优炫数据库uxmpp使用手册 2.1

UXSINO
优 炫 软 件

优炫数据库uxmpp使用手册 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

表格清单

前言

1. 文档目的

本文档介绍了uxmpp的部署和使用。

2. 文档对象

- 数据库管理员
- 开发工程师
- 测试工程师
- 技术支持工程师

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	2023-01-12	第一次正式发布。

第 1 章 引言

随着海量数据时代的发展，海量数据的管理能力，类型多，变化快，成本低，高可用，可扩展性等需求给企业带来了巨大的挑战。

大数据呈指数增长。结构化数据增长基本可控，随业务增长是线性关系；而对非结构化数据，尤其语音、图像、视频，增长巨大。精准营销、风险控制、运营等对大数据的应用提出更高的要求。

着眼未来，为更好地利用大数据领域新兴技术构造大数据平台，应对市场变幻、带动业务模式创新，寻求新的技术方向已成为必然选择。同时近年来信息安全问题越来越受中国政府和企业的关注，关键领域信息系统的国产化逐步开展。基于信息安全、技术革新等多方面考虑，坚定地走国产化替代道路。

为了应对海量数据时代，优炫软件通过实际应用进行研究、分析，在架构设计、资源管理、功能实现等诸多方面，最终形成了一个适用于PB级的大数据计算的数据库产品——优炫大规模并行处理数据库。

系统架构高可扩展，性能随着节点数的增加而提升，保证客户接入更全面的业务数据，满足市场营销、内部管理、内外监管的分析需求。为用户提供海量数据存储、管理能力，进一步降低客户数据仓库建设的成本，并进一步提升系统性能。

第 2 章 概述

- 概念

  优炫大规模并行处理uxmpp（UX Massive Parallel Process）是基于UXDB在多台机器上的横向扩展。这些服务器可以将传入的SQL查询并行化，以便在大型数据集群上实现实时响应。主要功能：1.以插件的形式对底层数据库的扩展，并非数据库的分支；2.通过分片和复制跨多个机器扩展；3.查询并行化的分布式引擎；4.用于扩展多租户应用程序的数据库。

- 使用场景

  - 多租户数据库

    大多数多租户应用程序已经有了租户或客户的概念，并将其构建到数据模型中。在此模型中，数据库为多租户提供服务，每个租户的数据都与其他租户分开。uxmpp支持对此模型提供标准的SQL，支持将关系型数据库扩展到100K+租户。uxmpp还为多租户添加了新功能，例如租户隔离，为大型租户提供性能保证；运用参考表的概念减少租户之间的数据重复。这些功能支持在多台机器上扩展租户数据，能够增加更多CPU、内存和磁盘资源。此外，跨多个租户共享相同数据库模式，能够有效利用硬件资源、方便数据库管理。

  - 实时分析

    uxmpp支持对大型数据集实时查询分析，例如具有亚秒级响应时间的监控图表和当前事物中数据进行探索性分析。

- 注意事项

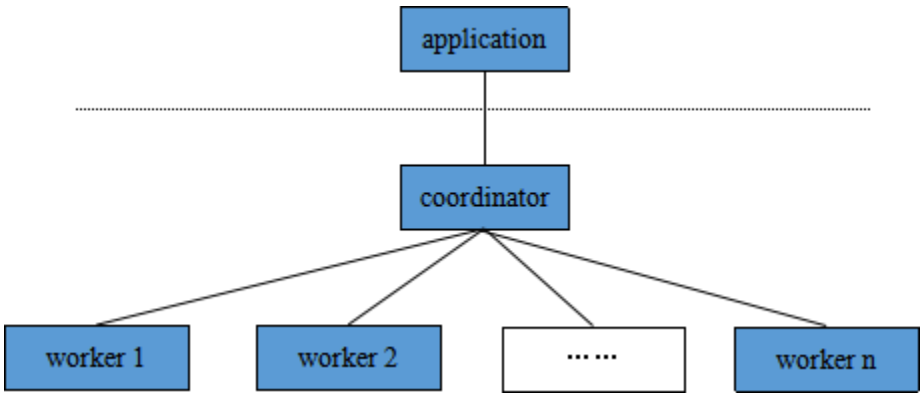
  uxmpp只扩展UXDB分布式功能，并非扩展UXDB所有功能，因此，uxmpp并非适用于所有UXDB集群。选择使用uxmpp的集群需要考虑数据模型、工具和SQL的选择。如果使用的数据模型是B2B或者实时分析，并且不考虑其他工具和特殊SQL的支持，那么适合使用uxmpp集群。

第 3 章 基本概念与原理

3.1. 分布式架构

uxmpp采用分布式计算架构SN（shared nothing）。多台数据库服务器（节点）形成一个集群，每个节点都是独立、自给的。在系统中不存在单点竞争、没有节点共享存储和磁盘。UXDB可以保存更多数据，比单台机器使用更多CPU，还可以添加更多节点来扩展数据库。

集群中的节点分为两类：一个协调节点（master或者coordinator）和若干工作节点（worker）。应用程序（application）将查询发送给协调节点，协调节点再将查询发送到工作节点上，并收集返回的结果。



对于每个查询，如果数据在单个节点上，那么协调节点将其路由到单个工作节点；如果数据跨多个节点上，那么协调节点将其并行路由到多个节点上。

> **注意**
>
> uxmpp支持直接在worker节点上直接进行读写，但不建议在worker节点上直接对跨节点的数据进行操作，这有可能会导致数据错乱。单节点的数据没有影响。

3.2. 表类型

uxmpp集群中包含以下三种类型的表。

- 分布式表

  分布式表是uxmpp中最常见的表，将一个表的数据跨多个worker节点水平分区。uxmpp使用分片算法将数据行分配给worker节点的碎片，每个节点碎片的总和组成这个节点的所有数据，所有节点的总和组成完整的表的数据。分片需要有一个特定的列值来做标志，用以对这些行进行分配，这个特定的列就叫做分布列，要在分发表的时候指定此列。

- 参考表

  参考表也算是一种分布式表，它的所有数据都集中在一个worker节点上的一个碎片上，其他碎片或worker节点上的数据都是它的复制，从任何一个worker节点上都可以访问到表的完整数据，因此不会产生从其他worker请求的网络开销。参考表不用区分每行形成碎片，因此不需要分布列。参考表通常比较小，用于存储和其他worker节点上有关联查询的数据。

- 本地表

  使用uxmpp实际上是在普通的UXDB数据库的协调节点和与之交互的工作节点安装uxmpp扩展，因此，可以在这些节点上创建不进行分片的普通表，比如一些不参与连接查询的小型管理表，这就是本地表。实际上uxmpp本身也是使用本地表来保存集群元数据。

3.3. 分片

一个分片（shard）指该分布式表在worker节点上的某个小表，即分布式表行的子集。根据分布式规则，一个worker上一般有多个分片（shards）。ux_dist_shard可查看到表的分片信息。

```
uxdb=# select * from ux_dist_shard;
 logicalrelid | shardid | shardstorage | shardminvalue | shardmaxvalue
--------------+---------+--------------+---------------+---------------
 employee     |  102572 | t            | -2147483648   | -1073741825
 employee     |  102573 | t            | -1073741824   | -1
 employee     |  102574 | t            | 0             | 1073741823
 employee     |  102575 | t            | 1073741824    | 2147483647
(4 行记录)
```

> **提示**
>
> 分布式表的创建和分片设置请参见第 4.3.1 节 “创建”和第 5.1 节 “分片数量”。

上例中employee有4个分片，那么数据就会分别写入对应的分片中，可直接从该分片中读取对应的数据。至于这些分片到worker上的映射，即放在哪个worker上，是通过系统的元数据表决定的，可以通过查询表ux_dist_placement和ux_dist_node得出。

```
SELECT shardid,node.nodename,node.nodeport
  FROM ux_dist_placement placement
  JOIN ux_dist_node node
    ON placement.groupid = node.groupid
   AND node.noderole = 'primary'::noderole
 WHERE shardid = 102572;
shardid |   nodename   | nodeport
--------+--------------+----------
102572  | 192.168.1.83 |   5432
```

> **提示**
>
> uxmpp支持复制分片，以防止数据丢失。

- 分片的意义

  根据需要将分片及其副本放在不同的节点上，将包含相关表的相关行的分片放在同一个节点上。这样，它们之间的连接查询可以避免更多的网络消耗，在单节点上就可以执行，提高处理能力。

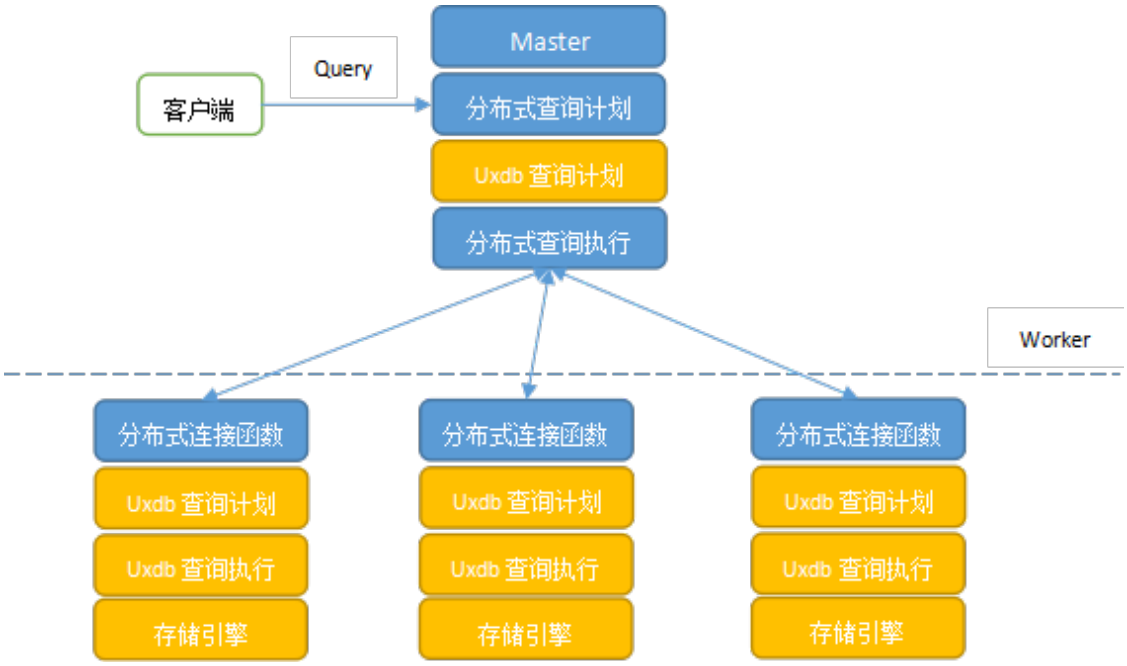
  例如一个具有库存、产品和销售的数据库，如果这三个表都有一个商品id作为分发列进行分片，那么当限制为某一个商品的时候，对于该商品的所有查询可以直接在单个worker上高效的运行，包括这些表之间的联合查询。

另外，跨多台计算机分布查询，可以让多台机器同时执行查询。并且可以通过增加新机器来提高处理速度。实现最大并行性，即最大CPU利用率。

虽然将表分布在多个节点上，以达到查询可以“实时”运行。但是，查询结果仍然需要通过coordinator（或master）节点来进行分发请求命令和收集结果并传回，因此当查询是计数或者统计等聚会函数时，这个加速是最明显的。

3.4. 查询处理

uxmpp集群由master实例和多个worker实例组成。master上存储有关分片的元数据，worker上存储分片数据和复制数据。发布到集群上的所有查询通过master执行，master将查询分为较小的查询片段，其中每个查询片段可在分片上独立运行；然后，master将查询片段分配给worker，监督其执行，合并其结果，并最终返回给用户。



uxmpp的查询涉及到两个组件：分布式查询计划和执行程序、uxdb查询计划和执行程序。

3.4.1. 分布式查询计划程序

uxmpp的分布式查询计划程序可对SQL进行查询并计划分布式执行。

对于SELECT查询，计划程序首先创建输入查询的计划树，并将其转化为可交换和关联的形式，以便于并行化。它还应用了一些其他优化，以可伸缩方式执行查询，并最大限度减少网络I/O。

接下来，程序将查询分为两部分，在master运行的查询和在各个worker上运行的查询片段。master将这些查询片段分配给worker，以便有效利用它们的所有资源。在此之后，将分布式查询计划传递给分布式执行程序以供执行。

3.4.2. 分布式查询执行程序

uxmpp分布式查询执行程序运行分布式查询计划并处理查询执行期间发生的故障。执行程序连接worker节点，将执行任务分配给它们并监督它们执行。如果执行程序无法将任务分配给指定

worker执行程序或者任务执行失败，那么执行程序会将该任务重新动态分配给其他worker副本。执行程序处理故障时，只处理查询失败的查询子树，而非处理整个查询。

uxmpp默认执行器是adaptive，默认开启。adaptive能快速响应涉及过滤，聚合和亲合连接的查询，也能很好适应具有完全SQL支持的单租户查询。根据需要，为每个分片建立一个连接，将所有的查询分片分配到这些连接上，然后取回各查询分片的查询结果，合并查询结果，最终将查询结果返回用户。

> **提示**
>
> 运行UXDB的EXPLAIN命令可以查看查询执行过程。

第 4 章 uxmpp的部署和使用

本章介绍如何在linux上进行uxmpp的部署和使用。

4.1. 部署

4.1.1. 手动部署

表 4.1. 环境信息

	IP address	备注
master	192.168.1.82	主节点，进行数据操作
worker1	192.168.1.83	worker节点，数据分布节点
worker2	192.168.1.84	worker节点，数据分布节点
……		worker节点，数据分布节点

> **注意**
>
> 各节点的uxdb服务器版本、操作系统、时区时间均应保持一致，且各节点网络相通。本文示例操作系统均为CentOS7.4。

- 安装UXDB

  1. 分别在master和worker节点上安装uxdb（具体安装过程请参见《优炫数据库安装手册V2.1》）。
  2. 分别给master和worker节点加载license（联系优炫相关技术人员获取license）。
  3. 分别在master和worker节点上进入uxdb安装目录的dbsql/bin目录下，初始化集群：./initdb -W -D mpptest。

- 配置uxmpp

  1. 修改master和各worker节点的集群配置文件uxsinodb.conf，打开shared_preload_libraries开关，并添加uxmpp。

```
150 #       like uxAdmin-III, EDB enterprise manager etc.
151 shared_preload_libraries = 'postgres_adaptor,uxmpp'
152
```

> **注意**
>
> 当加载多个改变规划器或执行器行为的插件时，请确保uxmpp是第一个被加载的。上图所示postgres_adaptor没有改变规划器行为，所以允许放在uxmpp之前。

  2. 在master节点上创建.uxpass，并写入worker节点的信息，包括worker节点的IP（不能省略localhost）、集群port、搭建uxmpp的数据库名、用户名、密码。

     touch ~/.uxpass

chmod 0600 ~/.uxpass
vi ~/.uxpass

```
#hostname:port:database:username:password
localhost:5432:uxdb:uxdb:123456
192.168.1.82:5432:uxdb:uxdb:123456
192.168.1.83:5432:uxdb:uxdb:123456
192.168.1.84:5432:uxdb:uxdb:123456
```

> **注意**
>
> 在.uxpass文件中，前两行master节点的连接信息可不写。如果不写，在连接数据库时，会提示输入密码。

3. 在worker1和worker2上分别创建.uxpass，并写入相关信息，如下图（不能省略localhost）。操作命令和master相同。

```
#hostname:port:database:username:password
localhost:5432:uxdb:uxdb:123456
192.168.1.83:5432:uxdb:uxdb:123456
192.168.1.84:5432:uxdb:uxdb:123456
```

4. 分别启动master和worker节点上的集群。

   ./ux_ctl -D mpptest start

5. 分别在master和worker节点以需要搭建uxmpp的用户登录数据库控制台。

   ./uxsql -p 5432 -d uxdb -U uxdb

6. 分别在master和worker节点上控制台中加载uxmpp插件。

   create extension uxmpp;

```
uxdb=# create extension uxmpp;
CREATE EXTENSION
uxdb=# \dx
                 List of installed extensions
  Name   | Version |   Schema   |         Description
---------+---------+------------+------------------------------
 pluxsql | 1.0     | ux_catalog | PL/uxSQL procedural language
 uxmpp   | 7.3-3   | ux_catalog | UXmpp distributed database
(2 rows)
```

7. master节点登录控制台添加worker节点。

select master_add_node('192.168.1.83',5432);
select master_add_node('192.168.1.84',5432);

```
uxdb=# create extension uxmpp;
CREATE EXTENSION
uxdb=# select master_add_node('192.168.1.83',5432);
 master_add_node
-----------------
               1
(1 row)

uxdb=# select master_add_node('192.168.1.84',5432);
 master_add_node
-----------------
               2
(1 row)
```

8. master节点上查看在线的worker节点。

select master_get_active_worker_nodes();

```
uxdb=# select master_get_active_worker_nodes();
 master_get_active_worker_nodes
--------------------------------
 (192.168.1.84,5432)
 (192.168.1.83,5432)
(2 行记录)
```

> **提示**
>
> 验证uxmpp环境是否搭建成功，可在master上创建一个分布表（创建分布表请参见第 4.3.1 节 “创建”），在worker节点上查看是否分布成功。

4.1.2. 自动部署

自动部署可以简化部署过程，提高部署效率，减少部署过程中出现的人为错误。

示例节点情况如下：包含一个协调节点，一个执行节点，一个数据节点。

表 4.2. 环境信息

	IP address	备注
Coordinator	192.71.0.204	管理节点，对各个节点的集群进行部署和配置。可以和其他节点部署在同台机器
executor1	192.71.0.203	执行节点，可进行DML操作
……	……	……

	IP address	备注
datanode1	192.71.0.204	数据分布节点
……	……	……

> **注意**
>
> 各节点的uxdb服务器版本、操作系统、时区时间均应保持一致，且各节点网络相通。本文示例操作系统均为CentOS7.4。

1. 安装ansible

   可通过yum或者rpm包进行安装。

   yum install ansible -y

   查看ansible是否安装成功。

```
[uxdb@uxdev204 ~]$ ansible --version
/usr/lib/python2.7/site-packages/ansible/parsing/vault/__init__.py:44: CryptographyDeprecationWarn
, and will be removed in the next release.
  from cryptography.exceptions import InvalidSignature
ansible 2.9.27
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/home/uxdb/.ansible/plugins/modules', u'/usr/share/ansible/pl
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Jun 28 2022, 15:30:04) [GCC 4.8.5 20150623 (Red Hat 4.8.5-44)]
```

   在所有其他节点都安装ansible，步骤相同。

2. ssh免密配置认证

   以uxdb用户生成公钥，Coordinator节点执行ssh-keygen -t rsa（生成过程中默认回车）。

```
[uxdb@localhost ~]$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/uxdb/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/uxdb/.ssh/id_rsa.
Your public key has been saved in /home/uxdb/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:29KPK3vT2FexZrdQBFNWKJ8PK5PtCM6V7g99ZszSN1I uxdb@localhost.localdomain
The key's randomart image is:
+---[RSA 2048]----+
|             oo+o|
|            . +. |
|             o.. |
|              +o |
|        S    +.Eo|
|         +. *o+B+|
|        oooB.**o@|
|        ..*o=.+*o|
|        .+o+oo.  |
+----[SHA256]-----+
```

   同步公钥文件id_rsa.pub到目标主机，如下所示。

ssh-copy-id -i /home/uxdb/.ssh/id_rsa.pub uxdb@192.71.0.204
ssh-copy-id -i /home/uxdb/.ssh/id_rsa.pub uxdb@192.71.0.203

验证免密配置是否成功：Coordinator节点上uxdb用户分别执行ssh 192.71.0.204、ssh 192.71.0.203，免密连接则为配置成功。

在其他节点也进行ssh免密配置认证，操作同Coordinator节点。

为各节点root用户同样配置ssh免密环境。

3. 协调节点安装UXDB

在Coordinator节点安装uxdb（具体安装过程请参见《优炫数据库安装手册 V2.1》）。

4. 部署配置

配置各节点信息。

进入Coordinator节点uxdb安装路径下的deploy目录，修改inventory.ini文件，配置各节点IP、集群名称和端口号。

```
## UXDB engine
## 变量重复定义时的生效规则:
## 节点变量留空则默认使用全局变量，否则会覆盖全局变量
##
#############################################################################
### the master node
[uxdb_coordinators]
uxdb_coordinator  ansible_host=192.168.102.100  clustername=uxdata  listen_port=5432

### the data nodes
[uxdb_datanodes]
uxdb_datanode101  ansible_host=192.168.102.101  clustername=uxdata  listen_port=5432
uxdb_datanode102  ansible_host=192.168.102.102  clustername=uxdata  listen_port=5432  installdir=/home/uxdb/datainstall

### the executors
[uxdb_executors]
uxdb_executor103  ansible_host=192.168.102.103  clustername=uxdata  listen_port=5432
uxdb_executor104  ansible_host=192.168.102.104  clustername=uxdata  listen_port=5432  installdir=/home/uxdb/execinstall
```

在自动化部署工具中，各种节点的区别如下所示。

- [uxdb_coordinators]：协调节点，即master节点，不存放实际数据，可以执行DDL，DML
- [uxdb_datanodes]：数据节点，存放分片表等实际数据
- [uxdb_executors]：执行节点，既可以存放分片数据，也能执行DML

表 4.3. 部署配置字段说明

名称	说明
uxdb_coordinator	节点名称
ansible_host	节点ip
clustername	集群名称
listen_port	集群端口
Installdir	具体安装路径

修改bootstrap.yml文件，根据实际NTP server修改，如Coordinator节点做NTPserver。

```
become_method: sudo

roles:
    - { role: ntp, ntp_server: 10.10.10.101 }

tasks:
- name: ensure ntp is at the latest version
```

启动NTP时间同步服务，如下所示。

ansible-playbook --verbose bootstrap.yml -k -u root

修改group_vars/all.yml文件，配置数据库安装路径和执行路径，如下所示。

```
#### CAUTION: Both of control machine and the orchastrated machines
####          the location of our install file is
####          {{ deploy_path }}/{{ installfile }}
deploy_path                : /home/uxdb
installfile                : uxdb-install.tar.gz

#### The uxdbinstall directory to be packaged and archived
#### The directory from where we execute initdb/ux_ctl/removedb/uxsql
#### relative to {{ deploy_path }}
archivedir                 : /home/uxdb/uxdbinstall
installdir                 : /home/uxdb/uxdbinstall
enable_xtfs                : False
```

表　4.4.　配置数据库安装路径和执行路径字段说明

名称	说明
installdir	数据库在各个节点的默认安装路径
archivedir	分布式安装包的原路径，即已经安装好的主节点数据库路径

修改group_vars/uxdb.yml文件配置集群信息，如下所示。

```
### database engine parameters
ux_data                     : uxsino
clustername                 : uxdata
default_password            : 123456
listen_port                 : 5432
shared_preload_libraries    : "'uxmpp, postgres_adaptor'"
uxmpp_replication_model     : "'streaming'"

### Database name used during automatic deployment
Database                    : uxdb
### Database administrator name used during automatic deployment
Administrator               : uxdb
### Database running mode, support [standard] and [compatible]
running_mode                : standard
```

表 4.5. 配置集群信息字段说明

名称	说明
ux_data	集群存放路径
clustername	集群名（会被inventory.ini中的设置覆盖）
default_password	默认密码
listen_port	集群启动端口（会被inventory.ini中的设置覆盖）
shared_preload_libraries	加载扩展
uxmpp_replication_model	复制模式
Database	数据库名
Administrator	管理员名
running_mode	运行模式

注意

自动化工具默认为部署标准数据库集群，如果希望部署兼容模式集群，需要修改配置文件uxdb.yml如下所示。

Database: UXDB
Administrator: UXDB
running_mode: compatible

5. 分布式安装uxdb

分布式安装数据库，如下所示。

sudo ansible-playbook --verbose deploy.yml
或
cd /home/uxdb/uxdbinstall/deploy/uxmpp
sudo ./deploympp -d

在各个节点上可以查看到uxdb已经从协调节点拷贝安装。

> **注意**
>
> 分布式安装和卸载都需要sudo权限。

> **提示**
>
> deploy脚本下的配置文件，只需要在Coordinator节点上配置一次即可，可以将其备份至其他目录，后续再安装其他的版本时，就不需要重新配置了。

6. 集群操作

   如果数据库版本需要license，需要先在各个节点分别配置（license找优炫相关技术人员获取）。

   a. Coordinator节点上进入deploy目录，执行如下命令。

      初始化集群

      ansible-playbook --verbose initdb.yml

      启动

      ansible-playbook --verbose start.yml

      重启

      ansible-playbook --verbose restart.yml

      查询状态

      ansible-playbook --verbose status.yml

      停止

      ansible-playbook --verbose stop.yml

      删除

      ansible-playbook --verbose removedb.yml

b. 快捷命令

Coordinator节点上进入deploy目录，使用deploympp工具进行操作，具体使用方法如下所示。

```
[uxdb@uxdev204 uxmpp]$ ./deploympp -h
deploympp: deploy uxmpp
Usage:
       deploympp  -d|--deploy
                          分布安装，等同于: 'ansible-playbook -v deploy.yml'
       deploympp  -i|--initdb
                          初始化集群，等同于: 'ansible-playbook -v initdb.yml'
       deploympp  -a|--start
                          启动集群，等同于: 'ansible-playbook -v start.yml'
       deploympp  -o|--stop
                          停止集群，等同于: 'ansible-playbook -v stop.yml'
       deploympp  -r|--restart
                          重启集群，等同于: 'ansible-playbook -v restart.yml'
       deploympp  -u|--status
                          查看集群状态，等同于: 'ansible-playbook -v status.yml'
       deploympp  -m|--removedb
                          删除集群，等同于: 'ansible-playbook -v removedb.yml'
       deploympp  -p|--purge
                          清理节点，等同于: 'ansible-playbook -v purge.yml'
       deploympp  -v|--version
                          显示工具版本信息
       deploympp  -h|--help
                          显示工具帮助页面
```

