被外推到与实际测试的非常不同的情况。例如，一个很小的表上的结果不能被假定成适合大型表。规划器的开销估计不是线性的，并且因此它可能为一个更大或更小的表选择一个不同的计划。一个极端例子是，在一个只占据一个磁盘页面的表上，将几乎总是得到一个

顺序扫描计划，而不管索引是否可用。规划器认识到它在任何情况下都将采用一次磁盘页面读取来处理该表，因此用额外的页面读取去查看一个索引是没有价值的（我们已经在前面的polygon_tbl例子中见过）。

在一些情况中，实际的值和估计的值不会匹配得很好，但是这并非错误。一种这样的情况发生在计划结点的执行被LIMIT或类似的效果很快停止。例如，在我们之前用过的LIMIT查询中：

```
EXPLAIN ANALYZE SELECT * FROM tenk1 WHERE unique1 < 100 AND unique2 > 9000 LIMIT
 2;

                                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.29..14.71 rows=2 width=244) (actual time=0.177..0.249 rows=2 loops=1)
   ->  Index Scan using tenk1_unique2 on tenk1  (cost=0.29..72.42 rows=10 width=244) (actual
 time=0.174..0.244 rows=2 loops=1)
         Index Cond: (unique2 > 9000)
         Filter: (unique1 < 100)
         Rows Removed by Filter: 287
 Planning time: 0.096 ms
 Execution time: 0.336 ms
```

索引扫描结点的估计开销和行计数被显示成好像它会运行到完成。但是实际上限制结点在得到两个行之后就停止请求行，因此实际的行计数只有 2 并且运行时间远低于开销估计所建议的时间。这并非估计错误，这仅仅一种估计值和实际值显示方式上的不同。

归并连接也有类似的现象。如果一个归并连接用尽了一个输入并且其中的最后一个键值小于另一个输入中的下一个键值，它将停止读取另一个输入。在这种情况下，不会有更多的匹配并且因此不需要扫描第二个输入的剩余部分。这会导致不读取一个子结点的所有内容，其结果就像在LIMIT中所提到的。另外，如果 outer （第一个）子结点包含带有重复键值的行，inner（第二个）子结点会被倒退并且被重新扫描来找能匹配那个键值的行。EXPLAIN ANALYZE会统计相同 inner 行的重复发出，就好像它们是真实的额外行。当有很多 outer 重复时，对 inner 子计划结点所报告的实际行计数会显著地大于实际在 inner 关系中的行数。

由于实现的限制，BitmapAnd 和 BitmapOr 结点总是报告它们的实际行计数为零。

通常，EXPLAIN将显示规划器生成的每个计划节点。 但是，在某些情况下，执行器可以不执行某些节点，因为根据规划时不可用的参数值能确定这些节点无法产生任何行。 （当前，这仅会在扫描分区表的Append或MergeAppend节点的子节点中发生。） 发生这种情况时，将从EXPLAIN输出中省略这些计划节点，并显示Subplans Removed：*N*的标识。

11.2. 规划器使用的统计信息

11.2.1. 单列统计信息

如我们在上一节所见，查询规划器需要估计一个查询要检索的行数，这样才能对查询计划做出好的选择。 本节对系统用于这些估计的统计信息进行一个快速的介绍。

统计信息的一个部分就是每个表和索引中的项的总数，以及每个表和索引占用的磁盘块数。这些信息保存在ux_class表的*reltuples*和*relpages*列中。 我们可以用类似下面的查询查看这些信息：

```
SELECT relname, relkind, reltuples, relpages
FROM ux_class
```

```
WHERE relname LIKE 'tenk1%';

      relname        | relkind | reltuples | relpages
---------------------+---------+-----------+----------
 tenk1               | r       |     10000 |      358
 tenk1_hundred       | i       |     10000 |       30
 tenk1_thous_tenthous | i      |     10000 |       30
 tenk1_unique1       | i       |     10000 |       30
 tenk1_unique2       | i       |     10000 |       30
(5 rows)
```

这里我们可以看到tenk1包含 10000 行， 它的索引也有这么多行，但是索引远比表小得多（不奇怪）。

出于效率考虑，*reltuples*和*relpages*不是实时更新的 ，因此它们通常包含有些过时的值。它们被VACUUM、ANALYZE和几个 DDL 命令（例如CREATE INDEX）更新。一个不扫描全表的VACUUM或ANALYZE操作（常见情况）将以它扫描的部分为基础增量更新*reltuples*计数，这就导致了一个近似值。在任何情况中，规划器将缩放它在ux_class中找到的值来匹配当前的物理表尺寸，这样得到一个较紧的近似。

大多数查询只是检索表中行的一部分，因为它们有限制要被检查的行的WHERE子句。 因此规划器需要估算WHERE子句的选择度，即符合WHERE子句中每个条件的行的比例。 用于这个任务的信息存储在ux_statistic系统目录中。 在ux_statistic中的项由ANALYZE和VACUUM ANALYZE命令更新， 并且总是近似值（即使刚刚更新完）。

除了直接查看ux_statistic之外， 手工检查统计信息的时候最好查看它的视图ux_stats。ux_stats被设计为更容易阅读。 而且，ux_stats是所有人都可以读取的，而ux_statistic只能由超级用户读取（这样可以避免非授权用户从统计信息中获取一些其他人的表的内容的信息。ux_stats视图被限制为只显示当前用户可读的表）。例如，我们可以：

```
SELECT attname, inherited, n_distinct,
    array_to_string(most_common_vals, E'\n') as most_common_vals
FROM ux_stats
WHERE tablename = 'road';

 attname | inherited | n_distinct |         most_common_vals
---------+-----------+------------+----------------------------------
 name    | f         |  -0.363388 | I- 580                  Ramp+
         |           |            | I- 880                Ramp+
         |           |            | Sp Railroad             +
         |           |            | I- 580                +
         |           |            | I- 680                Ramp
 name    | t         |  -0.284859 | I- 880                  Ramp+
         |           |            | I- 580                Ramp+
         |           |            | I- 680                Ramp+
         |           |            | I- 580                +
         |           |            | State Hwy 13            Ramp
(2 rows)
```

注意，这两行显示的是相同的列，一个对应开始于road表（inherited=t）的完全继承层次， 另一个只包括road表本身（inherited=f）。

ANALYZE在ux_statistic中存储的信息量（特别是每个列的*most_common_vals*中的最大项数和*histogram_bounds*数组）可以用ALTER TABLE SET STATISTICS命令为每一列设置， 或者通过设

置配置变量default_statistics_target进行全局设置。 目前的默认限制是 100 个项。提升该限制可能会让规划器做出更准确的估计（特别是对那些有不规则数据分布的列）， 其代价是在ux_statistic中消耗了更多空间，并且需要略微多一些的时间来计算估计数值。 相比之下，比较低的限制可能更适合那些数据分布比较简单的列。

11.2.2. 扩展统计信息

常常可以看到由于查询子句中用到的多个列相互关联而运行着糟糕的执行计划的慢查询。规划器通常会假设多个条件是彼此独立的，这种假设在列值相互关联的情况下是不成立的。由于常规的统计信息天然的针对个体列的性质，它们无法捕捉到跨列关联的知识。不过，UXDB有能力计算多元统计信息，它能捕捉这类信息。

由于可能的列组合数非常巨大，所以不可能自动计算多元统计信息。可以创建扩展统计信息对象（更常被称为统计信息对象）来指示服务器获得跨感兴趣列集合的统计信息。

统计信息对象可以使用CREATE STATISTICS(7)命令创建。这样一个对象的创建仅仅是创建了一个目录项来表示对统计信息有兴趣。实际的数据收集是由ANALYZE（或者是一个手工命令，或者是后台的自动分析）执行的。收集到的值可以在ux_statistic_ext_data目录中看到。

ANALYZE基于它用来计算常规单列统计信息的表行样本来计算扩展统计信息。由于样本的尺寸会随着表或者表列的统计信息目标（如前一节所述）增大而增加，更大的统计信息目标通常将会导致更准确的扩展统计信息，同时也会导致更多花在计算扩展统计信息之上的时间。

下面的小节介绍当前支持的扩展统计信息类型。

11.2.2.1. 函数依赖

最简单的一类扩展统计信息跟踪函数依赖，这是在数据库范式定义中使用的概念。如果列*a*的值的知识足以决定列*b*的值，即不会有两个行具有相同的*a*值但是有不同的*b*值，我们就说列*b*函数依赖于列*a*。在一个完全规范化的数据库中，函数依赖应该仅存在于主键和超键上。不过，实际上很多数据集合会由于各种原因无法被完全规范化，常见的例子是为了性能而有意地反规范化。即使在一个完全规范化的数据库中，也会有某些列之间的部分关联，这些可以表达成部分函数依赖。

函数依赖的存在直接影响了特定查询中估计的准确性。如果一个查询包含独立列和依赖列上的条件，依赖列上的条件不会进一步降低结果的尺寸。但是如果没有函数依赖的知识，查询规划器将假定条件是独立的，导致对结果尺寸的低估。

要告知规划器有关函数依赖的信息，ANALYZE可以收集跨列依赖的测度。评估所有列组之间的依赖程度可能会昂贵到不可实现，因此数据收集被限制为针对那些在一个统计信息对象中一起出现的列组（用dependencies选项定义）。建议只对强相关的列组创建dependencies统计信息，以避免ANALYZE以及后期查询规划中不必要的开销。

这里是一个收集函数依赖统计信息的例子：

```
CREATE STATISTICS stts (dependencies) ON city, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxname, stxkeys, stxddependencies
  FROM ux_statistic_ext join ux_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts';
 stxname | stxkeys |             stxddependencies
---------+---------+-----------------------------------------
```

```
 stts   | 1 5    | {"1 => 5": 1.000000, "5 => 1": 0.423130}
(1 row)
```

这里可以看到列1（邮编）完全决定列5（城市），因此系数为1.0，而城市仅决定42%的邮编，意味着有很多城市（58%）有多个邮编。

在为涉及函数依赖列的查询计算选择度时，规划器会使用依赖系数来调整针对条件的选择度估计，这样就不会产生低估。

11.2.2.1.1. 函数依赖的限制

当前只有在考虑简单等值条件（将列与常量值比较）时，函数依赖才适用。不会使用它们来改进比较两个列或者比较列和表达式的等值条件的估计，也不会用它们来改进范围子句、LIKE或者任何其他类型的条件。

在用函数依赖估计时，规划器假定在涉及的列上的条件是兼容的并且因此是冗余的。如果它们是不兼容的，正确的估计将是零行，但是那种可能性不会被考虑。例如，给定一个这样的查询

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '94105';
```

规划器将会忽视*city*子句，因为它不改变选择度，这是正确的。不过，即便真地只有零行满足下面的查询，规划器也会做出同样的假设

```
SELECT * FROM zipcodes WHERE city = 'San Francisco' AND zip = '90210';
```

不过，函数依赖统计信息无法提供足够的信息来排除这种情况。

在很多实际情况中，这种假设通常是能满足的。例如，在应用程序中可能有一个GUI仅允许选择兼容的城市和邮编值用在查询中。但是如果不是这样，函数依赖可能就不是一个可行的选项。

11.2.2.2. 多元可区分值计数

单列统计信息存储每一列中可区分值的数量。在组合多个列（例如GROUP BY a, b）时，如果规划器只有单列统计数据，则对可区分值数量的估计常常会错误，导致选择不好的计划。

为了改进这种估计，`ANALYZE`可以为列组收集可区分值统计信息。和以前一样，为每一种可能的列组合做这件事情是不切实际的，因此只会为一起出现在一个统计信息对象（用ndistinct选项定义）中的列组收集数据。将会为列组中列出的列的每一种可能的组合都收集数据。

继续之前的例子，ZIP代码表中的可区分值计数可能像这样：

```
CREATE STATISTICS stts2 (ndistinct) ON city, state, zip FROM zipcodes;

ANALYZE zipcodes;

SELECT stxkeys AS k, stxdndistinct AS nd
  FROM ux_statistic_ext join ux_statistic_ext_data on (oid = stxoid)
  WHERE stxname = 'stts2';
-[ RECORD 1 ]----------------------------------------------------
k  | 1 2 5
nd | {"1, 2": 33178, "1, 5": 33178, "2, 5": 27435, "1, 2, 5": 33178}
(1 row)
```

这表示有三种列组合有33178个可区分值：ZIP代码和州、ZIP代码和城市、ZIP代码+城市+周（事实上对于表中给定的一个唯一的ZIP代码，它们本来就应该是相等的）。另一方面，城市和州的组合只有27435个可区分值。

建议只对实际用于分组的列组合以及分组数错误估计导致了糟糕计划的列组合创建ndistinct统计信息对象。否则，**ANALYZE**循环只会被浪费。

11.2.2.3. 多元MCV列表

为每列存储的另一种统计信息是频繁值列表。 这样可以对单个列进行非常准确的估计，但是对于在多个列上具有条件的查询，可能会导致严重的错误估计。

为了改善这种估计，**ANALYZE**可以收集列组合上的MCV列表。 与功能依赖和n-distinct系数类似，对每种可能的列分组进行此操作都是不切实际的。 在这种情况下，甚至更是如此，因为MCV列表（与功能依赖性和n-distinct系数不同）存储了公共列值。 因此，仅收集在使用mcv选项定义的统计对象中同时出现的那些列组的数据。

继续前面的示例，邮政编码表的MCV列表可能类似于以下内容（与更简单的统计信息不同，它需要一个函数来检查MCV内容）：

```
CREATE STATISTICS stts3 (mcv) ON city, state FROM zipcodes;

ANALYZE zipcodes;

