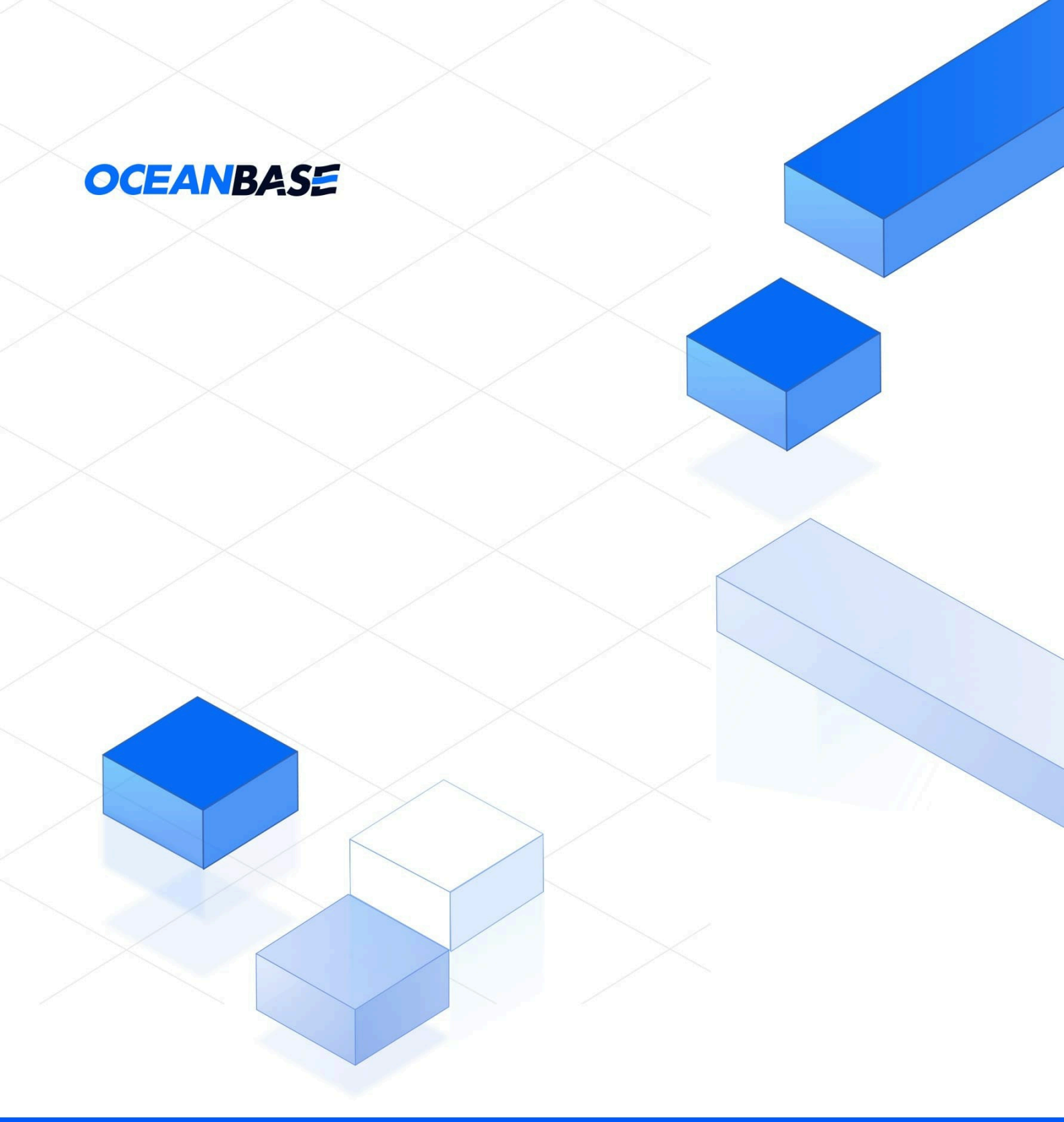

OceanBase 数据库

快速上手

| 产品版本：V4.5.0 | 文档版本：20251219

声明

北京奥星贝斯科技有限公司版权所有©2024，并保留一切权利。

商标声明

免责声明

通用约定

格式	说明	样例
危险	该类警示信息将导致系统重大变更甚至故障，或者导致人身伤害等结果。	**危险** 重置操作将丢失用户配置数据。
警告	该类警示信息可能会导致系统重大变更甚至故障，或者导致人身伤害等结果。	**警告** 重启操作将导致业务中断，恢复业务时间约十分钟。
注意	用于警示信息、补充说明等，是用户必须了解的内容。	**注意** 权重设置为0，该服务器不会再接受新请求。
说明	用于补充说明、最佳实践、窍门等，不是用户必须了解的内容。	**说明** 您也可以通过按Ctrl+A选中全部文件。
>	多级菜单递进。	单击**设置**> **网络**> **设置网络类型**。
粗体	表示按键、菜单、页面名称等UI元素。	在**结果确认**页面，单击**确定**。
`Courier字体`	命令或代码。	执行 `cd /d C:/window` 命令，进入Windows系统文件夹。
斜体	表示参数、变量。	`bae log list --instanceid` *Instance_ID*
[] 或者 [a|b]	表示可选项，至多选择一个。	`ipconfig [-all|-t]`
{} 或者 {a|b}	表示必选项，至多选择一个。	`switch {active|stand}`

目录

1 快速体验 OceanBase 社区版

本文根据使用场景详细介绍如何快速部署 OceanBase 数据库，旨在帮助您快速掌握并成功使用 OceanBase 数据库。

1.0.0.1 说明

本文档**不适用于生产环境**，如需在生产环境中部署 OceanBase 数据库，请参见 OceanBase 数据库社区版部署概述 选择合适的部署方法。

1.1 组件介绍

- obd

  OceanBase Deployer，OceanBase 安装部署工具，简称为 obd。详细信息请参考官网文档 OceanBase 安装部署工具。

- ODP

  OceanBase Database Proxy，OceanBase 数据库代理，是 OceanBase 数据库专用的代理服务器，简称为 ODP（又称为 OBProxy）。详细信息请参考官网文档 OceanBase 数据库代理。

- OBAgent

  OBAgent 是 OceanBase 数据库监控采集框架，支持推、拉两种数据采集模式，可以满足不同的应用场景。

- Grafana

  Grafana 是一款开源的数据可视化工具，它可以将数据源中的各种指标数据进行可视化展示，以便更直观地了解系统运行状态和性能指标。详细信息可参见 Grafana 官网。

- Prometheus

  Prometheus 是一个开源的服务监控系统和时序数据库，其提供了通用的数据模型以及快捷数据采集、存储和查询接口。详细信息可参见 Prometheus 官网。

1.2 前提条件

在参考本文安装 OceanBase 数据库之前，确保您的软硬件环境满足以下要求：

项目	描述

系统	• Anolis OS 8.X 版本（内核 Linux 4.19 版本及以上） • Alibaba Cloud Linux 2/3 版本（内核 Linux 4.19 版本及以上） • Red Hat Enterprise Linux Server 7.X 版本、8.X 版本（内核 Linux 4.19 版本及以上） • CentOS Linux 7.X 版本、8.X 版本（内核 Linux 4.19 版本及以上） • Rocky Linux 9（内核 Linux 5.14） • Debian 9.X 版本及以上版本（内核 Linux 4.19 版本及以上） • Ubuntu 16.X 版本及以上版本（内核 Linux 4.19 版本及以上） • SUSE / openSUSE 15.X 版本及以上版本（内核 Linux 4.19 版本及以上） • openEuler 22.03 和 24.03 版本（内核 Linux 5.10.0 版本及以上） • KylinOS V10 版本 • 统信 UOS V20 版本 • 中科方德 NFSChina 4.0 版本及以上 • 浪潮 Inspur kos 5.8 版本
CPU	最低要求 2 核，推荐 4 核及以上。
内存	最低要求 6 GB，推荐设置在 16 GB 至 1024 GB 范围内。
磁盘类型	使用 SSD 存储。
磁盘存储空间	最低要求 20 GB。
文件系统	EXT4 或 XFS，当数据超过 16 TB 时，使用 XFS。
端口	需保证组件的默认端口未被占用： • OceanBase 数据库默认使用 2881、2882、2886 端口。 • ODP 默认使用 2883、2884、2885 端口。 • OBAgent 默认使用 8088、8089 端口。 • Prometheus 默认使用 9090 端口。 • Grafana 默认使用 3000 端口。
all-in-one 安装包	all-in-one 安装包需选择 V4.1.0 及以上版本。