4.1.3. 单节点部署

- 安装UXDB

1. 在单节点上安装uxdb（具体安装过程请参见《优炫数据库安装手册 V2.1》）。
2. 加载license（联系优炫相关技术人员获取license）。
3. 进入uxdb安装目录的dbsql/bin目录下，初始化集群。

./initdb -W -D mpptest

- 配置uxmpp

1. 修改集群配置文件uxsinodb.conf，打开shared_preload_libraries开关，并添加uxmpp。
2. 启动集群。

./ux_ctl -D mpptest start

3. 搭建uxmpp的用户登录数据库控制台。

./uxsql -p 5432 -d uxdb -U uxdb

4. 在控制台中加载uxmpp插件。

create extension uxmpp;

如果需要验证uxmpp环境是否搭建成功，可以创建一个分布表，创建成功，则搭建成功。

4.2. 升级

以uxdb-server-linux7-2.0.4.10-standard版本为例，将uxmpp从7.3升级到8.2（下述操作步骤在master和各worker节点都要执行）。

1. 获取uxmpp_8.2_v2.0.4.10.tar.gz压缩包并解压。

tar xvf uxmpp_8.2_v2.0.4.10.tar.gz

2. 进入安装包路径内，安装高版本uxmpp。

cd uxmpp_8.2_v2.0.4.10/
./install.sh -p /home/uxdb/uxdbinstall

```
[uxdb@localhost ~]$  cd uxmpp_8.2_v2.0.4.10/
[uxdb@localhost uxmpp_8.2_v2.0.4.10]$ ./install.sh -p /home/uxdb/uxdbinstall
Installing uxmpp...
Done.
```

3. 进入uxdb安装目录下的dbsql/bin目录下，重启dbserver。

cd /home/uxdb/uxdbinstall/dbsql/bin
./ux_ctl -D mpptest restart

```
[uxdb@localhost uxmpp_8.2_v2.0.4.10]$ cd /home/uxdb/uxdbinstall/dbsql/bin
[uxdb@localhost bin]$ ./ux_ctl -D mpptest restart
waiting for server to shut down.... done
server stopped
waiting for server to start.....2019-06-04 16:45:34.057 CST [73160] LOG:  Load License file successfully!
2019-06-04 16:45:34.101 CST [73160] LOG:  loaded library "postgres_adaptor"
2019-06-04 16:45:34.212 CST [73160] LOG:  number of prepared transactions has not been configured, overriding
2019-06-04 16:45:34.212 CST [73160] DETAIL:  max_prepared_transactions is now set to 200
2019-06-04 16:45:34.212 CST [73160] LOG:  loaded library "uxmpp"
2019-06-04 16:45:34.216 CST [73160] LOG:  listening on IPv4 address "0.0.0.0", port 5432
2019-06-04 16:45:34.216 CST [73160] LOG:  listening on IPv6 address "::", port 5432
2019-06-04 16:45:34.225 CST [73160] LOG:  listening on Unix socket "/tmp/.s.UXSQL.5432"
2019-06-04 16:45:34.248 CST [73160] LOG:  redirecting log output to logging collector process
2019-06-04 16:45:34.248 CST [73160] HINT:  Future log output will appear in directory "log".
uxmaster status ready
 done
server started
```

4. 登录数据库控制台并查看升级前uxmpp版本。

./uxsql -U uxdb -d uxdb
\dx

```
[uxdb@localhost bin]$ ./uxsql -U uxdb -d uxdb

The license due date is: 2028-12-01 00:00:00
It's commercial license.
uxsql (10.0)
Type "help" for help.

uxdb=# \dx
                List of installed extensions
  Name   | Version |   Schema   |         Description
---------+---------+------------+-----------------------------
 pluxsql | 1.0     | ux_catalog | PL/uxSQL procedural language
 uxmpp   | 7.3-3   | ux_catalog | UXmpp distributed database
(2 rows)
```

5. 升级uxmpp并查看升级后uxmpp版本。

   alter extension uxmpp update;
   \dx

```
uxdb=# alter extension uxmpp update;
ALTER EXTENSION
uxdb=# \dx
                  List of installed extensions
  Name   | Version |   Schema   |         Description
---------+---------+------------+-------------------------------
 pluxsql | 1.0     | ux_catalog | PL/uxSQL procedural language
 uxmpp   | 8.2-2   | ux_catalog | UXmpp distributed database
(2 rows)
```

4.3. 分布式表的应用

4.3.1. 创建

- 分布表

1. 先定义一个表。

   CREATE TABLE github_events(
   event_id bigint,
   event_type text,
   event_public boolean,
   repo_id bigint,
   payload jsonb,
   repo jsonb,
   actor jsonb,
   org jsonb,
   created_at timestamp);

2. 使用create_distributed_table()函数指定表分发列并创建分片。

   SELECT create_distributed_table ('github_events','repo_id');

   此函数通知uxmpp通过repo_id列对表github_events进行分发。根据uxmpp.shard_count和uxmpp.shard_replication_factor的值在worker上创建分片。总共创建

uxmpp.shard_count个分片数，并根据uxmpp.shard_replication_factor的值进行复制，复制的副本具有和原表相同的表结构，包括模式、索引和约束。

每个分片都有一个唯一的分片ID，但是副本和原表具有相同的分片ID。每个分片在worker节点上是以名为”tablename_shardid”的常规表显示的，其中tablename是分布式表的名称，shardid是该分片的唯一ID。

> **提示**
>
> 前面使用的create_distributed_table创建分布式表适用于空表和非空表。在非空表中使用会自动对数据进行分布，并会有相应的提示。
>
> **CREATE TABLE series AS SELECT i FROM generate_series(1,1000000) i;**
> **SELECT create_distributed_table('series','i');**
>
> ```
> uxdb=# CREATE TABLE series AS SELECT i FROM generate_series(1,1000000) i;
> SELECT 1000000
> uxdb=# SELECT create_distributed_table('series', 'i');
> 注意:  Copying data from local table...
>  create_distributed_table
> --------------------------
>
> (1 行记录)
> ```

> **注意**
>
> 非空表的分布式会有数据迁移的操作，因此分发过程中禁止对该表写入数据。如果是挂起的写入，则在数据分发完成之后，按分布式处理。同时，读取也是按分布式处理。

3. 协同定位

协同定位是在相同的机器上保留相关信息，用以实现高效的关系操作，有效利用整个数据集的水平可伸缩性。

如需手动控制表的分组，那么在创建分布式表create_distributed_table时使用可选参数colocate_with。该参数默认为default，该值表示将具有相同的分发列类型、分片数和复制因子的表分为一组。如下所示，t1表和t2表分片表分发列类型相同，默认分片数和复制因子，故表t1和t2拥有相同的分组，例如a=1行和m=1的行放在同一个节点上。

CREATE TABLE t1(a int,b char(10));
CREATE TABLE t2(m int,n char(10));
SELECT create_distributed_table('t1','a');
SELECT create_distributed_table('t2','m');

如果需要每个表有自己单独的分组，则可以指定colocate_with参数值为none。

SELECT create_distributed_table ('t1','a',colocate_with => 'none');

如果共同定位多个表，可以先分配一个表，然后再将其他表放入该表的组中。如下所示。

CREATE TABLE test(id int, id2 int, t text);
CREATE TABLE test1(id int, id2 int, t text);

```
CREATE TABLE test2(id int, id2 int, t text);
SELECT create_distributed_table('test', 'id');
SELECT create_distributed_table('test1', 'id2', colocate_with=>'test');
SELECT create_distributed_table('test2', 'id2', colocate_with=>'test');
```

> 提示
>
> 表ux_dist_partition中可以看到哪些表分配给哪些组（colocationid字段），有关组的信息可以通过表ux_dist_colocation查看。

- 参考表

  上述方法是将表分配到多个水平分片中，另一种方法是将表分配到单个分片中并将分片复制到每个worker节点上，这种方式分发的表称为参考表。主要是为了存储多个节点频繁访问的数据。常见的参考表有：需要与较大的分布式表连接的小表；多租户应用中的缺少租户ID列或者与租户无关的表；跨多列的唯一约束的小表。使用create_reference_table()函数创建参考表。

```
SELECT create_reference_table('tablename');
```

4.3.2. 修改

uxmpp会自动传播多种DDL语句。在协调节点（master）上修改表，同时也会更新到对应worker节点上对应的分片表。当然，不是所有的DDL语句都可以传播，有一部分需要手动在各个节点上进行修改，还有一部分目前是不支持的，例如修改分发列。可以通过设置参数uxmpp.enable_ddl_propagation来启用或禁止自动传播DDL，默认是启用。

uxmpp自动传播大多数ALTER TABLE命令，如修改表名、修改列名、添加列、修改列、删除列、修改列类型，这些操作和在单机版uxdb数据库中是一样的。但是，不建议直接对分布列进行相关修改操作，且分布列不支持列的删除和类型修改。

使用uxmpp不影响数据库约束的使用，例如在分布式表上创建主键和外键。

```
//创建一个货物、订单、库存的关系表
CREATE TABLE goods(gid varchar(20),gname varchar(20),gprice int);
CREATE TABLE orders(oid varchar(20),gid varchar(20),onumber int);
CREATE TABLE stock(sid varchar(20),gid varchar(20),snumber int);
//添加主键（唯一约束必须在分发列之前创建）
ALTER TABLE goods ADD PRIMARY KEY (gid);
ALTER TABLE orders ADD PRIMARY KEY (oid,gid);
ALTER TABLE stock ADD PRIMARY KEY (sid,gid);
//分发表
SELECT create_distributed_table ('goods','gid');
SELECT create_distributed_table ('orders','gid');
SELECT create_distributed_table ('stock','gid');
//添加外键
ALTER TABLE orders ADD CONSTRAINT goods_orders_fk FOREIGN KEY(gid)
 REFERENCES goods(gid);
ALTER TABLE stock ADD CONSTRAINT goods_stock_fk FOREIGN KEY(gid) REFERENCES
 goods(gid);
//非空约束
ALTER TABLE orders ALTER COLUMN onumber SET NOT NULL;
```

uxmpp支持添加和删除索引。

CREATE INDEX date_idx ON orders USING BRIN(oid);
DROP INDEX date_idx ;

由于添加索引时会锁定表，所以在uxmpp适用的多租户模型中是不可取的。为了避免这种问题，可以使用下述方法创建索引。虽然可能会花费较多时间，但是不会锁定表，因此，对于多租户模型来说是非常有用的。

CREATE INDEX CONCURRENTLY date_idx ON orders USING BRIN(oid);

4.3.3. 操作数据

- 插入

  要将数据插入分布式表，可以使用标准INSERT命令。

  INSERT INTO test VALUES (0,0,'uxsino');
  INSERT INTO test VALUES (1,1,'agent'),(2,2,'mpp');

  给分布式表插入行时，必须指定分发列，即分发列不能为空。数据会根据分发列将数据路由到对应分片上。

  还支持批量加载，可以直接使用uxdb的copy命令：

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

  > **注意**
  >
  > 分片没有快照隔离的概念，那么当在copy的同时进行select，可能会在某些分片上看到copy的数据，但在其他分片上可能没有。如果copy无法连接到其他分片，那么它的行为相当于insert，如果连接到分片发生故障，则回滚事务，不会对元数据进行更改。

  > **提示**
  >
  > 在很多uxmpp数据模型下，对大量数据进行快速查询，需要亚秒级反应。那么快速查询的一种方法就是可以提前计算和保存聚合。例如重复执行一个聚合查询的时，它必须遍历每个相关的行并进行重新计算整个数据集的结果，那么将数据分别汇总到每小时或者每天进行保存，这样就避免了再运行的时候处理原始数据的成本。当聚合的汇总足够多的时候，且不在需要完整的详细信息时候，旧数据则可以被删除，这样也会节省存储空间。

  综上所述，需要注意：

  - 查询和插入表由类似的列分发
  - 选择查询时，条件应该带分布列
  - 插入必须包括分布列

- 更新

```
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
   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 ] [, ...] ]
```

> **注意**
>
> 更新分布式表时，不能修改分布式列。

- 删除

```
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
   [ USING using_list ]
   [ WHERE condition | WHERE CURRENT OF cursor_name ]
   [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

4.3.4. 查询

uxmpp是一个扩展，扩展了UXDB以实现分布式。那么可以在master上使用标准的UXDB的SELECT查询。uxmpp可以并行化复杂的选择、分组、排序以及JOIN的SELECT查询，以加快查询性能。uxmpp将SELECT查询分成小的查询片段分配给下面的worker，并将其结果合并返回给用户。

- 聚合查询

  uxmpp支持并行化uxdb的大多数聚合函数。

  uxmpp以多种方式支持count（distinct）聚合。如果count（distinct）聚合在分发列上，则可以直接下发到worker；如果没有在分发列上，那么将会在每个worker上运行select distinct语句，然后将结果返回给master，在master上进行最终的count。

> **注意**
>
> 对于包含多个count（distinct）聚合查询时，出具传输会很慢，如下所示。
>
> **SELECT count(distinct a), count(distinct b), count(distinct c) FROM table_abc;**

- join查询

  uxmpp支持任意数量的表之间的equi-join，并且可以不考虑表的大小和分布方式。查询规划器会根据表的分布方式选择最优的连接方法和连接顺序。并创建一个连接计划，用以使用最小的网络资源传输数据。

当两个表是协同表的时候，它们可以在公共分布列上实现最高效的join连接。这也是分布式表中join连接的最优方法。

> **注意**
>
> join连接要确保表分布到相同数量的分片中，且分发列类型有相应的类型。如果连接不同类型的列（如int和bigint）可能会出现问题。

参考表可用作“维度”表，以便与大表连接。由于参考表是在所有worker节点上完全复制，因此可以将与参考表的连接分解为每个worker上的本地连接，并且可以并行执行。这类似于协同表连接的升级版，因为参考表不用在任何特定的列上分发，并且可以用任何列进行join连接。

4.4. SQL支持

由于uxmpp通过UXDB提供分布式功能，因此，可以使用UXDB自身提供的工具和特性用于分布式表。

uxmpp支持分布式表上的大部分SQL查询，以下除外：

- 相关子查询
- WITH RECURSIVE(递归)
- TABLESAMPLE
- SELECT ... FOR UPDATE/SHARE
- 分组集合（GROUPING SETS、CUBE、ROLLUP）

uxmpp并非支持所有场景，所以用户在构建应用集群时应该先考虑实际业务场景，目前比较适合的是多租户和实时分析两个场景。

多租户应用程序中，当通过分布列将查询过滤到单个租户的时候，所有的SQL都可以支持。在一般多租户场景下，基本可以支持所有该场景下使用的SQL语句。

在实时分析的场景中，通过跨节点查询，大部分SQL都是支持的，跨节点 SQL 查询的具体限制如下所示。

- SELECT … FOR UPDATE仅适用于单分片查询。
- TABLESAMPLE仅适用于单分片查询。
- 仅当相关位于Distribution Column时，才支持相关子查询。
- 分布式表之间的外连接仅在 分布列上受支持。
- 仅当分布式表在外侧时，才支持分布式表和引用表或本地表之间的外连接。
- 递归 CTE仅适用于单分片查询。
- 分组集仅适用于单分片查询。

可以使用cte和临时表两种方式，示例如下。

- CTE解决限制

SELECT * FROM ref LEFT JOIN dist USING (value);
ERROR: cannot pushdown the subquery
DETAIL: There exist a reference table in the outer part of the outer join

要解决此限制，可以通过将部分分布式包装在CTE中，将查询转换为路由器查询。

WITH x AS (SELECT * FROM dist WHERE dist.value > 10)
SELECT * FROM ref LEFT JOIN x USING (id);

但是协调节点会将CTE结果发送给所有需要处理的节点。因此，最好是尽可能向内部查询添加最具体的筛选条件和限制条件，或者聚合表。这么操作可以减少此类查询可能导致的网络开销。

- 临时表解决限制

即使使用CTE执行，仍会有一些查询不受支持。例如分布式表上使用分组集。创建一个名为github_events的表，由列user_id分发，为查找出最早的管理事件。根据事件类型和事件公开字段组合进行查询即可。但是，如上所述，分布式查询中还不支持此SQL。

SELECT repo_id, event_type, event_public,
grouping(event_type, event_public),
min(created_at)
FROM github_events
WHERE repo_id IN (8514, 15435, 19438, 21692)
GROUP BY repo_id, ROLLUP(event_type, event_public);
ERROR:could not run distributed query with GROUPING
HINT:Consider using an equality filter on the distributed table's partition column.

解决这个限制，可以使用临时表，如下所示。

//grab the data, minus the aggregate, into a local table
CREATE TEMP TABLE results AS (
SELECT repo_id, event_type, event_public, created_at
FROM github_events
WHERE repo_id IN (8514, 15435, 19438, 21692)
);
//now run the aggregate locally
SELECT repo_id, event_type, event_public,
grouping(event_type, event_public),
min(created_at)
FROM results
GROUP BY repo_id, ROLLUP(event_type, event_public);

```
 repo_id |    event_type     | event_public | grouping |         min
---------+-------------------+--------------+----------+---------------------
    8514 | PullRequestEvent  | t            |        0 | 2016-12-01 05:32:54
    8514 | IssueCommentEvent | t            |        0 | 2016-12-01 05:32:57
   19438 | IssueCommentEvent | t            |        0 | 2016-12-01 05:48:56
   21692 | WatchEvent        | t            |        0 | 2016-12-01 06:01:23
   15435 | WatchEvent        | t            |        0 | 2016-12-01 05:40:24
   21692 | WatchEvent        |              |        1 | 2016-12-01 06:01:23
   15435 | WatchEvent        |              |        1 | 2016-12-01 05:40:24
    8514 | PullRequestEvent  |              |        1 | 2016-12-01 05:32:54
    8514 | IssueCommentEvent |              |        1 | 2016-12-01 05:32:57
   19438 | IssueCommentEvent |              |        1 | 2016-12-01 05:48:56
```

```
15435 |            |          |      3 | 2016-12-01 05:40:24
21692 |            |          |      3 | 2016-12-01 06:01:23
19438 |            |          |      3 | 2016-12-01 05:48:56
 8514 |            |          |      3 | 2016-12-01 05:32:54
```

在协调器上创建临时表是最后的手段。它受节点的磁盘大小和CPU的限制。

4.5. 函数

4.5.1. 创建分布式表函数

4.5.1.1. alter_columnar_table_set

- 功能

  alter_columnar_table_set()函数可以更改列式表上的设置。在非列式表上调用此函数会产生错误。除表名之外的所有参数都是可选的。

  若要查看所有列式表的当前选项，请参阅此表：

  SELECT * FROM columnar.options;

  可以使用以下参数覆盖新创建的表的列式设置的默认值。

  - columnar.compression
  - columnar.compression_level
  - columnar.stripe_row_limit
  - columnar.chunk_row_limit

- 参数

  表 4.6. alter_columnar_table_set 参数说明

名称	描述
table_name	列式表的名称。
chunk_row_limit	（可选）新插入的数据的每个chunk的最大行数。现有数据不会更改，并且可能具有超过此最大值的行数。默认值为10000。
stripe_row_limit	（可选）新插入的数据的每个列存条带的最大行数。现有的数据不会更改，并且可能具有超过此最大值的行数。默认值为150000。
compression	（可选）[none|pglz|zstd|lz4|lz4hc]新插入数据的压缩类型。不会重新压缩或解压缩现有数据。默认的和通常建议的值为 zstd（如果已编译支持）。

名称	描述
compression_level	（可选）有效设置介于 1 到 19 之间。如果压缩方法不支持所选的级别，则将选择最接近的级别。

- 返回值

  N/A

- 示例

```
SELECT alter_columnar_table_set(
'my_columnar_table',
compression => 'none',
stripe_row_limit => 10000);
```

4.5.1.2. alter_distributed_table

- 功能

  alter_distributed_table()函数可用于更改分布式表的分布列、分片计数或共置属性。

- 参数

  表 4.7. alter_distributed_table 参数说明

名称	描述
table_name	将要更改的分布式表的名称。
distribution_column	（可选）新分发列的名称。
shard_count	（可选）新分片计数。
colocate_with	（可选）当前分布式表将与之共置的表。设置值为default、none，用于启动新的共置组；或设置值为另一个表的名称，用于与之共置。
cascade_to_colocated	（可选）当此参数设置为“true”时，shard_count和colocate_with更改也将应用于以前与该表共置的所有表，并且将保留共置。如果它是“false”，则此表的当前共置将被破坏。

- 返回值

  N/A

- 示例

```
//修改分布列
SELECT alter_distributed_table('github_events', distribution_column:='event_id');
//修改共置组中所有表的分片数
SELECT alter_distributed_table('github_events', shard_count:=6,
 cascade_to_colocated:=true);
//修改共置组
SELECT alter_distributed_table('github_events', colocate_with:='another_table');
```

4.5.1.3. alter_old_partitions_set_access_method

- 功能

  在时间序列数据用例中，表通常按时间分区，旧分区压缩为只读列式存储。

- 参数

  表 4.8. alter_old_partitions_set_access_method 参数说明

名称	描述
parent_table_name	要更改其分区的（reg类）表。该表必须按日期、时间戳或时间戳类型的一列进行分区。
older_than	（时间戳）更改上限范围小于或等于older_than分区。
new_access_method	（名称）“堆”表示基于行的存储，或“列式”表示列式存储。

- 返回值

  N/A

- 示例

  CALL alter_old_partitions_set_access_method('foo', now() - interval '6 months','columnar');

4.5.1.4. alter_table_set_access_method

- 功能

  alter_table_set_access_method()函数更改表的访问方法（堆或列式）。

- 参数

  表 4.9. alter_table_set_access_method 参数说明

名称	描述
table_name	访问方法将更改的表的名称。
access_method	新访问方法的名称。

- 返回值

  N/A

- 示例

  SELECT alter_table_set_access_method('github_events', 'columnar');

4.5.1.5. create_distributed_function

- 功能

将函数从协调器节点传播到工作节点，并将其标记为分布式执行。当在协调器上调用分布式函数时，uxmpp 使用“分布参数”的值来选取一个工作节点来运行该函数。在工作节点上执行函数可提高并行度，并且可以降低延迟。

请注意，在分布式函数执行期间，uxdb搜索路径不会从协调器传播到工作线程，因此分布式函数代码应完全限定数据库对象的名称。此外，函数发出的通知也不会向用户显示。

- 参数

表 4.10. create_distributed_function 参数说明

名称	描述
function_name	要分发的函数的名称。该名称必须在括号中包含函数的参数类型，因为多个函数在uxdb中可以具有相同的名称。例如，'foo(int)'与'foo(int, text)'是不同的。
distribution_arg_name	（可选）要分发的参数名称。允许使用位置占位符，例如'$1'。如果未指定此参数，则函数function_name仅在工作节点上创建。如果将来添加工作节点，则该函数也将自动在新节点创建。
colocate_with	（可选）当分布式函数读取或写入分布式表时，请确保使用参数colocate_with命名该表。这可确保函数的每次调用都在包含相关分片的工作节点上运行。

- 返回值

N/A

- 示例

```
//一个更新表event_responses的示例函数
//该表是以列event_id为分布列的分布表
CREATE OR REPLACE FUNCTION
register_for_event(p_event_id int, p_user_id int)
RETURNS void LANGUAGE plpgsql AS $fn$
BEGIN
INSERT INTO event_responses VALUES ($1, $2, 'yes')
ON CONFLICT (event_id, user_id)
DO UPDATE SET response = EXCLUDED.response;
END;
$fn$;
//使用p_event_id参数将函数分发到工作节点
//确定每个调用影响哪个碎片
//并显式地与函数更新的表event_responses共享
SELECT create_distributed_function(
'register_for_event(int, int)', 'p_event_id',
colocate_with := 'event_responses'
);
```

4.5.1.6. create_distributed_table

- 功能

  create_distributed_table()函数用于定义分布式表，如果分布方法是哈希，则直接创建分片。此函数参数有表名、分布列和分布方法。其中分布方法为可选参数，默认是哈希分布，哈希分布会根据分片数和复制因子创建分片。如果有数据，并将数据自动分发到worker节点。

- 函数

  create_distributed_table('table_name','distributed_column','distributed_method',colocate_with=>'options');

- 参数

  表 4.11. create_distributed_table 参数说明

名称	描述
table_name	需要分发的表的名称。
distributed_column	需要分发的列。
distributed_method	（可选）分发方法。允许的值是append或hash，默认是hash。
colocate_with	（可选）将当前表包含在另一个表的共置位置组中。默认情况下，当表按相同类型的列分布、具有相同的分片计数和相同的复制因子时，表将共存。如果以后要中断此共置，可以使用update_distributed_table_colocation。colocate_with的合法值为：default、none（以启动新的共置组）、或另一个表的名称。
shard_count	（可选）要为新分布式表创建的分片数。指定shard_count时，不能将colocate_with的值指定为none以外的值。要更改现有表或共置组的分片计数，请使用alter_distributed_table 函数。

- 返回值

  N/A

- 示例

  假设给表github_event通过repo_id列进行分发。

  SELECT create_distributed_table('github_events','repo_id');

4.5.1.7. create_reference_table

- 功能

  create_reference_table()函数用于定义小型参考表或维度表。此函数的参数为表名，创建仅包含一个分片的分布式表，并复制到每个worker节点。

- 函数

  create_reference_table('table_name');

- 参数

表 4.12. create_reference_table 参数说明

名称	描述
table_name	需要分发的表的名称。

- 返回值

N/A

- 示例

假设将表test定义为参考表。

SELECT create_reference_table('test');

4.5.1.8. create_time_partitions

- 功能

create_time_partitions()函数创建给定间隔的分区以覆盖给定的时间范围。

- 参数

表 4.13. create_time_partitions 参数说明

名称	描述
table_name	要为其创建新分区的（regclass）表。该表必须按日期、时间戳或时间戳类型的一列进行分区。
partition_interval	在新分区上设置范围时要使用的时间间隔，如'2 hours'、'1 month'。
end_at	（时间戳）所创建分区的截止时间。最后一个分区将包含点end_at，并且不会创建以后的分区。
start_from	（时间戳，可选）第一个分区，包含点start_from。缺省值为now()。

- 返回值

如果需要创建新分区，则为 true；如果它们都已存在，则为 false。

- 示例