SELECT m.* FROM ux_statistic_ext join ux_statistic_ext_data on (oid = stxoid),
            ux_mcv_list_items(stxdmcv) m WHERE stxname = 'stts3';

 index |        values         | nulls | frequency | base_frequency
-------+-----------------------+-------+-----------+----------------
     0 | {Washington, DC}      | {f,f} |  0.003467 |        2.7e-05
     1 | {Apo, AE}             | {f,f} |  0.003067 |        1.9e-05
     2 | {Houston, TX}         | {f,f} |  0.002167 |       0.000133
     3 | {El Paso, TX}         | {f,f} |     0.002 |       0.000113
     4 | {New York, NY}        | {f,f} |  0.001967 |       0.000114
     5 | {Atlanta, GA}         | {f,f} |  0.001633 |        3.3e-05
     6 | {Sacramento, CA}      | {f,f} |  0.001433 |        7.8e-05
     7 | {Miami, FL}           | {f,f} |    0.0014 |         6e-05
     8 | {Dallas, TX}          | {f,f} |  0.001367 |        8.8e-05
     9 | {Chicago, IL}         | {f,f} |  0.001333 |        5.1e-05
   ...
(99 rows)
```

这表明城市和州的最常见组合是华盛顿特区，实际频率（在样本中）约为0.35%。 组合的基本频率（根据简单的每列频率计算）仅为0.0027%，导致两个数量级的低估。

建议仅在实际在条件中一起使用的列的组合上创建MCV统计对象，对于这些组合，错误估计组数会导致糟糕的执行计划。 否则，只会浪费**ANALYZE**和规划时间。

11.3. 用显式JOIN子句控制规划器

我们可以在一定程度上用显式JOIN语法控制查询规划器。要明白为什么需要它，我们首先需要一些背景知识。

在一个简单的连接查询中，例如：

SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;

规划器可以自由地按照任何顺序连接给定的表。例如，它可以生成一个使用WHERE条件a.id = b.id连接 A 到 B 的查询计划，然后用另外一个WHERE条件把 C 连接到这个连接表。或者它可以先连接 B 和 C 然后再连接 A 得到同样的结果。 或者也可以连接 A 到 C 然后把结果与 B 连接 — 不过这么做效率不好，因为必须生成完整的 A 和 C 的迪卡尔积，而在WHERE子句中没有可用条件来优化该连接（UXDB执行器中的所有连接都发生在两个输入表之间， 所以它必须以这些形式之一建立结果）。 重要的一点是这些不同的连接可能性给出在语义等效的结果，但在执行开销上却可能有巨大的差别。 因此，规划器会对它们进行探索并尝试找出最高效的查询计划。

当一个查询只涉及两个或三个表时，那么不需要考虑很多连接顺序。但是可能的连接顺序数随着表数目的增加成指数增长。 当超过十个左右的表以后，实际上根本不可能对所有可能性做一次穷举搜索，甚至对六七个表都需要相当长的时间进行规划。 当有太多的输入表时，UXDB规划器将从穷举搜索切换为一种遗传概率搜索，它只需要考虑有限数量的可能性（切换的阈值用geqo_threshold运行时参数设置）。遗传搜索用时更少，但是并不一定会找到最好的计划。

当查询涉及外连接时，规划器比处理普通（内）连接时拥有更小的自由度。例如，考虑：

SELECT * FROM a LEFT JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);

尽管这个查询的约束表面上和前一个非常相似，但它们的语义却不同， 因为如果 A 里有任何一行不能匹配 B 和 C的连接表中的行，它也必须被输出。因此这里规划器对连接顺序没有什么选择：它必须先连接 B 到 C，然后把 A 连接到该结果上。 相应地，这个查询比前面一个花在规划上的时间更少。在其它情况下，规划器就有可能确定多种连接顺序都是安全的。例如，给定：

SELECT * FROM a LEFT JOIN b ON (a.bid = b.id) LEFT JOIN c ON (a.cid = c.id);

将 A 首先连接到 B 或 C 都是有效的。当前，只有FULL JOIN完全约束连接顺序。大多数涉及LEFT JOIN或RIGHT JOIN的实际情况都在某种程度上可以被重新排列。

显式连接语法（INNER JOIN、CROSS JOIN或无修饰的JOIN）在语义上和FROM中列出输入关系是一样的， 因此它不约束连接顺序。

即使大多数类型的JOIN并不完全约束连接顺序，但仍然可以指示UXDB查询规划器将所有JOIN子句当作有连接顺序约束来对待。例如，这里的三个查询在逻辑上是等效的：

SELECT * FROM a, b, c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a CROSS JOIN b CROSS JOIN c WHERE a.id = b.id AND b.ref = c.id;
SELECT * FROM a JOIN (b JOIN c ON (b.ref = c.id)) ON (a.id = b.id);

但如果我们告诉规划器遵循JOIN的顺序，那么第二个和第三个还是要比第一个花在规划上的时间少。 这个效果对于只有三个表的连接而言是微不足道的，但对于数目众多的表，可能就是救命稻草了。

要强制规划器遵循显式JOIN的连接顺序， 我们可以把运行时参数guc_join_collapse_limit设置为 1（其它可能值在下文讨论）。

不必为了缩短搜索时间来完全约束连接顺序，因为可以在一个普通FROM列表里使用JOIN操作符。例如，考虑：

SELECT * FROM a CROSS JOIN b, c, d, e WHERE ...;

如果设置join_collapse_limit = 1，那么这就强迫规划器先把 A 连接到 B， 然后再连接到其它的表上，但并不约束它的选择。在这个例子中，可能的连接顺序的数目减少了 5 倍。

按照这种方法约束规划器的搜索是一个有用的技巧，不管是对减少规划时间还是对引导规划器生成好的查询计划。 如果规划器按照默认选择了一个糟糕的连接顺序，可以通过JOIN语法强迫它选择一个更好的顺序 — 假设知道一个更好的顺序。我们推荐进行实验。

一个非常相近的影响规划时间的问题是把子查询压缩到它们的父查询中。例如，考虑：

```
SELECT *
FROM x, y,
   (SELECT * FROM a, b, c WHERE something) AS ss
WHERE somethingelse;
```

这种情况可能在使用包含连接的视图时出现；该视图的SELECT规则将被插入到引用视图的地方，得到与上文非常相似的查询。 通常，规划器会尝试把子查询压缩到父查询里，得到：

```
SELECT * FROM x, y, a, b, c WHERE something AND somethingelse;
```

这样通常会生成一个比独立的子查询更好些的计划（例如，outer 的WHERE条件可能先把 X 连接到 A 上，这样就消除了 A 中的许多行， 因此避免了形成子查询的全部逻辑输出）。但是同时，我们增加了规划的时间； 在这里，我们用五路连接问题替代了两个独立的三路连接问题。这样的差别是巨大的，因为可能的计划数的是按照指数增长的。 如果有超过from_collapse_limit个FROM项将会导致父查询，规划器将尝试通过停止提升子查询来避免卡在巨大的连接搜索问题中。可以通过调高或调低这个运行时参数在规划时间和计划的质量之间取得平衡。

from_collapse_limit和guc_join_collapse_limit的命名相似，因为它们做的几乎是同一件事：一个控制规划器何时将把子查询“平面化”，另外一个控制何时把显式连接平面化。通常，要么把join_collapse_limit设置成和from_collapse_limit一样（这样显式连接和子查询的行为类似），要么把join_collapse_limit设置为 1（如果想用显式连接控制连接顺序）。 但是可以把它们设置成不同的值，这样就可以细粒度地调节规划时间和运行时间之间的平衡。

11.4. 填充一个数据库

第一次填充数据库时可能需要插入大量的数据。本节包含一些如何让这个处理尽可能高效的建议。

11.4.1. 禁用自动提交

在使用多个INSERT时，关闭自动提交并且只在最后做一次提交（在普通 SQL 中，这意味着在开始发出BEGIN并且在结束时发出COMMIT。某些客户端库可能背着就做了这些，在这种情况下需要确定在需要做这些时该库确实帮做了）。如果允许每一个插入都被独立地提交，UXDB要为每一个被增加的行做很多工作。在一个事务中做所有插入的一个额外好处是：如果一个行的插入失败则所有之前插入的行都会被回滚，这样不会被卡在部分载入的数据中。

11.4.2. 使用COPY

使用COPY(7)在一条命令中装载所有记录，而不是一系列INSERT命令。 COPY命令是为装载大量行而优化过的； 它没INSERT那么灵活，但是在大量数据装载时导致的负荷也更少。 因为COPY是单条命令，因此使用这种方法填充表时无须关闭自动提交。

如果不能使用COPY，那么使用PREPARE(7)来创建一个预备INSERT语句也有所帮助，然后根据需要使用EXECUTE多次。这样就避免了重复分析和规划INSERT的负荷。不同接口以不同的方式提供该功能， 可参阅接口文档中的“预备语句”。

请注意，在载入大量行时，使用COPY几乎总是比使用INSERT快， 即使使用了PREPARE并且把多个插入被成批地放入一个单一事务。

同样的事务中，COPY比更早的CREATE TABLE或TRUNCATE命令更快。 在这种情况下，不需要写 WAL，因为在一个错误的情况下，包含新载入数据的文件不管怎样都将被移除。不过，只有当wal_level设置为minimal（此时所有的命令必须写 WAL）时才会应用这种考虑。

11.4.3. 移除索引

如果正在载入一个新创建的表，最快的方法是创建该表，用COPY批量载入该表的数据，然后创建表需要的任何索引。在已存在数据的表上创建索引要比在每一行被载入时增量地更新它更快。

如果正在对现有表增加大量的数据，删除索引、载入表然后重新创建索引可能是最好的方案。当然，在缺少索引的期间，其它数据库用户的数据库性能将会下降。 我们在删除唯一索引之前还需要仔细考虑清楚，因为唯一约束提供的错误检查在缺少索引的时候会丢失。

11.4.4. 移除外键约束

和索引一样，“成批地”检查外键约束比一行行检查效率更高。 因此，先删除外键约束、载入数据然后重建约束会很有用。 同样，载入数据和约束缺失期间错误检查的丢失之间也存在平衡。

更重要的是，当在已有外键约束的情况下向表中载入数据时， 每个新行需要一个在服务器的待处理触发器事件（因为是一个触发器的触发会检查行的外键约束）列表的条目。载入数百万行会导致触发器事件队列溢出可用内存， 造成不能接受的交换或者甚至是命令的彻底失败。因此在载入大量数据时，可能需要（而不仅仅是期望）删除并重新应用外键。如果临时移除约束不可接受，那唯一的其他办法可能是就是将载入操作分解成更小的事务。

11.4.5. 增加maintenance_work_mem

在载入大量数据时，临时增大maintenance_work_mem配置变量可以改进性能。这个参数也可以帮助加速CREATE INDEX命令和ALTER TABLE ADD FOREIGN KEY命令。 它不会对COPY本身起很大作用，所以这个建议只有在使用上面的一个或两个技巧时才有用。

11.4.6. 增加max_wal_size

临时增大max_wal_size配置变量也可以让大量数据载入更快。 这是因为向UXDB中载入大量的数据将导致检查点的发生比平常（由checkpoint_timeout配置变量指定）更频繁。无论何时发生一个检查点时，所有脏页都必须被刷写到磁盘上。 通过在批量数据载入时临时增加max_wal_size，所需的检查点数目可以被缩减。

11.4.7. 禁用 WAL 归档和流复制

当使用 WAL 归档或流复制向一个安装中载入大量数据时，在录入结束后执行一次新的基础备份比处理大量的增量 WAL 数据更快。为了防止载入时记录增量 WAL，通过将wal_level设置为minimal、将archive_mode设置为off以及将max_wal_senders设置为零来禁用归档和流复制。 但需要注意的是，修改这些设置需要重启服务。

除了避免归档器或 WAL 发送者处理 WAL 数据的时间之外，这样做将实际上使某些命令更快，因为它们被设计为在wal_level为minimal时完全不写 WAL （通过在最后执行一个fsync而不是写WAL，它们能以更小地代价保证崩溃安全）。这适用于下列命令：

- CREATE TABLE AS SELECT
- CREATE INDEX（以及类似 ALTER TABLE ADD PRIMARY KEY的变体）
- ALTER TABLE SET TABLESPACE
- CLUSTER
- COPY FROM，当目标表已经被创建或者在同一个事务的早期被截断

11.4.8. 事后运行ANALYZE

不管什么时候显著地改变了表中的数据分布后，我们都强烈推荐运行ANALYZE(7)。着包括向表中批量载入大量数据。运行ANALYZE（或者VACUUM ANALYZE）保证规划器有表的最新统计信息。如果没有统计数据或者统计数据过时，那么规划器在查询规划时可能做出很差劲决定，导致在任意表上的性能低下。需要注意的是，如果启用了 autovacuum 守护进程，它可能会自动运行ANALYZE。

11.4.9. 关于ux_dump的一些注记

ux_dump生成的转储脚本自动应用上面的若干个（但不是全部）技巧。 要尽可能快地载入ux_dump转储，需要手工做一些额外的事情（请注意，这些要点适用于恢复一个转储，而不是创建它的时候。同样的要点也适用于使用uxsql载入一个文本转储或用ux_restore从一个ux_dump归档文件载入）。

默认情况下，ux_dump使用COPY，并且当它在生成一个完整的模式和数据转储时， 它会先装载数据，然后创建索引和外键。因此在这种情况下，一些指导方针是被自动处理的。需要做的是：

- 为maintenance_work_mem和max_wal_size设置适当的（即比正常值大的）值。
- 如果使用 WAL 归档或流复制，在转储时考虑禁用它们。在载入转储之前，可通过将archive_mode设置为off、将wal_level设置为minimal以及将max_wal_senders设置为零（在录入dump前）来实现禁用。 之后，将它们设回正确的值并执行一次新的基础备份。
- 采用ux_dump和ux_restore的并行转储和恢复模式进行实验并且找出要使用的最佳并发任务数量。通过使用-j选项的并行转储和恢复应该能为带来比串行模式高得多的性能。
- 考虑是否应该在一个单一事务中恢复整个转储。要这样做，将-1或--single-transaction命令行选项传递给uxsql或ux_restore。 当使用这种模式时，即使是一个很小的错误也会回滚整个恢复，可能会丢弃已经处理了很多个小时的工作。根据数据间的相关性， 可能手动清理更好。如果使用一个单一事务并且关闭了 WAL 归档，COPY命令将运行得最快。
- 如果在数据库服务器上有多个 CPU 可用，可以考虑使用ux_restore的--jobs选项。这允许并行数据载入和索引创建。
- 之后运行ANALYZE。

一个只涉及数据的转储仍将使用COPY，但是它不会删除或重建索引，并且它通常不会触碰外键。[1] 因此当载入一个只有数据的转储时，如果希望使用那些技术，需要负责删除并重建索引和外

[1] 可以通过使用--disable-triggers选项的方法获得禁用外键的效果 — 不过要意识到这么做是消除（而不只是推迟）外键验证。因此如果使用该选项，就可能插入坏数据。

键。在载入数据时增加max_wal_size仍然有用，但是不要去增加maintenance_work_mem；不如说在以后手工重建索引和外键时已经做了这些。并且不要忘记在完成后执行ANALYZE。

11.5. 非持久设置

持久性是数据库的一个保证已提交事务的记录的特性（即使是发生服务器崩溃或断电）。 然而，持久性会明显增加数据库的负荷，因此如果站点不需要这个保证，UXDB可以被配置成运行更快。在这种情况下，可以调整下列配置来提高性能。除了下面列出的，在数据库软件崩溃的情况下也能保证持久性。当这些设置被使用时，只有突然的操作系统停止会产生数据丢失或损坏的风险。

- 将数据库集簇的数据目录放在一个内存支持的文件系统上（即RAM磁盘）。这消除了所有的数据库磁盘 I/O，但将数据存储限制到可用的内存量（可能有交换区）。
- 关闭fsync；不需要将数据刷入磁盘。
- 关闭synchronous_commit；可能不需要在每次提交时 强制把WAL写入磁盘。这种设置可能会在数据库崩溃时带来事务丢失的风险（但是没有数据破坏）。
- 关闭full_page_writes；不需要警惕部分页面写入。
- 增加max_wal_size和checkpoint_timeout； 这会降低检查点的频率，但会 增加/ux_wal的存储要求。
- 创建不做日志的表 来避免WAL写入，不过这会让表在崩溃时不安全。

第 12 章 并行查询

UXDB能设计出利用多 CPU 让查询更快的查询计划。这种特性被称为并行查询。由于现有实现的限制或者因为没有比连续查询计划更快的查询计划存在，很多查询并不能从并行查询获益。不过，对于那些可以从并行查询获益的查询来说，并行查询带来的速度提升是显著的。很多查询在使用并行查询时比之前快了超过两倍，有些查询是以前的四倍甚至更多的倍数。那些访问大量数据但只返回其中少数行给用户的查询最能从并行查询中获益。这一章介绍一些并行查询如何工作的细节以及哪些情况下可以使用并行查询，这样希望充分利用并行查询的用户可以理解他们能从并行查询得到什么。

12.1. 并行查询如何工作

当优化器判断对于某一个特定的查询，并行查询是最快的执行策略时，优化器将创建一个查询计划。该计划包括一个 Gather或者Gather Merge节点。下面是一个简单的例子：