Docker	使用 Docker 部署 OceanBase 数据库时需提前安装 Docker 并启动 Docker 服务，详细操作请参考 Docker 文档。 #### 1.2.0.1 说明 在使用 x86 架构的 MAC 机器时，仅支持使用 V4.9.0 及以下版本的 Docker 部署 OceanBase 数据库，可点击 链接 下载 Docker。

1.3 快速体验 OceanBase 数据库

此方案适用于仅有一台机器时，快速搭建一个可用的 OceanBase 数据库环境。部署的 OceanBase 数据库环境具备数据库的基本功能，可以有效地帮助您了解 OceanBase 数据库；但是该环境不具备任何分布式能力及高可用特性，不建议长期使用。

您可以使用以下四种方法快速体验 OceanBase 数据库。

1.3.1 方法一：使用 All in One 安装包

执行以下命令下载并安装 OceanBase All in One，该命令将**在线**下载并安装最新版本的 OceanBase All in One 包。

```
bash -c "$(curl -s https://obbusiness-private.oss-cn-shanghai.aliyuncs.com
/download-center/opensource/oceanbase-all-in-one/installer.sh)"
source ~/.oceanbase-all-in-one/bin/env.sh
```

obd 提供以下两种快速部署命令：

- 最小规格部署

  将以 OceanBase 数据库最小规格部署并启动单节点的 OceanBase 数据库及相关组件。

  ```
  obd demo
  ```

- 最大规格部署

  obd 自 V3.4.0 起提供最大规格部署命令，执行后将以 OceanBase 数据库最大规格部署并启动单节点的 OceanBase 数据库及相关组件。

  ```
  obd pref
  ```

以上为快速体验命令，将直接使用 **当前账号** 部署 OceanBase 数据库、ODP 和监控组件（OBAgent、Grafana、Prometheus），后续可以通过 obd 命令管理 OceanBase 数据库及组件。离线安装 OceanBase All in One 的步骤和 `obd demo` / `obd pref` 命令的更多介

绍可分别参见官网《OceanBase 安装部署工具》文档 快速上手/安装 obd 和 obd 命令/快速部署命令。

`obd demo` / `obd pref` 命令执行成功后会输出部署组件的连接方式，您可复制并执行 `oceanbase-ce` 组件下的连接串，使用 root 用户登录集群的 sys 租户。登录集群后可执行 SQL 语句进行简单的功能体验，具体可参见 SQL 基础操作（MySQL 模式）。

1.3.2 方法二：直接使用 RPM 包

1.3.2.1 说明

- 本方法依赖于 systemctl 命令，请在非容器环境下进行操作。
- 通过本方法安装 OceanBase 数据库时仅支持以下系统：
  - Anolis OS 8.X 版本（内核 Linux 4.19 版本及以上）
  - CentOS Linux 8.X 版本（内核 Linux 4.19 版本及以上）
  - Debian 10、11 和 12 版本（内核 Linux 4.19 版本及以上）
  - openEuler 22.03 和 24.03 版本（内核 Linux 5.10.0 版本及以上）
  - Ubuntu 18.04、20.04 和 22.04 版本（内核 Linux 4.19 版本及以上）

```
sudo bash -c "$(curl -s https://obbusiness-private.oss-cn-shanghai.aliyuncs.com/download-center/opensource/service/installer.sh)"
```

以上为快速体验命令，将 **在线下载** 最新版本的 RPM 包并直接使用 `root` 账号进行安装，后续可以通过 systemctl 命令管理 OceanBase 数据库。详细配置及离线安装步骤请参考官网《OceanBase 数据库》文档 使用 systemd 部署 OceanBase 数据库。

命令执行成功后会输出 OceanBase 数据库的连接方式，您可复制并执行输出的连接串，使用 root 用户登录集群的 sys 租户。登录集群后可执行 SQL 语句进行简单的功能体验，具体可参见 SQL 基础操作（MySQL 模式）。

1.3.3 方法三：使用 Docker 快速体验

此方案适用于非 Linux 操作系统的用户（例如 Windows、macOS），希望通过容器实现部署、管理 OceanBase 数据库的用户。该方案未经过规模化的验证，建议谨慎使用。

1.3.3.2 说明

在使用 x86 架构的 MAC 机器时，仅支持使用 V4.9.0 及以下版本的 Docker 部署 OceanBase 数据库，可点击 链接 下载 Docker。

```
sudo docker run -p 2881:2881 --name obstandalone -e MODE=MINI -d quay.io
/oceanbase/oceanbase-ce
```

以上为快速体验命令，将 **在线下载** 最新的镜像并启动最小规格的 OceanBase 数据库，后续可通过 docker 命令管理 OceanBase 数据库。详细配置请参考官网《OceanBase 数据库》文档 部署 OceanBase 数据库容器环境。

部署成功后可执行下述命令，使用 root 用户登录集群的 sys 租户。登录集群后可执行 SQL 语句进行简单的功能体验，具体可参见 SQL 基础操作（MySQL 模式）。

```
#进入 Docker 容器
sudo docker exec -it obstandalone bash
#连接集群，密码默认为空
obclient -uroot@sys -h127.0.0.1 -Doceanbase -P2881 -p
```

1.3.4 方法四：安装桌面版 OceanBase 数据库

OceanBase 桌面版是一个用于管理和操作 OceanBase 数据库的桌面应用程序。它提供了图形化界面，使用户能够方便地进行数据库管理、查询执行、数据导入导出等操作。

1.3.4.3 说明

- 仅支持在 Windows（x86-64 芯片）和 macOS 操作系统下安装 OceanBase 桌面版。
- 安装 OceanBase 桌面版前需安装所需依赖：wsl（Windows）或 OrbStack（macOS）。您可自行搜索安装方法进行安装，也可参考我们提供的文档 安装 OceanBase 桌面版 进行安装。

1. 下载安装包

   访问 OceanBase 软件下载中心，搜索 **OceanBase桌面版**，根据所用机器的操作系统下载所需安装包。

2. 安装应用并启动 OceanBase 桌面版

   安装后会自动部署一个处于 `已停止` 状态的 OceanBase 数据库，单击图形化页面中的 `启动` 即可启动 OceanBase 数据库。

3. 连接 OceanBase 桌面版

   单击 OceanBase 桌面版中的 **连接** 按钮，即可自动连接 OceanBase 数据库。登录集群后可执行 SQL 语句进行简单的功能体验，具体可参见 SQL 基础操作（MySQL 模式）。

1.4 相关文档

- OceanBase 数据库社区版部署方式及适用场景的介绍可参见 OceanBase 数据库社区版部署概述。
- 清理集群的具体操作可参见 清理旧集群。

2 在您开始前

为了能更好的体验和上手 OceanBase 分布式数据库，请您在开始尝试使用前，先了解 **内存** 和 **超时时间** 这两个最常见的特性差异。

2.1 关于内存

OceanBase 数据库是基于 LSM-Tree 的存储引擎，不同于传统数据库实时刷脏页的机制，OceanBase 数据库将数据分为内存中的 MemTable 和磁盘中的 SSTable，其中所有的数据更新写入操作都在内存的 MemTable 中完成，并且在内存使用量达到一定阈值后触发 Compaction，转储至 SSTable，并释放活跃的内存。这种架构的优势是可以将随机 I/O 转化为顺序 I/O，提供更大的写入吞吐能力。详细介绍请参考 存储架构概述。

由于 LSM-Tree 将增量数据都存放在内存中，达到一定阈值后才触发转储，这会导致小规格的租户实例运行在超过其可承载能力的密集写入场景时（例如数据导入或者运行大量数据批处理场景），会因为 MemTable 达到上限而无法接受新的请求。OceanBase 数据库有如下几种处理方式：

- 开启写入限速：设置内存写入达到一定阈值后，OceanBase 数据库主动限制客户端导入速度。
- 租户内存扩容：环境中节点总内存资源相对充足，可扩大租户内存。
- 调整租户内存中 MemTable 的比例：当节点总内存有限，无法扩容时，还可调整租户内存中 MemTable 的比例，扩大可写入内存，并且调低转储阈值，让转储更快发生。

2.1.1 开启写入限速

OceanBase 数据库具备写入过载保护功能，当资源有限，无法扩展内存时，可以通过服务端写入限速来保护内存，避免写入超限。可通过设置如下两个配置项来开启服务端的写入限速功能：