```
//在表foo中创建一年的月度分区
//以当前时间为起始时间
SELECT create_time_partitions(
 table_name         := 'foo',
 partition_interval := '1 month',
 end_at             := now() + '12 months'
);
```

4.5.1.9. drop_old_time_partitions

- 功能

  drop_old_time_partitions()函数删除间隔在给定时间戳之前的所有分区。除了使用此函数之外，还可以考虑alter_old_partitions_set_access_method使用列式存储压缩旧分区。

- 参数

  表 4.14. drop_old_time_partitions 参数说明

名称	描述
table_name	要为其删除分区的（reg类）表。该表必须按日期、时间戳或时间戳类型的一列进行分区。
older_than	（时间戳）删除上限小于或等于older_than的分区。

- 返回值

  N/A

- 示例

  //删除超过一年的分区
  CALL drop_old_time_partitions('foo', now() - interval '12 months');

4.5.1.10. master_create_empty_shard

- 功能

  master_create_empty_shard()函数可用于为append分发的表追加空的分片，使用该函数前，首先用create_distributed_table()函数将表按append方法分发，然后使用该函数进行空的分片追加。

- 函数

  master_create_empty_shard('table_name');

- 参数

  表 4.15. master_create_empty_shard 参数说明

名称	描述
table_name	要为其创建分片追加的append分布式表的表名。

- 返回值

  shard_id：返回新追加的分片的唯一ID。

- 示例

  假设给表append分布方式的分布表empty追加一个空分片。

```
SELECT * FROM master_create_empty_shard('empty');
master_create_empty_shard
---------------------------
                103846
(1 行记录)
```

4.5.1.11. mark_tables_colocated

- 功能

  mark_tables_colocated()函数可以将分布式源表，和其他分布式目标表列表，放入相同的共置组中。如果源表尚未在组中，则此函数将创建一个新的共置组，并将源表和目标表列表放入其中。

  通常，共置表应该在表分布时通过函数create_distributed_table的colocate_with参数完成。但如果有必要，mark_tables_colocated可以修改维护它。

  如果要中断表的共置，可以使用函数update_distributed_table_colocation。

- 参数

  表 4.16. mark_tables_colocated 参数说明

名称	描述
source_table_name	希望目标表与之共置的分布式表的名称。
target_table_names	分布式目标表的名称数组必须为非空。这些分布式表必须与源表的以下属性匹配：分布方法。、分布列类型、发布类型和分片计数。 如果没能和源表属性相匹配，uxmpp将引发错误。 例如，尝试共置分布列类型不同的表apples和oranges会导致： ERROR: cannot colocate tables apples and oranges DETAIL: Distribution column types don't match for apples and oranges.

- 返回值

  N/A

- 示例

  本示例将products和line_items放在与stores相同的共置组中。该示例假定这些表都分布在具有匹配类型的列上。

```
SELECT mark_tables_colocated('stores', ARRAY['products', 'line_items']);
```

4.5.1.12. remove_local_tables_from_metadata

- 功能

  remove_local_tables_from_metadata()函数从 uxmpp 的元数据中删除不再需要的本地表。（请参见uxmpp.enable_local_reference_table_foreign_keys(boolean)）。

通常，如果本地表位于 uxmpp 的元数据中，说明表和引用表之间存在外键。但是，如果enable_local_reference_foreign_keys禁用，uxmpp 将不再管理在这种情况下的元数据，并且不必要的元数据可以一直保留，除非手动清理。

- 参数

  N/A

- 返回值

  N/A

4.5.1.13. truncate_local_data_after_distributing_table

- 功能

  分发表后截断所有本地行，并防止约束因本地记录过时而失败。截断将级联到具有指定表的外键的表。

  如果引用表本身不是分布式的，则在它们分布之前禁止截断，以保护引用完整性：ERROR: cannot truncate a table referenced in a foreign key constraint by a local table 截断本地协调器节点表数据对于分布式表是安全的，因为它们的行（如果有）会在分发过程中复制到工作节点。

- 参数

  表 4.17. truncate_local_data_after_distributing_table 参数说明

名称	描述
table_name	应截断其在协调器节点上的本地对应项的分布式表的名称。

- 返回值

  N/A

- 示例

  //要求参数是分布式表
  SELECT truncate_local_data_after_distributing_table('public.github_events');

4.5.1.14. undistribute_table

- 功能

  undistribute_table()函数撤消create_distributed_table或create_reference_table的操作。取消分发会将所有数据从分片移回协调器节点上的本地表（假设数据可以容纳），然后删除分片。

  uxmpp 不会取消分发具有外键或被外键引用的表，除非 cascade_via_foreign_keys 参数设置为 true。如果此参数为 false（或省略），则必须在取消分发之前手动删除有冲突的外键约束。

- 参数

表 4.18. undistribute_table 参数说明

名称	描述
table_name	要取消分发的分布式表或引用表的名称。
cascade_via_foreign_keys	（可选）当此参数设置为“true”时，undistribute_table还会通过外键取消分布与table_name相关的所有表。请谨慎使用此参数，因为它可能会影响许多表。

- 返回值

  N/A

- 示例

  此示例分发一个表，然后取消分发该表。

  //首先设置表为分布式表
  SELECT create_distributed_table('github_events', 'repo_id');
  //回退分布操作，使该表重新成为本地表
  SELECT undistribute_table('github_events');

4.5.1.15. update_distributed_table_colocation

- 功能

  update_distributed_table_colocation()函数用于更新分布式表的共置。此函数还可用于中断分布式表的共置。如果分布列是同一类型，uxmpp 将隐式共置两个表。如果表 A 和 B 是共置的，并且表 A 被重新平衡，则表 B 也将重新平衡。如果表 B 没有副本标识，则重新平衡将失败。因此，在这种情况下，此函数可以有效地打破隐式共置。

  这两个表必须是哈希分布式表，目前我们不支持 APPEND 分布式表的共置。请注意，此函数不会在物理上移动任何数据。

- 参数

  表 4.19. update_distributed_table_colocation 参数说明

名称	描述
table_name	将更新其共置的表。
colocate_with	待更新的表将会与之共置的表。如果要中断表的共置，则应指定colocate_with => 'none'。

- 返回值

  N/A

- 示例

  //此示例将table A的共置更新为table B的共置。
  SELECT update_distributed_table_colocation('A', colocate_with => 'B');
  //假设table A和table B是共置的（可能是隐式的），如果你想打破共置，执行如下命令。

SELECT update_distributed_table_colocation('A', colocate_with => 'none');
//假设当前table A、table B、table C和table D是共置的，如果要拆分，改为分别共置table A和table B，以及table C和table D，执行如下命令。
SELECT update_distributed_table_colocation('C', colocate_with => 'none');
SELECT update_distributed_table_colocation('D', colocate_with => 'C');

4.5.2. 修改表相关函数

4.5.2.1. master_append_table_to_shard

- 功能

  master_append_table_to_shard()函数可用于将UXDB的表中的数据拷贝到指定的append分布式表的某一个分片中。使用该函数需要先创建append分发方式的分布式表，然后对该表追加空分片（或者使用已有的分片，即必须有分片存在），最后使用该函数将某个表（表结构相近）中的数据拷贝到指定的分片中。

- 函数

  master_append_table_to_shard(shard_id,'source_table_name','source_node_name',source_node_port);

- 参数

  表 4.20. master_append_table_to_shard 参数说明

名称	描述
shard_id	需要拷贝数据的目的分片表的分片ID。
source_table_name	需要拷贝数据的源表名称。
source_node_name	源表所在节点。
source_node_port	源表所在节点的端口。

- 返回值

  shard_fill_ratio：该函数返回分片的填充率，该分片定义为当前分片大小与配置参数shard_max_size的比率。

- 示例

  假设表em是一个本地表，且表结构和empty类似，将表em中的数据拷贝到empty_103846的分片表中。

  SELECT * FROM master_append_table_to_shard(103846,'em','192.168.1.82',5432);
  master_append_table_to_shard

  0.000450134
  (1 行记录)

4.5.2.2. master_apply_delete_command

- 功能

  master_apply_delete_command()函数用于删除与delete命令指定的条件匹配的分片。仅当分片中的所有行都与删除条件匹配时，此函数才会删除分片。由于该函数使用分片元数据来决定

是否需要删除分片，因此它要求DELETE语句中的WHERE子句位于分发列上。如果未指定条件，则删除该表的所有分片。

> **注意**
>
> 该函数适用于append分片方式的分布式表。

- 函数

  master_apply_delete_command('delete_command');

- 参数

表 4.21. master_apply_delete_command 参数说明

名称	描述
delete_command	有效的SQL DELETE命令。

- 返回值

  deleted_shard_count：该函数返回与条件匹配并被删除的分片数。请注意，这是分片的数量，而不是删除的行数。

- 示例

  表empty是以id为分布列的append分布式表，有3个分片，每个分片上各有一行数据。删除其中两个分片。

  SELECT * from master_apply_delete_command('DELETE FROM empty WHERE id < 3');
  master_apply_delete_command

  2
  (1 行记录)

4.5.3. 节点管理函数

4.5.3.1. master_update_node

- 功能

  master_update_node()函数更改uxmpp元数据表ux_dist_node中注册的节点的主机名和端口。

- 函数

  master_update_node(node_id,'node_name',node_port);

- 参数

表 4.22. master_update_node 参数说明

名称	描述
node_id	需要更新的节点的ID，来自ux_dist_node表的nodeid。
node_name	更新的新节点的IP。

名称	描述
node_port	更新的新节点的端口。

- 返回值

  N/A

- 示例

  将192.168.1.84:5432节点更新为192.168.1.83:5432。

  SELECT * FROM master_update_node(3,'192.168.1.83',5432);
  master_update_node

  (1 行记录)

4.5.3.2. master_add_inactive_node

- 功能

  该master_add_inactive_node函数与uxmpp_add_node()类似，添加一个新节点。但是，它将新节点标记为非活动状态，即不会在其中放置任何分片。可查看表ux_dist_node的isactive值。也可以使用函数uxmpp_add_inactive_node，它们具有相同功能。

- 函数

  master_add_inactive_node('node_name',node_port);

- 参数

  表 4.23. master_add_inactive_node 参数说明

名称	描述
node_name	要添加的新节点的IP地址。
node_port	要添加的新节点的端口。
group_id	（可省略）服务器组，仅与流复制相关。默认值为-1。
node_role	（可省略）primary或secondary。默认primary。
node_cluster	（可省略）集群名称。默认default。

- 返回值

  表ux_dist_node中新插入行中的 nodeid 列。

- 示例

  select * from master_add_inactive_node('new-node', 12345);
  master_add_inactive_node

  7

(1 row)

4.5.3.3. master_activate_node

- 功能

  master_activate_node()函数将uxmpp中非活动状态的节点激活。可在master_add_inactive_node()函数之后用。也可以使用函数uxmpp_activate_node，它们具有相同功能。

- 函数

  master_activate_node('node_name',node_port);

- 参数

  表 4.24. master_activate_node 参数说明

名称	描述
node_name	要激活的节点的IP地址。
node_port	要激活的节点的端口。

- 返回值

  表ux_dist_node中新插入行中的 nodeid 列。

- 示例

```
select * from master_activate_node('new-node', 12345);
 master_activate_node
----------------------
                    7
(1 row)
```

4.5.3.4. master_disable_node

- 功能

  master_disable_node()函数与master_activate_node()函数相反。该函数将活动状态的节点标记为非活动状态，即反激活。

- 函数

  master_disable_node('node_name',node_port);

- 参数

  表 4.25. master_disable_node 参数说明

名称	描述
node_name	要反激活的节点的IP地址。
node_port	要反激活的节点的端口。

- 返回值

N/A

- 示例

反激活一个活动状态的节点。

```
SELECT * FROM master_disable_node('192.168.1.84',5432);
 master_disable_node
---------------------

(1 行记录)
```

4.5.3.5. master_add_secondary_node

- 功能

master_add_secondary_node()函数是给一个现有的主节点新添加一个辅助节点。它将更新表ux_dist_node。与函数uxmpp_add_secondary_node具有相同功能。

- 函数

master_add_secondary_node('node_name',node_port,'primary_name',primary_port,'node_cluster');

- 参数

表 4.26. master_add_secondary_node 参数说明

名称	描述
node_name	要添加的辅助新节点的IP地址。
node_port	要添加的辅助新节点的端口。
primary_name	主节点的IP地址。
primary_port	主节点的端口。
node_cluster	群集名称。默认default。

- 返回值

表ux_dist_node辅助节点的 nodeid 列。

- 示例

```
select * from master_add_secondary_node('new-node', 12345, 'primary-node', 12345);
 master_add_secondary_node
---------------------------
                         7
(1 row)
```

4.5.3.6. master_remove_node

- 功能

master_remove_node()函数是删除指定的节点。如果删除的节点上存在分片，则此函数会报错。因此，在使用此函数前，需要将分片移出节点。与函数uxmpp_remove_node具有相同功能。

- 函数

  master_remove_node('node_name',node_port);

- 参数

  表 4.27. master_remove_node 参数说明

名称	描述
node_name	要删除的节点的IP地址。
node_port	要删除的节点的端口。

- 返回值

  N/A

- 示例

  select master_remove_node('new-node', 12345);
  master_remove_node

  (1 row)

4.5.3.7. uxmpp_add_node

- 功能

  uxmpp_add_node()函数在 uxmpp 元数据表ux_dist_node的群集中注册新的节点添加。它还将引用表复制到新节点。

  如果在单节点群集上运行，请确保先运行citus_set_coordinator_host。

- 函数

  uxmpp_add_node('node_name',node_port,group_id,'node_role','node_cluster');

- 参数

  表 4.28. uxmpp_add_node 参数说明

名称	描述
node_name	要添加的新节点的IP地址。
node_port	要添加的新节点的端口。
group_id	（可省略）服务器组，仅与流复制相关。请确保设置为大于零的值，因为零是为协调器节点保留的。默认值为 -1。
node_role	（可省略）primary或secondary。默认primary。
node_cluster	（可省略）集群名称。默认default。

- 返回值

表ux_dist_node中新插入行中的 nodeid 列。

- 示例

select * from uxmpp_add_node('new-node', 12345);
 uxmpp_add_node

              7
(1 row)

4.5.3.8. uxmpp_set_coordinator_host

- 功能

如果要将工作节点添加到最初作为单节点群集创建的 uxmpp 群集，需要通过次函数设置协调器节点的 DNS 名称。因为当协调器注册新工作线程时，它会从 uxmpp.local_hostname(text)的值添加一个协调器主机名，默认情况下为localhost。工作节点会尝试连接到localhost与协调节点交互。

因此，系统管理员应在调用单节点群集中的uxmpp_add_node之前调用uxmpp_set_coordinator_host。

- 参数

表 4.29. uxmpp_set_coordinator_host 参数说明

名称	描述
host	协调器节点的 DNS 名称。
port	（可选）协调器为其列出UXDB连接的端口。缺省值为current_setting('port')。
node_role	（可选）默认为primary。
node_cluster	（可选）默认为default。

- 返回值

N/A

- 示例

//假设当前为单节点集群
//首先设置协调节点信息，以便工作节点连接访问
SELECT uxmpp_set_coordinator_host('coord.example.com', 5432);
//然后添加工作节点
SELECT * FROM uxmpp_add_node('worker1.example.com', 5432);

4.5.3.9. uxmpp_set_node_property

- 功能

uxmpp_set_node_property()函数更改 uxmpp 元数据表ux_dist_node中的属性。目前，它只能更改shouldhaveshards属性。

- 参数

表 4.30. uxmpp_set_node_property 参数说明

名称	描述
node_name	节点的DNS名称或IP地址。
node_port	UXDB在工作节点上侦听的端口。
property	当前仅支持要更改ux_dist_node的shouldhaveshard列。
value	列的新值。

- 返回值

  N/A

- 示例

  SELECT * FROM uxmpp_set_node_property('localhost', 5433, 'shouldhaveshards', false);

4.5.4. 资源查询函数

4.5.4.1. master_get_active_worker_nodes

- 功能

  master_get_active_worker_nodes()函数时查看活动状态的节点。

- 参数

  N/A

- 返回值

  处于活动状态的节点列表。

  node_name：活动状态的节点IP。

  node_port：活动状态的节点端口。

- 示例

```
SELECT * FROM master_get_active_worker_nodes();
 node_name  | node_port
------------+-----------
 192.168.1.83 |     5432
(1 行记录)
```

4.5.4.2. master_get_table_metadata

- 功能

  master_get_table_metadata()可用于查看分布式表的相关分布元数据。包括标的oid，存储类型，分发发放，分发列，复制因子。最大分片大小和分片分发策略。

- 函数

master_get_table_metadata('table_name');

- 参数

表 4.31. master_get_table_metadata 参数说明

名称	描述
table_name	要获取分布元数据的分布式表名。

- 返回值

包含以下信息的元组。

logical_relid：分布式表的oid。此值引用ux_class系统目录表中的relfilenode列。

part_storage_type：表的存储类型。可以是't'（标准表），'f'（外部表）或'c'（列存表）。

part_method：表的分发类型。可以是'a'（append）或'h'（hash）。

part_key：表的分发列。

part_replica_count：当前分片复制因子。

part_max_size：当前最大分片大小（以字节为单位）。

part_placement_policy：分片分发策略。可以是1（本地节点优先）或2（循环）。

- 示例

```
SELECT * FROM master_get_table_metadata('empty');
 logical_relid | part_storage_type | part_method | part_key | part_replica_count | part_max_size |
 part_placement_policy
---------------+-------------------+-------------+----------+--------------------+---------------
+----------------------
         57770 | t                 | a           | id       |                  1 |    1073741824 |
                  2
(1 行记录)
```

4.5.4.3. get_shard_id_for_distribution_column

- 功能

get_shard_id_for_distribution_column()函数根据分发列的值确定给行所在的分片表。该函数在hash分布方式的分布式表上使用。不适用append分布方式。

- 函数

get_shard_id_for_distribution_column('table_name', distribution_value);

- 参数

表 4.32. get_shard_id_for_distribution_column 参数说明

名称	描述
table_name	分布式表名。

名称	描述
distribution_value	分发列的值。

- 返回值

  shardID：指定分发列的行所在表的分片ID。

- 示例

  查看hash分布式表github_events中分布列值为5的行所在的分片表。

```
SELECT  get_shard_id_for_distribution_column('github_events',5);
 get_shard_id_for_distribution_column
--------------------------------------
                               102742
(1 行记录)
```

4.5.4.4. column_to_column_name

- 功能

  column_to_column_name()函数查看分布式表的分布列。

- 函数

  column_to_column_name(logicalrelid,partkey)

- 参数

  表 4.33. column_to_column_name 参数说明

名称	描述
table_name	分布式表名。

- 返回值

  distribution_column：指定分布式表的分发列。

- 示例

  查看分布式表github_events的分发列。

```
SELECT  column_to_column_name(logicalrelid,partkey )AS  dist_col_name FROM
 ux_dist_partition WHERE  logicalrelid = 'github_events' :: regclass ;
 dist_col_name
---------------
 repo_id
(1 行记录)
```

4.5.4.5. ux_relation_size

- 功能

  ux_relation_size()函数获取指定分布式表所有分片使用的磁盘空间。不包括空闲空间(FSM)和和映射空间(VM)。与函数uxmpp_relation_size具有相同功能。

- 函数

  ux_size_pretty(uxmpp_relation_size('table_name'));

- 参数

  表 4.34. ux_size_pretty 参数说明

名称	描述
table_name	分布式表名。

- 返回值

  table_size：指定分布式表的大小。

- 示例

  查看分布式表github_events所有分片的大小。

```
SELECT ux_size_pretty(uxmpp_relation_size('github_events'));
 ux_size_pretty
----------------
 16 kB
(1 行记录)
```

4.5.4.6. ux_table_size

- 功能

  ux_table_size ()函数获取指定分布式表所有分片使用的磁盘空间，不包括索引，但是包括TOSTA、空闲空间(FSM)和和映射空间(VM)。与函数uxmpp_table_size具有相同功能。

- 函数

  ux_size_pretty(uxmpp_table_size('table_name'));

- 参数

  表 4.35. ux_table_size 参数说明

名称	描述
table_name	分布式表名。

- 返回值

  table_size：指定分布式表的大小。

- 示例

  查看分布式表github_events所有分片的大小。

```
SELECT  ux_size_pretty(uxmpp_table_size('github_events'));
 ux_size_pretty
----------------
 272 kB
```

(1 行记录)

4.5.4.7. ux_total_relation_size

- 功能

  ux_total_relation_size()函数获取指定分布式表所有分片使用的磁盘空间，包括索引和TOAST数据。与函数uxmpp_total_relation_size具有相同功能。

- 函数

  ux_size_pretty(uxmpp_total_relation_size('table_name'));

- 参数

  表 4.36. ux_total_relation_size 参数说明

名称	描述
table_name	分布式表名。

- 返回值

  table_size：指定分布式表的大小。

- 示例

  查看分布式表github_events所有分片的大小。