```
EXPLAIN SELECT * FROM uxbench_accounts WHERE filler LIKE '%x%';
                                QUERY PLAN
------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..217018.43 rows=1 width=97)
   Workers Planned: 2
   ->  Parallel Seq Scan on uxbench_accounts  (cost=0.00..216018.33 rows=1 width=97)
         Filter: (filler ~~ '%x%'::text)
(4 rows)
```

在所有的情形下，Gather或Gather Merge节点都只有一个子计划，它是将被并行执行的计划的一部分。如果Gather或Gather Merge节点位于计划树的最顶层，那么整个查询将并行执行。如果它位于计划树的其他位置，那么只有查询中在它之下的那一部分会并行执行。在上面的例子中，查询只访问了一个表，因此除Gather节点本身之外只有一个计划节点。因为该计划节点是Gather节点的孩子节点，所以它会并行执行。

使用 EXPLAIN命令， 能看到规划器选择的工作者数量。当查询执行期间到达Gather节点时，实现用户会话的进程将会请求和规划器选中的工作者数量一样多的后台工作者进程 。规划器将考虑使用的后台工作者的数量被限制为最多 max_parallel_workers_per_gather个。任何时候能够存在的后台工作者进程的总数由max_worker_processes和max_parallel_workers限制。因此，一个并行查询可能会使用比规划中少的工作者来运行，甚至有可能根本不使用工作者。最优的计划可能取决于可用的工作者的数量，因此这可能会导致不好的查询性能。如果这种情况经常发生，那么就应当考虑一下提高max_worker_processes和max_parallel_workers的值，这样更多的工作者可以同时运行；或者降低max_parallel_workers_per_gather，这样规划器会要求少一些的工作者。

为一个给定并行查询成功启动的后台工作者进程都将会执行计划的并行部分。这些工作者的领导者也将执行该计划，不过它还有一个额外的任务：它还必须读取所有由工作者产生的元组。当整个计划的并行部分只产生了少量元组时，领导者通常将表现为一个额外的加速查询执行的工作者。反过来，当计划的并行部分产生大量的元组时，领导者将几乎全用来读取由工作者产生的元组并且执行Gather或Gather Merge节点上层计划节点所要求的任何进一步处理。在这些情况下，领导者所作的执行并行部分的工作将会很少。

当计划的并行部分的顶层节点是Gather Merge而不是Gather时，它表示每个执行计划并行部分的进程会产生有序的元组，并且领导者执行一种保持顺序的合并。相反，Gather会以任何方便的顺序从工作者读取元组，这会破坏可能已经存在的排序顺序。

12.2. 何时会用到并行查询？

有几种设置会导致查询规划器在任何情况下都不生成并行查询计划。为了让并行查询计划能够被生成，必须配置好下列设置。

- max_parallel_workers_per_gather必须被设置为大于零的值。这是一种特殊情况，更加普遍的原则是所用的工作者数量不能超过max_parallel_workers_per_gather所配置的数量。

此外，系统一定不能运行在单用户模式下。因为在单用户模式下，整个数据库系统运行在单个进程中，没有后台工作者进程可用。

如果下面的任一条件为真，即便对一个给定查询通常可以产生并行查询计划，规划器都不会为它产生并行查询计划：

- 查询要写任何数据或者锁定任何数据库行。如果一个查询在顶层或者 CTE 中包含了数据修改操作，那么不会为该查询产生并行计划。一种例外是，CREATE TABLE ... AS、SELECT INTO以及CREATE MATERIALIZED VIEW这些创建新表并填充它的命令可以使用并行计划。
- 查询可能在执行过程中被暂停。只要在系统认为可能发生部分或者增量式执行，就不会产生并行计划。例如：用DECLARE CURSOR创建的游标将永远不会使用并行计划。类似地，一个FOR x IN query LOOP .. END LOOP形式的 PL/uxSQL 循环也永远不会使用并行计划，因为当并行查询进行时，并行查询系统无法验证循环中的代码执行起来是安全的。
- 使用了任何被标记为PARALLEL UNSAFE的函数的查询。大多数系统定义的函数都被标记为PARALLEL SAFE，但是用户定义的函数默认被标记为PARALLEL UNSAFE。参见第 12.4 节 “并行安全性”中的讨论。
- 该查询运行在另一个已经存在的并行查询内部。例如，如果一个被并行查询调用的函数自己发出一个 SQL 查询，那么该查询将不会使用并行计划。这是当前实现的一个限制，但是或许不值得移除这个限制，因为它会导致单个查询使用大量的进程。

即使对于一个特定的查询已经产生了并行查询计划，在一些情况下执行时也不会并行执行该计划。如果发生这种情况，那么领导者将会自己执行该计划在Gather节点之下的部分，就好像Gather节点不存在一样。上述情况将在满足下面的任一条件时发生：

- 因为后台工作者进程的总数不能超过max_worker_processes，导致不能得到后台工作者进程。
- 由于为并行查询目的启动的后台工作者数量不能超过max_parallel_workers这一限制而不能得到后台工作者。
- 客户端发送了一个执行消息，并且消息中要求取元组的数量不为零。执行消息可见扩展查询协议中的讨论。因为libuxsql当前没有提供方法来发送这种消息，所以这种情况只可能发生在不依赖 libuxsql 的客户端中。如果这种情况经常发生，那在它可能发生的会话中设置 max_parallel_workers_per_gather为零是一个很好的主意，这样可以避免产生连续运行时次优的查询计划。

12.3. 并行计划

因为每个工作者只执行完成计划的并行部分，所以不可能简单地产生一个普通查询计划并使用多个工作者运行它。每个工作者都会产生输出结果集的一个完全拷贝，因而查询并不会比普通查询运行得更快甚至还会产生不正确的结果。相反，计划的并行部分一定被查询优化器在内部当作一个部分计划，即它必须被构建出来，这样每一个执行该计划的进程将以无重复地方式产生输出行的一个子集，即保证每一个所需要的输出行正好只被一个合作进程生成。通常，这意味着该查询的驱动表上的扫描必须是一种可并行的扫描。

12.3.1. 并行扫描

当前支持下列可并行的表扫描。

- 在一个*并行顺序扫描*中，表块将在合作进程之间被划分。一次会分发一个块，这样对表的访问还是保持顺序方式。
- 在一个*并行位图堆扫描*中，一个进程被选为领导者。这个进程执行对一个或者多个索引的扫描并且构建出一个位图指示需要访问哪些表块。这些表块接着会在合作进程之间划分（和并行顺序扫描中一样）。换句话说，堆扫描以并行方式进行但底层的索引扫描不是并行。
- 在一个*并行索引扫描*或者*并行只用索引的扫描*中，合作进程轮流从索引读取数据。当前，并行索引扫描仅有B-树索引支持。每一个进程将认领一个索引块并且扫描和返回该索引块引用的所有元组，其他进程可以同时地从一个不同的索引块返回元组。并行B-树扫描的结果会以每个工作者进程内的顺序返回。

其他扫描类型（例如非B-树索引的扫描）可能会在未来支持并行扫描。

12.3.2. 并行连接

正如在非并行计划中那样，驱动表可能被使用嵌套循环、哈希连接或者归并连接连接到一个或者多个其他表。连接的内侧可以是任何类型的被规划器支持的非并行计划，假设它能够安全地在并行工作者中运行。根据连接类型，内侧还可以是一种并行计划。

- 在一个*嵌套循环连接*中，内侧总是非并行的。尽管它会被完全执行，如果内侧是一个索引扫描也会很高效，因为外侧元组以及在索引中查找值的循环会被划分到多个合作进程。
- 在一个*归并连接*中，内侧总是一个非并行计划并且因此会被完全执行。这可能是不太高效的，特别是在排序必须被执行时，因为在每一个合作进程中工作数据和结果数据是重复的。
- 在一个*哈希连接*（没有“并行”前缀）中，每个合作进程都会完全执行内侧以构建哈希表的相同拷贝。如果哈希表很大或者该计划开销很大，这种方式就很低效。在一个*并行哈希连接*中，内侧是一个*并行哈希*，它把构建共享哈希表的工作划分到多个合作进程。

12.3.3. 并行聚集

UXDB通过按两个阶段进行聚集来支持并行聚集。首先，每个参与到查询并行部分的进程执行一个聚集步骤，为该进程注意到的每个分组产生一个部分结果。这在计划中反映为一个Partial Aggregate节点。然后，部分结果通过Gather或者Gather Merge被传输到领导者。最后，领导者对来自所有工作者的结果进行重新聚集得到最终的结果。这在计划中反映为一个Finalize Aggregate节点。

因为Finalize Aggregate节点运行在领导者进程上，如果查询产生的分组数相对于其输入行数来说比较大，则查询规划器不会喜欢它。例如，在最坏的情况下，Finalize Aggregate节点看到的分组数可能与所有工作者进程在Partial Aggregate阶段看到的输入行数一样多。对于这类情况，使用并行聚集显然得不到性能收益。查询规划器会在规划过程中考虑这一点并且不太会在这种情况下选择并行聚集。

并行聚集并非在所有情况下都被支持。每一个聚集都必须是对并行安全的并且必须有一个组合函数。如果该聚集有一个类型为internal的转移状态，它必须有序列化和反序列化函数。更多细节请参考CREATE AGGREGATE(7)。如果任何聚集函数调用包含DISTINCT或ORDER BY子句，则不支持并行聚集。对于有序集聚集或者当查询涉及GROUPING SETS时，也不支持并行聚集。只有在查询中涉及的所有连接也是该计划并行部分的组成部分时，才能使用并行聚集。

12.3.4. 并行Append

只要当UXDB需要从多个源中整合行到一个单一结果集时，它会使用Append或MergeAppend计划节点。在实现UNION ALL或扫描分区表时常常会发生这种情况。就像这些节点可以被用在任何其他计划中一样，它们可以被用在并行计划中。不过，在并行计划中，规划器使用的是Parallel Append节点。

当一个Append节点被用在并行计划中时，每个进程将按照子计划出现的顺序执行子计划，这样所有的参与进程会合作执行第一个子计划直到它被完成，然后同时移动到第二个计划。而在使用Parallel Append时，执行器将把它的子计划尽可能均匀地散布在参与进程中，这样多个子计划会被同时执行。这避免了竞争，也避免了子计划在那些不执行它的进程中产生启动代价。

此外，和常规的Append节点不同（在并行计划中使用时仅有部分子计划），Parallel Append节点既可以有部分子计划也可以有非部分子计划。非部分子计划将仅被单个进程扫描，因为扫描它们不止一次会产生重复的结果。因此涉及到追加多个结果集的计划即使在没有有效的部分计划可用时，也能实现粗粒度的并行。例如，考虑一个针对分区表的查询，它只能通过使用一个不支持并行扫描的索引来实现。规划器可能会选择常规Index Scan计划的Parallel Append。每个索引扫描必须被单一的进程执行完，但不同的扫描可以由不同的进程同时执行。

enable_parallel_append可以被用来禁用这种特性。

12.3.5. 并行计划小贴士

如果我们想要一个查询能产生并行计划但事实上又没有产生，可以尝试减小parallel_setup_cost或者parallel_tuple_cost。当然，这个计划可能比规划器优先产生的顺序计划还要慢，但也不总是如此。如果将这些设置为很小的值（例如把它们设置为零）也不能得到并行计划，那就可能是有某种原因导致查询规划器无法为查询产生并行计划。可能的原因可见第 12.2 节 “何时会用到并行查询？”和第 12.4 节 “并行安全性”。

在执行一个并行计划时，可以用EXPLAIN (ANALYZE,VERBOSE)来显示每个计划节点在每个工作者上的统计信息。这些信息有助于确定是否所有的工作被均匀地分发到所有计划节点以及从总体上理解计划的性能特点。

12.4. 并行安全性

规划器把查询中涉及的操作分类成*并行安全*、*并行受限*或者*并行不安全*。并行安全的操作不会与并行查询的使用产生冲突。并行受限的操作不能在并行工作者中执行，但是能够在并行查询的领导者中执行。因此，并行受限的操作不能出现在Gather或者Gather Merge节点之下，但是能够出现在包含这类节点的计划的其他位置。并行不安全的操作不能在并行查询中执行，甚至不能在领导者中执行。当一个查询包含任何并行不安全操作时，并行查询对这个查询是完全被禁用的。

下面的操作总是并行受限的。

- 公共表表达式（CTE）的扫描。
- 临时表的扫描。
- 外部表的扫描，除非外部数据包装器有一个IsForeignScanParallelSafe API。
- InitPlan所挂接到的计划节点。
- 引用一个相关的SubPlan的计划节点。

12.4.1. 为函数和聚集加并行标签

规划器无法自动判定一个用户定义的函数或者聚集是并行安全、并行受限还是并行不安全，因为这需要预测函数可能执行的每一个操作。一般而言，这就相当于一个停机问题，因此是不可能的。甚至对于可以做到判定的简单函数我们也不会尝试，因为那会非常昂贵而且容易出错。相反，除非是被标记出来，所有用户定义的函数都被认为是并行不安全的。在使用CREATE FUNCTION(7)或者ALTER FUNCTION(7)时，可以通过指定PARALLEL SAFE、PARALLEL RESTRICTED或者PARALLEL UNSAFE来设置标记。在使用CREATE AGGREGATE(7)时，PARALLEL选项可以被指定为SAFE、RESTRICTED或者 UNSAFE。

如果函数和聚集会写数据库、访问序列、改变事务状态（即便是临时改变，例如建立一个EXCEPTION块来捕捉错误的 PL/uxsql）或者对设置做持久化的更改，它们一定要被标记为PARALLEL UNSAFE。类似地，如果函数会访问临时表、客户端连接状态、游标、预备语句或者系统无法在工作者之间同步的后端本地状态，它们必须被标记为PARALLEL RESTRICTED。例如，setseed和 random由于后一种原因而是并行受限的。

一般而言，如果一个函数是受限或者不安全的却被标记为安全，或者它实际是不安全的却被标记为受限，把它用在并行查询中时可能会抛出错误或者产生错误的回答。如果 C 语言函数被错误标记，理论上它会展现出完全不明确的行为，因为系统中无法保护自身不受任意 C 代码的影响。但是，在最有可能的情况下，结果不会比其他任何函数更糟糕。如果有疑虑，最好还是标记函数为UNSAFE。

如果在并行工作者中执行的函数要求领导者没有持有的锁，例如读该查询中没有引用的表，那么工作者退出时会释放那些锁（而不是在事务结束时释放）。如果写了一个这样做的函数并且这种不同的行为对很重要，把这类函数标记为PARALLEL RESTRICTED以确保它们只在领导者中执行。

注意查询规划器不会为了获取一个更好的计划而考虑延迟计算并行受限的函数或者聚集。所以，如果一个被应用到特定表的WHERE子句是并行受限的，查询规划器就不会考虑对处于计划并行部分的表执行一次扫描。在一些情况中，可以（甚至效率更高）把对表的扫描包括在查询的并行部分并且延迟对WHERE子句的计算，这样它会出现在Gather节点之上。不过，规划器不会这样做。

第 13 章 资源限制

13.1. 表空间最大限额

13.1.1. 概述

表空间是数据库的逻辑划分，一个表空间只能属于一个数据库。所有的数据库对象都存放在指定的表空间中。但主要存放的是表，所以称作表空间。

表空间限额，指的是表空间的最大存储量，用于控制表空间的大小。当设置了表空间的最大限额后，在表空间中存储数据达到一定数量（告警阀值）会进行告警；继续向表空间存储数据，当达到最大限额时，系统会进行报错，提示用户表空间达到了最大限额。

UXDB默认提供了两个表空间ux_default和ux_global，ux_default存储用户数据，ux_global存储全局数据，因为二者是系统的表空间，所以不受表空间最大限额限制。

13.1.2. 语法

ALTER TABLESPACE name SET (tablespace_option = value [, ...]) ;
ALTER TABLESPACE name RESET (tablespace_option [, ...]);

表 13.1. ALTER TABLESPACE参数说明

参数	说明
name	一个现有表空间的名称。
tablespace_option	要设置或者重置的一个表空间参数。本次新增max_space_size参数。

示例

//设置表空间最大限额为10M bytes
uxdb=# **alter tablespace myspace set (max_space_size='10M');**
//重置表空间限额，即删除表空间限额
uxdb=# **alter tablespace myspace reset (max_space_size);**

max_space_size='10M'，其中10M代表10M bytes。另外，max_space_size的数值单位支持k K、m M、g G、t T，分别代表K bytes、M bytes、G bytes、T bytes。

13.1.3. 支持平台

Linux、Windows

13.1.4. 支持模式

标准模式、兼容模式、标准安全、兼容安全

13.1.5. 使用限制

1. ux_default和ux_global 表空间不受限额控制。

2. 集群处于恢复过程中，不受表空间限额控制，要确保恢复成功。

3. 新增GUC参数和表空间（ux_tablespace）表选项，标准安全和兼容安全模式下只有uxsmo可以设置。标准模式下只有超级用户(uxdb)可以设置。

   a. enable_resource_limit

      开启关闭限额功能。该参数默认关闭。

   b. tblspc_alarm_threshold

      表空间最大限额告警阈值。取值为0.0到1.0，默认值为0.8。

   c. 表空间（ux_tablespace）表选项max_space_size

      用于设置表空间的最大限额值。设置方法为：alter tablespace myspace set (max_space_size='10M'); ---代表设置表空间myspace的最大限额为10M。

4. 当前只有uxsmo可以创建表空间。Uxsmo可以将表空间的使用权限赋予普通用户。

13.1.6. 用法示例

标准安全模式举例