- writing_throttling_trigger_percentage：用于设置写入速度的阈值，即当 MemStore 已使用的内存达到该阈值（百分比）时，触发写入限速。该配置项的取值范围为 [1，100]，默认值为 60，取值为 100 表示关闭写入限速机制。
- writing_throttling_maximum_duration：指定触发写入限速后，剩余 MemStore 内存分配完所需的时间。默认值为 2h，该配置项一般不做修改。

在租户的管理员账号中，设置内存写入达到 80% 开始限速，并保证剩余内存足够提供 2h 的写入限速，示例如下：

```
obclient> ALTER SYSTEM SET writing_throttling_trigger_percentage = 80;
Query OK, 0 rows affected
```

```
obclient> ALTER SYSTEM SET writing_throttling_maximum_duration = '2h';
Query OK, 0 rows affected
```

2.1.2 租户内存扩容

当环境中的内存资源相对充足时，最佳处理方案是增大租户内存。

内存配置步骤如下：

1. 使用 root 用户登录 OceanBase 集群的 sys 租户，执行以下 SQL 语句，确认当前租户使用的 UNIT_CONFIG NAME 。

   ```
   obclient> SELECT NAME FROM DBA_OB_UNIT_CONFIGS;
   +-----------------+
   | NAME |
   +-----------------+
   | sys_unit_config |
   | test_unit |
   +-----------------+
   2 rows in set
   ```

2.1.2.1 说明

- sys_unit_config 是管控租户的参数，一般不做修改。
- 本示例中租户 test 的 unit_config name 为 test_unit。

2. 复制租户的 unit_config name ，使用如下命令，完成内存扩容。

   ```
   obclient> ALTER RESOURCE UNIT test_unit MIN_CPU = 2, MAX_CPU = 2,
   MEMORY_SIZE = '10G', MAX_IOPS = 10000, MIN_IOPS = 10000;
   ```

2.1.2.2 注意

当前版本中，仅 CPU、Memory 配置生效，其他 I/O 参数（例如 IOPS）暂不生效。

2.1.3 调整租户内存中 MemStore 的比例

通过如下配置项来调整租户内存中 MemStore 的比例：

- freeze_trigger_percentage：当租户的 MemStore 内存的使用量达到此配置项所限制使用的百分比时，就会自动触发转储，转储后会释放占用的内存。该配置项取值范围 [1,99]，默认值为 20，表示当 MemStore 使用率超过 20%，就会触发转储。
- memstore_limit_percentage：该配置项用于控制租户内存中可用于 MemStore 写入的比例，取值范围 [0, 100)，默认值为 0，表示租户使用 MemStore 的内存占其总可用内存的百分比由系统进行自适应调整。

当内存不足时，可以调高 memstore_limit_percentage 的取值，并调低 freeze_trigger_percentage 的取值，从而达到临时扩容和尽快转储释放的效果。

示例如下：

```
obclient> ALTER SYSTEM SET freeze_trigger_percentage = 20;
Query OK, 0 rows affected


obclient> ALTER SYSTEM SET memstore_limit_percentage = 70;
Query OK, 0 rows affected
```

2.2 关于超时时间

在 OceanBase 数据库中，您可能在查询或执行 DML 操作时遇到 timeout 或 Transaction is timeout 的错误，这是因为 OceanBase 数据库对查询和事务超时做了默认配置，方便用户针对不同业务场景进行调整。

OceanBase 数据库提供了以下超时时间相关的变量，可使用 SHOW VARIABLES LIKE '%timeout%'; 命令进行查看。

- ob_query_timeout：查询超时时间，单位 us，默认值为 10000000。
- ob_trx_timeout：事务超时时间，单位 us，默认值 86400000000。
- ob_trx_idle_timeout：事务空闲超时时间，单位 us，默认值 86400000000。

2.2.4 设置超时时间

超时时间的设置方法如下：

- 在会话/全局进行变量设置。示例如下：

```
obclient> SET ob_query_timeout = 10000000;
Query OK, 0 rows affected

obclient> SET GLOBAL ob_query_timeout = 10000000;
Query OK, 0 rows affected
```

- 在 JDBC 连接串中设置。示例如下：

```
jdbc:mysql://10.1.0.0:1001/unittests?
user=**u**@sys&password=******&sessionVariables = ob_query_timeout =
60000000000,ob_trx_timeout = 60000000000&xxxx
```

- 在 SQL 级别添加 Hint 设置。示例如下：

2.2.4.1 说明

此方式只对当前 SQL 语句生效。

```
SELECT /*+query_timeout(100000000) */ c1 FROM t1;
```

更多关于数据库开发、管理方面的详细内容，请参见《应用开发》和《管理数据库》。

3 SQL 基础操作（MySQL 模式）

本节主要介绍 OceanBase 数据库 MySQL 模式下的一些 SQL 基本操作。

3.1 创建数据库

使用 `CREATE DATABASE` 语句创建数据库。

示例：创建数据库 `db1`，指定字符集为 `utf8mb4`，并创建读写属性。

```
obclient(root@mysqltenant)[(none)]> CREATE DATABASE db1 DEFAULT CHARACTER SET utf8mb4 READ WRITE;
```

更多 `CREATE DATABASE` 语句相关的语法说明，参见 CREATE DATABASE。

创建完成后，可以通过 `SHOW DATABASES` 命令查看当前数据库服务器中所有的数据库。

```
obclient(root@mysqltenant)[(none)]> SHOW DATABASES;
```

返回结果如下：

```
+--------------------+
| Database |
+--------------------+
| db1 |
| information_schema |
| mysql |
| oceanbase |
| test |
| test_db |
+--------------------+
6 rows in set
```

3.2 表操作

在 OceanBase 数据库中，表是最基础的数据存储单元，包含了所有用户可以访问的数据，每个表包含多行记录，每个记录由多个列组成。本节主要提供数据库中表的创建、查看、修改和删除的语法和示例。

3.2.1 创建表

使用 `CREATE TABLE` 语句在数据库中创建新表。

示例：在数据库 `db1` 中创建表 `test` 。

1. 切换到数据库 `db1` 中。

   ```
   obclient(root@mysqltenant)[(none)]> USE db1;
   ```

2. 创建表 `test` 。

   ```
   obclient(root@mysqltenant)[db1]> CREATE TABLE test (c1 INT PRIMARY KEY, c2 VARCHAR(3));
   ```

更多 `CREATE TABLE` 语句相关的语法说明，参见 CREATE TABLE。

3.2.2 查看表

使用 `SHOW CREATE TABLE` 语句查看建表语句。

示例：查看表 `test` 的建表语句。

```
obclient(root@mysqltenant)[db1]> SHOW CREATE TABLE test\G
```

返回结果如下：

```
*************************** 1. row ***************************
Table: test
Create Table: CREATE TABLE `test` (
`c1` int(11) NOT NULL,
`c2` varchar(3) DEFAULT NULL,
PRIMARY KEY (`c1`)
) ORGANIZATION INDEX DEFAULT CHARSET = utf8mb4 ROW_FORMAT = DYNAMIC
COMPRESSION = 'zstd_1.3.8' REPLICA_NUM = 1 BLOCK_SIZE = 16384
```

```
USE_BLOOM_FILTER = FALSE ENABLE_MACRO_BLOCK_BLOOM_FILTER = FALSE
TABLET_SIZE = 134217728 PCTFREE = 0
1 row in set
```

还可以使用 `SHOW TABLES` 查看指定数据库下的所有表。

示例：查看 `db1` 数据库下的所有表。

```
obclient(root@mysqltenant)[db1]> SHOW TABLES FROM db1;
```

返回结果如下：

```
+---------------+
| Tables_in_db1 |
+---------------+
| test |
+---------------+
1 row in set
```

更多 `SHOW` 语句相关的语法说明，参见 SHOW。

3.2.3 修改表

使用 `ALTER TABLE` 语句来修改已存在的表的结构，包括修改表及表属性、新增列、修改列及属性、删除列等。

示例：

- 将表 `test` 的字段 `c2` 改名为 `c3`，并同时修改其字段类型。

  1. 查看表 `test` 的列定义。

     ```
     obclient(root@mysqltenant)[db1]> DESCRIBE test;
     ```

     返回结果如下：

     ```
     +-------+------------+------+-----+---------+-------+
     | Field | Type | Null | Key | Default | Extra |
     +-------+------------+------+-----+---------+-------+
     ```

```
| c1 | int(11) | NO | PRI | NULL | |
| c2 | varchar(3) | YES | | NULL | |
+-------+------------+------+-----+---------+-------+
2 rows in set
```