```
SELECT ux_size_pretty(uxmpp_total_relation_size('github_events'));
 ux_size_pretty
----------------
 272 kB
(1 行记录)
```

4.5.5. 故障修复函数

4.5.5.1. get_rebalance_progress

- 功能

  分片重新平衡开始后，函数get_rebalance_progress()会列出所涉及的每个分片的进度。它监视由rebalance_table_shards()计划和执行的移动。

- 参数

  N/A

- 返回值

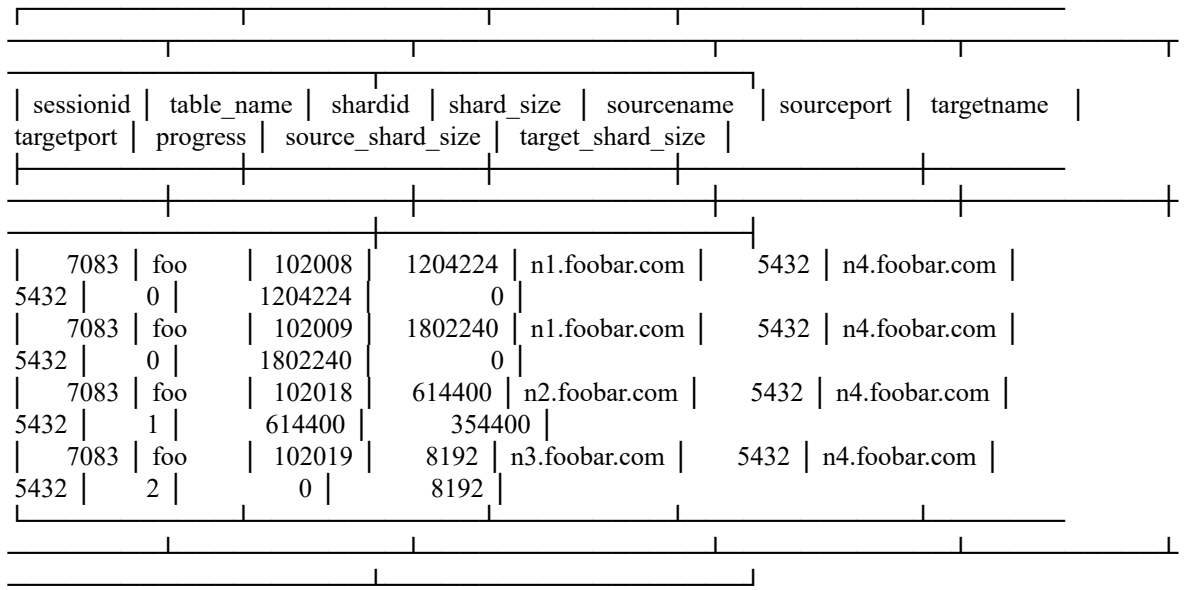
  包含以下列的元组。

  - sessionid：再平衡监视器的后置 PID
  - table_name：分片正在移动的表
  - shardid：要移动的分片

- shard_size：分片的大小（以字节为单位）
- sourcename：源节点的主机名
- sourceport：源节点的端口
- targetname：目标节点的主机名
- targetport：目标节点的端口
- progress： 0 = 等待移动;1 = 移动;2 = 完成
- source_shard_size：源节点上分片的大小（以字节为单位）
- target_shard_size：目标节点上分片的大小（以字节为单位）

- 示例

SELECT * FROM get_rebalance_progress();



4.5.5.2. get_rebalance_table_shards_plan

- 功能

输出rebalance_table_shards的分片移动计划，但并不实际执行。get_rebalance_table_shards_plan输出的计划可能与rebalance_table_shards略有不同。发生这种情况可能是因为它们不是同时执行的，因为有关群集的物理属性（例如磁盘空间）在两次函数调用之间可能会有变化。

- 参数

与rebalance_table_shards相同的参数：
relation、threshold、max_shard_moves、excluded_shard_list和drain_only。有关参数的含义，请参阅该函数的文档。

- 返回值

包含以下列的元组。

- table_name：分片将移动的表
- shardid：要移动的分片
- shard_size：大小（以字节为单位）
- sourcename：源节点的主机名
- sourceport：源节点的端口
- targetname：目标节点的主机名
- targetport：目标节点的端口

4.5.5.3. master_copy_shard_placement

- 功能

  master_copy_shard_placement()用于修复故障分片。在操作期间导致某个分片变成非活动状态，然后通过该函数将活动的副本分片中的数据拷贝到故障的非活动分片，并使其变成活动状态，以确保新插入的数据分片正常。与函数uxmpp_copy_shard_placement具有相同功能。

- 函数

  master_copy_shard_placement(shard_id,'source_node_name',source_node_port,'target_node_name',target_node_port);

- 参数

  表 4.37. master_copy_shard_placement 参数说明

名称	描述
shard_id	要修复的分片的ID。
source_node_name	正常的分片所在节点IP（“源”节点）。
source_node_port	正常的分片所在节点IP端口。
target_node_name	故障的分片所在节点IP（“目标”节点）。
target_node_port	故障的分片所在节点端口。

- 返回值

  N/A

- 示例

  //创建测试表，并设置分片和副本
  CREATE TABLE test_table(id int, name varchar(16));
  //插入数据
  INSERT INTO test_table VALUES(1,'a'),(2,'b'),(3,'c'),(4,'d');
  //断开其中一个节点（断网或停止集群）之后，进行写操作
  INSERT INTO test_table VALUES(4,'99');
  //查看分片分布情况数据shardstate字段值为3表示故障，1表示正常
  SELECT * from ux_dist_shard_placement;

//恢复上面断开的节点，修复故障
SELECT master_copy_shard_placement(102048, '192.168.1.84', 5432, '192.168.1.83', 5432);

> **注意**
>
> 修复故障的时候涉及到worker与worker之间数据传输，因此需要将.uxpass文件添加到worker节点的uxdb用户主目录下（可以直接将master上.uxpass复制到各个worker节点）。

4.5.5.4. rebalance_table_shards

- 功能

rebalance_table_shards()函数移动给定表的分片，使它们在工作节点中均匀分布。该函数首先计算它需要进行移动的列表，以确保集群在给定阈值内保持平衡。然后，它将分片逐个从源节点移动到目标节点，并更新相应的分片元数据信息。

在确定分片是否“均匀分布”时，每个分片都会分配一个成本值。默认情况下，每个分片具有相同的成本值（值为 1），因此均衡工作节点之间的成本值等同于，均衡每个节点上的分片数量。“by_shard_count”是默认的再平衡策略，该策略下成本值恒定。

1. 分片的大小大致相同
2. 分片获得的流量大致相同
3. 工作节点的大小/类型都相同
4. 分片尚未固定到特定工作节点

如果这些假设中的任何一个不成立，那么默认的再平衡策略可能不会得到理想的平衡结果。在这种情况下，可以使用参数rebalance_strategy自定义再平衡策略。

建议在运行rebalance_table_shards之前调用get_rebalance_table_shards_plan，以查看和验证要执行的操作。

- 参数

表 4.38. rebalance_table_shards 参数说明

名称	描述
table_name	（可选）需要重新平衡其分片的表的名称。如果为 NULL，则重新平衡所有现有的共置组。
threshold	（可选）介于 0.0 和 1.0 之间的浮点数，指示节点利用率与平均利用率的最大差值。例如，指定 0.1 将导致分片重新平衡器尝试平衡所有节点以容纳相同数量的分片 ±10%。具体来说，分片重新平衡器将尝试将所有工作节点的利用率收敛到（1 - 阈值）*average_utilization...（1 + 阈值） * average_utilization范围。
max_shard_moves	（可选）要移动的最大分片数。
excluded_shard_list	（可选）在重新平衡操作期间不应移动的分片标识符。

名称	描述
shard_transfer_mode	（可选）指定复制方法，是使用 UXDB 逻辑复制还是跨工作节点 COPY 命令。可能的值包括： • auto：默认值，等价于block_writes。 • block_writes：对缺少主键或副本标识的表使用 COPY（阻塞写入）。
drain_only	（可选）如果为 true，则在工作节点表ux_dist_node中将分片从shouldhaveshards设置为 false 的工作节点上移开;不移动其他分片。
rebalance_strategy	（可选）再平衡策略表中的策略名称。如果省略此参数，则该函数将选择默认策略，如表ux_dist_rebalance_strategy中所示。

- 返回值

  N/A

- 示例

  下面的示例将尝试在默认阈值内重新平衡github_events表的分片。

  SELECT rebalance_table_shards('github_events');

  此示例用法将尝试重新平衡github_events表，而不移动 ID 为 1 和 2 的分片。

  SELECT rebalance_table_shards('github_events', excluded_shard_list:='{1,2}');

4.5.5.5. replicate_table_shards

- 功能

  replicate_table_shards()函数复制给定表备份数不足复制因子的分片。该函数首先计算备份不足的分片列表，以及可以从中获取这些分片以进行复制的位置。然后，该函数复制这些分片并更新相应的分片元数据。

- 参数

  表 4.39. replicate_table_shards 参数说明

名称	描述
table_name	需要复制其分片的表的名称。
shard_replication_factor	（可选）每个分片需要达到备份数（复制因子）。
max_shard_copies	（可选）要复制的最大分片数以达到所需的复制因子。
excluded_shard_list	（可选）在复制操作期间不进行复制的分片标识符。

- 返回值

  N/A

- 示例

  下面的示例将尝试以shard_replication_factor复制github_events表的分片。

  SELECT replicate_table_shards('github_events');

  此示例将尝试将github_events表的分片设置为所需的复制因子，最多包含 10 个分片副本。这意味着重新平衡器在尝试达到所需的复制因子时最多将只复制 10 个分片。

  SELECT replicate_table_shards('github_events', max_shard_copies:=10);

4.5.5.6. uxmpp_add_rebalance_strategy

- 功能

  将一行追加到ux_dist_rebalance_strategy，增加一个新的再平衡策略。

- 参数

  表 4.40. uxmpp_add_rebalance_strategy 参数说明

名称	描述
name	新策略的标识符。
shard_cost_function	标识用于确定每个分片的“成本”的函数。
node_capacity_function	识别测量节点容量的功能。
shard_allowed_on_node_function	标识确定哪些分片可以放置在哪些节点上的函数。
default_threshold	一个浮点阈值，用于调整节点之间应平衡累积分片成本的精确程度。
minimum_threshold	（可选）阈值参数 rebalance_table_shards()所允许的最小值。其默认值为 0。

- 返回值

  N/A

4.5.5.7. uxmpp_create_restore_point

- 功能

  临时阻止对群集的写入，并在所有节点上创建命名还原点。此功能类似于 ux_create_restore_point，但适用于所有节点，并确保还原点在它们之间保持一致。此函数非常适合执行时间点恢复和集群备份。

- 参数

表 4.41. uxmpp_create_restore_point 参数说明

名称	描述
name	要创建的还原点的名称。

- 返回值

  coordinator_lsn：协调器节点 WAL 中还原点的日志序列号。

- 示例

  select uxmpp_create_restore_point('foo');

uxmpp_create_restore_point
0/1EA2808

4.5.5.8. uxmpp_drain_node

- 功能

  uxmpp_drain_node()函数将分片从指定节点，移动到ux_dist_table中shouldhaveshards设置为 true 的其他节点上。函数一般在从群集中删除节点之前调用。

- 参数

表 4.42. uxmpp_drain_node 参数说明

名称	描述
nodename	要清空的节点的主机名。
nodeport	要清空的节点的端口号。
shard_transfer_mode	（可选）指定复制方法，是使用 UXDB逻辑复制还是跨工作线程 COPY 命令。包括的值如下所示。 • auto：默认值，等价于block_writes。 • block_writes：对缺少主键或副本标识的表使用 COPY（阻止写入）。
rebalance_strategy	（可选）再平衡策略表中的策略名称。如果省略此参数，则该函数将选择默认策略。

- 返回值

  N/A

- 示例

  下面是删除单个节点的一般步骤（例如标准UXDB端口上的“10.0.0.1”）。

  1. 清空节点。

SELECT * from uxmpp_drain_node('10.0.0.1', 5432);

2. 等到命令完成。

3. 删除节点。

排空多个节点时，建议改用rebalance_table_shards。这样做可以让uxmpp提前计划并以最少的次数移动分片。

1. 对要删除的每个节点运行如下命令。

SELECT * FROM uxmpp_set_node_property(node_hostname, node_port,
'shouldhaveshards', false);

2. 用rebalance_table_shards一次将它们全部清理。

SELECT * FROM rebalance_table_shards(drain_only := true);

3. 等待再平衡完成。

4. 删除节点。

4.5.5.9. uxmpp_move_shard_placement

- 功能

此函数将给定的分片（以及与之共置的分片）从一个节点移动到另一个节点。函数通常在分片重新平衡期间间接自动调用，而不是由数据库管理员直接调用。

移动数据有两种方式：阻塞或不阻塞。阻塞意味着在移动过程中，对分片的所有修改都将暂停。不阻塞则避免了阻塞分片写入，依赖于UXDB逻辑复制。

移动操作成功后，源节点中的分片将被删除。如果移动失败，此函数将引发错误，并使源节点和目标节点保持不变。

- 参数

表 4.43. uxmpp_move_shard_placement 参数说明

名称	描述
shard_id	要移动的分片的 ID。
source_node_name	“源”节点的 DNS 名称。
source_node_port	数据库服务器正在侦听的源工作线程节点上的端口。
target_node_name	“目标”节点的 DNS 名称。
target_node_port	数据库服务器正在侦听的目标工作线程节点上的端口。
shard_transfer_mode	（可选）指定复制方式，是使用 UXDB 逻辑复制还是跨节点 COPY 命令。可以设置的值如下所示。 • auto：默认值，等价于block_writes。

名称	描述
	• block_writes：对缺少主键或副本标识的表使用 COPY（阻塞写入）。

- 返回值

  N/A

- 示例

  SELECT uxmpp_move_shard_placement(12345, 'from_host', 5432, 'to_host', 5432);

4.5.5.10. uxmpp_remote_connection_stats

- 功能

  uxmpp_remote_connection_stats()函数显示与每个远程节点的活动连接数。

- 参数

  N/A

- 示例

  SELECT * from uxmpp_remote_connection_stats();
  ```
   hostname      | port | database_name | connection_count_to_node
  ---------------+------+---------------+--------------------------
  uxmpp_worker_1 | 5432 |   uxdb        |            3
  (1 row)
  ```

4.5.5.11. uxmpp_set_default_rebalance_strategy

- 功能

  更新再平衡策略表ux_dist_rebalance_strategy，按其参数更改重新平衡分片时选择的默认策略。

- 参数

  表 4.44. uxmpp_set_default_rebalance_strategy 参数说明

名称	描述
name	策略在ux_dist_rebalance_strategy中的名称。

- 返回值

  N/A

- 示例

  SELECT uxmpp_set_default_rebalance_strategy('by_disk_size');

4.6. 元数据表

uxmpp根据分发列将每个分布式表划分为多个逻辑分片。master将这些的分片的状态、位置和相关信息统计起来放在对应的元数据表中。可以在master节点通过SQL查看这些表。

4.6.1. time_partitions

- 功能

  uxmpp 提供 UDF 来管理时间序列数据用例的分区。同时维护一个视图time_partitions来检查它管理的分区。

- 列

  表 4.45. time_partitions 列说明

名称	描述
parent_table	分区的表。
partition_column	父表分区的列。
partition	分区表的名称。
from_value	此分区中行的时间下限。
to_value	此分区中行的时间上限。
access_method	heap行式存储和columnar列式存储。

- 示例

SELECT * FROM time_partitions;

```
┌──────────────────────┬──────────────────┬──────────────────────────────────────┬─────────────────────┬─────────────────────┬───────────────┐
│     parent_table     │ partition_column │              partition               │     from_value      │      to_value       │ access_method │
├──────────────────────┼──────────────────┼──────────────────────────────────────┼─────────────────────┼─────────────────────┼───────────────┤
│ github_columnar_events │ created_at     │ github_columnar_events_p2015_01_01_0000 │ 2015-01-01 00:00:00 │ 2015-01-01 02:00:00 │ columnar      │
│ github_columnar_events │ created_at     │ github_columnar_events_p2015_01_01_0200 │ 2015-01-01 02:00:00 │ 2015-01-01 04:00:00 │ columnar      │
│ github_columnar_events │ created_at     │ github_columnar_events_p2015_01_01_0400 │ 2015-01-01 04:00:00 │ 2015-01-01 06:00:00 │ columnar      │
│ github_columnar_events │ created_at     │ github_columnar_events_p2015_01_01_0600 │ 2015-01-01 06:00:00 │ 2015-01-01 08:00:00 │ heap          │
└──────────────────────┴──────────────────┴──────────────────────────────────────┴─────────────────────┴─────────────────────┴───────────────┘
```

4.6.2. ux_dist_rebalance_strategy

- 功能

此表定义了rebalance_table_shards可用于确定将分片移动到何处的策略。

默认策略by_shard_count，为每个分片分配相同的成本。它的效果是均衡节点之间的分片计数。另一个预定义的策略by_disk_size，将cost分配给与其磁盘大小相匹配的每个分片（以字节为单位），再加上与之共存的分片。磁盘大小是使用ux_total_relation_size计算的，因此它包括索引。此策略尝试在每个节点上实现相同的磁盘空间。请注意阈值0.1，它可防止因磁盘空间的差异而导致的不必要的分片移动。

- 列

表 4.46. ux_dist_rebalance_strategy 列说明

名称	类型	描述
name	name	策略的唯一名称。
default_strategy	boolean	是否是rebalance_table_shards的默认策略。可以使用uxmpp_set_default_rebalance_strategy修改此列。
shard_cost_function	regproc	成本函数的标识符，它必须将分片作为bigint，并将其cost返回为real类型。
node_capacity_function	regproc	容量函数的标识符，它必须将 nodeid 作为int，并将其节点容量返回为real类型。
shard_allowed_on_node_function	regproc	给定 shardid bigint 和 nodeidarg int 的函数的标识符，返回是否允许将分片存储在节点。
default_threshold	float4	将节点视为太满或太空的阈值，rebalance_table_shards将依据此值判断何时应尝试移动分片。
minimum_threshold	float4	防止阈值参数的保障措施rebalance_table_shards()设置得太低。
improvement_threshold	float4	确定在重新平衡期间何时值得移动分片。重新平衡器在移动分片时，不得超过阈值。这对于by_disk_size策略最有用。

在uxmpp安装之后，附带以下策略，如下所示。

```
SELECT * FROM ux_dist_rebalance_strategy;
-[ RECORD 1 ]----------------+---------------------------------
name                         | by_shard_count
default_strategy             | t
shard_cost_function          | uxmpp_shard_cost_1
node_capacity_function       | uxmpp_node_capacity_1
shard_allowed_on_node_function | uxmpp_shard_allowed_on_node_true
default_threshold            | 0
minimum_threshold            | 0
improvement_threshold        | 0
-[ RECORD 2 ]----------------+---------------------------------
name                         | by_disk_size
default_strategy             | f
shard_cost_function          | uxmpp_shard_cost_by_disk_size
```

```
node_capacity_function        | uxmpp_node_capacity_1
shard_allowed_on_node_function | uxmpp_shard_allowed_on_node_true
default_threshold             | 0.1
minimum_threshold             | 0.01
improvement_threshold         | 0.5
```

4.6.3. ux_dist_partition

ux_dist_partition存储有关分布表的分布元数据。包括分布方法以及分发列的相关详细信息。

表 4.47. ux_dist_partition元数据表

名称	类型	描述
logicalrelid	regclass	分布式表的表名。
partmethod	char	分布式表的分布方法，h表示哈希分布，a表示append分布，n表示参考表。
partkey	text	分发列的详细信息。
colocationid	integer	此表所属的共址组（请参考协同表），该值引用自ux_dist_colocation表的colocationid列。
repmodel	char	数据复制的方法。c表示基于SQL语句的复制，s表示流复制。

4.6.4. ux_dist_shard

ux_dist_shard存储有关分布式表的各个分片的元数据。包括有关该分片所属的分布式表的信息以及该分片的分发列的统计信息。

表 4.48. ux_dist_shard元数据表

名称	类型	描述
logicalrelid	regclass	分布式表的表名。
shardid	bigint	分配给分片的唯一标识符。
shardstorage	char	用于此分片的存储类型。t表示存储常规存储，c表示列存，f表示外部存储。
shardminvalue	text	append：分布列最小值。hash：分布列最小键值。
shardmaxvalue	text	append：分布列最大值。hash：分布列最大键值。

4.6.5. ux_dist_placement

ux_dist_placement存储worker节点上的分片和副本的统计信息。

表 4.49. ux_dist_placement元数据表

名称	类型	描述
placementid	bigint	自动生成的唯一标识符。
shardid	bigint	分片的唯一标识符，该值引用自ux_dist_shard表的shardid列。
shardstate	integer	碎片状态。1表示活动状态，3表示非活动（故障）状态。
shardlength	bigint	append：分片的大小。hash：为零。
groupid	integer	节点所在组的唯一标识符，该值引用自ux_dist_node表的groupid列。

4.6.6. ux_dist_shard_placement

ux_dist_shard_placement是一个视图，显示了worker节点上的分片和副本的统计信息。包括worker的IP和端口。

表 4.50. ux_dist_shard_placement元数据表

名称	类型	描述
shardid	bigint	分片的唯一标识符，该值引用自ux_dist_shard表的shardid列。
shardstate	integer	碎片状态。1表示活动状态，3表示非活动（故障）状态。
shardlength	bigint	append：分片的大小。hash：为零。
nodename	text	分片所在节点IP。
nodeport	integer	分片所在节点端口。
placementid	bigint	自动生成的唯一标识符。

4.6.7. ux_dist_node

ux_dist_node存储worker节点的信息。

表 4.51. ux_dist_node元数据表

名称	类型	描述
nodeid	integer	自动生成的单个节点的唯一标识符。
groupid	integer	节点所在组的唯一标识符，通常和nodeid一样。
nodename	text	节点IP。
nodeport	integer	节点端口。

名称	类型	描述
noderack	text	（可选）worker节点位置信息。
hasmetadata	boolean	保留字段。
isactive	boolean	节点活动状态。
noderole	noderole	节点级别。primary或secondary。
nodecluster	text	节点集群名称。（需要手动添加，默认是default）。
metadatasynced	boolean	显示此节点是否具有最新的元数据。
shouldhaveshards	boolean	显示此节点是否具有资格包含分布表的数据。

4.6.8. ux_dist_colocation

ux_dist_colocation存储协同表的相关信息。

表 4.52. ux_dist_colocation元数据表

名称	类型	描述
colocationid	integer	协同组的唯一标识符。
shardcount	integer	协同表的分片数。
replicationfactor	integer	协同表的复制因子。
distributioncolumntype	oid	分发列的分发类型代码。
distributioncolumncollation	oid	协同表分发列的排序代码。

4.6.9. uxmpp_dist_stat_activity

- 功能

  显示在所有节点上执行的分布式查询ux_stat_activity的超集。

- 示例

  考虑对分布式表中的行进行计数，如下所示。

  ```
  //run from worker on localhost:9701
  SELECT count(*) FROM users_table;
  ```

  我们可以在uxmpp_dist_stat_activity看到查询记录出现，如下所示。

  ```
  SELECT * FROM uxmpp_dist_stat_activity;
  -[ RECORD 1 ]----------+---------------------------------
  query_hostname         | localhost
  query_hostport         | 9701
  master_query_host_name | localhost
  master_query_host_port | 9701
  transaction_number     | 1
  ```

```
transaction_stamp      | 2018-10-05 13:27:20.691907+03
datid                  | 12630
datname                | uxdb
pid                    | 23723
usesysid               | 10
usename                | uxmpp
application_name       | uxsql
client_addr            |
client_hostname        |
client_port            | -1
backend_start          | 2018-10-05 13:27:14.419905+03
xact_start             | 2018-10-05 13:27:16.362887+03
query_start            | 2018-10-05 13:27:20.682452+03
state_change           | 2018-10-05 13:27:20.896546+03
wait_event_type        | Client
wait_event             | ClientRead
state                  | idle in transaction
backend_xid            |
backend_xmin           |
query                  | SELECT count(*) FROM users_table;
backend_type           | client backend
```

4.6.10. uxmpp_lock_waits

- 功能

  阻止整个群集中的查询。

- 示例

  要了解它是如何工作的，我们可以手动生成锁定情况。首先，在协调节点创建测试表，如下所示。

```
CREATE TABLE numbers AS
SELECT i, 0 AS j FROM generate_series(1,10) AS i;
SELECT create_distributed_table('numbers', 'i');
```

然后，在协调节点上使用两个会话，我们可以运行以下语句序列，如下所示。

```
//会话1
--------------------------------------
BEGIN;
UPDATE numbers SET j = 2 WHERE i = 1;
//会话2
--------------------------------------
BEGIN;
UPDATE numbers SET j = 3 WHERE i = 1;
//(阻塞)
```

查看视图uxmpp_lock_waits。

```
SELECT * FROM uxmpp_lock_waits;
-[ RECORD 1 ]-------------------------+----------------------------------------
waiting_pid                           | 88624
```

```
blocking_pid                    | 88615
blocked_statement               | UPDATE numbers SET j = 3 WHERE i = 1;
current_statement_in_blocking_process | UPDATE numbers SET j = 2 WHERE i = 1;
waiting_node_id                 | 0
blocking_node_id                | 0
waiting_node_name               | coordinator_host
blocking_node_name              | coordinator_host
waiting_node_port               | 5432
blocking_node_port              | 5432
```

在此示例中，查询源自协调器。同样的，视图uxmpp_lock_waits也可以列出源自工作节点的查询之间的锁，不再列举。

4.6.11. uxmpp_shards

- 功能

  除了上面描述的低级分片元数据表之外，uxmpp 还提供了一个视图uxmpp_shards，可以用于查看分片相关信息，包括内容如下所示。

  - 每个分片的所在位置（节点和端口）
  - 属于哪种表
  - 分片大小

  通过此视图，可以方便地分析节点之间的大小不平衡。

- 示例

SELECT * FROM uxmpp_shards;

```
 table_name | shardid | shard_name   | uxmpp_table_type | colocation_id | nodename  | nodeport |
 shard_size
------------+---------+--------------+------------------+---------------+-----------+----------+------------
 dist       | 102170 | dist_102170  | distributed      |            34 | localhost |     9701 |   90677248
 dist       | 102171 | dist_102171  | distributed      |            34 | localhost |     9702 |   90619904
 dist       | 102172 | dist_102172  | distributed      |            34 | localhost |     9701 |   90701824
 dist       | 102173 | dist_102173  | distributed      |            34 | localhost |     9702 |   90693632
 ref        | 102174 | ref_102174   | reference        |             2 | localhost |     9701 |       8192
 ref        | 102174 | ref_102174   | reference        |             2 | localhost |     9702 |       8192
 dist2      | 102175 | dist2_102175 | distributed      |            34 | localhost |     9701 |     933888
 dist2      | 102176 | dist2_102176 | distributed      |            34 | localhost |     9702 |     950272
 dist2      | 102177 | dist2_102177 | distributed      |            34 | localhost |     9701 |     942080
 dist2      | 102178 | dist2_102178 | distributed      |            34 | localhost |     9702 |     933888
```

其中，colocation_id是指共置组。

4.6.12. uxmpp_tables

- 功能

  视图uxmpp_tables显示 uxmpp 管理的所有表（分布式表和参考表）的摘要。该视图合并了uxmpp 元数据表中的信息，用户可以更方便地了解uxmpp表的相关属性。

- 列

表 4.53. uxmpp_tables 列参数

名称	描述
table_name	表名
uxmpp_table_type	表类型
colocation_id	共置组 ID
table_size	表大小
shard_count	分片数
table_owner	所有者（数据库用户）
access_method	访问方法（堆或列式）

- 示例

SELECT * FROM uxmpp_tables;