```
//初始化标准安全集群的数据库实例、启动集群、连接数据库：
[uxdb@localhost bin]$ ./initdb -W -D test_sec --security
[uxdb@localhost bin]$ ./ux_ctl -D test_sec start
[uxdb@localhost bin]$ ./uxsql -d uxdb -u uxsmo
//创建表空间目录：
[uxdb@localhost ~]$ mkdir test8
[uxdb@localhost ~]$ cd test8
[uxdb@localhost test8]$pwd
/home/uxdb/test8
//创建表空间、创建表并指定表空、查看表空间信息：
uxdb=> create tablespace myspace8 location '/home/uxdb/test8';
CREATE TABLESPACE
uxdb=> create table test8 (id1 int,id2 int) tablespace myspace8;
CREATE TABLE
uxdb=> select *,ux_size_pretty(ux_tablespace_size(spcname)) from ux_tablespace where
 spcname='myspace8';
  oid  | spcname  | spcowner | spcacl | spcoptions | ux_size_pretty
-------+----------+----------+--------+------------+----------------
 16384 | myspace8 |     6015 |        |            | 19 bytes
(1 row)
//设置表空间限额、查看资源限额开关是否打开（默认关闭）并打开、重启集群或者reload：
uxdb=> alter tablespace myspace8 set (max_space_size='10M');
ALTER TABLESPACE
uxdb=> select *,ux_size_pretty(ux_tablespace_size(spcname)) from ux_tablespace where
 spcname='myspace8';
  oid  | spcname  | spcowner | spcacl |      spcoptions      | ux_size_pretty
-------+----------+----------+--------+----------------------+----------------
 16384 | myspace8 |     6015 |        | {max_space_size=10M} | 19 bytes
(1 row)
uxdb=> show enable_resource_limit;
 enable_resource_limit
```

```
----------------------
 off
(1 row)
uxdb=> alter system set enable_resource_limit to on;
ALTER SYSTEM
uxdb=> \q
[uxdb@localhost bin]$ ./ux_ctl -D test_sec/ reload
server signaled
//重新连接数据库：
[uxdb@localhost bin]$ ./uxsql -d uxdb -U uxsmo
//插入数据，当表空间大小超过0.8*max_size则报警告，当表空间大小超过max_size则报错：
//查询表空间大小，发现已经到达限额值：
uxdb=> insert into test8 select generate_series(1,100000),generate_series(1,100000);
INSERT 0 100000
uxdb=> insert into test8 select generate_series(1,100000),generate_series(1,100000);
WARNING:  Tablespace myspace8 exceeds threshold size(maxsize * threshold)
HINT:  Max size is 10485760, current size is 8355876, request size is 32768, threshold is 0.8
.....
ERROR:  Tablespace myspace8 size exceeds maxsize
HINT:  Max size is 10485760, current size is 10453028, request size is 32768, threshold is 0.8
uxdb=> select *,ux_size_pretty(ux_tablespace_size(spcname)) from ux_tablespace where
 spcname='myspace8';
  oid  | spcname  | spcowner | spcacl |      spcoptions      | ux_size_pretty
-------+----------+----------+--------+----------------------+----------------
 16384 | myspace8 |     6015 |        | {max_space_size=10M} | 10208 kB
(1 row)
//重置表空间限额即取消限额，查看表空间大小，继续插入数据则可以成功：
uxdb=> alter tablespace myspace8 reset (max_space_size);
ALTER TABLESPACE
uxdb=> select *,ux_size_pretty(ux_tablespace_size(spcname)) from ux_tablespace where
 spcname='myspace8';
  oid  | spcname  | spcowner | spcacl | spcoptions | ux_size_pretty
-------+----------+----------+--------+------------+----------------
 16384 | myspace8 |     6015 |        |            | 10208 kB
(1 row)
uxdb=> insert into test8 select generate_series(1,100000),generate_series(1,100000);
INSERT 0 100000
```

兼容安全模式举例

```
//初始化兼容安全集群的数据库实例、启动集群、连接数据库：
[uxdb@localhost bin]$ ./initdb -W -D comp_test_sec --running-mode=compatible --security
[uxdb@localhost bin]$ ./ux_ctl -D comp_test_sec/ start
[uxdb@localhost bin]$ ./uxsql -d UXDB -U UXDB
//创建表空间目录：
[uxdb@localhost ~]$mkdir test9
[uxdb@localhost ~]$cd test9
[uxdb@localhost test9]pwd
/home/uxdb/test9
//创建表空间、创建表并指定表空、查看表空间信息：
UXDB=> create tablespace myspace9 location '/home/uxdb/test9';
CREATE TABLESPACE
UXDB=> create table test9 (id1 int,id2 int) tablespace myspace9;
CREATE TABLE
```

```
UXDB=> select *,ux_size_pretty(ux_tablespace_size(spcname)) from ux_tablespace where
 spcname='MYSPACE9';
  OID  | SPCNAME  | SPCOWNER | SPCACL | SPCOPTIONS | UX_SIZE_PRETTY
-------+----------+----------+--------+------------+----------------
 17072 | MYSPACE9 |     6015 |        |            | 19 bytes
//设置表空间限额、查看资源限额开关是否打开（默认关闭）并打开、重启集群或者reload：
UXDB=> alter tablespace myspace9 set (max_space_size='10M');
ALTER TABLESPACE
UXDB=> show enable_resource_limit;
 enable_resource_limit
-----------------------
 off
(1 row)
UXDB=> alter system set enable_resource_limit to on;
ALTER SYSTEM
UXDB=> \q
[uxdb@localhost bin]$ ./ux_ctl -D comp_test_sec/ reload
server signaled
//重新连接数据库：
[uxdb@localhost bin]$ ./uxsql -d UXDB -U UXSMO
//插入数据，当表空间大小超过0.8*max_size则报警告，当表空间大小超过max_size则报错：
//查询表空间大小，发现已经到达限额值：
UXDB=> insert into test9 select generate_series(1,100000),generate_series(1,100000);
INSERT 0 100000
UXDB=> insert into test9 select generate_series(1,100000),generate_series(1,100000);
WARNING:  Tablespace MYSPACE9 exceeds threshold size(maxsize * threshold)
HINT:  Max size is 10485760, current size is 8355876, request size is 32768, threshold is 0.8
.............
ERROR:  Tablespace MYSPACE9 size exceeds maxsize
HINT:  Max size is 10485760, current size is 10453028, request size is 32768, threshold is 0.8
UXDB=> select *,ux_size_pretty(ux_tablespace_size(spcname)) from ux_tablespace where
 spcname='MYSPACE9';
  OID  | SPCNAME  | SPCOWNER | SPCACL |     SPCOPTIONS      | UX_SIZE_PRETTY
-------+----------+----------+--------+---------------------+----------------
 17072 | MYSPACE9 |     6015 |        | {MAX_SPACE_SIZE=10M} | 10208 kB
(1 row)
//重置表空间限额即取消限额，继续插入数据则可以成功，查看表空间大小：
UXDB=> alter tablespace myspace9 reset (max_space_size);
ALTER TABLESPACE
UXDB=> insert into test9 select generate_series(1,100000),generate_series(1,100000);
INSERT 0 100000
UXDB=> select *,ux_size_pretty(ux_tablespace_size(spcname)) from ux_tablespace where
 spcname='MYSPACE9';
  OID  | SPCNAME  | SPCOWNER | SPCACL | SPCOPTIONS | UX_SIZE_PRETTY
-------+----------+----------+--------+------------+----------------
 17072 | MYSPACE9 |     6015 |        |            | 16 MB
(1 row)
```

部分 III. 大数据存储及相关操作

目录

第 14 章 大对象

14.1. 简介

所有的大对象都存在一个名为ux_largeobject的系统表中。每一个大对象还在系统表ux_largeobject_metadata中有一个对应的项。大对象可以通过类似于标准文件操作的读/写API来进行创建、修改和删除。

UXDB也支持一种称为“TOAST”的存储系统， 它自动把大于一个数据库页的值存储到一个二级存储区域，每一个表都有专属的二级存储区域。这使得大对象功能显得有些陈旧。但是大对象功能仍然有一个优势是能够支持高达4TB的值，而TOAST域只能支持最大1GB。此外，读写一个大对象的片段会很高效，但是大部分在TOAST域上的操作都将把整个值作为一个单元进行读或写。

14.2. 接口

本节描述UXDB的libuxsql客户端接口为访问大对象所提供的功能。UXDB的大对象接口按照Unix文件系统的接口建模，也有相似的open、read、write、lseek等。

所有使用这些函数对大对象的操作都必须发生在一个SQL事务块中，因为大对象文件描述符只在事务期间有效。

在执行任何一个这种函数期间如果发生一个错误，该函数将会返回一个其他的不可能值，典型的是0或-1。一个关于该错误的消息亦会被保存在连接对象中，可以通过UXSQLerrorMessage检索到。

使用这些函数的客户端应用应该包括头文件libuxsql/libuxsql-fs.h并链接libuxsql库。

14.2.1. 创建一个大对象

函数

Oid lo_creat(UXconn *conn, int mode);

创建一个新的大对象。其返回值是分配给这个新大对象的OID或者InvalidOid（0）表示失败。*mode*在UXDB 会被忽略。但是，它最好被设置为INV_READ、INV_WRITE或INV_READ | INV_WRITE（这些符号常量定义在头文件libuxsql/libuxsql-fs.h中）。

一个例子：

```
inv_oid = lo_creat(conn, INV_READ|INV_WRITE);
```

函数

Oid lo_create(UXconn *conn, Oid lobjId);

也创建一个新的大对象。分配给该大对象的OID可以通过*lobjId*指定，如果这样做，该OID已经被某个大对象使用时会产生错误。如果*lobjId*是InvalidOid（0），则lo_create会分配一个未使用的OID（这时和lo_creat的行为相同）。返回值是分配给新大对象的OID或InvalidOid（0）表示发生错误。

一个例子：

```
inv_oid = lo_create(conn, desired_oid);
```

14.2.2. 导入一个大对象

要将一个操作系统文件导入成一个大对象，调用：

```
Oid lo_import(UXconn *conn, const char *filename);
```

*filename*指定了要导入为大对象的操作系统文件名。返回值是分配给新大对象的OID或InvalidOid（0）表示发生错误。注意该文件是被客户端接口库而不是服务器所读取，因此它必须存在于客户端文件系统中并且对于客户端应用是可读的。

函数

```
Oid lo_import_with_oid(UXconn *conn, const char *filename, Oid lobjId);
```

也可以导入一个新大对象。分配给新大对象的OID可以用*lobjId*指定，如果这样做，该OID已经被某个大对象使用时会产生错误。如果*lobjId*是InvalidOid（0），则lo_import_with_oid会分配一个未使用的OID（这和lo_import的行为相同）。返回值是分配给新大对象的OID或InvalidOid（0）表示发生错误。

14.2.3. 导出一个大对象

要把一个大对象导出到一个操作系统文件，调用：

```
int lo_export(UXconn *conn, Oid lobjId, const char *filename);
```

*lobjId*参数指定要导出的大对象的OID，*filename*参数指定操作系统文件名。注意该文件是被客户端接口库而不是服务器写入。成功返回1，错误返回-1。

14.2.4. 打开一个现有的大对象

要打开一个现有的大对象进行读写，调用：

```
int lo_open(UXconn *conn, Oid lobjId, int mode);
```

*lobjId*参数指定要打开的大对象的OID。*mode*位控制着打开对象是为了只读（INV_READ）、只写（INV_WRITE）或者读写（这些符号常量定义在头文件libuxsql/libuxsql-fs.h中）。lo_open返回一个（非负）大对象描述符以便后面用于lo_read、lo_write、lo_lseek、lo_lseek64、lo_tell、lo_tell64、lo_truncate、lo_truncate64以及lo_close。该描述符只在当前事务期间有效。如果打开错误将会返回-1。

服务器目前并不区分模式INV_WRITE和INV_READ |INV_WRITE：在两种情况中都允许从描述符读取。但是在这些模式和单独的INV_READ之间有明显的区别：使用INV_READ我们不能向描述符写入，从中读取的数据则反映了该大对象在活动事务快照时刻的内容（该快照在lo_open被执行时创建），而不管之后被该事务或其他事务写入的内容。从一个以INV_WRITE模式打开的描述符读取的数据所有其他已提交事务以及当前事务所作的写入。这与普通SQL命令SELECT的REPEATABLE READ和READ COMMITTED事务模式之间的区别相似。

如果大对象的SELECT特权不可用，或者如果在指定了INV_WRITE时UPDATE特权不可用，则lo_open将会失败（在UXDB　2.1.1.2之前，这些特权的检查是在使用该描述符的第一次实际读取或写入时进行）。这些特权检查可以用lo_compat_privileges运行时参数禁用。

一个例子：