2. 将表 `test` 的字段 `c2` 改名为 `c3`，并同时修改其字段类型为 `char`。

```
obclient(root@mysqltenant)[db1]> ALTER TABLE test CHANGE COLUMN c2 c3
CHAR(10);
```

3. 再次查看表 `test` 的列定义，确认修改成功。

```
obclient(root@mysqltenant)[db1]> DESCRIBE test;
```

返回结果如下：

```
+-------+----------+------+-----+---------+-------+
| Field | Type | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| c1 | int(11) | NO | PRI | NULL | |
| c3 | char(10) | YES | | NULL | |
+-------+----------+------+-----+---------+-------+
2 rows in set
```

- 在表 `test` 中增加、删除列。

1. 查看当前表 `test` 的列定义。

```
obclient(root@mysqltenant)[db1]> DESCRIBE test;
```

返回结果如下：

```
+-------+----------+------+-----+---------+-------+
| Field | Type | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| c1 | int(11) | NO | PRI | NULL | |
```

```
| c3 | char(10) | YES | | NULL | |
+-------+----------+------+-----+---------+-------+
2 rows in set
```

2. 向表 `test` 中添加一列 `c4` 。

```
obclient(root@mysqltenant)[db1]> ALTER TABLE test ADD c4 int;
```

3. 再次查看表 `test` 的列定义，确认 `c4` 列添加成功。

```
obclient(root@mysqltenant)[db1]> DESCRIBE test;
```

返回结果如下：

```
+-------+----------+------+-----+---------+-------+
| Field | Type | Null | Key | Default | Extra |
+-------+----------+------+-----+---------+-------+
| c1 | int(11) | NO | PRI | NULL | |
| c3 | char(10) | YES | | NULL | |
| c4 | int(11) | YES | | NULL | |
+-------+----------+------+-----+---------+-------+
3 rows in set
```

4. 删除表 `test` 中 `c3` 列。

```
obclient(root@mysqltenant)[db1]> ALTER TABLE test DROP c3;
```

5. 再次查看表 `test` 的列定义，确认 `c3` 列删除成功。

```
obclient(root@mysqltenant)[db1]> DESCRIBE test;
```

返回结果如下：

```
+-------+---------+------+-----+---------+-------+
| Field | Type | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
```

```
| c1 | int(11) | NO | PRI | NULL | |
| c4 | int(11) | YES | | NULL | |
+-------+---------+------+-----+---------+-------+
2 rows in set
```

更多 `ALTER TABLE` 语句相关的语法说明，参见 ALTER TABLE。

3.2.4 删除表

使用 `DROP TABLE` 语句删除表。

示例：删除表 `test` 。

```
obclient(root@mysqltenant)[db1]> DROP TABLE test;
```

更多 `DROP TABLE` 语句相关的语法说明，参见 DROP TABLE。

3.3 索引操作

索引是创建在表上并对数据库表中一列或多列的值进行排序的一种结构。其作用主要在于提高查询的速度，降低数据库系统的性能开销。本节主要介绍数据库中索引的创建、查看、删除的语法和示例。

3.3.5 创建索引

使用 `CREATE INDEX` 语句创建表的索引。

示例：在表 `test` 中创建索引。

1. 查看表 `test` 的列定义。

   ```
   obclient(root@mysqltenant)[db1]> DESCRIBE test;
   ```

   返回结果如下：

   ```
   +-------+----------+------+-----+---------+-------+
   | Field | Type | Null | Key | Default | Extra |
   +-------+----------+------+-----+---------+-------+
   | c1 | int(11) | NO | PRI | NULL | |
   ```

```
| c2 | char(3) | YES | | NULL | |
+-------+----------+------+-----+---------+-------+
2 rows in set
```

2. 在表 `test` 的 `c1` 、 `c2` 列上创建一个名为 `test_index` 的复合索引。

```
obclient(root@mysqltenant)[db1]> CREATE INDEX test_index ON test (c1, c2);
```

更多 `CREATE INDEX` 语句相关的语法说明，参见 CREATE INDEX。

3.3.6 查看索引

使用 `SHOW INDEX` 语句查看表的索引。

示例：查看表 `test` 中的索引信息。

```
obclient(root@mysqltenant)[db1]> SHOW INDEX FROM test\G
```

返回结果如下：

```
*************************** 1. row ***************************
Table: test
Non_unique: 0
Key_name: PRIMARY
Seq_in_index: 1
Column_name: c1
Collation: A
Cardinality: NULL
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
Comment: available
Index_comment:
Visible: YES
```

```
Expression: NULL
*************************** 2. row ***************************
Table: test
Non_unique: 1
Key_name: test_index
Seq_in_index: 1
Column_name: c1
Collation: A
Cardinality: NULL
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
Comment: available
Index_comment:
Visible: YES
Expression: NULL
*************************** 3. row ***************************
Table: test
Non_unique: 1
Key_name: test_index
Seq_in_index: 2
Column_name: c2
Collation: A
Cardinality: NULL
Sub_part: NULL
Packed: NULL
Null: YES
Index_type: BTREE
```

```
Comment: available
Index_comment:
Visible: YES
Expression: NULL
3 rows in set
```

更多 SHOW 语句相关的语法说明，参见 SHOW。

3.3.7 删除索引

使用 DROP INDEX 语句删除表的索引。

示例：删除表 test 中的索引。

```
obclient(root@mysqltenant)[db1]> DROP INDEX test_index ON test;
```

更多 DROP INDEX 语句相关的语法说明，参见 DROP INDEX。

3.4 插入数据

使用 INSERT 语句在已经存在的表中插入数据。

示例：

- 创建表 t1 并插入一行数据。
  1. 创建表 t1 。

     ```
     obclient(root@mysqltenant)[db1]> CREATE TABLE t1(c1 INT PRIMARY KEY, c2 int)
     PARTITION BY KEY(c1) PARTITIONS 4;
     ```

  2. 查看表中的数据。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t1;
     ```

     查询结果为空，表中无数据。

  3. 向表 t1 中插入一行数据。

     ```
     obclient(root@mysqltenant)[db1]> INSERT t1 VALUES(1,1);
     ```

4. 再次查看表中的数据，确认插入一行数据成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t1;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 1 | 1 |
+----+------+
1 row in set
```

- 向表 `t1` 中插入多行数据。

1. 查看当前表中的数据。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t1;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 1 | 1 |
+----+------+
1 row in set
```

2. 向表 `t1` 中同时插入多行数据。

```
obclient(root@mysqltenant)[db1]> INSERT t1 VALUES(2,2),(3,default),(2+2,3*4);
```

3. 再次查看表中的数据，确认数据插入成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t1;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 1 | 1 |
| 2 | 2 |
| 3 | NULL |
| 4 | 12 |
+----+------+
4 rows in set
```

更多 INSERT 语句相关的语法，请参见 INSERT 章节。

3.5 删除数据

使用 DELETE 语句删除数据，支持单表删除和多表删除数据。

为了便于示例说明，下面先创建示例表 t2 和 t3 ，并向表中插入数据。

1. 创建示例表 t2 并插入数据。
   a. 创建一个非分区表 t2 。

      ```
      obclient(root@mysqltenant)[db1]> CREATE TABLE t2(c1 INT PRIMARY KEY, c2 INT);
      ```

   b. 向表 t2 中插入多行数据。

      ```
      obclient(root@mysqltenant)[db1]> INSERT t2 VALUES(1,1),(2,2),(3,3),(5,5);
      ```

   c. 查看表中的数据，确认插入成功。

      ```
      obclient(root@mysqltenant)[db1]> SELECT * FROM t2;
      ```

      返回结果如下：

      ```
      +----+------+
      | c1 | c2 |
      ```

```
+----+------+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 5 | 5 |
+----+------+
4 rows in set
```

2. 创建示例表 t3 并插入数据。

   a. 创建一个 KEY 分区的一级分区表 t3，其分区名由系统根据分区命令规则自动生成，即分区名为 p0 、 p1 、 p2 、 p3 。

   ```
   obclient(root@mysqltenant)[db1]> CREATE TABLE t3(c1 INT PRIMARY KEY, c2 INT) PARTITION BY KEY(c1) PARTITIONS 4;
   ```

   b. 向表 t3 中插入多行数据。

   ```
   obclient(root@mysqltenant)[db1]> INSERT INTO t3 VALUES(5,5),(1,1),(2,2),(3,3);
   ```

   c. 查看表中的数据，确认插入成功。

   ```
   obclient(root@mysqltenant)[db1]> SELECT * FROM t3;
   ```

   返回结果如下：

   ```
   +----+------+
   | c1 | c2 |
   +----+------+
   | 5 | 5 |
   | 1 | 1 |
   | 2 | 2 |
   | 3 | 3 |
   +----+------+
   4 rows in set
   ```

示例：

- 删除表中指定的行。

  1. 查看删除前表 t2 中的数据。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t2;
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 1 | 1 |
     | 2 | 2 |
     | 3 | 3 |
     | 5 | 5 |
     +----+------+
     4 rows in set
     ```

  2. 删除表 t2 中 c1=2 的行，其中 c1 列为表 t2 中的 PRIMARY KEY 。

     ```
     obclient(root@mysqltenant)[db1]> DELETE FROM t2 WHERE c1 = 2;
     ```

  3. 查看删除后表 t2 中的数据，确认删除成功。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t2;
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 1 | 1 |
     | 3 | 3 |
     | 5 | 5 |
     ```