```
┌────────────┬──────────────────┬──────────────────┬──────────────┬────────────┬─────────────┬──────────────┬─────────────────┐
│ table_name │ uxmpp_table_type │ distribution_column │ colocation_id │ table_size │ shard_count │ table_owner │ access_method │
├────────────┼──────────────────┼──────────────────┼──────────────┼────────────┼─────────────┼──────────────┼─────────────────┤
│ foo.test   │ distributed      │ test_column      │            1 │ 0 bytes    │          32 │ uxmpp        │ heap            │
└────────────┴──────────────────┴──────────────────┴──────────────┴────────────┴─────────────┴──────────────┴─────────────────┘
┌────────────┬──────────────────┬──────────────────┬──────────────┬────────────┬─────────────┬──────────────┬─────────────────┐
│ ref        │ reference        │ <none>           │            2 │ 24 GB      │           1 │ uxmpp        │ heap            │
│ test       │ distributed      │ id               │            1 │ 248 TB     │          32 │ uxmpp        │ heap            │
└────────────┴──────────────────┴──────────────────┴──────────────┴────────────┴─────────────┴──────────────┴─────────────────┘
```

4.6.13. uxmpp.ux_dist_object

- 功能

表ux_dist_object包含已在协调器节点上创建，并传播到工作节点的对象列表，例如类型和函数。当管理员将新的工作节点添加到群集时，uxmpp 会自动在新节点上创建分布式对象的副本。

- 列

表 4.54. uxmpp.ux_dist_object 列参数

名称	类型	描述
classid	oid	分布式对象的类
objid	oid	分布式对象的对象 ID
objsubid	integer	分布式对象的对象子 ID，例如attnum
type	text	uxdb升级期间使用的稳定地址的一部分
object_names	text[]	uxdb升级期间使用的稳定地址的一部分

名称	类型	描述
object_args	text[]	uxdb升级期间使用的稳定地址的一部分
distribution_argument_index	integer	仅对分布式函数/过程有效
colocationid	integer	仅对分布式函数/过程有效

4.6.14. uxmpp_worker_stat_activity

- 功能

  显示有关工作节点的查询，包括针对单个分片的片段查询。

- 示例

  我们可以通过查看视图uxmpp_worker_stat_activity来查看访问分片的查询。

```
SELECT * FROM uxmpp_worker_stat_activity;
-[ RECORD 1 ]----------+-----------------------------------------------------------------------------------------------
query_hostname         | localhost
query_hostport         | 9700
master_query_host_name | localhost
master_query_host_port | 9701
transaction_number     | 1
transaction_stamp      | 2018-10-05 13:27:20.691907+03
datid                  | 12630
datname                | uxdb
pid                    | 23781
usesysid               | 10
usename                | uxmpp
application_name       | uxmpp
client_addr            | ::1
client_hostname        |
client_port            | 51773
backend_start          | 2018-10-05 13:27:20.75839+03
xact_start             | 2018-10-05 13:27:20.84112+03
query_start            | 2018-10-05 13:27:20.867446+03
state_change           | 2018-10-05 13:27:20.869889+03
wait_event_type        | Client
wait_event             | ClientRead
state                  | idle in transaction
backend_xid            |
backend_xmin           |
query                  | COPY (SELECT count(*) AS count FROM users_table_102038 users_table
 WHERE true) TO STDOUT
backend_type           | client backend
```

  query字段显示从要计数的分片中复制的数据。

> **注意**
>
> 如果路由器查询SELECT * FROM table WHERE tenant_id = X（例如，多租户应用程序中的单租户）在没有事务块的情况下执行，则

> uxmpp_query_host_name和uxmpp_query_host_port列在uxmpp_worker_stat_activity中将为 NULL。

可以使用以下命令构建查询示例。

```
//特定节点上的活跃查询等待事件
SELECT query, wait_event_type, wait_event
 FROM uxmpp_worker_stat_activity
 WHERE query_hostname = 'xxxx' and state='active';
//活跃查询的最大等待事件
SELECT wait_event, wait_event_type, count(*)
 FROM uxmpp_worker_stat_activity
 WHERE state='active'
 GROUP BY wait_event, wait_event_type
 ORDER BY count(*) desc;
//uxmpp为每个节点生成的内部连接总数
SELECT query_hostname, count(*)
 FROM uxmpp_worker_stat_activity
 GROUP BY query_hostname;
//uxmpp为每个节点生成的内部活跃连接总数
SELECT query_hostname, count(*)
 FROM uxmpp_worker_stat_activity
 WHERE state='active'
 GROUP BY query_hostname;
```

4.7. 参数配置

一些配置参数会影响到uxmpp的使用，包括uxdb本身的参数和uxmpp特有的参数。要了解有关uxdb本身参数，请参见《优炫数据库用户手册 V2.1》。本节主要讨论uxmpp特有的参数。修改uxmpp参数类似于uxdb参数的修改，可以修改配置文件或使用SET命令。

4.7.1. 通用

- uxmpp.max_worker_nodes_tracked (integer)

  此配置值限制哈希表的大小，从而限制工作节点数。此设置的默认值为2048。此参数只能在服务器启动时设置，并且在master节点上有效。

- uxmpp.use_secondary_nodes (enum)

  设置在为SELECT查询选择节点时使用的策略。如果将其设置为“always”，则规划器将仅查询在ux_dist_node中noderole列标记为"secondary"的节点。枚举值为：never：（默认）所有读取都发生在主节点上；always：读取针对secondary节点运行，并且禁用插入/更新语句。

- uxmpp.cluster_name (text)

  通知协调器节点规划器它协调哪个集群。设置cluster_name后，计划程序将仅查询该群集的工作节点。

- uxmpp.enable_version_checks (boolean)

  升级uxmpp版本需要重新启动服务器（以获取新的共享库），以及运行ALTER EXTENSION UPDATE命令。执行这两个步骤失败可能会导致错误或崩溃。此参数是检查代码的版本和扩展匹

配的版本。默认是on。在极少数情况下，复杂的升级过程可能需要将此参数设置为off，禁用检查。

- uxmpp.log_distributed_deadlock_detection（boolean）

  是否在服务器日志中记录分布式死锁检测相关处理。它默认为off。

- uxmpp.distributed_deadlock_detection_factor（floating point）

  设置在检查分布式死锁之前等待的时间。默认值为2。设置值为-1则禁用分布式死锁检测。

- uxmpp.node_connection_timeout（integer）

  设置等待连接建立的最大持续时间（以毫秒为单位）。如果在建立至少一个工作节点连接之前超时，uxmpp将引发错误。此 GUC 影响主节点与工作节点之间的连接，以及工作节点与工作节点之间的连接。

  参数默认值为30秒，最小值为10毫秒，最大值为1小时。

  //设置为60秒
  ALTER DATABASE foo SET uxmpp.node_connection_timeout = 60000;

- uxmpp.node_conninfo（text）

  设置用于所有节点间连接的非敏感libpq连接参数。

  //由空格分隔的key=value键值对
  //例如，ssl选项
  ALTER DATABASE foo SET uxmpp.node_conninfo = 'sslrootcert=/path/to/uxmpp.crt sslmode=verify-full';

  uxmpp 仅支持允许选项的特定子集，如下所示。

  - application_name
  - connect_timeout
  - gsslib
  - keepalives
  - keepalives_count
  - keepalives_idle
  - keepalives_interval
  - krbsrvname
  - sslcompression
  - sslcrl
  - sslmode（默认为“require”）
  - sslrootcert
  - tcp_user_timeout

该设置仅在新打开的连接上生效。要强制所有连接使用新设置，请确保重新加载 uxdb 配置：

SELECT ux_reload_conf();

- uxmpp.local_hostname（text）

  uxmpp 节点偶尔需要连接到自身以进行系统操作。默认情况下，他们使用localhost来引用自己，但这可能会导致问题。例如，当主机需要sslmode=verify-full传入连接时，在SSL证书上添加localhost为备用主机名并不总是可取的 - 甚至不可行。

  uxmpp.local_hostname选择节点用于连接到自身的主机名。缺省值为 localhost。

ALTER SYSTEM SET uxmpp.local_hostname TO 'mynode.example.com';

4.7.2. 加载数据

- uxmpp.multi_shard_commit_protocol（enum）

  该参数时在哈希分布式表上执行复制时要使用的提交协议。对于每个单独的节点，复制在事务块中执行，以确保在复制期间发生错误时不会写入数据。但是，有一种特定情况，即复制在所有节点上都成功，但是提交事务之前发生了错误。此参数可用于在这种情况下通过选择以下提交协议来防止数据丢失：

  - 2pc：（默认）在执行复制的时候，首先在各节点使用uxdb的提交为两段提交做准备，然后再由协调节点决定是否执行提交。可以使用COMMIT PREPARED或ROLLBACK PREPARED手动恢复或中止失败的提交。使用2pc时，应在各个节点上增大参数max_prepared_transactions的值，建议与max_connections相同。
  - 1pc：在各个节点中直接提交。有极小的可能会造成数据丢失。

- uxmpp.shard_replication_factor（integer）

  设置分片的复制因子，即在节点上放置某一个碎片的数量。默认为1，即在节点放置的某一个碎片数为1。此参数可以在运行时设置，即对不同表有不同的值。该参数只对master上设置有效。参数值一般取决于集群的大小和节点故障率。

- uxmpp.shard_count（integer）

  设置hash分布式表的分片总数，默认为32。使用create_distributed_table创建hash分布式表的时候，使用这个值。此参数可在运行过程中设置，即不同的hash分布式表可以有不同的分片总数，此参数只对master上设置有效。

- uxmpp.shard_max_size（integer）

  设置append分布式表的分片表最大大小，默认是1GB。即append分布式表的一个分片超过1GB的时候，就会创建一个新的分片。此参数可在运行过程中设置，即不同的append分布式表可以有不同大小的分片，此参数只对master上设置有效。

- uxmpp.replicate_reference_tables_on_activate（boolean）

  参考表分片必须放置在具有分布式表的所有节点上。默认情况下，引用表分片在节点激活时复制到节点，即调用uxmpp_add_node或uxmpp_activate_node等函数时。但是，节点激活可能会给复制带来不便，因为当存在大型引用表时可能需要很长时间。

对此，可以通过将 GUC 设置为“off”来延迟引用表复制。设置之后，引用表复制将发生于在节点上创建新分片时。例如，调用create_distributed_table，create_reference_table，或者分片重新平衡器将分片移动到新节点。此 GUC 的默认值为“on”。

4.7.3. 计划程序

- uxmpp.limit_clause_row_fetch_count (integer)

  该参数为limit子句优化设置每个任务获取的行数。在某些情况下，带有limit的查询子句可能需要从每个任务中获取所有行来生成结果。在这种情况下，设置从每个分片中获取指定的行数，会产生和实际结果有点偏差的近似结果。默认情况下，该参数值是-1，即禁用的。这个参数可以在运行时设置，对master生效。

- uxmpp.count_distinct_error_rate (floating point)

  该参数设置计算count（distinct）的容错率。默认是0。如果需要用到此值，建议设置为0.005。此参数可在运行过程中设置，对master生效。

- uxmpp.task_assignment_policy (enum)

  设置将任务分配给worker时的策略。master通过计划程序将任务分发到各worker节点上。该参数指定分发任务时使用哪种策略。目前，有三种策略：

  - greedy：默认策略，在worker之间平均分配任务。
  - round-robin：以循环的方式在不同的副本之间交替为worker分配任务。当分片总数小于worker数的时候，可使用这种策略更合适。
  - first-replica：根据插入顺序分配任务，即将任务分配给第一副本所在的worker。

  这个参数可以在运行时设置，对master生效。

- uxmpp.local_table_join_policy (enum)

  此 GUC 确定 uxmpp 在本地表和分布式表之间进行join联接时如何移动数据。自定义联接策略有助于减少在工作节点之间发送的数据量。

  uxmpp 将根据需要将本地表或分布式表发送到节点以支持join联接。复制表数据称为“转换”。如果转换了本地表，则该表将发送给需要其数据来执行联接的任何工作节点。如果转换了分布式表，则会在协调器中收集该表以支持联接。uxmpp 规划器将仅发送执行转换的必要行。

  有四种模式可用于表达转换首选项，如下所示。

  - auto：（默认）uxmpp 将转换所有本地表或所有分布式表以支持本地和分布式表联接。uxmpp 使用启发式方法决定要转换哪个。如果对唯一索引（如主键）使用常量筛选器联接分布式表，它将转换分布式表。这可确保在工作节点之间移动的数据更少。
  - never：uxmpp 将不允许在本地表和分布式表之间进行联接。
  - prefer-local：uxmpp 更倾向转换本地表以支持本地和分布式表联接。
  - prefer-distributed：uxmpp 更倾向转换分布式表以支持本地和分布式表连接。如果分布式表很大，则使用此选项可能会导致在工作节点之间移动大量数据。

  例如，假设uxmpp_table是一个由列x分布的分布式表，并且uxdb_table是一个本地表，如下所示。

```
CREATE TABLE uxmpp_table(x int primary key, y int);
SELECT create_distributed_table('uxmpp_table', 'x');
CREATE TABLE uxdb_table(x int, y int);
//联接在主键上，没有常量筛选器
//因此，表uxdb_table将被发送至工作节点以支持联接
SELECT * FROM uxmpp_table JOIN uxdb_table USING (x);
//主键上有一个常量过滤器
//因此分布式表中的过滤行将被发送至协调节点以支持联接
SELECT * FROM uxmpp_table JOIN uxdb_table USING (x) WHERE uxmpp_table.x = 10;
SET uxmpp.local_table_join_policy to 'prefer-distributed';
//表uxmpp_table将被发送至协调节点以支持联接
//这里需要注意表uxmpp_table数据量很大的情况
SELECT * FROM uxmpp_table JOIN uxdb_table USING (x);
SET uxmpp.local_table_join_policy to 'prefer-local';
//尽管在表uxmpp_table的主键上有一个常量过滤器
//但表uxdb_table仍将被发送给必要的工作节点，因为使用的是'prefer-local'
SELECT * FROM uxmpp_table JOIN uxdb_table USING (x) WHERE uxmpp_table.x = 10;
```

4.7.4. 中间数据传输

- uxmpp.binary_worker_copy_format (boolean)

  使用二进制复制格式在worker之间传输中间数据。在大型表连接期间，uxmpp可能必须在不同worker之间动态地对数据进行重新分区和随机排列。默认情况下，GUC的值为 false，此数据以文本格式传输。启用此参数指示数据库使用uxdb的二进制序列化格式来传输此数据。此参数对worker有效。需要在uxsinodb.conf文件中进行更改。编辑配置文件后，用户可以发送 SIGHUP 信号或重新启动服务器，以使此更改生效。

- uxmpp.max_intermediate_result_size (integer)

  通用表表达式（CTE）和复杂子查询的中间结果的最大大小。默认是1GB，值为-1表示没有限制。超出限制查询会被取消并打印错误消息。

4.7.5. DDL参数

- uxmpp.enable_ddl_propagation (boolean)

  该参数指定是否自动将DDL传播到所有的worker节点。默认是on。由于某些情况下，需要对表进行访问并且独占锁定，并自动传播到worker节点上，这会影响到uxmpp集群的性能。此时可以选择关闭该参数。

- uxmpp.enable_local_reference_table_foreign_keys (boolean)

  默认情况下启用此设置，允许在引用表和本地表之间创建外键。要使该功能正常工作，必须使用uxmpp_add_node向自身注册协调器节点。

  请注意，引用表和本地表之间的外键会需要一些额外的成本。创建外键时，uxmpp 必须将普通表添加到 uxmpp 的元数据中，并在分区表中对其进行跟踪。添加到元数据的本地表继承与引用表相同的限制。

  如果删除外键，uxmpp 将自动从元数据中删除此类本地表，从而消除对这些表的此类限制。

4.7.6. 执行程序

- uxmpp.all_modifications_commutative (boolean)

  uxmpp强制执行交换规则并获取适当的锁定以进行修改操作，以保证行为的正确性。例如，它假定INSERT语句与另一个INSERT语句交换，但不与UPDATE或DELETE语句交换。同样，它假定UPDATE或DELETE语句不与另一个UPDATE或DELETE语句交换。这意味着UPDATE和DELETE要求uxmpp获得更强大的锁。

  如果有与INSERT或其他UPDATE交换的UPDATE语句，那么可以通过将此参数设置为on来放宽这些交换假设。当此参数设置为on时，所有命令都被视为可交换，并声明共享锁，这可以提高整体吞吐量。此参数可以在运行时设置，并且对master有效。

- uxmpp.remote_task_check_interval (integer)

  设置uxmpp检查任务跟踪器执行程序管理的任务状态的频率。默认为10ms。master将任务分配给worker，然后定期检查每个任务的进度。这个配置值设置两个检查结果之间的时间间隔。此参数可以在运行时设置，对master生效。

- uxmpp.task_executor_type (enum)

  uxmpp默认执行器类型是adaptive，用于运行分布式SELECT查询。

- uxmpp.multi_task_query_log_level (enum)

  为生成多个任务的查询（访问多个分片）设置日志级别。对多租户应用程序很有用，可以设置error或warning级别，可以通过租户id过滤到具体的租户。此参数可以在运行时设置，对master生效。默认值是off。此参数可选的值为：

  - off：关闭生成多个任务查询（跨多个分片）的日志。
  - debug：生成调试级别的日志。
  - log：生成日志级别的日志，包含运行的SQL查询。
  - notice：生成通知级别的日志。
  - warning：生成警告级别的日志。
  - error：生成错误级别的日志。

  > **注意**
  >
  > error一般在开发测试期间使用，在实际使用时建议使用log级别。

- uxmpp.enable_repartition_joins (boolean)

  当实时执行程序执行重新分区连接失败时，会打印错误信息。但是当打开该参数时，可以临时切换到任务跟踪程序执行连接。默认是off。

- uxmpp.partition_buffer_size (integer)

  设置用于分区操作的缓冲区大小，默认为8MB。uxmpp允许在连接两个大表时将表数据重新分区为多个文件。在此分区缓冲区填满后，重新分区的数据将刷新到磁盘上的文件中。此参数可以在运行的时候设置，对worker生效。

- uxmpp.propagate_set_commands（enum）

  确定将哪些 SET 命令从协调器传播到工作线程。此参数的默认值为“none”。

  支持的值为：

  - none：不传播 SET 命令。
  - local：仅传播 SET LOCAL 命令。

- uxmpp.enable_repartitioned_insert_select（boolean）

  默认情况下，无法向下推的INSERT INTO … SELECT语句将尝试重新划分SELECT语句中的行，并在worker之间传输这些行以进行插入。但是，如果目标表具有太多分片，则重新分区可能不会很好地执行。在确定如何对结果进行分区时，处理分片间隔的开销太大。可以通过设置为 false 来手动禁用重新分区。

- uxmpp.enable_binary_protocol（boolean）

  将此参数设置为 true 将指示协调器节点使用UXDB的二进制序列化格式（如果适用）与工作节点一起传输数据。某些列类型不支持二进制序列化。当工作节点必须返回大量数据时，启用此参数最有用。例如，当请求大量行时，行具有许多列。

  默认值为false，这意味着所有结果都以文本格式编码和传输。

- uxmpp.max_shared_pool_size（integer）

  指定允许协调器节点跨所有并发会话的每个工作节点建立的最大连接数。UXDB必须为每个连接分配固定资源，这个GUC有助于减轻工作节点的连接压力。

  如果没有连接限制，每个多分片查询都会在每个工作线程上创建与其访问的分片数成比例的连接（取决于#shards/#workers）。一次运行数十个多分片查询很容易达到工作节点的max_connections限制，从而导致查询失败。

  max_connections限制，从而导致查询失败。默认情况下，该值会自动设置为等于协调员自己的max_connections，这不保证与工作人员的值匹配（请参见下面的注释）。值-1禁用限制。

  > **注意**
  >
  > 某些操作不遵守 uxmpp.max_shared_pool_size，最重要的是重新分区联接。这就是为什么将worker的max_connections提高至比master的max_connections高一点的原因。这为工作线程上的重新分区查询所需的连接提供了额外的空间。

- uxmpp.max_adaptive_executor_pool_size（integer）

  uxmpp.max_shared_pool_size（integer）限制所有会话中的工作节点连接，而max_adaptive_executor_pool_size仅限制当前会话中的工作节点连接。此 GUC 可用于：

  - 防止单个后端获取所有辅助角色资源。
  - 提供优先级管理：指定低优先级会话和低max_adaptive_executor_pool_size值的高优先级会话。

  默认值为16。

- uxmpp.executor_slow_start_interval（integer）

  在打开与同一工作节点的连接之间等待的时间（以毫秒为单位）。

  当多分片查询的各个任务花费很少的时间时，它们通常可以通过单个（通常已经缓存的）连接完成。为避免以冗余方式打开其他连接，执行程序会在两次连接尝试之间等待配置的毫秒数。在间隔结束时，它会增加下次允许打开的连接数。

  对于长查询（需要>500毫秒的查询），慢启动可能会增加延迟，但对于短查询，它更快。默认值为 10 毫秒。

- uxmpp.max_cached_conns_per_worker（integer）

  每个后端都会打开与工作节点的连接以查询分片。在事务结束时，配置的连接数保持打开状态，以加快后续命令的速度。增加此值将减少多分片查询的延迟，但也会增加工作节点的开销。

  默认值为 1。较大的值（如 2）对于使用少量并发会话的群集可能会有所帮助，但不建议将该值设置过高（例如，16 会太高）。

- uxmpp.force_max_query_parallelization（boolean）

  模拟已弃用且现在不存在的实时执行器。这用于打开尽可能多的连接，以最大限度地提高查询并行化。

  启用此 GUC 后，uxmpp 将强制自适应执行程序在执行并行分布式查询时使用尽可能多的连接。如果未启用，执行程序可能会选择使用较少的连接来优化整体查询执行吞吐量。在内部，将此设置为 true 最终将为每个任务使用一个连接。

  在一些事务中，第一个查询是轻量级的，需要很少的连接，而后续查询又比较复杂，需要较多连接，这时，就可以考虑设置此参数。因为uxmpp 会根据第一个语句决定在事务中使用多少个连接。

  BEGIN;
  //添加提示
  SET uxmpp.force_max_query_parallelization TO ON;
  //一个不需要很多连接的轻量级查询
  SELECT count(*) FROM table WHERE filter = x;
  //因为强制了最大并行化
  //查询可以从更多连接中得到优化
  SELECT ... very .. complex .. SQL;
  COMMIT;

  默认值为 false。

4.7.7. 解释输出

- uxmpp.explain_all_tasks（boolean）

  默认情况下，在分布式查询上运行EXPLAIN时显示单个任意任务的输出。大多数情况下，EXPLAIN在各个任务中都是差不多的。但是有些时候，某些任务的计划方式不同或执行时间特别长。在此情况下，启用此参数可以输出所有任务的EXPLAIN。当然，这也会导致EXPLAIN花费更长的时间。

- uxmpp.explain_analyze_sort_method（enum）

确定 EXPLAIN ANALYZE输出中任务的排序方法。缺省值为execution-time。

支持的值如下所示。

- execution-time：按执行时间排序。
- taskId：按任务 ID 排序。

第 5 章 uxmpp集群管理

5.1. 分片数量

集群中的节点数量很容易更改(更改方法请参见第 5.3 节 “节点平衡”)，但是当创建分布式表后，修改分布式表分片数量会比较复杂，所以在创建分布式表之前，应该选择合适的分片数。合适的分片数既可以实现多分片的灵活性，也可以兼顾到对于跨分片进行查询计划和查询执行的开销。

- 多租户用例

  对于分片数的选择取决于对数据的访问模式。建议选择32个~128个分片。对于较小的工作负载，如小于100GB，可以选择32个分片；对于较大的工作负载，可以选择64或128个分片。

- 实时分析用例

  在实时分析的模型中，分片总数和worker的CPU核数有关。为了确保最大并行性，每个核数至少有一个分片。建议一个worker节点上的分片数为该节点CPU核数的2倍或者4倍。

5.2. 初始硬件规模

就节点数量和硬件自身大小容量来说，uxmpp集群的大小可以通过扩展来进行更改。但是，仍需选择一个合适的初始大小。以下是一些建议，不作绝对说明。

- 多租户用例

  对于从现有的单节点数据库实例迁移到uxmpp实例中，建议选择的worker节点和原单节点的CPU核数、内存相等。这样基本可以得到2~3倍的的性能提升，因为分片提高了资源利用率。

  master节点相比worker节点，对内存的需求更小一点。但是对内核数的要求取决于实际的工作负载情况（读/写吞吐量）。

- 实时分析用例

  总核数：当内存合适的时候，那个uxmpp的性能和核数数量成正比。要确定一个合适的核数，需要考虑单节点中查询的延时和uxmpp本身所需的延时。将单节点的延时除以uxmpp本身所需的延迟，将得到的结果值四舍五入，即为合适的核数。

  工作节点内存：理论上内存越大越好。应用程序使用的查询类型会对内存大小有影响。

5.3. 节点平衡

uxmpp 基于逻辑分片的架构，支持在不停机的情况下横向扩展集群。本节介绍如何向 uxmpp 集群添加更多节点，以提高查询性能与可伸缩性。

- 添加新节点

  uxmpp 将分布式表的所有数据存储在工作节点上。因此，如果要通过添加更多计算能力来横向扩展群集，则可以通过添加worker节点来实现。

  要想添加新节点，首先需要在ux_dist_node中添加该节点的IP和端口。可以使用 master_add_node来添加。

SELECT * FROM master_add_node ('node-name',node-port);

新节点可用于新分布式表分片，原来的分布式表分片保留原样，除非重新分配。因此，如果不重新分配数据或者不新建分布式表的话，增加新节点没有任何意义。

> 注意
>
> 添加新节点之后，将新节点的集群信息写入.uxpass文件中（请参见第 4.1 节 “部署”）。

如果您的集群具有非常大的引用表，它们可能会减慢节点的添加速度。在这种情况下，请考虑设置uxmpp.replicate_reference_tables_on_activate（布尔）GUC。

首先，从协调器节点检查其他工作线程是否使用SSL，命令如下所示。

SELECT run_command_on_workers('show ssl');

如果没有，则连接到新节点，并允许它在必要时通过纯文本进行通信，命令如下所示。

ALTER SYSTEM SET uxmpp.node_conninfo TO 'sslmode=prefer';
SELECT ux_reload_conf();

- 在不停机的情况下重新平衡分片

将现有的分布式表要重新分配到新的节点rebalance_table_shards函数和uxmpp_drain_node函数进行重新分配。

扩容：对于新增的节点，可以将之前已经存在的分布式表重新平衡，即扩容。rebalance_table_shards函数将指定的分布式表重新分配给新增的worker中。示例如下所示。

SELECT rebalance_table_shards('github_events');

缩容：对于之前有分布式表分片的节点，需要删除这个节点，则需要把这个节点中的分片 收回，再分布到其他worker节点中，即缩容。uxmpp_drain_node函数将指定节点中的分 布式表的分片收回，并重新分配到其他worker节点中。示例如下所示。

SELECT * from uxmpp_drain_node('10.0.0.1', 5432);

> 注意
>
> 分片重新平衡在移动分片时阻止写入访问。

如果要将现有分片移动到新添加的工作节点，uxmpp 提供了一个rebalance_table_shards函数来简化此操作。此函数将移动给定表的分片，以在工作线程之间均匀分布它们。

该函数可配置为根据多种策略重新平衡分片，以最好地匹配您的数据库工作负载。请参阅函数参考，了解要选择的策略。下面是使用默认策略重新平衡分片的示例：

SELECT rebalance_table_shards();

已发布的表必须配置“副本标识”，以便能够复制 UPDATE 和 DELETE 操作，以便在订阅服务器端标识要更新或删除的相应行。默认情况下，副本标识将是主键（如果有）。也可以将另一个唯一索引设置为副本标识。

也就是说，如果分布式表定义了主键，那么它就可以进行分片重新平衡，而无需额外的工作。但是，如果它没有主键或显式定义的副本标识，则尝试重新平衡它将导致错误。示例如下所示。