```
inv_fd = lo_open(conn, inv_oid, INV_READ|INV_WRITE);
```

14.2.5.　向一个大对象写入数据

函数

```
int lo_write(UXconn *conn, int fd, const char *buf, size_t len);
```

从*buf*（大小必须是 *len*）中写出*len*字节到大对象描述符*fd*。参数*fd*必须是已经由前面的lo_open返回的大对象描述符。函数将返回实际写入的字节数（在当前的实现中，除非出错，返回的字节数总是等于*len*）。在出错时，返回值为-1。

尽管参数*len*被声明为类型size_t，该函数会拒绝超过INT_MAX的长度值。在实际中，被传送的数据最好是每块最多数兆字节。

14.2.6.　从一个大对象读取数据

函数

```
int lo_read(UXconn *conn, int fd, char *buf, size_t len);
```

从大对象描述符*fd*中读取最多*len*字节到*buf*　　（大小必须是*len*）中。参数*fd*必须是已经由前面的lo_open返回的大对象描述符。实际读出的字节数将被返回，如果先到达了大对象的末尾返回值可能会小于*len*。出错时返回值为-1。

尽管参数*len*被声明为类型size_t，该函数会拒绝超过INT_MAX的长度值。在实际中，被传送的数据最好是每块最多数兆字节。

14.2.7.　在一个大对象中查找

要改变一个大对象描述符的当前读或写位置，调用：

```
int lo_lseek(UXconn *conn, int fd, int offset, int whence);
```

该函数将大对象文件描述符*fd*的当前位置指针移动到由*offset*指定的新位置。*whence*的可用值是SEEK_SET（从对象开头定位）、SEEK_CUR　　（从当前位置定位）以及SEEK_END（从对象末尾定位）。返回值是新位置的指针，或者是-1表示出错。

在处理可能超过2GB大小的大对象时，换用

```
ux_int64 lo_lseek64(UXconn *conn, int fd, ux_int64 offset, int whence);
```

该函数的行为和lo_lseek相同，但是它能接受一个超过2GB的*offset*并/或传送一个超过2GB的结果。注意如果新位置的指针超过2GB，lo_lseek会失败。

14.2.8. 获取一个大对象的查找位置

要得到一个大对象描述符的当前读或写位置，调用：

```
int lo_tell(UXconn *conn, int fd);
```

如果出现错误，返回值是-1。

在处理可能超过2GB大小的大对象时，换用：

```
ux_int64 lo_tell64(UXconn *conn, int fd);
```

该函数和lo_tell的行为相同，但是它能传递超过2GB的结果。注意如果当前读/写位置超过2GB，lo_tell将会失败。

14.2.9. 截断一个大对象

要将一个大对象截断成一个给定长度，调用：

```
int lo_truncate(PGcon *conn, int fd, size_t len);
```

该函数将大对象描述符*fd*截断为长度*len*。参数*fd*必须是已经由前面的lo_open返回的大对象描述符。如果*len*超过了大对象的当前长度，大对象将会被使用空字节（'\0'）扩展到指定长度。成功时lo_truncate返回0，失败时返回值为-1。

描述*fd*的读/写位置不变。

尽管参数*len*被声明为类型size_t，lo_truncate会拒绝超过INT_MAX的长度值。

在处理可能超过2GB大小的大对象时，换用：

```
int lo_truncate64(PGcon *conn, int fd, ux_int64 len);
```

该函数和lo_truncate的行为相同，但它能够接受超过2GB的*len*值。

14.2.10. 关闭一个大对象描述符

要关闭一个大对象描述符，调用：

```
int lo_close(UXconn *conn, int fd);
```

其中*fd*是由lo_open返回的大对象描述符。成功时，lo_close返回0，失败时返回-1。

在事务末尾仍然保持打开的任何大对象描述符都会自动被关闭。

14.2.11. 移除一个大对象

要从数据库中移除一个大对象，调用：

```
int lo_unlink(UXconn *conn, Oid lobjId);
```

*lobjId*参数指定要移除的大对象的OID。成功时返回1，失败时返回-1。

14.3. 相关插件

14.3.1. lo

lo模块提供管理大对象（也被称为LO或BLOB）的支持。这包括一种数据类型lo以及一个触发器lo_manage。

14.3.1.1. 原理

JDBC（java数据库连接）驱动的问题之一是规范假定对BLOB（二进制大对象）的引用被存储在一个表中，如果该表被改变，相关的BLOB会被从数据库删除；ODBC（开放数据库连接）驱动也存在这样的问题。

但对于UXDB来说这并不会发生。大对象被当做自主的对象，表对象通过OID引用大对象，可以有多个表对象通过OID引用同一个大对象，因此系统不会因为表对象的改变或者删除而删除大对象。但是由于使用JDBC或ODBC的标准代码不会删除大对象，从而导致孤立大对象—不被任何表引用的大对象，会始终占据磁盘空间。

lo模块允许通过附加一个触发器到包含LO引用列的表来修复这种问题。该触发器本质上只是在删除或修改一个引用大对象的值时做lo_unlink。使用这个触发器时，触发器控制的列中的大对象必须只有一个数据库引用。

这个模块也提供了一种数据类型lo，是oid类型的一个域。有助于区分包含大对象引用的数据库列和包含其他类的数据库列。并非必须使用lo类型来使用触发器，但是用它来追踪数据库中哪些列表示用触发器管理的大对象非常方便。

14.3.1.2. 示例

```
CREATE EXTENSION lo;
CREATE TABLE image (title TEXT, raster lo);

CREATE TRIGGER t_raster BEFORE UPDATE OR DELETE ON image
   FOR EACH ROW EXECUTE PROCEDURE lo_manage(raster);
```

对每一个将包含大对象唯一引用的列，创建一个BEFORE UPDATE OR DELETE触发器，并且将该列名作为唯一的触发器参数。也可以用BEFORE UPDATE OF column_name来限制该触发器只对该列上的更新事件执行。如果需要在同一个表中有多个lo列，为每一个lo列创建一个独立的触发器，记住为同一个表上的每个触发器指定不同的名称。

14.3.1.3. 限制

清空或删除表仍将让表中包含的大对象变成孤立的，因为触发器在这种情况下不会被执行。可以在TRUNCATE TABLE或DROP TABLE之前使用DELETE FROM table来避免这种问题。如果已经有或者怀疑有孤立的大对象，使用客户端应用vacuumlo模块可以进行清理。

14.3.2. ux_lob_operate

14.3.2.1. 概述

ux_lob_operate提供了对大对象进行操作的接口，包括对大对象的追加、拷贝、删除、截断等操作,及临时大对象的功能。

14.3.2.2. 大对象操作

为了方便演示设置**set bytea_output TO escape;**以字符形式显示lob的内容。

14.3.2.2.1. lo_append

14.3.2.2.1.1. lo_append(fd_dest int4, fd_src int4)

- 功能描述

  通过fd向一个大对象尾追加一个大对象。

- 函数

  lo_append(fd_dest int4, fd_src int4)

- 参数说明

  表 14.1. lo_append 参数说明

名称	描述	说明
fd_dest	目的大对象的描述符fd	需要事先以写方式打开
fd_src	源大对象的描述符fd	需要事先以读方式打开

- 示例

```
uxdb=# select lo_create(16788); --创建大对象
 lo_create
-----------
    16788
(1 row)

uxdb=# select lo_put(16788, 0, 'aaaaaa');
 lo_put
--------

(1 row)

uxdb=# select lo_create(16789);
 lo_create
-----------
    16789
(1 row)

uxdb=# select lo_put(16789, 0, 'bbbbbb');
 lo_put
--------

(1 row)

uxdb=# select lo_get(16788); --获取大对象内容
 lo_get
--------
 aaaaaa
```

```
(1 row)

uxdb=# select lo_get(16789);
 lo_get
--------
 bbbbbb
(1 row)

uxdb=# select lo_open(16788, set_mode('read')); --以读方式打开
 lo_open
---------
       0
(1 row)

uxdb=# select lo_open(16789, set_mode('write')); --以写方式打开
 lo_open
---------
       1
(1 row)

uxdb=# select lo_append(1,0); --通过文件描述符操作追加
 lo_append
-----------
         0
(1 row)

uxdb=# select lo_get(16789);
    lo_get
--------------
 bbbbbbaaaaaa
(1 row)
```

14.3.2.2.1.2. lo_append(dest_oid oid, src_oid oid)

- 功能描述

  通过OID向一个大对象尾追加一个大对象。

- 函数

  lo_append(dest_oid oid, src_oid oid)

- 参数说明

  表 14.2. lo_append 参数说明

名称	描述	说明
dest_oid	目标大对象的OID	不需要提前打开大对象
src_oid	源大对象的OID	不需要提前打开大对象

- 示例

```
uxdb=# select lo_get(16789);
    lo_get
```

```
--------------
 bbbbbbaaaaaa
(1 row)

uxdb=# select lo_get(16788);
 lo_get
--------
 aaaaaa
(1 row)

uxdb=# select lo_append(16789::oid, 16788::oid);--通过oid进行追加
 lo_append
-----------
         0
(1 row)

uxdb=# select lo_get(16789);
      lo_get
--------------------
 bbbbbbaaaaaaaaaaaa
(1 row)
```

14.3.2.2.2. lo_erase

14.3.2.2.2.1. lo_erase(lob_fd int4, amount int8, lob_offset int8, fill bytea)

- 功能描述

  通过fd删除大对象的固定内容。

- 函数

  lo_erase(lob_fd int4, amount int8, lob_offset int8, fill bytea)

- 参数说明

  表 14.3. lo_erase 参数说明

名称	描述	说明
lob_fd	大对象的描述符fd	需要事先以写方式打开
amount	需要删除的字节数	如果偏移量+需要删除的字节数超出大对象长度，就以大对象长度删除；超出的部分忽略
lob_offset	偏移量	删除的起始位置
fill	填充的字节	CLOB类型一般以空格填充；BLOB类型一般以\0填充。只能是一个字节

- 示例

```
uxdb=# select lo_get(16789);
   lo_get
--------------
```

```
 bbbbbbaaaaaa
(1 row)

uxdb=# select lo_erase(1, 3, 7, ' '); --从第7个位置删除3个字节，以空格填充
 lo_erase
----------
        3
(1 row)

uxdb=# select lo_get(16789);
   lo_get
--------------
 bbbbbba   aa
(1 row)
```

14.3.2.2.2.2. lo_erase(lob_oid oid, amount int8, lob_offset int8, fill bytea)

- 功能描述

  通过OID删除大对象的固定内容。

- 函数

  lo_erase(lob_oid oid, amount int8, lob_offset int8, fill bytea)

- 参数说明

  表 14.4. lo_erase 参数说明

名称	描述	说明
lob_oid	大对象的OID	不需要提前打开大对象
amount	需要删除的字节数	如果偏移量+需要删除的字节数超出大对象长度，就以大对象长度删除；超出的部分忽略
lob_offset	偏移量	删除的起始位置
fill	填充的字节	CLOB类型一般以空格填充；BLOB类型一般以\0填充

- 示例

```
uxdb=# select lo_get(16789);
      lo_get
--------------------
 bbbbbbaaaaaaaaaaaa
(1 row)

uxdb=# select lo_erase(16789::oid, 6, 14, ' ');--从第14个位置开始删除6个字节，以--空格填充
 lo_erase
----------
        4
(1 row)

uxdb=# select lo_get(16789);
```

```
     lo_get
--------------------
 bbbbbbaaaaaaaa
(1 row)
```

14.3.2.2.3. lo_copy

14.3.2.2.3.1. lo_copy(fd_dest int4, fd_src int4, amount int8, dest_offset int8, src_offset int8)

- 功能描述

  通过fd拷贝大对象内容到另一个大对象。

- 函数

  lo_copy(fd_dest int4, fd_src int4, amount int8, dest_offset int8, src_offset int8)

- 参数说明

  表 14.5. lo_copy 参数说明

名称	描述	说明
fd_dest	目标大对象的描述符fd	需要事先以写方式打开
fd_src	源大对象的描述符fd	需要事先以读方式打开
amount	需要拷贝的字节数	如果偏移量+需要删除的字节数超出大对象长度，就拷贝大对象长度；超出的部分忽略
dest_offset	目标大对象的偏移量	不能大于目标大对象的长度
src_offset	源大对象的偏移量	不能大于源大对象的长度

- 示例

```
uxdb=# select lo_copy(1, 0, 6, 7, 0); --从src起始位置拷贝7个字节到dest第6位
 lo_copy
---------
       0
(1 row)

uxdb=# select lo_get(16789);
   lo_get
---------------
 bbbbbbaaaaaaa
(1 row)
```

14.3.2.2.3.2. lo_copy(dest_oid oid, src_oid oid, amount int8, dst_offset int8, src_offset int8)

- 功能描述

  通过OID拷贝大对象内容到另一个大对象。

- 函数

lo_copy(dest_oid oid, src_oid oid, amount int8, dst_offset int8, src_offset int8)

- 参数说明

表 14.6. lo_copy 参数说明

名称	描述	说明
dest_oid	目标大对象的OID	不需要提前打开大对象
src_oid	源大对象的OID	不需要提前打开大对象
amount	需要拷贝的字节数	如果偏移量+需要删除的字节数超出大对象长度，就拷贝大对象长度；超出的部分忽略
dest_offset	目标大对象的偏移量	偏移位置不能大于大对象长度
src_offset	源大对象的偏移量	偏移位置不能大于大对象长度

- 示例

```
uxdb=# select lo_get(16789);
      lo_get
--------------------
 bbbbbbaaaaaaaa
(1 row)

uxdb=# select lo_copy(16789::oid, 16788::oid, 6, 0, 0);
 lo_copy
---------
       0
(1 row)

uxdb=# select lo_get(16789);
      lo_get
--------------------
 aaaaaaaaaaaaaa
(1 row)
```

14.3.2.2.4. lo_try_lseek64

- 功能描述

获取大对象的查找位置。

- 函数

lo_try_lseek64(lob_fd int4, lob_offset int8)

- 参数说明

表 14.7. lo_try_lseek64 参数说明

名称	描述	说明
lob_fd	大对象的描述符fd	需要事先打开大对象

名称	描述	说明
lob_offset	偏移量	偏移量不能超过大对象的长度

- 示例

```
uxdb=# select lo_try_lseek64(1, 5);
 lo_try_lseek64
----------------
              5
(1 row)
```

14.3.2.2.5. ux_getpid

- 功能描述

  获取当前服务端的PID。

- 函数

  ux_getpid()

- 示例