```
+----+------+
3 rows in set
```

- 删除表中满足条件的行。

  1. 查看删除前表 `t2` 中的数据。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t2;
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 1 | 1 |
     | 3 | 3 |
     | 5 | 5 |
     +----+------+
     3 rows in set
     ```

  2. 删除表 `t2` 中按照 `c2` 列排序之后的第一行的数据。

     ```
     obclient(root@mysqltenant)[db1]> DELETE FROM t2 ORDER BY c2 LIMIT 1;
     ```

  3. 查看删除后表 `t2` 中的数据，确认删除成功。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t2;
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 3 | 3 |
     | 5 | 5 |
     ```

```
+----+------+
3 rows in set
```

- 删除表中指定分区的数据。

  1. 查看删除前表 `t3` 中 `p2` 分区的数据。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t3 PARTITION(p2);
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 1 | 1 |
     | 2 | 2 |
     | 3 | 3 |
     +----+------+
     3 rows in set
     ```

  2. 删除表 `t3` 的 `p2` 分区的数据。

     ```
     obclient(root@mysqltenant)[db1]> DELETE FROM t3 PARTITION(p2);
     ```

  3. 查看删除后表 `t3` 中的数据，确认删除成功。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t3;
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 5 | 5 |
     +----+------+
     1 row in set
     ```

- 同时删除多个表中的数据。

  1. 查看删除前表 t2 、t3 中的数据。

     查看表 t2 中的数据。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t2;
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 3 | 3 |
     | 5 | 5 |
     +----+------+
     2 rows in set
     ```

     查看表 t3 中的数据。

     ```
     obclient(root@mysqltenant)[db1]> SELECT * FROM t3;
     ```

     返回结果如下：

     ```
     +----+------+
     | c1 | c2 |
     +----+------+
     | 5 | 5 |
     +----+------+
     1 row in set
     ```

  2. 删除 t2 、t3 表中 t2.c1 = t3.c1 的数据。

     ```
     obclient(root@mysqltenant)[db1]> DELETE t2, t3 FROM t2, t3 WHERE t2.c1 = t3.c1;
     ```

     该语句等价于

```
obclient(root@mysqltenant)[db1]> DELETE FROM t2, t3 USING t2, t3 WHERE t2.c1 = t3.c1;
```

3. 查看删除后表 `t2` 、 `t3` 中的数据，确认删除成功。

   查看表 `t2` 中的数据。

   ```
   obclient(root@mysqltenant)[db1]> SELECT * FROM t2;
   ```

   返回结果如下：

   ```
   +----+------+
   | c1 | c2 |
   +----+------+
   | 3 | 3 |
   +----+------+
   1 row in set
   ```

   查看表 `t3` 中的数据。

   ```
   obclient(root@mysqltenant)[db1]> SELECT * FROM t3;
   ```

   返回结果为空，表 `t3` 中的数据已删除。

更多 `DELETE` 语句相关的语法说明，参见 DELETE。

3.6 更新数据

使用 `UPDATE` 语句修改表中的字段值。

为了便于示例说明，下面先创建示例表 `t4` 和 `t5` ，并向表中插入数据。

1. 创建示例表 `t4` 并插入数据。

   a. 创建一个非分区表 `t2` 。

   ```
   obclient(root@mysqltenant)[db1]> CREATE TABLE t4(c1 INT PRIMARY KEY, c2 INT);
   ```

b. 向表 t4 中插入多行数据。

```
obclient(root@mysqltenant)[db1]> INSERT t4 VALUES(10,10),(20,20),(30,30),
(40,40);
```

c. 查看表中的数据，确认插入成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t4;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 10 |
| 20 | 20 |
| 30 | 30 |
| 40 | 40 |
+----+------+
4 rows in set
```

2. 创建示例表 t5 并插入数据。

a. 创建一个 KEY 分区的一级分区表 t5，其分区名由系统根据分区命令规则自动生成，即分区名为 p0 、 p1 、 p2 、 p3 。

```
obclient(root@mysqltenant)[db1]> CREATE TABLE t5(c1 INT PRIMARY KEY, c2
INT) PARTITION BY KEY(c1) PARTITIONS 4;
```

b. 向表 t5 中插入多行数据。

```
obclient(root@mysqltenant)[db1]> INSERT t5 VALUES(50,50),(10,10),(20,20),
(30,30);
```

c. 查看表中的数据，确认插入成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t5;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 20 | 20 |
| 10 | 10 |
| 50 | 50 |
| 30 | 30 |
+----+------+
4 rows in set
```

示例：

- 更新表中指定行指定列的值。

1. 查看更新前表 `t4` 中的数据。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t4;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 10 |
| 20 | 20 |
| 30 | 30 |
| 40 | 40 |
+----+------+
4 rows in set
```

2. 将表 `t4` 中 `t4.c1=10` 对应的那一行数据的 `c2` 列值修改为 `100` 。

```
obclient(root@mysqltenant)[db1]> UPDATE t4 SET t4.c2 = 100 WHERE t4.c1 = 10;
```

3. 查看更新后表 t4 中的数据，确认更新成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t4;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 100 |
| 20 | 20 |
| 30 | 30 |
| 40 | 40 |
+----+------+
4 rows in set
```

- 将表 t4 中按照 c2 列排序的前两行数据的 c2 列值修改为 100 。

1. 查看更新前表 t4 中的数据。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t4;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 100 |
| 20 | 20 |
| 30 | 30 |
| 40 | 40 |
+----+------+
4 rows in set
```

2. 将表 t4 中按照 c2 列排序的前两行数据的 c2 列值修改为 100 。

```
obclient(root@mysqltenant)[db1]> UPDATE t4 set t4.c2 = 100 ORDER BY c2 LIMIT
2;
```

3. 查看更新后表 t4 中的数据，确认更新成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t4;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 100 |
| 20 | 100 |
| 30 | 100 |
| 40 | 40 |
+----+------+
4 rows in set
```

- 更新表中指定分区的值。
  1. 查看更新前表 t5 中 p1 分区的值。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t5 PARTITION (p1);
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 10 |
| 50 | 50 |
+----+------+
2 rows in set
```

  2. 将表 t5 中 p1 分区的数据中 t5.c1 > 20 的对应行数据的 c2 列值修改为 100 。

```
obclient(root@mysqltenant)[db1]> UPDATE t5 PARTITION(p1) SET t5.c2 = 100
WHERE t5.c1 > 20;
```

3. 查看更新后表 `t5` 中 `p1` 分区的值，确认更新成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t5 PARTITION (p1);
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 10 |
| 50 | 100 |
+----+------+
2 rows in set
```

- 对于表 `t4` 和表 `t5` 中满足 `t4.c2 = t5.c2` 对应行的数据，将表 `t4` 中的 `c2` 列值修改为 `100`，表 `t5` 中的 `c2` 列值修改为 `200`。

1. 查看更新前表 `t4`、`t5` 中的数据。

查看表 `t4` 中的数据。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t4;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 10 | 100 |
| 20 | 100 |
| 30 | 100 |
| 40 | 40 |
```

```
+----+------+
4 rows in set
```

查看表 t5 中的数据。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t5;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 20 | 20 |
| 10 | 10 |
| 50 | 100 |
| 30 | 30 |
+----+------+
4 rows in set
```

2. 对于表 t4 和表 t5 中满足 t4.c2 = t5.c2 对应行的数据，将表 t4 中的 c2 列值修改为 100，表 t5 中的 c2 列值修改为 200。

```
obclient(root@mysqltenant)[db1]> UPDATE t4,t5 SET t4.c2 = 100, t5.c2 = 200
WHERE t4.c2 = t5.c2;
```

3. 查看更新后表 t4、t5 中的数据，确认更新成功。

查看表 t4 中的数据。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t4;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
```

```
| 10 | 100 |
| 20 | 100 |
| 30 | 100 |
| 40 | 40 |
+----+------+
4 rows in set
```