```
//创建表时不指定 REPLICA IDENTITY 或 PRIMARY KEY
CREATE TABLE test_table (key int not null, value text not null);
SELECT create_distributed_table('test_table', 'key');
//添加新的工作节点
//以模拟分片重新平衡
//使用默认策略执行分片重新平衡
SELECT rebalance_table_shards('test_table');

/*
NOTICE:  Moving shard 102040 from localhost:9701 to localhost:9700 ...
ERROR: cannot use logical replication to transfer shards of the
  relation test_table since it doesn't have a REPLICA IDENTITY or
  PRIMARY KEY
DETAIL:  UPDATE and DELETE commands on the shard will error out during
  logical replication unless there is a REPLICA IDENTIY or PRIMARY KEY.
HINT:  If you wish to continue without a replica identity set the
  shard_transfer_mode to 'force_logical' or 'block_writes'.
*/
```

下面介绍了如何修复此错误。

1. 如果要复制的表已具有包含分布列的唯一索引，请选择该索引作为副本标识。

   ```
   //假设表my_table具有唯一索引my_table_idx
   //并且包含分布列
   ALTER TABLE my_table REPLICA IDENTITY USING INDEX my_table_idx;
   ```

   > **注意**
   >
   > 虽然REPLICA IDENTITY USING INDEX没问题，但我们建议不要添加REPLICA IDENTITY FULL到表中。此设置将导致每次UPDATE/DELETE在订阅服务器端执行完整表扫描，以查找包含这些行的元组。这会导致性能大幅下降。

2. 如果表不具有包含分布列的唯一索引，可以向表中添加主键。添加的主键需要在分布列上或者包含分布列。

3. 如果分布式表没有主键或副本身份，并且不能添加一个副本身份，可以在仅接收读取和插入（无删除或更新）的表上执行此操作。设置以下参数：

   ```
   SELECT rebalance_table_shards('test_table', shard_transfer_mode => 'force_logical');
   ```

   在这种情况下，如果应用程序在复制过程中尝试更新或删除，则请求将仅返回错误。直到复制完成后，才可以继续删除和写入。

第 6 章 用例指南

6.1. 广告分析（多租户模型）

该模型记录多个公司（companies），每个公司都有广告系列（campaigns），每个活动里有多个广告（ads），每个广告都有相关的点击量（clicks）和显示次数（impressions）。

```
CREATE TABLE companies (
  id bigserial PRIMARY KEY,
  name text NOT NULL,
  image_url text,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);
CREATE TABLE campaigns (
  id bigserial PRIMARY KEY,
  company_id bigint REFERENCES companies (id),
  name text NOT NULL,
  cost_model text NOT NULL,
  state text NOT NULL,
  monthly_budget bigint,
  blacklisted_site_urls text[],
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);
CREATE TABLE ads (
  id bigserial PRIMARY KEY,
  campaign_id bigint REFERENCES campaigns (id),
  name text NOT NULL,
  image_url text,
  target_url text,
  impressions_count bigint DEFAULT 0,
  clicks_count bigint DEFAULT 0,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);
CREATE TABLE clicks (
  id bigserial PRIMARY KEY,
  ad_id bigint REFERENCES ads (id),
  clicked_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_click_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL
);
CREATE TABLE impressions (
  id bigserial PRIMARY KEY,
  ad_id bigint REFERENCES ads (id),
  seen_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_impression_usd numeric(20,10),
  user_ip inet NOT NULL,
```

```
  user_data jsonb NOT NULL
);
```

对这个模型进行一些修改，可让其在uxmpp中可以更高效的运行。

关系数据模型非常适合应用程序。它保护数据完整性，允许灵活的查询，并适应不断变化的数据。但是，关系数据库不能满足大型SaaS应用程序所需的工作负载。必须通过NoSQL数据库才能达到这个规模。

使用uxmpp，可以保留数据模型并使其扩展。uxmpp将应用程序看作是单个uxdb数据库，但它在内部将查询路由到可调数量的物理服务器（节点），这些服务器可以并行处理请求。

多租户应用程序有一个很好的属性：查询通常总是一次请求一个租户的信息，而不是租户的混合。例如，当销售人员在搜索潜在客户信息时，搜索结果特定于其客户，其他企业的潜在客户和信息不会包括在内。

由于应用程序查询仅限于单个租户（例如商店或公司），因此快速进行多租户应用程序查询的一种方法是将给定租户的所有数据存储在同一节点上。这可以最大限度地减少节点之间的网络开销，并允许uxmpp有效地支持所有应用程序的连接，关键约束和事务。有了这个，可以跨多个节点进行扩展，而无需完全重写或重新构建应用程序。

需要确保每个表都有一列来清楚地标记哪个租户拥有哪些行，从而在uxmpp中执行此操作。在广告分析应用程序中，租户是公司，因此必须确保所有表都有company_id列。

当行标记为同一公司时，可以使用此列将对应的行读取和写入行到同一节点。

6.1.1. 准备表和数据

为多租户应用程序确定正确的分发列：company_id。

创建的模式使用单独的id列作为每个表的主键。uxmpp要求主键和外键约束包括分发列。可以通过包含company_id来生成主键和外键组合。这与多租户案例兼容，因为真正需要的是确保每个租户的唯一性。

首先需要下载这些表的示例数据，如下所示，

```
curl https://examples.citusdata.com/tutorial/companies.csv > companies.csv
curl https://examples.citusdata.com/tutorial/campaigns.csv > campaigns.csv
curl https://examples.citusdata.com/tutorial/ads.csv > ads.csv
curl https://examples.citusdata.com/mt_ref_arch/clicks.csv > clicks.csv
curl https://examples.citusdata.com/mt_ref_arch/impressions.csv > impressions.csv
```

接下来对这些表进行改造。

```
CREATE TABLE companies (
  id bigserial PRIMARY KEY,
  name text NOT NULL,
  image_url text,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL
);
CREATE TABLE campaigns (
  id bigserial,      --  PRIMARY KEY
  company_id bigint REFERENCES companies (id),
  name text NOT NULL,
  cost_model text NOT NULL,
```

```
  state text NOT NULL,
  monthly_budget bigint,
  blacklisted_site_urls text[],
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL,
  PRIMARY KEY (company_id, id) -- added
);
CREATE TABLE ads (
  id bigserial,       -- PRIMARY KEY
  company_id bigint,  -- added
  campaign_id bigint, -- REFERENCES campaigns (id)
  name text NOT NULL,
  image_url text,
  target_url text,
  impressions_count bigint DEFAULT 0,
  clicks_count bigint DEFAULT 0,
  created_at timestamp without time zone NOT NULL,
  updated_at timestamp without time zone NOT NULL,
  PRIMARY KEY (company_id, id),         -- added
  FOREIGN KEY (company_id, campaign_id) -- added
    REFERENCES campaigns (company_id, id)
);
CREATE TABLE clicks (
  id bigserial,       -- was: PRIMARY KEY
  company_id bigint,   -- added
  ad_id bigint,       -- was: REFERENCES ads (id),
  clicked_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_click_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL,
  PRIMARY KEY (company_id, id),      -- added
  FOREIGN KEY (company_id, ad_id)    -- added
    REFERENCES ads (company_id, id)
);
CREATE TABLE impressions (
  id bigserial,        -- PRIMARY KEY
  company_id bigint,    -- added
  ad_id bigint,        -- REFERENCES ads (id),
  seen_at timestamp without time zone NOT NULL,
  site_url text NOT NULL,
  cost_per_impression_usd numeric(20,10),
  user_ip inet NOT NULL,
  user_data jsonb NOT NULL,
  PRIMARY KEY (company_id, id),       -- added
  FOREIGN KEY (company_id, ad_id)     -- added
    REFERENCES ads (company_id, id)
);
//分发表
SELECT create_distributed_table('companies',   'id');
SELECT create_distributed_table('campaigns',   'company_id');
SELECT create_distributed_table('ads',         'company_id');
SELECT create_distributed_table('clicks',      'company_id');
SELECT create_distributed_table('impressions', 'company_id');
```

```
//使用COPY命令进行批量加载数据（COPY时指定正确的数据路径）
\copy companies from 'companies.csv' with csv
\copy campaigns from 'campaigns.csv' with csv
\copy ads from 'ads.csv' with csv
\copy clicks from 'clicks.csv' with csv
\copy impressions from 'impressions.csv' with csv
```

6.1.2. 简单应用

对单租户进行一个简单的查询和更新。

```
//某公司预算最高的广告系列
SELECT name, cost_model, state, monthly_budget
  FROM campaigns
 WHERE company_id = 5
 ORDER BY monthly_budget DESC
 LIMIT 10;
//将其预算翻倍
UPDATE campaigns
   SET monthly_budget = monthly_budget*2
 WHERE company_id = 5;
```

对于使用NoSQL数据库扩展应用程序的用户来说，一个常见的问题是缺少事务和连接。然而，在uxmpp中，事务可以正常的使用。

```
//更改广告系列的预算资金
BEGIN;
UPDATE campaigns
   SET monthly_budget = monthly_budget + 1000
 WHERE company_id = 5
   AND id = 40;
UPDATE campaigns
   SET monthly_budget = monthly_budget - 1000
 WHERE company_id = 5
   AND id = 41;
COMMIT;
```

下面的示例演示一个包含聚合和窗口函数的查询，它在uxmpp中的工作方式与在uxdb中的工作方式相同。根据显示次数（impressions）对每个广告系列中的广告进行排名。

```
SELECT a.campaign_id,
       RANK() OVER (
         PARTITION BY a.campaign_id
         ORDER BY a.campaign_id, count(*) desc
       ), count(*) as n_impressions, a.id
  FROM ads as a,
       impressions as i
 WHERE a.company_id = 5
   AND i.company_id = a.company_id
   AND i.ad_id      = a.id
 GROUP BY a.campaign_id, a.id
 ORDER BY a.campaign_id, n_impressions desc;
```

简而言之，当查询范围限定为租户时，插入，更新，删除，复杂SQL和事务都可以正常使用。

6.1.3. 租户共享数据

到目前为止，所有表都已经分发company_id，但有时候租户需要共享数据，并且特是“不属于”任何租户的数据。例如，使用此示例广告平台的所有公司可能希望根据IP地址获取其受众的地理信息。在单个机器数据库中，这可以通过geo-ip的查找表来完成，如下所示。（实际的表可能会使用GIS。）

```
CREATE TABLE geo_ips (addrs cidr NOT NULL PRIMARY KEY, latlon point NOT NULL
 CHECK (-90  <= latlon[0] AND latlon[0] <= 90 AND
      -180 <= latlon[1] AND latlon[1] <= 180)
);
CREATE INDEX ON geo_ips USING gist (addrs inet_ops);
```

要在分布式设置中有效地使用此表，需要找到一种方法来共同定位geo_ips表。这样，在查询时不需要产生网络流量。通过指定geo_ips作为参考表。

```
//为所有工人制作geo_ips的同步副本
SELECT create_reference_table('geo_ips');
```

参考表在所有工作节点上复制，uxmpp会自动保持同步。

```
//加载数据
\copy geo_ips from 'geo_ips.csv' with csv
//查询
SELECT c.id, clicked_at, latlon
  FROM geo_ips, clicks c
 WHERE addrs >> c.user_ip
   AND c.company_id = 5
   AND c.ad_id = 290;
```

6.1.4. 租户不共享数据

例如，使用广告数据库的一个租户应用程序可能希望通过单击存储跟踪cookie信息，而另一个租户可能关心浏览器代理。传统上，对于多租户，使用共享模式方法的数据库只能创建固定数量的预先分配的“自定义”列，或者使用外部的“扩展表”。一种更简单非结构化列类型，JSONB格式。

请注意，表clicks已经调用了一个JSONB字段user_data。每个租户都可以使用它进行灵活存储。

假设公司包含用户是否在移动设备上的字段信息。该公司可以查询移动设备的点击次数。

```
SELECT user_data->>'is_mobile' AS is_mobile, count(*) AS count
  FROM clicks
 WHERE company_id = 5
 GROUP BY user_data->>'is_mobile'
 ORDER BY count DESC;
```

数据库管理员甚至可以创建部分索引来提高单个租户查询模式的速度。如通过用户在移动设备上的点击对公司5的过滤。

```
CREATE INDEX click_user_data_is_mobile
```

```
 ON clicks ((user_data->>'is_mobile'))
 WHERE company_id = 5;
```

此外，uxdb支持JSONB上的GIN索引。在JSONB列上创建GIN索引将在JSON文档中的每个键和值上创建索引。这将加速一些JSONB操作符。

```
CREATE INDEX click_user_data ON clicks USING gin (user_data);
//这加快了查询，例如“哪些点击有在user_data中存在is_mobile密钥？”
SELECT id
  FROM clicks
 WHERE user_data ? 'is_mobile'
   AND company_id = 5;
```

6.2. 实时分析

uxmpp提供大型数据集的实时查询。例如，一个云服务提供商，帮助其他企业监控其HTTP流量。每当一个客户端收到HTTP请求时，云服务就会收到一条日志记录。希望获取所有这些记录并创建HTTP分析仪表板，为客户提供洞察，例如其网站所服务的HTTP错误数量。重点是希望此数据显示尽可能少的延迟，以便客户可以修复其网站的问题。

或者，也许正在建立一个广告网络，并希望在其广告系列上向客户展示点击率。在此示例中，延迟也很关键，原始数据量也很高，历史数据和实时数据都很重要。

接下来构建一个类似的数据模型。使用一个简单的模式来读取HTTP事件数据。

```
//在master节点上运行。
CREATE TABLE http_request (
  site_id INT,
  ingest_time TIMESTAMPTZ DEFAULT now(),
  url TEXT,
  request_country TEXT,
  ip_address TEXT,
  status_code INT,
  response_time_msec INT
);
SELECT create_distributed_table('http_request', 'site_id');
```

对http_request使用site_id列进行hash分配。这意味着特定站点的所有数据都将存在于同一个分片中。

这里分片计数使用的默认值。建议在实际群集中分片数设置为CPU核数的2-4倍。

生成测试数据，如下所示。

1. 在uxsql控制台后台执行下面的SQL语句，创建一个pluxsql函数insert_http_request。该函数向表http_request插入一条数据，并随机休眠一段时间。

   ```
   CREATE OR REPLACE FUNCTION insert_http_request() RETURNS VOID AS $$
   BEGIN
     INSERT INTO http_request (
       site_id, ingest_time, url, request_country,
       ip_address, status_code, response_time_msec
     ) VALUES (
       trunc(random()*32), clock_timestamp(),
       concat('http://example.com/', md5(random()::text)),
   ```

```
    ('{China,India,USA,Indonesia}'::text[])[ceil(random()*4)],
    concat(
      trunc(random()*250 + 2), '.',
      trunc(random()*250 + 2), '.',
      trunc(random()*250 + 2), '.',
      trunc(random()*250 + 2)
    )::inet,
    ('{200,404}'::int[])[ceil(random()*2)],
    5+trunc(random()*150)
   );
  PERFORM ux_sleep(random() * 0.25);
  END;
  $$LANGUAGE pluxsql;
```

2. 新建一个SHELL脚本generate.sh，作用是循环不断地向表http_request插入数据。

```
#!/bin/bash
while [ 1 ]
do
   ./uxsql -c "SELECT insert_http_request()"
done
```

3. 执行SHELL脚本，生成实时数据。（如需停止生成数据，使用CTRL+Z中断脚本执行）

```
bash generate.sh
```

生成数据后，进行简单查询。

```
SELECT
 site_id,
 date_trunc('minute', ingest_time) as minute,
 COUNT(1) AS request_count,
 SUM(CASE WHEN (status_code between 200 and 299) THEN 1 ELSE 0 END) as success_count,
 SUM(CASE WHEN (status_code between 200 and 299) THEN 0 ELSE 1 END) as error_count,
 SUM(response_time_msec) / COUNT(1) AS average_response_time_msec
 FROM http_request
WHERE date_trunc('minute', ingest_time) > now() - '5 minutes'::interval
GROUP BY site_id, minute
ORDER BY minute ASC;
```

上述设置可以有效运行，但是有两个缺点：

1. 每次都必须遍历每一行。例如，如果客户对过去一年的趋势感兴趣，那么查询将从头开始汇总过去一年的每一行。

2. 存储成本将与数据摄取率和可查询历史记录的长度成比例增长。实际中，可能希望将原始事件保留较短的时间段（一个月），并查看较长时间段（年）的历史图表。

可以通过将原始数据汇总到预先聚合的表中来克服这两个缺点。在这里，将原始数据聚合到一个表单中，该表存储1分钟间隔的摘要。在生产系统中，可能还需要1小时和1天的间隔，这些间隔对应相应的缩放级别即可。当用户想要上个月的请求时间时，可以简单地读取并绘制过去30天中每一天的值。

```
CREATE TABLE http_request_1min (
```

```
 site_id INT,
 ingest_time TIMESTAMPTZ, --每分钟
 error_count INT,
 success_count INT,
 request_count INT,
 average_response_time_msec INT,
 CHECK (request_count = error_count + success_count),
 CHECK (ingest_time = date_trunc('minute', ingest_time))
);
SELECT create_distributed_table('http_request_1min', 'site_id');
CREATE INDEX http_request_1min_idx ON http_request_1min (site_id, ingest_time);
```

在这里，通过site_id来进行分布，并且分片数数和复制因子和http_request相同。因为这三个都是匹配的，所以分http_request片和分http_request_1min片之间存在一对一的对应关系，这形成了共址表，使连接等查询更快。

为了填充表http_request_1min将定期运行INSERT INTO SELECT。可以借助下面的存储过程，可以将其设置为定时任务，定时执行SELECT rollup_http_request();。

```
CREATE OR REPLACE FUNCTION rollup_http_request() RETURNS void AS $$
BEGIN
 INSERT INTO http_request_1min (
  site_id, ingest_time, request_count,
  success_count, error_count, average_response_time_msec
 ) SELECT
  site_id,
  minute,
  COUNT(1) as request_count,
  SUM(CASE WHEN (status_code between 200 and 299) THEN 1 ELSE 0 END) as
success_count,
  SUM(CASE WHEN (status_code between 200 and 299) THEN 0 ELSE 1 END) as error_count,
  SUM(response_time_msec) / COUNT(1) AS average_response_time_msec
 FROM (
  SELECT *,
   date_trunc('minute', ingest_time) AS minute
  FROM http_request
 ) AS h
 WHERE minute > (
  SELECT COALESCE(max(ingest_time), timestamp '1901-10-10')
  FROM http_request_1min
  WHERE http_request_1min.site_id = h.site_id
 )
  AND minute <= date_trunc('minute', now())
 GROUP BY site_id, minute
 ORDER BY minute ASC;
END;
$$LANGUAGE pluxsql;
```

再对其进行查询。

```
SELECT site_id, ingest_time as minute, request_count,
     success_count, error_count, average_response_time_msec
 FROM http_request_1min
WHERE ingest_time > date_trunc('minute', now()) - '5 minutes'::interval;
```

汇总使查询更快，但仍需要使旧数据过期以避免无限制的存储成本。只需确定希望为每个粒度保留数据的时间长度，并使用标准查询来删除过期数据。例如，决定将原始数据保留一天，每分钟保留一个月的聚合。

DELETE FROM http_request WHERE ingest_time < now() - interval '1 day';
DELETE FROM http_request_1min WHERE ingest_time < now() - interval '1 month';

> 注意
>
> 可以将这些查询包装在一个函数中，并在cron作业中每分钟调用一次。

6.3. 高可用实例

uxmpp的作为UXDB的扩展插件，也可以实现高可用。

uxmpp的高可用是通过给master搭建主备流复制，分片使用uxmpp的副本的形式来实现的。

1. 搭建uxmpp环境（包含一个master节点，至少两个worker节点，详细步骤请参见第 4.1 节“部署”）。

2. 搭建异步流复制环境，为master节点创建一个备服务器，详细步骤请参见《优炫数据库备份与还原手册 V2.1》中异步流复制章节。

3. master节点上的uxmpp集群修改配置参数。

   修改uxsinodb.conf。

   max_connections = 1000
   wal_level = replica
   hot_standby = on
   max_prepared_transactions = 2000

   修改ux_hba.conf。

   host replication all 0.0.0.0/0 md5

   新建recovery.done。

   vi recovery.done

   recovery_target_timeline = 'latest'
   standby_mode = on
   primary_conninfo = 'host=IP port=port user=username password=password'

   IP为master上uxmpp集群所在机器IP，port为master上uxmpp集群端口，username为master上uxmpp集群的用户名，password为master上uxmpp集群的用户密码。

   上述配置完成之后可以重启master节点上的uxmpp集群。

4. 修改master节点的复制因子。

   登录master集群控制台执行。

   alter system set uxmpp.shard_replication_factor TO 2;

重启集群使参数生效。

5. 备节点上生成备库。备节点请参见《优炫数据库备份与还原手册 V2.1》。

   在master备节点上的uxdb的bin目录下执行。

   ./ux_basebackup -D uxdbdata -F p -X stream -h masterIP -p 5432 -U uxdb

   控制选项

   -D，--uxdata

   指定把备份写到那个目录，如果这个目录或这个目录路径中的各级父目录不存在，则ux_basebackup就会自动创建这个目录，如果目录存在，但目录不为空，则会导致ux_basebackup执行失败。

   -F，--format=p|t

   指定输出格式。p表示原样输出，即把主数据库中的各个数据文件，配置文件、目录结构都完全一样的写到备份目录；t表示把输出的备份文件打包到一个tar文件中。

   -X，--xlog(wal)-method=fetch|stream

   指定WAL日志的存储方法。

   连接选项

   -h，--host=HOSTNAME

   指定需要连接的数据库所在地址。

   -p，--port=PORT

   指定需要连接的数据库的端口。

   -U，--username=NAME--username=NAME

   指定需要连接的数据库的用户名。

   进入备库集群目录执行。

   mv recovery.done recovery.conf

   启动备库。

6. 模拟故障。

   - 登录master主库创建测试表（分布式表），并插入数据。
   - 模拟主库故障（停止主库）。
   - 进入备库集群目录执行mv recovery.conf recovery.done（重启备节点集群，备节点转为读写主节点）。
   - 在备节点上对原数据进行读写。

- 模拟worker节点故障（停止一个worker）。
- 在备节点上对原表进行读操作。

第 7 章 查询性能调优

在本节中，将介绍如何调整uxmpp集群以获得最佳性能。首先了解一下选择正确的分发列对性能的影响。然后，将先在一台UXDB服务器上调优数据库以提高性能，再在所有的集群上进行扩展。在本节中，还将讨论一些与性能相关的配置参数。

7.1. 选择分发列

创建分布式表的第一步是选择正确的分发列。这有助于uxmpp将多个操作直接下推到worker分片，并排除不相关的分片，从而显着提高查询速度。

通常，对于分发列的选择。可以选择常用的连接键，或者大多数查询都用到的过滤键。对于过滤键，uxmpp使用分发列范围排除不相关分片，确保查询只匹配与WHERE子句重叠的分片。对于连接键，如果连接键和分发列相同，则uxmpp只在具有匹配/重叠分布列范围的碎片之间执行连接。所有这些碎片连接都可以在worker上并行执行，因此效率更高。

此外，uxmpp还可以根据分发列将多个操作直接下发到各个worker上，这大大减少了每个节点上的计算量和跨节点传输数据产生的网络负荷。

7.2. UXDB调优

uxmpp协调器将传入的查询分为片段查询，并将它们下发到worker上进行并行处理。worker其实只是扩展的UXDB服务器，这些查询将应用UXDB的标准规划和执行逻辑。因此，需要先调优worker上的UXDB配置参数以提高性能。

调整参数是一个实验的过程，通常需要多次尝试才能达到理想的性能。因此，在调整参数的时候，可以选择加载一小部分数据，加快迭代速度。

开始调优前，先创建一个uxmpp集群并加载一些数据。在master上执行EXPLAIN命令，查看worker是如何处理查询的，以及master如何汇总查询结果。

以github_events表为例。