```
uxdb=# select ux_getpid();
 ux_getpid
-----------
    105726
(1 row)
```

14.3.2.2.6. open_mode

- 功能描述

  获取打开模式的整型值。

- 函数

  open_mode(text)

- 参数说明

  当传入的字符串是'write'的时候返回131072，代表写方式的整型值；当传入的字符串是'read'的时候返回262144，代表读方式的整型值；传入其他字符串的时候返回0。

- 示例

```
uxdb=# select lo_open(16788, open_mode('read')); --以读的方式打开
 lo_open
---------
       0
(1 row)

uxdb=# select lo_open(16789, open_mode('write'));--以写的方式打开
 lo_open
---------
       1
```

(1 row)

14.3.2.2.7. set_mode

- 功能描述

  获取位置的整型值。

- 函数

  set_mode(text)

- 参数说明

  传入的字符串是'set'时返回0，代表设置模式的整型值；传入字符串是'cur'时返回1，代表当前位置。传入字符串是'end'时返回2，代表设置到最后位置。传入其他字符串是返回-1。

- 示例

```
uxdb=# select lo_open(16789, open_mode('write')); --以写的方式打开
 lo_open
---------
       0
(1 row)

uxdb=# select lo_lseek64(0, 0, set_mode('set')); --使用绝对偏移
 lo_lseek64
------------
          0
(1 row)

uxdb=# select lo_lseek64(0, 0, set_mode('cur'));--使用当前位置的相对偏移
 lo_lseek64
------------
          0
(1 row)

uxdb=# select lo_lseek64(0, 0, set_mode('end'));--使用最后位置的相对偏移
 lo_lseek64
------------
          5
(1 row)
```

14.3.2.2.8. lo_get_length

- 功能描述

  获取大对象的长度(字节数)。

- 函数

  lo_get_length(oid)

- 参数说明

  传入大对象的oid。

- 示例

```
uxdb=# select lo_get(16789);
   lo_get
--------------
 bbbbbbaaaaaa
(1 row)

uxdb=# select lo_get_length(16789);
 lo_get_length
---------------
            12
(1 row)
```

14.3.2.2.9. lo_trim

14.3.2.2.9.1. lo_trim(fd int4, newlen int8)

- 功能描述

  通过fd截断大对象。

- 函数

  lo_trim(fd int4, newlen int8)

- 参数说明

  表 14.8. lo_trim 参数说明

名称	描述	说明
fd	大对象的描述符	需要事先以写方式打开大对象
newlen	大对象的剩余长度	不能大于大对象长度

- 示例

```
uxdb=# select lo_trim(0, 5);
 lo_trim
---------
       0
(1 row)

uxdb=# select lo_get(16789);
 lo_get
--------
 aaaaa
(1 row)
```

14.3.2.2.9.2. lo_trim(lob_oid oid, newlen int8)

- 功能描述

  通过OID截断大对象。

- 函数

lo_trim(lob_oid oid, newlen int8)

- 参数说明

表 14.9. lo_trim 参数说明

名称	描述	说明
lob_oid	大对象的OID	不需要提前打开大对象
newlen	大对象的剩余长度	不能大于大对象长度

- 示例

```
uxdb=# select lo_trim(16789::oid, 5);
 lo_trim
---------
       0
(1 row)

uxdb=# select lo_get(16789);
 lo_get
--------
 aaaaa
(1 row)
```

14.3.2.3. 临时大对象

14.3.2.3.1. 临时大对象记录

临时大对象被记录在表ux_temp_lob中。

表 14.10. 表ux_temp_lob 列说明

名称	类型	描述
oid	oid	大对象的oid
type	duration	大对象的类型
pid	int4	创建大对象的服务进程pid

14.3.2.3.2. 临时大对象类型

临时大对象被分成两类，一类是连接级别的临时大对象用0表示，可以调用free_session_temp函数一次释放当前连接创建的连接级别大对象，也可以调用free_temp_lob逐个释放；另一类是调用级别的临时大对象用1表示，只能调用free_temp_lob逐个释放。另外临时大对象的类型被指定为新的类型duration。

14.3.2.3.3. create_temp_lob

- 功能描述

创建临时大对象。

- 参数说明

表 14.11. create_temp_lob 参数说明

名称	描述
oid	大对象oid，如果输入0则会分配一个大对象oid，否则按照给定的oid创建
duration	大对象类型

- 返回值

  大对象oid。

- 示例

```
uxdb=# select create_temp_lob(0,0::duration);
 create_temp_lob
-----------------
           40985
(1 row)
```

14.3.2.3.4. free_temp_lob

- 功能描述

  删除临时大对象。

- 参数说明

表 14.12. free_temp_lob 参数说明

名称	描述
oid	临时大对象oid

- 返回值

  临时大对象存在返回1，否则返回空。

- 示例

```
uxdb=# select free_temp_lob(40985);
 free_temp_lob
---------------
             1
(1 row)
```

14.3.2.3.5. free_temp_session_lob

- 功能描述

  清除连接级别临时大对象。

- 返回值

  临时大对象存在返回1，否则返回空。

- 示例

```
uxdb=# select free_temp_session_lob();
 free_temp_session_lob
-----------------------
                     1
(1 row)
```

14.3.2.3.6. lob_is_temp

- 功能描述

  判断大对象是不是连接级别临时大对象。

- 参数说明

  表 14.13. lob_is_temp 参数说明

名称	描述
oid	大对象oid

- 返回值

  所给定oid是连接级别临时大对象返回1，否则返回0。

- 示例

```
uxdb=# select lob_is_temp(40988);
 lob_is_temp
-------------
           1
(1 row)
```

14.4. 工具

14.4.1. vacuumlo

vacuumlo — 从UXDB数据库中移除孤立的大对象

14.4.1.1. 用法

vacuumlo [*option...*] *dbname...*

14.4.1.2. 描述

vacuumlo是一个从UXDB数据库中移除“孤立”大对象的简单使用程序。一个孤立的大对象（LO）是指其OID不出现在数据中任何oid或lo数据列中的LO。

如果你使用该程序，你也许还会对lo模块中的lo_manage触发器感兴趣。lo_manage对于避免创建孤立LO有用处。

在命令行中提到的所有数据库都将被处理。

14.4.1.3. 选项

vacuumlo接受下列命令行参数：

-l *limit*
--limit= *limit*

在每一个事务中移除不超过*limit*个大对象（默认值为1000）。因为移除每一个LO时服务器都将要求一个锁，所以在一个事务中移除过多的LO会有超过max_locks_per_transaction的风险。如果你想在一个事务中就完成所有的移除工作，请将这个限制设置为0。

-n
--dry-run

不移除任何东西，只是显示下一步操作。

-v
--verbose

显示进度消息。

-V
--version

打印vacuumlo的版本并退出。

-?
--help

显示关于vacuumlo的命令行参数，并且退出。

vacuumlo也接受下列命令行参数用于连接：

-h *host*
--host=*host*

数据库服务器的主机名。

-p *port*
--port=*port*

数据库服务器的端口。

-U *username*
--username= *username*

用于连接的用户名。

-w
--no-password

不要发出一个口令提示。如果服务器要求口令认证并且没有其他方式可以提供一个口令（例如一个.uxpass文件），连接尝试将会失败。这个选项可用于批处理任务以及脚本，因为在这些情况下不会有用户输入口令。

-W
--password

强制vacuumlo在连接到数据库之前提示要求一个口令。

这个选项不是不可缺少的，因为如果服务器要求口令认证，vacuumlo会自动提示要求一个口令。但是，vacuumlo将会浪费一次连接尝试来了解到服务器需要口令。在某些情况，值得用-W来避免这种额外的连接尝试。

14.4.1.4. 环境变量

UXHOST
UXPORT
UXUSER

默认连接参数。

和大部分其他UXDB工具相似，这个工具也使用libuxsql支持的环境变量。

14.4.1.5. 说明

vacuumlo按照下列方法工作：首先vacuumlo建立一个临时表，其中包含所选择数据库中所有大对象的OID。然后它会扫描数据库中所有类型为oid或lo的列，并且从临时表中移除在这些列中出现过的OID（注意：只有类型为这些名字的才被考虑，特别的，在这些类型之上的域是不被考虑的）。临时表中剩下的项就标识了孤立LO。它们将被移除。

14.5. 兼容类型

14.5.1. blob

- 类型介绍

  BLOB (binary large object)二进制大对象，是一个可以存储二进制文件的容器，图片、音频文件等二进制文件均可。支持empty_blob、blob_import、blob_export等函数。

- 用法示例

  1. 创建表包含BLOB，并使用empty_blob函数。

     ```
     UXDB=# create table tb_blob (pictureid blob);
     CREATE TABLE
     UXDB=# insert into tb_blob values (empty_blob());
     INSERT 0 1
     UXDB=# select * from tb_blob;
      pictureid
     -----------
      16387
     (1 row)
     ```

  2. 导入函数blob_import。

     ```
     UXDB=# insert into tb_blob values (blob_import('/home/uxdb/Desktop/soft/zhuzi.jpg'));
     INSERT 0 1
     UXDB=# select * from tb_blob;
      pictureid
     -----------
      16387
      16388
     (2 rows)
     ```

3. 导出函数blob_export

```
UXDB=# select blob_export(16388, '/home/uxdb/Desktop/soft/zhuzi_2.jpg');
 blob_export
-------------
           1
(1 row)
```

14.5.2. clob

- 类型介绍

CLOB（Character Large Object）字符串对象，存储大的单字节字符集数据，如文档等。支持empty_clob、clob_import、clob_export等函数。

- 用法示例

1. 创建表包含CLOB，并使用empty_clob 函数。

```
UXDB=# create table tb_clob (textid clob);
CREATE TABLE
UXDB=# insert into tb_clob values (empty_clob());
INSERT 0 1
UXDB=# select * from tb_clob;
 textid
--------
 16392
(1 row)
```

2. 导入函数clob_import。

```
UXDB=# insert into tb_clob values (clob_import('/home/uxdb/Desktop/soft/test.txt'));
INSERT 0 1
UXDB=# select * from tb_clob;
 textid
--------
 16392
 16393
(2 rows)
```

3. 导出函数clob_export

```
UXDB=# select clob_export(16393, '/home/uxdb/Desktop/soft/test_2.txt');
 clob_export
-------------