查看表 `t5` 中的数据。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t5;
```

返回结果如下：

```
+----+------+
| c1 | c2 |
+----+------+
| 20 | 20 |
| 10 | 10 |
| 50 | 200 |
| 30 | 30 |
+----+------+
4 rows in set
```

更多 `UPDATE` 语句相关的语法，参见 UPDATE。

3.7 查询数据

使用 `SELECT` 语句查询表中的内容。

示例：

- 通过 `CREATE TABLE` 创建表 `t6` 。从表 `t6` 中读取 `name` 的数据。

  1. 创建表 `t6` 。

```
obclient(root@mysqltenant)[db1]> CREATE TABLE t6 (id INT, name VARCHAR
(50), num INT);
```

2. 向表 t6 中插入多行数据。

```
obclient(root@mysqltenant)[db1]> INSERT INTO t6 VALUES(1,'a',100),(2,'b',200),
(3,'a',50);
```

3. 查看表 t6 中的数据，确认插入成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t6;
```

返回结果如下：

```
+------+------+------+
| ID | NAME | NUM |
+------+------+------+
| 1 | a | 100 |
| 2 | b | 200 |
| 3 | a | 50 |
+------+------+------+
3 rows in set
```

4. 从表 t6 中读取 name 的数据。

```
obclient(root@mysqltenant)[db1]> SELECT name FROM t6;
```

返回结果如下：

```
+------+
| NAME |
+------+
| a |
| b |
```

```
| a |
+------+
3 rows in set
```

- 在查询结果中对 `name` 进行去重处理。

  ```
  obclient(root@mysqltenant)[db1]> SELECT DISTINCT name FROM t6;
  ```

  返回结果如下：

  ```
  +------+
  | NAME |
  +------+
  | a |
  | b |
  +------+
  2 rows in set
  ```

- 从表 `t6` 中根据筛选条件 `name = 'a'`，输出对应的 `id` 、`name` 和 `num` 。

  ```
  obclient(root@mysqltenant)[db1]> SELECT id, name, num FROM t6 WHERE name =
  'a';
  ```

  返回结果如下：

  ```
  +------+------+------+
  | ID | NAME | NUM |
  +------+------+------+
  | 1 | a | 100 |
  | 3 | a | 50 |
  +------+------+------+
  2 rows in set
  ```

更多 `SELECT` 语句相关的语法说明，请参见 SELECT 章节。

3.8 提交事务

使用 COMMIT 语句提交事务。

在提交事务（COMMIT）之前：

- 您的修改只对当前会话可见，对其他数据库会话均不可见。
- 您的修改没有持久化，您可以通过 ROLLBACK 语句撤销修改。

在提交事务（COMMIT）之后：

- 您的修改对所有数据库会话可见。
- 您的修改持久化成功，不能通过 ROLLBACK 语句回滚修改。

示例：通过 CREATE TABLE 创建表 t_insert 。使用 COMMIT 语句提交事务。

1. 创建表 t_insert 。

```
obclient(root@mysqltenant)[db1]> CREATE TABLE t_insert(
id number NOT NULL PRIMARY KEY,
name varchar(10) NOT NULL,
value number,
gmt_create DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

2. 开启事务。

```
obclient(root@mysqltenant)[db1]> BEGIN;
```

3. 向表 t_insert 中插入多行数据。

```
obclient(root@mysqltenant)[db1]> INSERT INTO t_insert(id, name, value,
gmt_create) VALUES(1,'CN',10001, current_timestamp),(2,'US',10002,
current_timestamp),(3,'EN',10003, current_timestamp);
```

4. 查看修改后表 t_insert 中的数据。可以看到，数据插入成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t_insert;
```

返回结果如下：

```
+------+------+-------+--------------------+
| id | name | value | gmt_create |
+------+------+-------+--------------------+
| 1 | CN | 10001 | 2025-11-07 17:46:35 |
| 2 | US | 10002 | 2025-11-07 17:46:35 |
| 3 | EN | 10003 | 2025-11-07 17:46:35 |
+------+------+-------+--------------------+
3 rows in set
```

5. 再次向表 `t_insert` 中插入一行数据。

```
obclient(root@mysqltenant)[db1]> INSERT INTO t_insert(id,name) VALUES(4,'JP');
```

6. 提交事务。

```
obclient(root@mysqltenant)[db1]> COMMIT;
```

7. 退出当前登录。

```
obclient(root@mysqltenant)[db1]> exit;
```

8. 重新登录数据库。

```
obclient -h127.0.0.1 -ur**t@mysqltenant -P2881 -p****** -Ddb1
```

9. 再次查看表 `t_insert` 中的数据。发现表 `t_insert` 中数据的修改持久化成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t_insert;
```

返回结果如下：

```
+------+------+-------+--------------------+
| id | name | value | gmt_create |
+------+------+-------+--------------------+
| 1 | CN | 10001 | 2025-11-07 17:46:35 |
| 2 | US | 10002 | 2025-11-07 17:46:35 |
| 3 | EN | 10003 | 2025-11-07 17:46:35 |
```

```
| 4 | JP | NULL | 2025-11-07 17:46:53 |
+------+------+-------+---------------------+
4 rows in set
```

更多事务控制语句相关的说明，参见 事务管理概述。

3.9 回滚事务

使用 ROLLBACK 语句回滚事务。

回滚一个事务指将事务的修改全部撤销。可以回滚当前整个未提交的事务，也可以回滚到事务中任意一个保存点。如果要回滚到某个保存点，必须结合使用 ROLLBACK 和 TO SAVEPOINT 语句。 其中：

- 如果回滚整个事务，则：
  - 事务会结束
  - 所有的修改会被丢弃
  - 清除所有保存点
  - 释放事务持有的所有锁
- 如果回滚到某个保存点，则：
  - 事务不会结束
  - 保存点之前的修改被保留，保存点之后的修改被丢弃
  - 清除保存点之后的保存点（不包括保存点自身）
  - 释放保存点之后事务持有的所有锁

示例：回滚事务的全部修改。

1. 查看修改前表 t_insert 中的数据。

   ```
   obclient(root@mysqltenant)[db1]> SELECT * FROM t_insert;
   ```

   返回结果如下：

   ```
   +------+------+-------+---------------------+
   | id | name | value | gmt_create |
   +------+------+-------+---------------------+
   | 1 | CN | 10001 | 2025-11-07 17:46:35 |
   | 2 | US | 10002 | 2025-11-07 17:46:35 |
   ```

```
| 3 | EN | 10003 | 2025-11-07 17:46:35 |
+------+------+-------+---------------------+
3 rows in set
```

2. 开启事务。

```
obclient(root@mysqltenant)[db1]> BEGIN;
```

3. 向表 t_insert 中插入多行数据。

```
obclient(root@mysqltenant)[db1]> INSERT INTO t_insert(id, name, value) VALUES
(4,'JP',10004),(5,'FR',10005),(6,'RU',10006);
```

4. 查看修改后表 t_insert 中的数据。可以看到，数据插入成功。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t_insert;
```

返回结果如下：

```
+------+------+-------+---------------------+
| id | name | value | gmt_create |
+------+------+-------+---------------------+
| 1 | CN | 10001 | 2025-11-07 17:46:35 |
| 2 | US | 10002 | 2025-11-07 17:46:35 |
| 3 | EN | 10003 | 2025-11-07 17:46:35 |
| 4 | JP | 10004 | 2025-11-07 17:48:01 |
| 5 | FR | 10005 | 2025-11-07 17:48:01 |
| 6 | RU | 10006 | 2025-11-07 17:48:01 |
+------+------+-------+---------------------+
6 rows in set
```

5. 回滚事务。

```
obclient(root@mysqltenant)[db1]> ROLLBACK;
```

6. 回滚后，再次查看表 t_insert 中的数据，发现插入的数据已回滚。

```
obclient(root@mysqltenant)[db1]> SELECT * FROM t_insert;
```

返回结果如下：

```
+------+------+-------+---------------------+
| id | name | value | gmt_create |
+------+------+-------+---------------------+
| 1 | CN | 10001 | 2025-11-07 17:46:35 |
| 2 | US | 10002 | 2025-11-07 17:46:35 |
| 3 | EN | 10003 | 2025-11-07 17:46:35 |
+------+------+-------+---------------------+
3 rows in set
```