```
EXPLAIN
 SELECT date_trunc('minute', created_at) AS minute,
       sum((payload->>'distinct_size')::int) AS num_commits
  FROM github_events
 WHERE event_type = 'PushEvent'
 GROUP BY minute
 ORDER BY minute;
```

```
                                                   QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Sort  (cost=0.00..0.00 rows=0 width=0)
   Sort Key: remote_scan.minute
   ->  HashAggregate  (cost=0.00..0.00 rows=0 width=0)
         Group Key: remote_scan.minute
         ->  Custom Scan (UXmpp Real-Time)  (cost=0.00..0.00 rows=0 width=0)
               Task Count: 32
               Tasks Shown: One of 32
               ->  Task
                     Node: host=192.168.1.83 port=5432 dbname=uxdb
                     ->  HashAggregate  (cost=31.58..33.08 rows=120 width=16)
                           Group Key: date_trunc('minute'::text, created_at)
                           ->  Seq Scan on github_events_104169 github_events  (cost=0.00..30.04 rows=124 width=533)
                                 Filter: (event_type = 'PushEvent'::text)
(13 行记录)
```

通过执行结果，可以了解执行过程。首先有32个分片，执行器选择的是adaptive：

```
->  Custom Scan (UXmpp Adaptive)  (cost=0.00..0.00 rows=0 width=0)
      Task Count: 32
      Tasks Shown: One of 32
```

接下来，展示了其中一个分片上的查询行为，指明了worker的主机、端口和数据库：

```
->  Task
      Node: host=192.168.1.83 port=5432 dbname=uxdb
```

最后，显示了查询片段在worker的UXDB服务器的EXPLAIN的结果：

```
->  HashAggregate  (cost=31.58..33.08 rows=120 width=16)
     Group Key: date_trunc('minute'::text, created_at)
     ->  Seq Scan on github_events_104169 github_events  (cost=0.00..30.04 rows=124 width=533)
           Filter: (event_type = 'PushEvent'::text)
```

现在，可以通过EXPLAIN的执行结果，连接到192.168.1.83:5432的worker上，调整分片github_events_104169的查询性能。调优过程中，可以通过在master上执行EXPLAIN来查看调优结果。

先进行配置参数调优。UXDB的默认设置都是比较保守的，其中shard_buffers和work_mem是对读取性能优化比较重要的参数。

shard_buffers定义了分配给数据库缓存数据的内存。默认是128MB。建议改值为当前节点的RAM的1/4—1/2。

如果查询中有很多复杂的排序，那么可以增加work_mem使排序在更大的内存中运行，这比在磁盘中排序要快的多。如果在worker节点中查看到有很多的磁盘活动，那么非常有必要增大该值。可以提高查询的效率，让更多的操作在内存中进行。

最后，可以在表上创建索引来增强数据库的性能。对于索引的选择，可以通过EXPLAIN查看查询计划并优化查询计划中较慢的部分。创建索引之后，索引必须和表保持同步，这会增加一部分数据操作的开销。因为，可以将不用的索引删除掉。

对于写入性能，可以使用常规的UXDB调优来提高INSERT的速率。通常建议增加checkpoint_timeout和max_wal_size设置。此外，根据应用程序的可靠性要求，可以选择更改fsync或者synchronous_commit的值（请参见《优炫数据库用户手册 V2.1》）。

在其中一个worker上调整了合适的参数，也同时需要手动去修改其他worker上的对应参数。可以在master上打开uxmpp.explain_all_tasks开关，并执行EXPLAIN来验证其他worker上的调优。这时候执行的EXPLAIN，将会显示所有分片的查询计划。

由于数据分布不均匀或者机器硬件的差异可能会导致worker程序执行的差异。可以运行EXPLAIN ANALYZE查询每个分片上运行的更详细的信息。

7.3. 其他调优

如上所述，一旦获取了单个分片所需的性能，可以在所有的worker上设置类似的参数配置。由于uxmpp在worker节点上运行所有的片段查询，因此可以将查询的扩展为集群中所有CPU的计算能力的累积。

应该尽可能多的使操作都在内存中进行，以获取更佳的性能。如果此举无法实现，建议使用SSD。因为当对连续的数据块进行连续的读取时，HDD确实不错，但是随机读/写时，性能会大大下降。如果出现大量的并发查询进行随机读写，SSD的性能要比HDD提高好几倍。此外，如果查询是高度计算密集型的，那么需要选择具有更强大的计算能力的CPU。

影响性能的另一个因素是每个worker的分片数。uxmpp将传入的查询分区为在各个工作分片上运行的片段查询。因此，每个查询的并行度由查询命中的分片数决定。要确保最大并行度，应在每

个节点上创建合适的分片，每个CPU核至少有一个分片。要记住的另一个考虑因素是，如果查询在分发列上有过滤，uxmpp将排除不相关的分片。因此，创建比CPU核数多的分片也是有益的，这样即使在分片修剪后也可以实现更大的并行性。

7.4. 分布式查询性能调优

在群集中分发数据后，每个worker都会针对最佳性能进行优化。在此之后，是调整一些分布式性能调整参数。

在讨论特定配置参数之前，建议查看分布式群集上的查询时间（启用\timing），并将它们与单个分片性能进行比较。这有助于确定在master和worker上花费的时间，找出瓶颈并进行相应的优化。

本节中，将讨论有助于优化分布式查询计划程序和执行程序的参数。分两部分讨论，一般和高级。一般性能调优满足大部分用例，涵盖所以常见的配置。高级性能调优主要是在特定用例中提供性能增益。

- 一般调优

  对于INSERT来说，对插入率影响最大的因素是并发级别。可以尝试并行运行多个INSERT语句。这样，如果具有强大的master节点并且能够同时使用该节点上的所有CPU核心，则可以实现非常高的插入速率。

  uxmpp有两种执行器类型用于运行SELECT查询。可以通过设置uxmpp.task_executor_type配置参数来选择所需的执行程序。如果用例主要需要简单的键值查找或需要对聚合和连接进行亚秒级响应，则可以选择实时执行程序。另一方面，如果有长时间运行的查询需要跨节点重新分区和重排数据，那么可以切换到任务跟踪器执行程序。

  除了上述之外，有两个配置参数在产生近似有意义的结果的情况下是有用的。这两个参数是uxmpp.limit_clause_row_fetch_count和uxmpp.count_distinct_error_rate。前者设置在计算时限制从每个任务获取的行数，而后者在计算近似不同计数时设置所需的容错率。

  对于子查询/CTE，最好的情况下，在一个步骤中包含子查询/CTE。因为常是因为主查询和子查询都按表的分布列以相同的方式过滤，并且可以一起下推到工作节点。但是，uxmpp有时会在执行主查询前强制执行子查询，将子查询产生的中间结果复制到其他worker节点上供主查询使用。

  如果子查询在单独的步骤中执行，则可以避免worker之间进行过多的数据传输，产生过多的网络开销，从而影响性能。

  此外，可以通过设置uxmpp.max_intermediate_result_size，调整中间结果的大小，默认是1GB，这足以允许一些低效查询。

- 高级调优

  - 任务分配策略

    uxmpp查询计划程序根据分片位置将任务分配给工作节点。可以通过设置uxmpp.task_assignment_policy配置参数来选择进行这些分配时使用的算法。用户可以更改此配置参数以选择最适合其用例的策略（修改参数方法请参见第 4.7 节 “参数配置”。

  - 中间数据传输格式

    有两个配置参数与中间数据将跨节点间传输的格式相关，uxmpp.binary_master_copy_format和uxmpp.binary_worker_copy_format。启用前者使

用二进制格式将中间查询结果从worker传输到master，而后者在worker间动态传输中间数据非常有用。

uxmpp默认以文本格式传输中间查询数据。因为文本文件通常具有比二进制表示更小的大小。因此，这会在写入和传输中间数据时导致较低的网络和磁盘I/O.但是，对于某些数据类型（如hll或hstore数组），序列化和反序列化数据的成本非常高。在这种情况下，使用二进制格式传输中间数据可以提高查询性能，因为CPU使用率降低。

- 实时执行程序（real-time）

  如果需要亚秒响应时间的SELECT查询，则应使用实时执行程序。

  实时执行程序打开一个连接，并为每个未修剪的分片使用两个文件描述符（在规划期间不相关的分片被排除）。因此，如果查询遇到大量分片，执行程序可能需要打开比max_connections更多的连接，或者使用比max_files_per_process更多的文件描述符。

  这种情况下，实时执行程序将开始限制任务，以防止worker的压倒性资源。由于此限制可能会降低查询性能，因此实时执行程序将发出警告，建议应增加max_connections或max_files_per_process。

- 任务跟踪执行程序

  如果查询需要重新分区数据或更有效的资源管理，则应使用任务跟踪器执行程序。有两个配置参数可用于调整任务跟踪器执行程序的性能。

  第一个是uxmpp.task_tracker_delay。任务跟踪器进程定期唤醒，遍历分配给它的所有任务，并安排和执行这些任务。此参数设置任务跟踪器在这些任务管理轮次之间的休眠时间。当分片查询很短并可以非定期更新其状态时，可以减少此参数的值。

  第二个参数是uxmpp.max_running_tasks_per_node。此配置值设置在任何给定时间在一个worker节点上并发执行的最大任务数。此配置条目可避免多个任务同时访问磁盘，并有助于避免磁盘I/O争用。如果查询是从内存或SSD提供的，则可以增加此参数的值。

第 8 章 uxmpp特性

8.1. 继承表

8.1.1. 概述

继承表是拥有父表角色或子表角色的表，系统表ux_inherit保存所有的继承关系，如果某个表的OID在字段ux_inherit.inhrelid出现，那么它拥有“子表”角色，如果表OID在字段ux_inherit.inhparent出现，则拥有 “父表”角色。两种情形中的任意一种，都称该表是一个“继承表”。

8.1.2. 详细功能

继承作为UXDB使用者的一种有用工具，具备若干特殊的性质。继承表存在两种角色：父表和子表，两者之间是多对多的关系，也就是说，一个父表可能存在多个子表，反过来，一个子表可能存在不止一个父表。继承表的性质具体体现在：

1. 数据结构的继承

   一个表可能存在零个或多个父表，子表继承它的所有父表的所有列，任一父表数据结构的变化自动传播到子表，这里的数据结构变化包含增加或删除列，修改列的数据类型等。子表可以包含自定义的列，这些自定义的列是子表区别于父表的特性部分。OID类型的系统字段tableoid能标识当前列的真实来源。如果子表包含与父表列名字相同，数据类型也相同的列，这些列与父表列“合并”，子表只存在一个这样的列。

2. check约束、NOT NULL约束及DEFAULT值的继承

   子表自动继承父表的所有check约束，NOT NULL约束，及DEFAULT值。如果子表与父表包含相同的列，子表列上的check约束,NOT NULL约束和DEFAULT值必须与父表完全相同，否则子表的列不能与父表列“合并”。

3. 查询、更新和删除父表

   默认情况下，查询，更新和删除父表引用了父表的行以及它的所有后代的所有行。如果仅查询父表中符合条件的元组，需要明确指定关键字ONLY。

4. 继承关系的创建和删除

   继承关系有两种创建方式，一种是创建子表时通过INHERITS子句指定父表，另一种是对于已存在的两个表，通过ALTER TABLE … INHERIT语法建立两者之间的继承关系。对于第二种方式，子表中必须已包含与父表具有相同名字，相同数据类型，相同DEFAULT值，相同NOT NULL约束的列，另外，如果父表包含CHECK约束，子表也必须包含具有相同名字和表达式的CHECK约束。而删除继承关系使用语法ALTER TABLE … NO INHERIT。

uxmpp支持对表分片，表中数据被分布存储在多个worker节点。目前uxmpp不支持对继承层次中的表进行分片（partition表除外）。另外，uxmpp无法把分片表作为子表或父表。结合继承表性质，uxmpp支持继承表的以下功能。

表 8.1. 继承表功能

一级功能	二级功能
分片	对于已存在继承关系的表，允许分片存储

一级功能	二级功能
修改	对于分片表，支持将其设置为其它表的父表或者子表
	父表数据结构的修改（增加或删除列，修改列的类型），应传播到worker节点的子表分片。
	父表check/not null/default约束的增加或删除，传播到worker节点的子表分片
查询	select的返回结果包含子表中符合条件的元组
	select only只包含父表中符合条件的元组
更新	update传播到子表分片
	update only只修改父表中符合条件的元组
删除	delete传播到子表
	delete only只删除父表中符合条件的元组

8.1.3. 限制条件

为了使父表分片和子表分片能一一对应，uxmpp继承表分片时，必须满足以下约束条件：

- 分片函数

  分片函数应使用函数create_distributed_table。另外，目前暂不支持将继承表创建为参考表，以后可能支持。

- 分片方法

  分片方法必须是hash。函数create_distributed_table接受分片方法作为第三个参数，支持的分片方法包括hash，append和range。由于append和range方法并不真正创建分片，因此，继承表必须使用hash分片。

- 复制因子

  为了简化问题，暂时规定继承表分片时复制因子必须1，每个分片都只有一份。

- worker节点

  父表分片和子表分片所在的worker节点必须完全相同。不能父表分片分布在节点1，2，而子表分片分布在节点2，3。这一点也是为了保证每个子表分片所在的worker节点存在一个父表分片。

8.1.4. 处理流程

通过改造函数create_distributed_table，以支持继承表分片。由于继承表分片必须满足若干条件，所以定义函数CheckInheritedTable实施必要的检查。

表 8.2. CheckInheritedTable函数

API	CheckInheritedTable
目的	检查继承表，如果不满足分片限制，报错
参数	参数描述
relationId	表的OID

API	CheckInheritedTable
目的	检查继承表，如果不满足分片限制，报错
参数	参数描述
relationId	表的OID
distributionColumn	分片列
distributionMethod	分片方法
返回值	无

表 8.3. checkIfTableSameDistributed函数

API	checkIfTableSameDistributed
目的	检查已存在的分片表是否与待分片表使用相同的复制因子，分片数和相同的worker节点，如果不是，提示错误。
参数	参数描述
existingRelationId	已分片表的OID
isParent	existingRelationId是否是父表
oDistributedRelationId	待分片表的OID
shardcount	待分片表的分片数
replicationfactor	待分片表使用的复制因子
workerNodes	待分片表的各个分片依次存储在哪些worker
返回值	无

8.1.5. 修改继承表

8.1.5.1. 修改继承关系

修改继承关系的语法是ALTER TABLE c INHERIT/NO INHERIT p …。uxmpp主要考虑当c和p中至少有一个表是哈希分片表。对于c和p均是本地表的情形，不需要特殊处理，直接交给UXDB的规划器处理就足够了。

8.1.5.2. 数据定义/CHECK约束/NOT NULL约束/DEFAULT

当父表操作涉及以下几方面时，传播到子表及子表的各个分片。

- 增加/删除列
- 修改列的数据类型
- 增加/删除可继承的CHECK约束
- 增加/删除NOT NULL约束
- 增加/删除DEFAULT值
- 重命名列
- 重命名可继承的CHECK约束

最后两项重命名操作对应于UXDB的语法ALTER TABLE … RENAME [COLUMN]…和ALTER TABLE … RENAME CONSTRAINT…，计划节点的类型是T_RenameStmt。前几种操作的计划节点类型是T_AlterTableStmt。

如果父表和子表均是分片表，worker节点记录了父表分片和子表分片的继承关系。当master节点修改父表的数据定义或约束时，父表分片的数据定义或约束相应地发生变化，进而传播到worker节点的子表分片。当父表或子表存在一个本地表时，由于本地表没有分片，worker节点无法记录分片间的继承关系，这种情况下父表数据定义及约束的修改无法在worker节点的分片之间传播，因此，当分片的父表有一个本地表作为孩子，或者本地的父表有一个分片表作为孩子时，必须限制用户修改父表的数据定义及约束。考虑到对多继承的支持，上述条件进而修正为，当分片的父表有一个本地表后代，或者本地的父表有一个分片表后代时，必须限制用户修改父表的数据定义及约束。

8.1.5.3. 查询/更新/删除继承表

查询、更新及删除继承表，主要涉及以下功能：

1. 如果当前表是分片表，存在至少一个本地表后代，或者当前表是本地表，存在至少一个分片表后代，不允许对表进行查询，更新和删除。这是因为查询正确性依赖于worker节点上记录的父表分片与子表分片的继承关系，而这两种情况下，worker节点并没有保存分片的继承关系。
2. 对于继承层次是均是分片表的情形，SELECT/UPDTAE/DELETE应当能够成功。
3. 对于继承层次是均是分片表的情形，SELECT ONLY/UPDTAE ONLY/DELETE ONLY应当只引用父表元组。

8.1.6. 注意事项

如果父表是本地表，且存在分片表后代，那么select/update/delete目前仍是成功的，预期结果是提示用户该SQL查询不支持。

在根据Query结构恢复查询字符串时，一些不需要ONLY关键字的查询目标之前有多余的ONLY关键字。这些不影响查询结果，但其实是多余的。

允许复制因子大于1，允许参考表成为继承表。

8.1.7. 示例

8.1.7.1. 环境部署

表 8.4. 机器及相关信息

机器	身份	IP address	Port	数据库
master	Coordinator	192.168.5.211	5432	uxdb
worker001	worker	192.168.5.214	5432	uxdb
worker002	worker	192.168.5.215	5432	uxdb

8.1.7.2. 创建继承关系

- 方法一：创建子表时通过INHERITS语句指定父表，建立继承关系。

  1. 设置复制因子。

set uxmpp.shard_replication_factor to 1;

2. 创建两个本地表，并将其分片，其中parent是child的父表。

create table parent(no int default 18, name text not null check(name is not null));
select create_distributed_table('parent', 'no');
create table child(hobbies text) inherits (parent);
select create_distributed_table('child', 'hobbies');

```
uxdb=# set uxmpp.shard_replication_factor to 1;
SET
uxdb=# create table parent(no int default 18, name text not null check(name is not null));
CREATE TABLE
uxdb=# select create_distributed_table('parent', 'no');
 create_distributed_table
--------------------------

(1 row)

uxdb=# create table child(hobbies text) inherits (parent);
CREATE TABLE
uxdb=# select create_distributed_table('child', 'hobbies');
 create_distributed_table
--------------------------

(1 row)
```

3. 查看子表分片的表结构。

select a.shardid as shardid, a.nodename as nodename, a.nodeport as nodeport
from ux_dist_shard_placement a, ux_dist_shard b
where a.shardid = b.shardid and b.logicalrelid = 'child'::regclass order by a.shardid limit 1;
\gset
\c - - :nodename :nodeport
\d child_:shardid

```
uxdb=# select  a.shardid as shardid, a.nodename as nodename, a.nodeport as nodeport from
uxdb-# ux_dist_shard_placement a, ux_dist_shard b where a.shardid = b.shardid and
uxdb-# b.logicalrelid = 'child'::regclass order by a.shardid limit 1;
 shardid |   nodename    | nodeport
---------+--------------+----------
  102168 | 192.168.5.214 |     5432
(1 row)

uxdb=# \gset
uxdb=# \c - - :nodename :nodeport
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
You are now connected to database "uxdb" as user "uxdb" on host "192.168.5.214" at port "5432".
uxdb=# \d child_:shardid
           Table "public.child_102168"
 Column  |  Type   | Collation | Nullable | Default
---------+---------+-----------+----------+---------
 no      | integer |           |          | 18
 name    | text    |           | not null |
 hobbies | text    |           |          |
Check constraints:
    "parent_name_check" CHECK (name IS NOT NULL)
Inherits: parent_102136
```

可看到子表分片继承了父表分片的表结构，默认值，not null，check约束等。

4. 登录父分片表所在的worker通过\d + 父分片表名，查看结果。

```
uxdb=# \d parent_102136
           Table "public.parent_102136"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 no     | integer |           |          | 18
 name   | text    |           | not null |
Check constraints:
    "parent_name_check" CHECK (name IS NOT NULL)
Number of child tables: 1 (Use \d+ to list them.)
```

- 方法二：通过ALTER TABLE ... INHERIT语句建立两张表之间的继承关系；这种方式子表中必须包含与父表具有相同名字，相同数据类型，相同NOT NULL约束的列，另外，如果父表包含CHECK约束，子表也必须包含具有相同名字和表达式的CHECK约束。

  1. 创建两个本地表，并将其分片。其中 liu_bei是liu_shan的父表。

     create table liu_bei(no int , name text);
     create table liu_shan(no int, name text, oname varchar(64));
     select create_distributed_table('liu_shan', 'oname');
     alter table liu_shan inherit liu_bei;
     select create_distributed_table('liu_bei', 'no');

```
uxdb=# create table liu_bei(no int, name text);
CREATE TABLE
uxdb=# create table liu_shan(no int, name text, oname varchar(64));
CREATE TABLE
uxdb=# select create_distributed_table('liu_shan', 'oname');
 create_distributed_table
--------------------------

(1 row)

uxdb=# alter table liu_shan inherit liu_bei;
ALTER TABLE
uxdb=# select create_distributed_table('liu_bei', 'no');
 create_distributed_table
--------------------------

(1 row)
```

  2. 查看子表分片的表结构。

     select a.shardid as shardid, a.nodename as nodename, a.nodeport as nodeport
     from ux_dist_shard_placement a, ux_dist_shard b
     where a.shardid = b.shardid and b.logicalrelid = 'liu_shan'::regclass order by a.shardid
     limit 1;
     \gset
     \c - - :nodename :nodeport
     \d liu_shan_:shardid

```
uxdb=# select  a.shardid as shardid, a.nodename as nodename, a.nodeport as nodeport from
uxdb-# ux_dist_shard_placement a, ux_dist_shard b where a.shardid = b.shardid and
uxdb-# b.logicalrelid = 'liu_shan'::regclass order by a.shardid limit 1;
 shardid |   nodename    | nodeport
---------+---------------+----------
  102200 | 192.168.5.214 |     5432
(1 row)

uxdb=# \gset
uxdb=# \c - - :nodename :nodeport
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
You are now connected to database "uxdb" as user "uxdb" on host "192.168.5.214" at port "5432".
uxdb=# \d  liu_shan_:shardid
                Table "public.liu_shan_102200"
 Column |         Type          | Collation | Nullable | Default
--------+-----------------------+-----------+----------+---------
 no     | integer               |           |          |
 name   | text                  |           |          |
 oname  | character varying(64) |           |          |
Inherits: liu_bei_102232
```

3. 登录父分片表所在的worker通过\d + 父分片表名，查看结果。

```
uxdb=# \d liu_bei_102232
            Table "public.liu_bei_102232"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 no     | integer |           |          |
 name   | text    |           |          |
Number of child tables: 1 (Use \d+ to list them.)
```

4. 测试建立继承关系时，对几种约束的要求。

create table sun_jian(name text not null check(name is not null));
create table sun_ce(name text not null);
create table sun_quan(name text);

需要有相同的“not null”、“check”。

```
uxdb=# create table sun_jian(name text not null check(name is not null));
CREATE TABLE
uxdb=# create table sun_ce(name text not null);
CREATE TABLE
uxdb=# create table sun_quan(name text);
CREATE TABLE
uxdb=# alter table sun_ce inherit sun_jian;
ERROR:  child table is missing constraint "sun_jian_name_check"
uxdb=# alter table sun_quan inherit sun_jian;
ERROR:  column "name" in child table must be marked NOT NULL
uxdb=#
```

create table cao_cao(age int default 18);
create table cao_pi(age_ int default 18);
create table cao_zhi(age smallint default 18);
create table cao_zhang(age int);

需要有相同的属性名、数据类型，但“default”可不一致。

```
uxdb=# create table cao_cao(age int default 18);
CREATE TABLE
uxdb=# create table cao_pi(age_ int default 18);
CREATE TABLE
uxdb=# create table cao_zhi(age smallint default 18);
CREATE TABLE
uxdb=# create table cao_zhang(age int);
CREATE TABLE
uxdb=# alter table cao_zhi inherit cao_cao ;
ERROR:  child table "cao_zhi" has different type for column "age"
uxdb=# alter table cao_pi inherit cao_cao ;
ERROR:  child table is missing column "age"
uxdb=# alter table cao_zhang inherit cao_cao ;
ALTER TABLE
```

8.1.7.3. 删除继承关系

1. 在ux_inherit系统表中查询所有的继承关系。

   select inhrelid::regclass, inhparent::regclass from ux_inherits;

```
uxdb=# select inhrelid::regclass, inhparent::regclass from ux_inherits;
 inhrelid | inhparent
----------+-----------
 child    | parent
 worker_1 | person
(2 rows)
```

2. 通过命令删除关系。

   alter table worker_1 no inherit person;

```
uxdb=# alter table worker_1 no inherit person;
ALTER TABLE
uxdb=# select inhrelid::regclass, inhparent::regclass from ux_inherits;
 inhrelid | inhparent
----------+-----------
 child    | parent
(1 row)
```

3. 在worker查看ux_inherit系统表中所有的继承关系。

   select inhrelid::regclass, inhparent::regclass from ux_inherits;