 1
(1 row)
```

部分 IV. 附录

目录

附录 A. SQL关键词

表 A.1 “SQL关键词”列出了在SQL标准以及UXDB 2.1中作为关键词的所有记号。背景资料可以在第 1.1.1 节 “标识符和关键词”中找到（由于篇幅的缘故，只包括了SQL标准的最近两个版本以及用于与历史比较的SQL-92。这些版本以及其他中间标准的版本之间的差别很小）。

SQL区分保留关键词和非保留关键词。根据标准，保留关键词才是真正的关键词，它们绝不会被允许作为标识符。非关键词仅仅是在特定上下文中具有特殊的含义并且可以在其他上下文中被用作标识符。大部分非保留关键词实际上是SQL指定的内建表和内建函数的名字。非保留关键词的概念存在的意义上实际上是声明某些上下文中的一个词被附加了某种预定义的含义。

在UXDB的解析器中情况更加复杂。其中有多种不同的记号分类，从那些决不能被用作标识符的加号到那些在解析器中与普通标识符比起来绝对没有特殊状态的记号（后者通常是SQL中指定的函数）。在UXDB中甚至保留关键词也不是完全被保留的，而是可以被用作列标签（例如可以写SELECT 55 AS CHECK，虽然CHECK是一个保留关键词）。

在表 A.1 “SQL关键词”的UXDB列中，我们把解析器明确知道但允许作为列名或者表名的那些关键词分类为“非保留”。有一些关键词是非保留的，但是不能被用作函数或数据类型名称，因此它们会被标记（大部分这些词表示有特殊语法的内建函数或数据类型。这种函数或类型仍然可用，但是不能被用户重新定义）。不允许作为列名或表名的记号被打上“保留”的标签。某些保留关键词被允许作为函数或数据类型的名字，这也显示在该表中。如果没有被那样标记，保留关键词仅被允许作为“AS”列的标签名。

作为一条一般性的规则，如果对包含所列出关键词作为标识符的命令得到了站不住脚的解析器错误，应该尝试将该标识符加上引号来看看是否能解决问题。

在学习表 A.1 “SQL关键词”之前有一件重要的事情是理解一个在UXDB中不被保留的关键字并不意味着与该词相关的特性没有被实现。反过来，一个关键词的存在也不表示相应特性的存在。

表 A.1. SQL关键词

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
A		非保留	非保留	
ABORT	非保留			
ABS		保留	保留	
ABSENT		非保留	非保留	
ABSOLUTE	非保留	非保留	非保留	保留
ACCESS	非保留			
ACCORDING		非保留	非保留	
ACTION	非保留	非保留	非保留	保留
ADA		非保留	非保留	非保留
ADD	非保留	非保留	非保留	保留
ADMIN	非保留	非保留	非保留	
AFTER	非保留	非保留	非保留	
AGGREGATE	非保留			
ALL	保留	保留	保留	保留
ALLOCATE		保留	保留	保留
ALSO	非保留			

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
ALTER	非保留	保留	保留	保留
ALWAYS	非保留	非保留	非保留	
ANALYSE	保留			
ANALYZE	保留			
AND	保留	保留	保留	保留
ANY	保留	保留	保留	保留
ARE		保留	保留	保留
ARRAY	保留	保留	保留	
ARRAY_AGG		保留	保留	
ARRAY_MAX_CARDINALITY		保留		
AS	保留	保留	保留	保留
ASC	保留	非保留	非保留	保留
ASENSITIVE		保留	保留	
ASSERTION	非保留	非保留	非保留	保留
ASSIGNMENT	非保留	非保留	非保留	
ASYMMETRIC	保留	保留	保留	
AT	非保留	保留	保留	保留
ATOMIC		保留	保留	
ATTACH	非保留			
ATTRIBUTE	非保留	非保留	非保留	
ATTRIBUTES		非保留	非保留	
AUTHORIZATION	保留（可以是函数或类型）	保留	保留	保留
AVG		保留	保留	保留
BACKWARD	非保留			
BASE64		非保留	非保留	
BEFORE	非保留	非保留	非保留	
BEGIN	非保留	保留	保留	保留
BEGIN_FRAME		保留		
BEGIN_PARTITION		保留		
BERNOULLI		非保留	非保留	
BETWEEN	非保留（不能是函数或类型）	保留	保留	保留
BIGINT	非保留（不能是函数或类型）	保留	保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
BINARY	保留（可以是函数或类型）	保留	保留	
BIT	非保留（不能是函数或类型）			保留
BIT_LENGTH				保留
BLOB		保留	保留	
BLOCKED		非保留	非保留	
BOM		非保留	非保留	
BOOLEAN	非保留（不能是函数或类型）	保留	保留	
BOTH	保留	保留	保留	保留
BREADTH		非保留	非保留	
BY	非保留	保留	保留	保留
C		非保留	非保留	非保留
CACHE	非保留			
CALL	非保留	保留	保留	
CALLED	非保留	保留	保留	
CARDINALITY		保留	保留	
CASCADE	非保留	非保留	非保留	保留
CASCADED	非保留	保留	保留	保留
CASE	保留	保留	保留	保留
CAST	保留	保留	保留	保留
CATALOG	非保留	非保留	非保留	保留
CATALOG_NAME		非保留	非保留	非保留
CEIL		保留	保留	
CEILING		保留	保留	
CHAIN	非保留	非保留	非保留	
CHAR	非保留（不能是函数或类型）	保留	保留	保留
CHARACTER	非保留（不能是函数或类型）	保留	保留	保留
CHARACTERISTICS	非保留	非保留	非保留	
CHARACTERS		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
CHARACTER_LENGTH		保留	保留	保留
CHARACTER_SET_CATALOG		非保留	非保留	非保留
CHARACTER_SET_NAME		非保留	非保留	非保留
CHARACTER_SET_SCHEMA		非保留	非保留	非保留
CHAR_LENGTH		保留	保留	保留
CHECK	保留	保留	保留	保留
CHECKPOINT	非保留			
CLASS	非保留			
CLASS_ORIGIN		非保留	非保留	非保留
CLOB		保留	保留	
CLOSE	非保留	保留	保留	保留
CLUSTER	非保留			
COALESCE	非保留（不能是函数或类型）	保留	保留	保留
COBOL		非保留	非保留	非保留
COLLATE	保留	保留	保留	保留
COLLATION	保留（可以是函数或类型）	非保留	非保留	保留
COLLATION_CATALOG		非保留	非保留	非保留
COLLATION_NAME		非保留	非保留	非保留
COLLATION_SCHEMA		非保留	非保留	非保留
COLLECT		保留	保留	
COLUMN	保留	保留	保留	保留
COLUMNS	非保留	非保留	非保留	
COLUMN_NAME		非保留	非保留	非保留
COMMAND_FUNCTION		非保留	非保留	非保留
COMMAND_FUNCTION_CODE		非保留	非保留	
COMMENT	非保留			
COMMENTS	非保留			
COMMIT	非保留	保留	保留	保留
COMMITTED	非保留	非保留	非保留	非保留
CONCURRENTLY	保留（可以是函数或类型）			
CONDITION		保留	保留	
CONDITION_NUMBER		非保留	非保留	非保留
CONFIGURATION	非保留			

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
CONFLICT	非保留			
CONNECT		保留	保留	保留
CONNECTION	非保留	非保留	非保留	保留
CONNECTION_NAME		非保留	非保留	非保留
CONSTRAINT	保留	保留	保留	保留
CONSTRAINTS	非保留	非保留	非保留	保留
CONSTRAINT_CATALOG		非保留	非保留	非保留
CONSTRAINT_NAME		非保留	非保留	非保留
CONSTRAINT_SCHEMA		非保留	非保留	非保留
CONSTRUCTOR		非保留	非保留	
CONTAINS		保留	非保留	
CONTENT	非保留	非保留	非保留	
CONTINUE	非保留	非保留	非保留	保留
CONTROL		非保留	非保留	
CONVERSION	非保留			
CONVERT		保留	保留	保留
COPY	非保留			
CORR		保留	保留	
CORRESPONDING		保留	保留	保留
COST	非保留			
COUNT		保留	保留	保留
COVAR_POP		保留	保留	
COVAR_SAMP		保留	保留	
CREATE	保留	保留	保留	保留
CROSS	保留（可以是函数或类型）	保留	保留	保留
CSV	非保留			
CUBE	非保留	保留	保留	
CUME_DIST		保留	保留	
CURRENT	非保留	保留	保留	保留
CURRENT_CATALOG	保留	保留	保留	
CURRENT_DATE	保留	保留	保留	保留
CURRENT_DEFAULT_TRANSFORM_GROUP		保留	保留	
CURRENT_PATH		保留	保留	
CURRENT_ROLE	保留	保留	保留	
CURRENT_ROW		保留		

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
CURRENT_SCHEMA	保留（可以是函数或类型）	保留	保留	
CURRENT_TIME	保留	保留	保留	保留
CURRENT_TIMESTAMP	保留	保留	保留	保留
CURRENT_TRANSFORM_GROUP_FOR_TYPE		保留	保留	
CURRENT_USER	保留	保留	保留	保留
CURSOR	非保留	保留	保留	保留
CURSOR_NAME		非保留	非保留	非保留
CYCLE	非保留	保留	保留	
DATA	非保留	非保留	非保留	非保留
DATABASE	非保留			
DATALINK		保留	保留	
DATE		保留	保留	保留
DATETIME_INTERVAL_CODE		非保留	非保留	非保留
DATETIME_INTERVAL_PRECISION		非保留	非保留	非保留
DAY	非保留	保留	保留	保留
DB		非保留	非保留	
DEALLOCATE	非保留	保留	保留	保留
DEC	非保留（不能是函数或类型）	保留	保留	保留
DECIMAL	非保留（不能是函数或类型）	保留	保留	保留
DECLARE	非保留	保留	保留	保留
DEFAULT	保留	保留	保留	保留
DEFAULTS	非保留	非保留	非保留	
DEFERRABLE	保留	非保留	非保留	保留
DEFERRED	非保留	非保留	非保留	保留
DEFINED		非保留	非保留	
DEFINER	非保留	非保留	非保留	
DEGREE		非保留	非保留	
DELETE	非保留	保留	保留	保留
DELIMITER	非保留			
DELIMITERS	非保留			
DENSE_RANK		保留	保留	
DEPENDS	非保留			

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
DEPTH		非保留	非保留	
DEREF		保留	保留	
DERIVED		非保留	非保留	
DESC	保留	非保留	非保留	保留
DESCRIBE		保留	保留	保留
DESCRIPTOR		非保留	非保留	保留
DETACH	非保留			
DETERMINISTIC		保留	保留	
DIAGNOSTICS		非保留	非保留	保留
DICTIONARY	非保留			
DISABLE	非保留			
DISCARD	非保留			
DISCONNECT		保留	保留	保留
DISPATCH		非保留	非保留	
DISTINCT	保留	保留	保留	保留
DLNEWCOPY		保留	保留	
DLPREVIOUSCOPY		保留	保留	
DLURLCOMPLETE		保留	保留	
DLURLCOMPLETEONLY		保留	保留	
DLURLCOMPLETEWRITE		保留	保留	
DLURLPATH		保留	保留	
DLURLPATHONLY		保留	保留	
DLURLPATHWRITE		保留	保留	
DLURLSCHEME		保留	保留	
DLURLSERVER		保留	保留	
DLVALUE		保留	保留	
DO	保留			
DOCUMENT	非保留	非保留	非保留	
DOMAIN	非保留	非保留	非保留	保留
DOUBLE	非保留	保留	保留	保留
DROP	非保留	保留	保留	保留
DYNAMIC		保留	保留	
DYNAMIC_FUNCTION		非保留	非保留	非保留
DYNAMIC_FUNCTION_CODE		非保留	非保留	
EACH	非保留	保留	保留	
ELEMENT		保留	保留	
ELSE	保留	保留	保留	保留
EMPTY		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
ENABLE	非保留			
ENCODING	非保留	非保留	非保留	
ENCRYPTED	非保留			
END	保留	保留	保留	保留
END-EXEC		保留	保留	保留
END_FRAME		保留		
END_PARTITION		保留		
ENFORCED		非保留		
ENUM	非保留			
EQUALS		保留	非保留	
ESCAPE	非保留	保留	保留	保留
EVENT	非保留			
EVERY		保留	保留	
EXCEPT	保留	保留	保留	保留
EXCEPTION				保留
EXCLUDE	非保留	非保留	非保留	
EXCLUDING	非保留	非保留	非保留	
EXCLUSIVE	非保留			
EXEC		保留	保留	保留
EXECUTE	非保留	保留	保留	保留
EXISTS	非保留（不能是函数或类型）	保留	保留	保留
EXP		保留	保留	
EXPLAIN	非保留			
EXPRESSION		非保留		
EXTENSION	非保留			
EXTERNAL	非保留	保留	保留	保留
EXTRACT	非保留（不能是函数或类型）	保留	保留	保留
FALSE	保留	保留	保留	保留
FAMILY	非保留			
FETCH	保留	保留	保留	保留
FILE		非保留	非保留	
FILTER	非保留	保留	保留	
FINAL		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
FIRST	非保留	非保留	非保留	保留
FIRST_VALUE		保留	保留	
FLAG		非保留	非保留	
FLOAT	非保留（不能是函数或类型）	保留	保留	保留
FLOOR		保留	保留	
FOLLOWING	非保留	非保留	非保留	
FOR	保留	保留	保留	保留
FORCE	非保留			
FOREIGN	保留	保留	保留	保留
FORTRAN		非保留	非保留	非保留
FORWARD	非保留			
FOUND		非保留	非保留	保留
FRAME_ROW		保留		
FREE		保留	保留	
FREEZE	保留（可以是函数或类型）			
FROM	保留	保留	保留	保留
FS		非保留	非保留	
FULL	保留（可以是函数或类型）	保留	保留	保留
FUNCTION	非保留	保留	保留	
FUNCTIONS	非保留			
FUSION		保留	保留	
G		非保留	非保留	
GENERAL		非保留	非保留	
GENERATED	非保留	非保留	非保留	
GET		保留	保留	保留
GLOBAL	非保留	保留	保留	保留
GO		非保留	非保留	保留
GOTO		非保留	非保留	保留
GRANT	保留	保留	保留	保留
GRANTED	非保留	非保留	非保留	
GREATEST	非保留（不能是函数或类型）			

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
GROUP	保留	保留	保留	保留
GROUPING	非保留（不能是函数或类型）	保留	保留	
GROUPS	非保留	保留		
HANDLER	非保留			
HAVING	保留	保留	保留	保留
HEADER	非保留			
HEX		非保留	非保留	
HIERARCHY		非保留	非保留	
HOLD	非保留	保留	保留	
HOUR	非保留	保留	保留	保留
ID		非保留	非保留	
IDENTITY	非保留	保留	保留	保留
IF	非保留			
IGNORE		非保留	非保留	
ILIKE	保留（可以是函数或类型）			
IMMEDIATE	非保留	非保留	非保留	保留
IMMEDIATELY		非保留		
IMMUTABLE	非保留			
IMPLEMENTATION		非保留	非保留	
IMPLICIT	非保留			
IMPORT	非保留	保留	保留	
IN	保留	保留	保留	保留
INCLUDE	非保留			
INCLUDING	非保留	非保留	非保留	
INCREMENT	非保留	非保留	非保留	
INDENT		非保留	非保留	
INDEX	非保留			
INDEXES	非保留			
INDICATOR		保留	保留	保留
INHERIT	非保留			
INHERITS	非保留			
INITIALLY	保留	非保留	非保留	保留
INLINE	非保留			

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
INNER	保留（可以是函数或类型）	保留	保留	保留
INOUT	非保留（不能是函数或类型）	保留	保留	
INPUT	非保留	非保留	非保留	保留
INSENSITIVE	非保留	保留	保留	保留
INSERT	非保留	保留	保留	保留
INSTANCE		非保留	非保留	
INSTANTIABLE		非保留	非保留	
INSTEAD	非保留	非保留	非保留	
INT	非保留（不能是函数或类型）	保留	保留	保留
INTEGER	非保留（不能是函数或类型）	保留	保留	保留
INTEGRITY		非保留	非保留	
INTERSECT	保留	保留	保留	保留
INTERSECTION		保留	保留	
INTERVAL	非保留（不能是函数或类型）	保留	保留	保留
INTO	保留	保留	保留	保留
INVOKER	非保留	非保留	非保留	
IS	保留（可以是函数或类型）	保留	保留	保留
ISNULL	保留（可以是函数或类型）			
ISOLATION	非保留	非保留	非保留	保留
JOIN	保留（可以是函数或类型）	保留	保留	保留
K		非保留	非保留	
KEY	非保留	非保留	非保留	保留
KEY_MEMBER		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
KEY_TYPE		非保留	非保留	
LABEL	非保留			
LAG		保留	保留	
LANGUAGE	非保留	保留	保留	保留
LARGE	非保留	保留	保留	
LAST	非保留	非保留	非保留	保留
LAST_VALUE		保留	保留	
LATERAL	保留	保留	保留	
LEAD		保留	保留	
LEADING	保留	保留	保留	保留
LEAKPROOF	非保留			
LEAST	非保留（不能是函数或类型）			
LEFT	保留（可以是函数或类型）	保留	保留	保留
LENGTH		非保留	非保留	非保留
LEVEL	非保留	非保留	非保留	保留
LIBRARY		非保留	非保留	
LIKE	保留（可以是函数或类型）	保留	保留	保留
LIKE_REGEX		保留	保留	
LIMIT	保留	非保留	非保留	
LINK		非保留	非保留	
LISTEN	非保留			
LN		保留	保留	
LOAD	非保留			
LOCAL	非保留	保留	保留	保留