更多事务控制语句相关的说明，参见 事务管理概述。

4 SQL 基础操作（Oracle 模式）

本节主要介绍 OceanBase 数据库 Oracle 模式下的一些 SQL 基本操作。

4.0.0.1 功能适用性

该内容仅适用于 OceanBase 数据库企业版。OceanBase 数据库社区版仅提供 MySQL 模式。

4.1 表操作

本节主要提供数据库中表的创建、查看、修改和删除的语法和示例。

4.1.1 创建表

使用 `CREATE TABLE` 语句在数据库中创建新表。

示例：创建表 `TEST` 。

```
obclient(SYS@oracletenant)[SYS]> CREATE TABLE TEST (C1 INT PRIMARY KEY, C2
VARCHAR(3));
```

更多 `CREATE TABLE` 语句相关的语法说明，参见 CREATE TABLE。

4.1.2 修改表

使用 `ALTER TABLE` 语句来修改已存在的表的结构，包括修改表及表属性、新增列、修改列及属性、删除列等。

示例：

- 修改表 `TEST` 的字段 `C2` 的字段类型。

  1. 查看表 `TEST` 的列定义。

     ```
     obclient(SYS@oracletenant)[SYS]> DESCRIBE TEST;
     ```

     返回结果如下：

     ```
     +-------+------------+------+-----+---------+-------+
     | FIELD | TYPE | NULL | KEY | DEFAULT | EXTRA |
     +-------+------------+------+-----+---------+-------+
     ```

```
| C1 | NUMBER(38) | NO | PRI | NULL | NULL |
| C2 | VARCHAR2(3) | YES | NULL| NULL | NULL |
+-------+------------+------+-----+---------+-------+
2 rows in set
```

2. 将表 `TEST` 中 `C2` 的字段类型修改为 `CHAR` 。

```
obclient(SYS@oracletenant)[SYS]> ALTER TABLE TEST MODIFY C2 CHAR(10);
```

3. 再次查看表 `TEST` 的列定义，确认修改成功。

```
obclient(SYS@oracletenant)[SYS]> DESCRIBE TEST;
```

返回结果如下：

```
+-------+------------+------+-----+---------+-------+
| FIELD | TYPE | NULL | KEY | DEFAULT | EXTRA |
+-------+------------+------+-----+---------+-------+
| C1 | NUMBER(38) | NO | PRI | NULL | NULL |
| C2 | CHAR(10) | YES | NULL| NULL | NULL |
+-------+------------+------+-----+---------+-------+
2 rows in set
```

- 在表 `TEST` 中增加、删除列。

1. 查看当前表 `TEST` 的列定义。

```
obclient(SYS@oracletenant)[SYS]> DESCRIBE TEST;
```

返回结果如下：

```
+-------+------------+------+-----+---------+-------+
| FIELD | TYPE | NULL | KEY | DEFAULT | EXTRA |
+-------+------------+------+-----+---------+-------+
| C1 | NUMBER(38) | NO | PRI | NULL | NULL |
| C2 | CHAR(10) | YES | NULL| NULL | NULL |
```

```
+-------+-----------+------+-----+---------+-------+
2 rows in set
```

2. 向表 `TEST` 中添加一列 `C3` 。

```
obclient(SYS@oracletenant)[SYS]> ALTER TABLE TEST ADD C3 int;
```

3. 再次查看表 `TEST` 的列定义，确认 `C3` 列添加成功。

```
obclient(SYS@oracletenant)[SYS]> DESCRIBE TEST;
```

返回结果如下：

```
+-------+-----------+------+-----+---------+-------+
| FIELD | TYPE | NULL | KEY | DEFAULT | EXTRA |
+-------+-----------+------+-----+---------+-------+
| C1 | NUMBER(38) | NO | PRI | NULL | NULL |
| C2 | CHAR(10) | YES | NULL | NULL | NULL |
| C3 | NUMBER(38) | YES | NULL | NULL | NULL |
+-------+-----------+------+-----+---------+-------+
3 rows in set
```

4. 删除表 `TEST` 中 `C3` 列。

```
obclient(SYS@oracletenant)[SYS]> ALTER TABLE TEST DROP COLUMN C3;
```

5. 再次查看表 `TEST` 的列定义，确认 `C3` 列删除成功。

```
obclient(SYS@oracletenant)[SYS]> DESCRIBE TEST;
```

返回结果如下：

```
+-------+-----------+------+-----+---------+-------+
| FIELD | TYPE | NULL | KEY | DEFAULT | EXTRA |
+-------+-----------+------+-----+---------+-------+
| C1 | NUMBER(38) | NO | PRI | NULL | NULL |
```

```
| C2 | CHAR(10) | YES | NULL | NULL | NULL |
+-------+------------+------+-----+---------+-------+
2 rows in set
```

更多 ALTER TABLE 语句相关的语法说明，参见 ALTER TABLE。

4.1.3 删除表

使用 DROP TABLE 语句删除表。

示例：删除表 TEST 。

```
obclient(SYS@oracletenant)[SYS]> DROP TABLE TEST;
```

更多 DROP TABLE 语句相关的语法说明，参见 DROP TABLE。

4.2 索引操作

索引是创建在表上并对数据库表中一列或多列的值进行排序的一种结构。其作用主要在于提高查询的速度，降低数据库系统的性能开销。

4.2.4 创建索引

使用 CREATE INDEX 语句创建表的索引。

示例：创建表 TEST 的索引。

1. 查看表 TEST 的列定义。

   ```
   obclient(SYS@oracletenant)[SYS]> DESCRIBE TEST;
   ```

   返回结果如下：

   ```
   +-------+------------+------+-----+---------+-------+
   | FIELD | TYPE | NULL | KEY | DEFAULT | EXTRA |
   +-------+------------+------+-----+---------+-------+
   | C1 | NUMBER(38) | NO | PRI | NULL | NULL |
   | C2 | CHAR(10) | YES | NULL| NULL | NULL |
   ```

```
+-------+-----------+------+-----+---------+-------+
2 rows in set
```

2. 在表 TEST 的 C1 、 C2 列上创建一个名为 TEST_INDEX 的复合索引。

```
obclient(SYS@oracletenant)[SYS]> CREATE INDEX TEST_INDEX ON TEST (C1, C2);
```

更多 CREATE INDEX 语句相关的语法说明，参见 CREATE INDEX。

4.2.5 查看索引

- 通过视图 ALL_INDEXES 查看表的所有索引。

```
obclient(SYS@oracletenant)[SYS]> SELECT OWNER,INDEX_NAME,INDEX_TYPE,
TABLE_OWNER,TABLE_NAME FROM ALL_INDEXES WHERE table_name='TEST';
```

返回结果如下：

```
+-------+---------------------------+-----------+------------+------------+
| OWNER | INDEX_NAME | INDEX_TYPE | TABLE_OWNER | TABLE_NAME |
+-------+---------------------------+-----------+------------+------------+
| SYS | TEST_OBPK_1762742804587195 | NORMAL | SYS | TEST |
| SYS | TEST_INDEX | NORMAL | SYS | TEST |
+-------+---------------------------+-----------+------------+------------+
2 rows in set
```

- 通过 USER_IND_COLUMNS 查看表索引的详细信息。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM USER_IND_COLUMNS WHERE
table_name='TEST'\G
```

返回结果如下：

```
*************************** 1. row ***************************
INDEX_NAME: TEST_OBPK_1762742804587195
TABLE_NAME: TEST
```

```
COLUMN_NAME: C1
COLUMN_POSITION: 1
COLUMN_LENGTH: 22
CHAR_LENGTH: 0
DESCEND: ASC
COLLATED_COLUMN_ID: NULL
*************************** 2. row ***************************
INDEX_NAME: TEST_INDEX
TABLE_NAME: TEST
COLUMN_NAME: C1
COLUMN_POSITION: 1
COLUMN_LENGTH: 22
CHAR_LENGTH: 0
DESCEND: ASC
COLLATED_COLUMN_ID: NULL
*************************** 3. row ***************************
INDEX_NAME: TEST_INDEX
TABLE_NAME: TEST
COLUMN_NAME: C2
COLUMN_POSITION: 2
COLUMN_LENGTH: 3
CHAR_LENGTH: 3
DESCEND: ASC
COLLATED_COLUMN_ID: NULL
3 rows in set
```

4.2.6 删除索引

使用 `DROP INDEX` 语句删除表的索引。

示例：删除索引 `TEST_INDEX` 。