```
uxdb=# select inhrelid::regclass, inhparent::regclass from ux_inherits;
   inhrelid   |   inhparent
--------------+---------------
 child_102041 | parent_102009
 child_102043 | parent_102011
 child_102045 | parent_102013
 child_102047 | parent_102015
 child_102049 | parent_102017
 child_102051 | parent_102019
 child_102053 | parent_102021
 child_102055 | parent_102023
 child_102057 | parent_102025
 child_102059 | parent_102027
 child_102061 | parent_102029
 child_102063 | parent_102031
 child_102065 | parent_102033
 child_102067 | parent_102035
 child_102069 | parent_102037
 child_102071 | parent_102039
(16 rows)
```

8.2. MX

8.2.1. 概述

uxmpp的架构中正常只有1个CN节点，有时候CN会成为性能瓶颈。可以通过减少分片数，垂直扩容CN节点等手段缓解CN的性能问题，但这些都不能治本。某些业务场景部署多个CN节点是非常必要的。

8.2.2. 技术方案

如果CN上主要的负载来自查询，可以为CN节点配置多个备机，做读写分离，这些备机可以分担读负载。但是这种方案不能称为多CN，它不具有均衡写负载的能力。

在uxmpp的具体实现中，CN和worker的区别就在于是否存储了相关的元数据，如果把CN的元数据拷贝一份到worker上，那么worker也可以向CN一样工作，这个多CN的模式早期被称做masterless。

对于当前的uxmpp版本，有一个开关，打开后，会自动拷贝CN的元数据到worker上，让worker也可以当CN用。下面进行验证。

在CN节点的uxsinodb.conf中添加下面的参数。

uxmpp.replication_model='streaming'

在CN节点上添加worker节点。

SELECT * from master_add_node('wk1', 5432);
SELECT * from master_add_node('wk2', 5432);

从CN复制元数据到第一个worker节点。

SELECT start_metadata_sync_to_node('wk1', 5432);

执行上面的函数后，uxmpp CN上的元数据会被拷贝到指定的worker上，并且ux_dist_node表中对应worker的hasmetadata字段值为true，标识这个worker存储了元数据，以后创建新的分片时，新产生的元数据也会自动同步到这个worker上。

```
uxdb=# select * from ux_dist_node;
 nodeid | groupid | nodename | nodeport | noderack | hasmetadata | isactive | noderole | nodecluster
--------+---------+----------+----------+----------+-------------+----------+----------+-------------
      1 |       1 |      wk1 |     5432 | default  | t           | t        | primary  | default
      2 |       2 |      wk2 |     5432 | default  | f           | t        | primary  | default
(2 rows)
```

在CN节点上创建一个测试分片表。

```
create table tb1(id int primary key, c1 int);
set uxmpp.shard_count=8;
select create_distributed_table('tb1','id');
insert into tb1 select id,random()*1000 from generate_series(1,100)id;
```

因为wk1带了元数据，可以当CN用，下面这个SQL可以在wk1上执行。

```
uxdb=# explain select * from tb1;
                              QUERY PLAN
-----------------------------------------------------------------------------
 Custom Scan (Uxmpp Real-Time)  (cost=0.00..0.00 rows=0 width=0)
   Task Count: 8
   Tasks Shown: One of 8
   ->  Task
         Node: host=wk1 port=5432 dbname=uxdb
         ->  Seq Scan on tb1_102092 tb1  (cost=0.00..32.60 rows=2260 width=8)
(6 rows)
```

为了说明方便，后面把这种带了元数据的worker称之为“扩展worker”。uxmpp会限制某些SQL在扩展worker上执行，比如DDL。

8.2.3. 注意事项

扩展worker其实把worker和CN两个角色混在一个节点里，维护上不是很方便。有下面几个表现：

1. 扩展worker上的负载可能不均衡
2. 如果出现性能问题增加了故障排查的难度
3. CN和worker将不能独立扩容
4. 扩大了插件兼容问题的影响

在扩展worker上尽量避免会产生分布式事务或死锁的操作。建议只执行这几类SQL，并且不使用事务。

- SELECT
- INSERT
- 条件中带分片字段的SQL

8.3. 列存

uxmpp为数据分析和数据仓库工作负载引入了仅追加列式表存储。当列连续存储在磁盘上时，可以提高数据压缩性，并且查询可以更快地请求列的子集。

8.3.1. 用法

要使用列式存储，请在创建表时指定：USING columnar。

```
CREATE TABLE contestant (
   handle TEXT,
   birthdate DATE,
   rating INT,
   percentile FLOAT,
   country CHAR(3),
   achievements TEXT[]
) USING columnar;
```

可以通过函数alter_table_set_access_method在行式存储（堆）和列式存储之间进行转换。

```
//转换为基于行（堆）的存储
SELECT alter_table_set_access_method('contestant', 'heap');
//转换为列式存储（将删除索引）
SELECT alter_table_set_access_method('contestant', 'columnar');
```

uxmpp在插入期间将行转换为列存条带中的列存储。每个列存条带包含一个事务的数据，或 150000 行数据，以较少者为准。（可以使用alter_columnar_table_set函数更改列式表的列存条带大小和其他参数。）

例如，以下语句将所有五行放入同一个列存条带中，因为所有值都插入到单个事务中，如下所示。

```
//将这些值插入同一个列存条带
INSERT INTO contestant VALUES
  ('a','1990-01-10',2090,97.1,'XA','{a}'),
  ('b','1990-11-01',2203,98.1,'XA','{a,b}'),
  ('c','1988-11-01',2907,99.4,'XB','{w,y}'),
  ('d','1985-05-05',2314,98.3,'XB','{}'),
  ('e','1995-05-05',2236,98.2,'XC','{a}');
```

最好尽可能选用大型列存条带，因为 uxmpp会按列存条带单独压缩列式数据。通过使用 VACUUM VERBOSE，我们可以看到有关列式表的实际属性，例如压缩率、列存条带数目和每个列存条带的平均行数，如下所示。

```
VACUUM VERBOSE contestant;
INFO:  statistics for "contestant":
storage id: 10000000000
total file size: 24576, total data size: 248
compression rate: 1.31x
total row count: 5, stripe count: 1, average rows per stripe: 5
chunk count: 6, containing data for dropped columns: 0, zstd compressed: 6
```

输出显示uxmpp使用zstd压缩算法获得1.31x的数据压缩率。压缩率是由将 a 插入的数据在内存中暂存时的大小，与 b 在其最终列存条带中压缩的数据的大小进行比较而得来。

由于它的测量方式，压缩率不一定和表的行存储与列式存储之间的大小差异匹配。只有构造包含相同数据的行表和列表，并进行比较，才能计算出确切的差异。

8.3.2. 测量压缩

创建一个包含多列的样表示例，以对压缩率进行基准测试。

```
//创建默认的行存储表
CREATE TABLE perf_row(
  c00 int8, c01 int8, c02 int8, c03 int8, c04 int8, c05 int8, c06 int8, c07 int8, c08 int8, c09 int8,
  c10 int8, c11 int8, c12 int8, c13 int8, c14 int8, c15 int8, c16 int8, c17 int8, c18 int8, c19 int8,
  c20 int8, c21 int8, c22 int8, c23 int8, c24 int8, c25 int8, c26 int8, c27 int8, c28 int8, c29 int8,
  c30 int8, c31 int8, c32 int8, c33 int8, c34 int8, c35 int8, c36 int8, c37 int8, c38 int8, c39 int8,
  c40 int8, c41 int8, c42 int8, c43 int8, c44 int8, c45 int8, c46 int8, c47 int8, c48 int8, c49 int8,
  c50 int8, c51 int8, c52 int8, c53 int8, c54 int8, c55 int8, c56 int8, c57 int8, c58 int8, c59 int8,
  c60 int8, c61 int8, c62 int8, c63 int8, c64 int8, c65 int8, c66 int8, c67 int8, c68 int8, c69 int8,
  c70 int8, c71 int8, c72 int8, c73 int8, c74 int8, c75 int8, c76 int8, c77 int8, c78 int8, c79 int8,
  c80 int8, c81 int8, c82 int8, c83 int8, c84 int8, c85 int8, c86 int8, c87 int8, c88 int8, c89 int8,
  c90 int8, c91 int8, c92 int8, c93 int8, c94 int8, c95 int8, c96 int8, c97 int8, c98 int8, c99 int8
);
//使用列存储创建具有相同列的表
CREATE TABLE perf_columnar(LIKE perf_row) USING COLUMNAR;
```

使用等量数据集填充两个表，如下所示。

```
INSERT INTO perf_row
 SELECT
  g % 00500, g % 01000, g % 01500, g % 02000, g % 02500, g % 03000, g % 03500, g % 04000, g
% 04500, g % 05000,
  g % 05500, g % 06000, g % 06500, g % 07000, g % 07500, g % 08000, g % 08500, g % 09000, g
% 09500, g % 10000,
  g % 10500, g % 11000, g % 11500, g % 12000, g % 12500, g % 13000, g % 13500, g % 14000, g
% 14500, g % 15000,
  g % 15500, g % 16000, g % 16500, g % 17000, g % 17500, g % 18000, g % 18500, g % 19000, g
% 19500, g % 20000,
  g % 20500, g % 21000, g % 21500, g % 22000, g % 22500, g % 23000, g % 23500, g % 24000, g
% 24500, g % 25000,
  g % 25500, g % 26000, g % 26500, g % 27000, g % 27500, g % 28000, g % 28500, g % 29000, g
% 29500, g % 30000,
  g % 30500, g % 31000, g % 31500, g % 32000, g % 32500, g % 33000, g % 33500, g % 34000, g
% 34500, g % 35000,
  g % 35500, g % 36000, g % 36500, g % 37000, g % 37500, g % 38000, g % 38500, g % 39000, g
% 39500, g % 40000,
  g % 40500, g % 41000, g % 41500, g % 42000, g % 42500, g % 43000, g % 43500, g % 44000, g
% 44500, g % 45000,
  g % 45500, g % 46000, g % 46500, g % 47000, g % 47500, g % 48000, g % 48500, g % 49000, g
% 49500, g % 50000
 FROM generate_series(1,50000000) g;
INSERT INTO perf_columnar
 SELECT
  g % 00500, g % 01000, g % 01500, g % 02000, g % 02500, g % 03000, g % 03500, g % 04000, g
% 04500, g % 05000,
  g % 05500, g % 06000, g % 06500, g % 07000, g % 07500, g % 08000, g % 08500, g % 09000, g
% 09500, g % 10000,
```

```
  g % 10500, g % 11000, g % 11500, g % 12000, g % 12500, g % 13000, g % 13500, g % 14000, g
 % 14500, g % 15000,
  g % 15500, g % 16000, g % 16500, g % 17000, g % 17500, g % 18000, g % 18500, g % 19000, g
 % 19500, g % 20000,
  g % 20500, g % 21000, g % 21500, g % 22000, g % 22500, g % 23000, g % 23500, g % 24000, g
 % 24500, g % 25000,
  g % 25500, g % 26000, g % 26500, g % 27000, g % 27500, g % 28000, g % 28500, g % 29000, g
 % 29500, g % 30000,
  g % 30500, g % 31000, g % 31500, g % 32000, g % 32500, g % 33000, g % 33500, g % 34000, g
 % 34500, g % 35000,
  g % 35500, g % 36000, g % 36500, g % 37000, g % 37500, g % 38000, g % 38500, g % 39000, g
 % 39500, g % 40000,
  g % 40500, g % 41000, g % 41500, g % 42000, g % 42500, g % 43000, g % 43500, g % 44000, g
 % 44500, g % 45000,
  g % 45500, g % 46000, g % 46500, g % 47000, g % 47500, g % 48000, g % 48500, g % 49000, g
 % 49500, g % 50000
  FROM generate_series(1,50000000) g;
VACUUM (FREEZE, ANALYZE) perf_row;
VACUUM (FREEZE, ANALYZE) perf_columnar;
```

对比数据，可以看到列式表的压缩比优于传统堆表约 8 倍。

```
SELECT ux_total_relation_size('perf_row')::numeric/ux_total_relation_size('perf_columnar') AS
 compression_ratio;
 compression_ratio
--------------------
 8.0196135873627944
(1 row)
```

8.3.3. 使用列式存储进行存档

某些应用程序的数据在逻辑上只有一小部分的会更新，较大的一大部分不会更新。在这种情况下，我们可以将分区与列式表存储相结合，以压缩磁盘上不更新的部分数据。uxmpp列式表目前为“仅追加”，这意味着列式表不支持更新或删除，但我们可以将它们用于不可变的历史分区。

分区表可以由行分区和列分区的任意组合组成。在时间戳键上使用范围分区时，我们可以将最新的分区设置为行表，并定期将最新的分区滚动到另一个历史列式分区中。

创建一个新的本地表github_columnar_events，并获取示例数据：数据路径：svn://192.30.1.2/svn/uxdb/33.研发专题/01.高可用专题/UXMPP专题/UXMPP升级/列存分区表示例数据。

```
gzip -c -d github_events-2015-01-01-*.gz >> github_events.csv

//创建新表
CREATE TABLE github_columnar_events ( LIKE github_events )
PARTITION BY RANGE (created_at);
//创建分区，每个分区保存两个小时的数据
SELECT create_time_partitions(
  table_name         := 'github_columnar_events',
  partition_interval := '2 hours',
  start_from         := '2015-01-01 00:00:00',
  end_at             := '2015-01-01 08:00:00'
);
```

```
//填充示例数据
//请注意，此数据要求数据库具有UTF8编码
\COPY github_columnar_events FROM 'github_events.csv' WITH (format CSV)
//列出分区，并确认它们是
//使用基于行的存储（堆访问方法）
SELECT partition, access_method
  FROM time_partitions
 WHERE parent_table = 'github_columnar_events'::regclass;
```

partition	access_method
github_columnar_events_p2015_01_01_0000	heap
github_columnar_events_p2015_01_01_0200	heap
github_columnar_events_p2015_01_01_0400	heap
github_columnar_events_p2015_01_01_0600	heap

```
//将旧分区转换为使用列存储
CALL alter_old_partitions_set_access_method(
  'github_columnar_events',
  '2015-01-01 06:00:00' /* older_than */,
  'columnar'
);
//旧分区现在是列式的
//而最新的分区使用行存储（可以更新）
SELECT partition, access_method
  FROM time_partitions
 WHERE parent_table = 'github_columnar_events'::regclass;
```

partition	access_method
github_columnar_events_p2015_01_01_0000	columnar
github_columnar_events_p2015_01_01_0200	columnar
github_columnar_events_p2015_01_01_0400	columnar
github_columnar_events_p2015_01_01_0600	heap

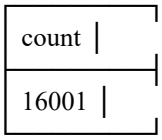
可以使用VACUUM VERBOSE查看列式表的压缩比，如下所示。

```
VACUUM VERBOSE github_columnar_events;
INFO:  statistics for "github_columnar_events_p2015_01_01_0000":
storage id: 10000000003
total file size: 4481024, total data size: 4444425
compression rate: 8.31x
total row count: 15129, stripe count: 1, average rows per stripe: 15129
chunk count: 18, containing data for dropped columns: 0, zstd compressed: 18

INFO:  statistics for "github_columnar_events_p2015_01_01_0200":
storage id: 10000000004
total file size: 3579904, total data size: 3548221
compression rate: 8.26x
total row count: 12714, stripe count: 1, average rows per stripe: 12714
chunk count: 18, containing data for dropped columns: 0, zstd compressed: 18

INFO:  statistics for "github_columnar_events_p2015_01_01_0400":
```

storage id: 10000000005
total file size: 2949120, total data size: 2917407
compression rate: 8.51x
total row count: 11756, stripe count: 1, average rows per stripe: 11756
chunk count: 18, containing data for dropped columns: 0, zstd compressed: 18

分区表github_columnar_events可以像普通表一样完整地查询，如下所示。

SELECT COUNT(DISTINCT repo_id) FROM github_columnar_events;



可以更新或删除条目，只要分区键上有一个 WHERE 子句，该子句完全筛选到行表分区中即可。

将行分区存档为列式存储，如下所示。

当行分区已填满其范围时，可以将其存档到压缩的列式存储中。我们可以借助ux_cron自动执行此操作，如下所示。

//按月度执行
SELECT cron.schedule('compress-partitions', '0 0 1 * *', $$
CALL alter_old_partitions_set_access_method(
'github_columnar_events', now() - interval '6 months' /* older_than */, 'columnar'
);
$$);

8.3.4. 隐式错误

列式存储按列存条带压缩。列存条带是按事务创建的，因此每个事务插入一行会将单个行放入它们自己的列存条带中。单行列存条带的压缩和性能将比行表差。始终批量插入到列式表中。

可以使用VACUUM命令来重新组织列存条带。

BEGIN;
CREATE TABLE foo_compacted (LIKE foo) USING columnar;
INSERT INTO foo_compacted SELECT * FROM foo;
DROP TABLE foo;
ALTER TABLE foo_compacted RENAME TO foo;
COMMIT;

列式存储相较于行式存储，在选择特定列时，加载到内存中的数据将更少。

在混合了行分区和列分区的分区表上，必须仔细定位或筛选更新以仅命中行分区。

如果操作针对特定的行分区（例如 UPDATE p2 SET i = i + 1），它将成功;如果以指定的列式分区为目标（例如 UPDATE p1 SET i = i + 1），它将失败。

如果操作以分区表为目标，并且具有排除所有列式分区的 WHERE 子句（例如，UPDATE 父级 SET i = i + 1 WHERE TIMESTAMP = '2020-03-15'），则操作将成功。

如果操作针对分区表，但不排除所有列式分区，则操作将失败;即使要更新的实际数据仅影响行表（例如，更新父 SET i = i + 1，其中 n = 300）。

8.3.5. 局限

- 仅追加（不支持更新/删除）
- 无空间回收（例如，回滚的事务可能仍会占用磁盘空间）
- 仅支持hash和 btree 索引
- 无索引扫描或位图索引扫描
- 无tid扫描
- 无样品扫描
- 不支持 TOAST
- 不支持 ON CONFLICT 语句（DO NOTHING操作除外）
- 不支持元组锁（SELECT … FOR SHARE, SELECT … FOR UPDATE）
- 不支持可序列化的隔离级别
- 不支持外键、唯一约束或排除约束
- 不支持逻辑解码
- 不支持节点内并行扫描
- 不支持AFTER … FOR EACH ROW触发器
- 不支持无记录的表
- 不支持临时表

第 9 章 常见问题

1. 可以在分布式表上创建主键吗？

   目前，uxmpp只在分布列是主键的一部分时才强制执行主键约束。以确保惟一性。

2. 如何将节点添加到现有uxmpp集群？

   可以通过使用新节点的主机名（或IP地址）和端口号调用第 4.5.3.7 节“uxmpp_add_node”函数来手动添加节点。在将节点添加到现有集群后，它将不包含任何数据（分片）。

3. uxmpp如何处理工作节点的故障？

   uxmpp支持两种复制模式，允许它容忍工作节点故障。在第一个模式中，使用uxdb的流复制来按原样复制整个工作节点。在第二个模式中，uxmpp在不同的工作节点之间复制分片。它们具有不同的优势，具体取决于工作负载和用例，如下所述。

   - uxdb流复制。此选项最适合繁重的OLTP工作负载。它通过不断地将WAL记录流式传输到备用数据库来复制整个工作节点。

   - uxmpp碎片复制。此选项最适合append分布式表。uxmpp通过自动复制DML语句和管理一致性来复制不同节点上的分片。如果节点发生故障，协调节点通过路由到副本来继续提供查询。要使用此种模式，只需启用分片复制：SET uxmpp.shard_replication_factor = 2; 。

4. uxmpp如何处理协调器节点的故障转移？

   由于uxmpp协调器节点类似于标准uxdb服务器，因此可以使用常规uxdb同步复制和故障转移来提供协调器节点的更高可用性。许多用户以这种方式使用同步复制来增加协调器节点故障的弹性。

5. 如何将查询结果摄取到分布式表中？

   当表位于同一位置时（协同表），uxmpp支持INSERT / SELECT语法，用于将分布式表上的查询结果复制到分布式表中。

   如果表不在同一位置，或者正在使用append分发，则可以使用COPY将数据复制出来然后再复制到目标表。

6. 可以在同一查询中将分布式和非分布式表连接在一起吗？

   如果要在小维度表（常规uxdb表）和大型表（分布式）之间进行连接，请将本地表包装在子查询中。uxmpp的子查询执行逻辑将允许联接工作。

7. 是否有uxmpp不支持的uxdb功能？

   由于uxmpp通过UXDB提供分布式功能，因此，可以使用UXDB自身提供的工具和特性用于分布式表。

   uxmpp支持分布式表上的大部分SQL查询，以下除外。

   - 相关子查询

   - WITH RECURSIVE(递归)

- TABLESAMPLE
- SELECT ... FOR UPDATE/SHARE
- 分组集合（GROUPING SETS、CUBE、ROLLUP）

uxmpp对访问数据库集群中单个节点的查询提供100％的SQL支持。

8. 在对数据进行hash分区时，如何选择分片数？

   首次分发表时可以选择其分片数。可以为每个表设置不同的分片数，最佳值取决于用例。在创建群集后也可以更改分片数，但是比较麻烦，因此建议首次分发的时候确定好分片数。

   在多租户数据库用例中，建议在32个~128个分片之间进行选择。对于较小的工作负载（例如<100GB），可以从32个分片开始，对于较大的工作负载，可以选择64个或128个。

   在实时分析用例中，分片数应与worker的CPU核数相关。要确保最大并行度，应在每个节点上创建足够的分片，以使每个CPU核至少有一个分片。建议创建大量初始分片，例如当前CPU核数的2倍或4倍。

   为要分发的表选择分片数，可以修改uxmpp.shard_count的值。这会影响对第 4.5.1.6 节“create_distributed_table”的后续调用。命令如下所示。

   SET uxmpp.shard_count = 64;

   此时分发的任何表都有64个分片。

9. 在哪些情况下，分布式表支持唯一性约束？

   只有当受约束的列包含分发列时，uxmpp才能强制执行主键或唯一性约束。特别地，这意味着如果单个列构成主键，那么它也必须是分发列。

   此限制允许uxmpp将唯一性检查本地化为单个分片，并让工作节点上的uxdb有效地进行检查。

10. 如何在uxmpp群集中创建数据库角色，功能，扩展等？

    某些命令在协调器节点上运行时，不会传播给worker，如下所示。

    CREATE ROLE/USER
    CREATE FUNCTION
    CREATE TYPE
    CREATE EXTENSION
    CREATE DATABASE
    ALTER … SET SCHEMA
    ALTER TABLE ALL IN TABLESPACE

    但是可以通过在所有节点上显式运行它们来使用这些命令。uxmpp提供了跨所有工作程序执行查询的功能，操作方法如下。

    ```
    SELECT  run_command_on_workers ($cmd$
     /*运行命令*/
     CREATE FUNCTION ...
    $cmd$);
    ```

11. 如果工作节点的地址发生变化怎么办？

    如果worker的主机名或IP地址发生更改，则可以通过第 4.5.3.1 节“master_update_node”函数来进行变更，操作方法如下。

    //更新协调器上的工作节点元数据（记得将’old-address’和’new-address' 替换为实际使用值）
    select master_update_node(nodeid, 'new-address', nodeport)
    from ux_dist_node
    where nodename = 'old-address';

    在执行此更新之前，协调器将无法与该工作程序进行通信。

12. 哪个分片包含特定租户的数据？

    uxmpp提供元数据表，以确定分发列值到特定分片的映射，以及工作节点上的分片放置。请参见第 4.6 节 “元数据表”

13. 为什么第 4.5.4.5 节 “ux_relation_size报告分布式表的零字节？

    分布式表中的数据存在于工作节点（在分片中），而不在协调器上。分布式表大小的真实度量是作为分片大小的总和获得的。uxmpp提供帮助函数来查询此信息。

14. 为什么看到有关max_intermediate_result_size的错误？

    uxmpp必须使用多个步骤来运行具有子查询或CTE的查询。它将子查询结果推送到所有工作节点以供主查询使用。如果这些结果太大，则可能会导致不可接受的网络开销，甚至协调器节点上的存储空间不足。

    uxmpp具有可配置的设置，uxmpp.max_intermediate_result_size用于指定取消查询的子查询结果大小阈值。如果遇到如下错误，可设置此参数。

    ERROR: the intermediate result size exceeds citus.max_intermediate_result_size (currently 1 GB)
    DETAIL: Citus restricts the size of intermediate results of complex subqueries and CTEs to
    avoid accidentally pulling large result sets into once place.HINT: To run the current query, set
    citus.max_intermediate_result_size to a higher value or -1 to disable.

15. cstore_fdw如何与uxmpp合作？

    uxmpp像普通的uxdb表一样处理cstore_fdw表。当cstore_fdw与uxmpp一起使用时，每个逻辑分片都被创建为外部cstore_fdw表而不是常规uxdb表。如果cstore_fdw用例适用于uxmpp的分布式特性（例如大型数据集归档和报告），则可以使用这两者来提供一个功能强大的工具，它结合了uxmpp的查询并行化，无缝分片和HA优势以及出色的压缩和I/O利用cstore_fdw。

16. 无法连接到新增worker节点？

    a. 查看添加节点信息是否正确。

    b. 查看master和worker的网络是否相通。

    c. 查看worker节点集群是否正常启动。

    d. 是否加载了uxmpp插件。

    e. 是否加载了license，可在启动集群的时候查看启动日志，license是否加载成功。

f. 重新设置分片规则，uxmpp使用hash分布时，默认遵循前一个分布式表的分布规则。可以设置uxmpp.shard_count来重新生成分布规则。

第 10 章 术语&缩略语

10.1. 术语

master（Coordinator）

协调节点，存储元数据和给worker节点下发任务并回收结果回显。

worker

工作节点，存储数据，并执行协调器下发的任务，用以实现并行化。

Shard

该分布式表在worker节点上的某个小表（分布式表行的子集）。

10.2. 缩略语

表 10.1. 缩略语详解

缩略语	英文全称	中文全称
UXDB	Uxsino Database	优炫数据库
uxmpp	UX Massive Parallel Process	优炫大规模并行处理