LOCALTIME	保留	保留	保留	
LOCALTIMESTAMP	保留	保留	保留	
LOCATION	非保留	非保留	非保留	
LOCATOR		非保留	非保留	
LOCK	非保留			
LOCKED	非保留			
LOGGED	非保留			
LOWER		保留	保留	保留
M		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
MAP		非保留	非保留	
MAPPING	非保留	非保留	非保留	
MATCH	非保留	保留	保留	保留
MATCHED		非保留	非保留	
MATERIALIZED	非保留			
MAX		保留	保留	保留
MAXVALUE	非保留	非保留	非保留	
MAX_CARDINALITY			保留	
MEMBER		保留	保留	
MERGE		保留	保留	
MESSAGE_LENGTH		非保留	非保留	非保留
MESSAGE_OCTET_LENGTH		非保留	非保留	非保留
MESSAGE_TEXT		非保留	非保留	非保留
METHOD	非保留	保留	保留	
MIN		保留	保留	保留
MINUTE	非保留	保留	保留	保留
MINVALUE	非保留	非保留	非保留	
MOD		保留	保留	
MODE	非保留			
MODIFIES		保留	保留	
MODULE		保留	保留	保留
MONTH	非保留	保留	保留	保留
MORE		非保留	非保留	非保留
MOVE	非保留			
MULTISET		保留	保留	
MUMPS		非保留	非保留	非保留
NAME	非保留	非保留	非保留	非保留
NAMES	非保留	非保留	非保留	保留
NAMESPACE		非保留	非保留	
NATIONAL	非保留（不能是函数或类型）	保留	保留	保留
NATURAL	保留（可以是函数或类型）	保留	保留	保留
NCHAR	非保留（不能是函数或类型）	保留	保留	保留

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
NCLOB		保留	保留	
NESTING		非保留	非保留	
NEW	非保留	保留	保留	
NEXT	非保留	非保留	非保留	保留
NFC		非保留	非保留	
NFD		非保留	非保留	
NFKC		非保留	非保留	
NFKD		非保留	非保留	
NIL		非保留	非保留	
NO	非保留	保留	保留	保留
NONE	非保留（不能是函数或类型）	保留	保留	
NORMALIZE		保留	保留	
NORMALIZED		非保留	非保留	
NOT	保留	保留	保留	保留
NOTHING	非保留			
NOTIFY	非保留			
NOTNULL	保留（可以是函数或类型）			
NOWAIT	非保留			
NTH_VALUE		保留	保留	
NTILE		保留	保留	
NULL	保留	保留	保留	保留
NULLABLE		非保留	非保留	非保留
NULLIF	非保留（不能是函数或类型）	保留	保留	保留
NULLS	非保留	非保留	非保留	
NUMBER		非保留	非保留	非保留
NUMERIC	非保留（不能是函数或类型）	保留	保留	保留
OBJECT	非保留	非保留	非保留	
OCCURRENCES_REGEX		保留	保留	
OCTETS		非保留	非保留	
OCTET_LENGTH		保留	保留	保留

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
OF	非保留	保留	保留	保留
OFF	非保留	非保留	非保留	
OFFSET	保留	保留	保留	
OIDS	非保留			
OLD	非保留	保留	保留	
ON	保留	保留	保留	保留
ONLY	保留	保留	保留	保留
OPEN		保留	保留	保留
OPERATOR	非保留			
OPTION	非保留	非保留	非保留	保留
OPTIONS	非保留	非保留	非保留	
OR	保留	保留	保留	保留
ORDER	保留	保留	保留	保留
ORDERING		非保留	非保留	
ORDINALITY	非保留	非保留	非保留	
OTHERS	非保留	非保留	非保留	
OUT	非保留（不能是函数或类型）	保留	保留	
OUTER	保留（可以是函数或类型）	保留	保留	保留
OUTPUT		非保留	非保留	保留
OVER	非保留	保留	保留	
OVERLAPS	保留（可以是函数或类型）	保留	保留	保留
OVERLAY	非保留（不能是函数或类型）	保留	保留	
OVERRIDING	非保留	非保留	非保留	
OWNED	非保留			
OWNER	非保留			
P		非保留	非保留	
PAD		非保留	非保留	保留
PARALLEL	非保留			
PARAMETER		保留	保留	
PARAMETER_MODE		非保留	非保留	
PARAMETER_NAME		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
PARAMETER_ORDINAL_POSITION		非保留	非保留	
PARAMETER_SPECIFIC_CATALOG		非保留	非保留	
PARAMETER_SPECIFIC_NAME		非保留	非保留	
PARAMETER_SPECIFIC_SCHEMA		非保留	非保留	
PARSER	非保留			
PARTIAL	非保留	非保留	非保留	保留
PARTITION	非保留	保留	保留	
PASCAL		非保留	非保留	非保留
PASSING	非保留	非保留	非保留	
PASSTHROUGH		非保留	非保留	
PASSWORD	非保留			
PATH		非保留	非保留	
PERCENT		保留		
PERCENTILE_CONT		保留	保留	
PERCENTILE_DISC		保留	保留	
PERCENT_RANK		保留	保留	
PERIOD		保留		
PERMISSION		非保留	非保留	
PLACING	保留	非保留	非保留	
PLANS	非保留			
PLI		非保留	非保留	非保留
POLICY	非保留			
PORTION		保留		
POSITION	非保留（不能是函数或类型）	保留	保留	保留
POSITION_REGEX		保留	保留	
POWER		保留	保留	
PRECEDES		保留		
PRECEDING	非保留	非保留	非保留	
PRECISION	非保留（不能是函数或类型）	保留	保留	保留
PREPARE	非保留	保留	保留	保留
PREPARED	非保留			
PRESERVE	非保留	非保留	非保留	保留
PRIMARY	保留	保留	保留	保留

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
PRIOR	非保留	非保留	非保留	保留
PRIVILEGES	非保留	非保留	非保留	保留
PROCEDURAL	非保留			
PROCEDURE	非保留	保留	保留	保留
PROCEDURES	非保留			
PROGRAM	非保留			
PUBLIC		非保留	非保留	保留
PUBLICATION	非保留			
QUOTE	非保留			
RANGE	非保留	保留	保留	
RANK		保留	保留	
READ	非保留	非保留	非保留	保留
READS		保留	保留	
REAL	非保留（不能是函数或类型）	保留	保留	保留
REASSIGN	非保留			
RECHECK	非保留			
RECOVERY		非保留	非保留	
RECURSIVE	非保留	保留	保留	
REF	非保留	保留	保留	
REFERENCES	保留	保留	保留	保留
REFERENCING	非保留	保留	保留	
REFRESH	非保留			
REGR_AVGX		保留	保留	
REGR_AVGY		保留	保留	
REGR_COUNT		保留	保留	
REGR_INTERCEPT		保留	保留	
REGR_R2		保留	保留	
REGR_SLOPE		保留	保留	
REGR_SXX		保留	保留	
REGR_SXY		保留	保留	
REGR_SYY		保留	保留	
REINDEX	非保留			
RELATIVE	非保留	非保留	非保留	保留
RELEASE	非保留	保留	保留	
RENAME	非保留			
REPEATABLE	非保留	非保留	非保留	非保留

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
REPLACE	非保留			
REPLICA	非保留			
REQUIRING		非保留	非保留	
RESET	非保留			
RESPECT		非保留	非保留	
RESTART	非保留	非保留	非保留	
RESTORE		非保留	非保留	
RESTRICT	非保留	非保留	非保留	保留
RESULT		保留	保留	
RETURN		保留	保留	
RETURNED_CARDINALITY		非保留	非保留	
RETURNED_LENGTH		非保留	非保留	非保留
RETURNED_OCTET_LENGTH		非保留	非保留	非保留
RETURNED_SQLSTATE		非保留	非保留	非保留
RETURNING	保留	非保留	非保留	
RETURNS	非保留	保留	保留	
REVOKE	非保留	保留	保留	保留
RIGHT	保留（可以是函数或类型）	保留	保留	保留
ROLE	非保留	非保留	非保留	
ROLLBACK	非保留	保留	保留	保留
ROLLUP	非保留	保留	保留	
ROUTINE	非保留	非保留	非保留	
ROUTINES	非保留			
ROUTINE_CATALOG		非保留	非保留	
ROUTINE_NAME		非保留	非保留	
ROUTINE_SCHEMA		非保留	非保留	
ROW	非保留（不能是函数或类型）	保留	保留	
ROWS	非保留	保留	保留	保留
ROW_COUNT		非保留	非保留	非保留
ROW_NUMBER		保留	保留	
RULE	非保留			
SAVEPOINT	非保留	保留	保留	
SCALE		非保留	非保留	非保留
SCHEMA	非保留	非保留	非保留	保留

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
SCHEMAS	非保留			
SCHEMA_NAME		非保留	非保留	非保留
SCOPE		保留	保留	
SCOPE_CATALOG		非保留	非保留	
SCOPE_NAME		非保留	非保留	
SCOPE_SCHEMA		非保留	非保留	
SCROLL	非保留	保留	保留	保留
SEARCH	非保留	保留	保留	
SECOND	非保留	保留	保留	保留
SECTION		非保留	非保留	保留
SECURITY	非保留	非保留	非保留	
SELECT	保留	保留	保留	保留
SELECTIVE		非保留	非保留	
SELF		非保留	非保留	
SENSITIVE		保留	保留	
SEQUENCE	非保留	非保留	非保留	
SEQUENCES	非保留			
SERIALIZABLE	非保留	非保留	非保留	非保留
SERVER	非保留	非保留	非保留	
SERVER_NAME		非保留	非保留	非保留
SESSION	非保留	非保留	非保留	保留
SESSION_USER	保留	保留	保留	保留
SET	非保留	保留	保留	保留
SETOF	非保留（不能是函数或类型）			
SETS	非保留	非保留	非保留	
SHARE	非保留			
SHOW	非保留			
SIMILAR	保留（可以是函数或类型）	保留	保留	
SIMPLE	非保留	非保留	非保留	
SIZE		非保留	非保留	保留
SKIP	非保留			
SMALLINT	非保留（不能是函数或类型）	保留	保留	保留

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
SNAPSHOT	非保留			
SOME	保留	保留	保留	保留
SOURCE		非保留	非保留	
SPACE		非保留	非保留	保留
SPECIFIC		保留	保留	
SPECIFICTYPE		保留	保留	
SPECIFIC_NAME		非保留	非保留	
SQL	非保留	保留	保留	保留
SQLCODE				保留
SQLERROR				保留
SQLEXCEPTION		保留	保留	
SQLSTATE		保留	保留	保留
SQLWARNING		保留	保留	
SQRT		保留	保留	
STABLE	非保留			
STANDALONE	非保留	非保留	非保留	
START	非保留	保留	保留	
STATE		非保留	非保留	
STATEMENT	非保留	非保留	非保留	
STATIC		保留	保留	
STATISTICS	非保留			
STDDEV_POP		保留	保留	
STDDEV_SAMP		保留	保留	
STDIN	非保留			
STDOUT	非保留			
STORAGE	非保留			
STRICT	非保留			
STRIP	非保留	非保留	非保留	
STRUCTURE		非保留	非保留	
STYLE		非保留	非保留	
SUBCLASS_ORIGIN		非保留	非保留	非保留
SUBMULTISET		保留	保留	
SUBSCRIPTION	非保留			
SUBSTRING	非保留（不能是函数或类型）	保留	保留	保留
SUBSTRING_REGEX		保留	保留	
SUCCEEDS		保留		

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
SUM		保留	保留	保留
SYMMETRIC	保留	保留	保留	
SYSID	非保留			
SYSTEM	非保留	保留	保留	
SYSTEM_TIME		保留		
SYSTEM_USER		保留	保留	保留
T		非保留	非保留	
TABLE	保留	保留	保留	保留
TABLES	非保留			
TABLESAMPLE	保留（可以是函数或类型）	保留	保留	
TABLESPACE	非保留			
TABLE_NAME		非保留	非保留	非保留
TEMP	非保留			
TEMPLATE	非保留			
TEMPORARY	非保留	非保留	非保留	保留
TEXT	非保留			
THEN	保留	保留	保留	保留
TIES	非保留	非保留	非保留	
TIME	非保留（不能是函数或类型）	保留	保留	保留
TIMESTAMP	非保留（不能是函数或类型）	保留	保留	保留
TIMEZONE_HOUR		保留	保留	保留
TIMEZONE_MINUTE		保留	保留	保留
TO	保留	保留	保留	保留
TOKEN		非保留	非保留	
TOP_LEVEL_COUNT		非保留	非保留	
TRAILING	保留	保留	保留	保留
TRANSACTION	非保留	非保留	非保留	保留
TRANSACTIONS_COMMITTED		非保留	非保留	
TRANSACTIONS_ROLLED_BACK		非保留	非保留	
TRANSACTION_ACTIVE		非保留	非保留	
TRANSFORM	非保留	非保留	非保留	
TRANSFORMS		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
TRANSLATE		保留	保留	保留
TRANSLATE_REGEX		保留	保留	
TRANSLATION		保留	保留	保留
TREAT	非保留（不能是函数或类型）	保留	保留	
TRIGGER	非保留	保留	保留	
TRIGGER_CATALOG		非保留	非保留	
TRIGGER_NAME		非保留	非保留	
TRIGGER_SCHEMA		非保留	非保留	
TRIM	非保留（不能是函数或类型）	保留	保留	保留
TRIM_ARRAY		保留	保留	
TRUE	保留	保留	保留	保留
TRUNCATE	非保留	保留	保留	
TRUSTED	非保留			
TYPE	非保留	非保留	非保留	非保留
TYPES	非保留			
UESCAPE		保留	保留	
UNBOUNDED	非保留	非保留	非保留	
UNCOMMITTED	非保留	非保留	非保留	非保留
UNDER		非保留	非保留	
UNENCRYPTED	非保留			
UNION	保留	保留	保留	保留
UNIQUE	保留	保留	保留	保留
UNKNOWN	非保留	保留	保留	保留
UNLINK		非保留	非保留	
UNLISTEN	非保留			
UNLOGGED	非保留			
UNNAMED		非保留	非保留	非保留
UNNEST		保留	保留	
UNTIL	非保留			
UNTYPED		非保留	非保留	
UPDATE	非保留	保留	保留	保留
UPPER		保留	保留	保留
URI		非保留	非保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
USAGE		非保留	非保留	保留
USER	保留	保留	保留	保留
USER_DEFINED_TYPE_CATALOG		非保留	非保留	
USER_DEFINED_TYPE_CODE		非保留	非保留	
USER_DEFINED_TYPE_NAME		非保留	非保留	
USER_DEFINED_TYPE_SCHEMA		非保留	非保留	
USING	保留	保留	保留	保留
VACUUM	非保留			
VALID	非保留	非保留	非保留	
VALIDATE	非保留			
VALIDATOR	非保留			
VALUE	非保留	保留	保留	保留
VALUES	非保留（不能是函数或类型）	保留	保留	保留
VALUE_OF		保留		
VARBINARY		保留	保留	
VARCHAR	非保留（不能是函数或类型）	保留	保留	保留
VARIADIC	保留			
VARYING	非保留	保留	保留	保留
VAR_POP		保留	保留	
VAR_SAMP		保留	保留	
VERBOSE	保留（可以是函数或类型）			
VERSION	非保留	非保留	非保留	
VERSIONING		保留		
VIEW	非保留	非保留	非保留	保留
VIEWS	非保留			
VOLATILE	非保留			
WHEN	保留	保留	保留	保留
WHENEVER		保留	保留	保留
WHERE	保留	保留	保留	保留
WHITESPACE	非保留	非保留	非保留	
WIDTH_BUCKET		保留	保留	
WINDOW	保留	保留	保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
WITH	保留	保留	保留	保留
WITHIN	非保留	保留	保留	
WITHOUT	非保留	保留	保留	
WORK	非保留	非保留	非保留	保留
WRAPPER	非保留	非保留	非保留	
WRITE	非保留	非保留	非保留	保留
XML	非保留	保留	保留	
XMLAGG		保留	保留	
XMLATTRIBUTES	非保留（不能是函数或类型）	保留	保留	
XMLBINARY		保留	保留	
XMLCAST		保留	保留	
XMLCOMMENT		保留	保留	
XMLCONCAT	非保留（不能是函数或类型）	保留	保留	
XMLDECLARATION		非保留	非保留	
XMLDOCUMENT		保留	保留	
XMLELEMENT	非保留（不能是函数或类型）	保留	保留	
XMLEXISTS	非保留（不能是函数或类型）	保留	保留	
XMLFOREST	非保留（不能是函数或类型）	保留	保留	
XMLITERATE		保留	保留	
XMLNAMESPACES	非保留（不能是函数或类型）	保留	保留	
XMLPARSE	非保留（不能是函数或类型）	保留	保留	
XMLPI	非保留（不能是	保留	保留	

关键词	UXDB	SQL:2016	SQL:2011	SQL-92
	函数或类型）			
XMLQUERY		保留	保留	
XMLROOT	非保留（不能是函数或类型）			
XMLSCHEMA		非保留	非保留	
XMLSERIALIZE	非保留（不能是函数或类型）	保留	保留	
XMLTABLE	非保留（不能是函数或类型）	保留	保留	
XMLTEXT		保留	保留	
XMLVALIDATE		保留	保留	
YEAR	非保留	保留	保留	保留
YES	非保留	非保留	非保留	
ZONE	非保留	非保留	非保留	保留