```
obclient(SYS@oracletenant)[SYS]> DROP INDEX TEST_INDEX;
```

更多 `DROP INDEX` 语句相关的语法说明，参见 DROP INDEX。

4.3 插入数据

使用 `INSERT` 语句添加一个或多个记录到表中。

示例：

- 通过 `CREATE TABLE` 创建表 `T1`，并向表 `T1` 中插入一行数据。
  1. 创建表 `t1` 。

     ```
     obclient(SYS@oracletenant)[SYS]> CREATE TABLE T1(C1 INT PRIMARY KEY, C2 INT);
     ```

  2. 查看表中的数据。

     ```
     obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
     ```

     查询结果为空，表中无数据。

  3. 向表 `T1` 中插入一行数据。

     ```
     obclient(SYS@oracletenant)[SYS]> INSERT INTO T1 VALUES(1,1);
     ```

  4. 再次查看表中的数据，确认插入一行数据成功。

     ```
     obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
     ```

     返回结果如下：

     ```
     +----+------+
     | C1 | C2 |
     +----+------+
     | 1 | 1 |
     +----+------+
     1 row in set
     ```

- 直接向子查询中插入数据。

```
obclient(SYS@oracletenant)[SYS]> INSERT INTO (SELECT * FROM T1) VALUES(2,2);
```

执行成功后，查看表中的数据，确认插入成功。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
```

返回结果如下：

```
+----+------+
| C1 | C2 |
+----+------+
| 1 | 1 |
| 2 | 2 |
+----+------+
2 rows in set
```

- 包含 `RETURNING` 子句的数据插入。

  1. 向表 `T1` 中插入一行数据，并返回插入行的 `C1` 列的数据。

     ```
     obclient(SYS@oracletenant)[SYS]> INSERT INTO T1 VALUES(3,3) RETURNING C1;
     ```

     返回结果如下：

     ```
     +----+
     | C1 |
     +----+
     | 3 |
     +----+
     1 row in set
     ```

  2. 再次查看表中的数据，确认插入成功。

     ```
     obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
     ```

     返回结果如下：

```
+----+------+
| C1 | C2 |
+----+------+
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
+----+------+
3 rows in set
```

更多 INSERT 语句相关的语法，参见 INSERT。

4.4 删除数据

使用 DELETE 语句删除数据。

示例：删除表 T1 中 C1=2 的行。

1. 查看删除数据前表 T1 中的数据。

   ```
   obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
   ```

   返回结果如下：

   ```
   +----+------+
   | C1 | C2 |
   +----+------+
   | 1 | 1 |
   | 2 | 2 |
   | 3 | 3 |
   +----+------+
   3 rows in set
   ```

2. 删除表 T1 中 C1=2 的行。

   ```
   
   ```

```
obclient(SYS@oracletenant)[SYS]> DELETE FROM T1 WHERE C1 = 2;
```

3. 查看删除数据后表 `T1` 中的数据。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
```

返回结果如下：

```
+----+------+
| C1 | C2 |
+----+------+
| 1 | 1 |
| 3 | 3 |
+----+------+
2 rows in set
```

更多 `DELETE` 语句相关的语法说明，参见 DELETE。

4.5 更新数据

使用 `UPDATE` 语句修改表中的字段值。

示例：

- 将表 `T1` 中 `T1.C1=1` 对应的那一行数据的 `C2` 列值修改为 `100` 。

1. 查看更新数据前表 `T1` 中的数据。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
```

返回结果如下：

```
+----+------+
| C1 | C2 |
+----+------+
| 1 | 1 |
| 3 | 3 |
```

```
+----+------+
2 rows in set
```

2. 将表 T1 中 T1.C1=1 对应的那一行数据的 C2 列值修改为 100 。

```
obclient(SYS@oracletenant)[SYS]> UPDATE T1 SET T1.C2 = 100 WHERE T1.C1 = 1;
```

3. 查看更新数据后表 T1 中的数据。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
```

返回结果如下：

```
+----+------+
| C1 | C2 |
+----+------+
| 1 | 100 |
| 3 | 3 |
+----+------+
2 rows in set
```

- 直接操作子查询，将子查询中 V.C1=3 对应的那一行数据的 C2 列值修改为 300 。

```
obclient(SYS@oracletenant)[SYS]> UPDATE (SELECT * FROM T1) V SET V.C2 = 300
WHERE V.C1 = 3;
```

语句执行成功后，查看更新数据后表 T1 中的数据。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T1;
```

返回结果如下：

```
+----+------+
| C1 | C2 |
+----+------+
| 1 | 100 |
```

```
| 3 | 300 |
+----+------+
2 rows in set
```

更多 UPDATE 语句相关的语法，参见 UPDATE。

4.6 查询数据

使用 SELECT 语句查询表中的内容。

示例：

- 通过 CREATE TABLE 创建表 T2 。从表 T2 中读取 NAME 的数据。

  1. 创建表 t2 。

     ```
     obclient(SYS@oracletenant)[SYS]> CREATE TABLE T2 (ID INT, NAME VARCHAR
     (50), NUM INT);
     ```

  2. 向表 T2 中插入多行数据。

     ```
     obclient(SYS@oracletenant)[SYS]> INSERT INTO T2 VALUES(1,'a',100),(2,'b',200),
     (3,'a',50);
     ```

  3. 查看表 T2 中的数据，确认插入成功。

     ```
     obclient(SYS@oracletenant)[SYS]> SELECT * FROM T2;
     ```

     返回结果如下：

     ```
     +------+------+------+
     | ID | NAME | NUM |
     +------+------+------+
     | 1 | a | 100 |
     | 2 | b | 200 |
     | 3 | a | 50 |
     +------+------+------+
     3 rows in set
     ```

4. 从表 T2 中读取字段 NAME 的数据。

```
obclient(SYS@oracletenant)[SYS]> SELECT NAME FROM T2;
```

返回结果如下：

```
+------+
| NAME |
+------+
| a |
| b |
| a |
+------+
3 rows in set
```

- 在查询结果中对 NAME 进行去重处理。

```
obclient(SYS@oracletenant)[SYS]> SELECT DISTINCT NAME FROM T2;
```

返回结果如下：

```
+------+
| NAME |
+------+
| a |
| b |
+------+
2 rows in set
```

- 从表 T2 中根据筛选条件 NAME = 'a'，输出对应的 ID 、NAME 和 NUM 。

```
obclient(SYS@oracletenant)[SYS]> SELECT ID, NAME, NUM FROM T2 WHERE NAME =
'a';
```

返回结果如下：

```
+------+------+------+
| ID | NAME | NUM |
+------+------+------+
| 1 | a | 100 |
| 3 | a | 50 |
+------+------+------+
2 rows in set
```

更多 SELECT 语句相关的语法说明，参见 SELECT。

4.7 提交事务

使用 COMMIT 语句提交事务。

在您提交事务之前，您的修改只对当前会话可见，对其他数据库会话是不可见的；您的修改没有持久化，可以用 ROLLBACK 语句撤销修改。

在您提交事务之后，您的修改对所有数据库会话可见。您的修改结果持久化成功，不可以用 ROLLBACK 语句回滚修改。

示例：通过 CREATE TABLE 创建表 T_INSERT 。使用 COMMIT 语句提交事务。

1. 创建表 T_INSERT 。

```
obclient(SYS@oracletenant)[SYS]> CREATE TABLE T_INSERT(
ID NUMBER NOT NULL PRIMARY KEY,
NAME VARCHAR(10) NOT NULL,
VALUE NUMBER NOT NULL,
GMT_CREATE DATE NOT NULL DEFAULT sysdate
);
```

2. 向表 T_INSERT 中插入多行数据。

```
obclient(SYS@oracletenant)[SYS]> INSERT INTO T_INSERT(ID, NAME, VALUE,
GMT_CREATE) VALUES(1,'CN',10001, sysdate),(2,'US',10002, sysdate),(3,'EN',
10003, sysdate);
```

3. 查看修改后表 T_INSERT 中的数据，确认插入数据成功。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T_INSERT;
```

返回结果如下：

```
+------+------+-------+------------+
| ID | NAME | VALUE | GMT_CREATE |
+------+------+-------+------------+
| 1 | CN | 10001 | 10-NOV-25 |
| 2 | US | 10002 | 10-NOV-25 |
| 3 | EN | 10003 | 10-NOV-25 |
+------+------+-------+------------+
3 rows in set
```

4. 再次向表 T_INSERT 中插入一行数据。

```
obclient(SYS@oracletenant)[SYS]> INSERT INTO T_INSERT(ID, NAME, VALUE)
VALUES(4,'JP',10004);
```

5. 提交事务。

```
obclient(SYS@oracletenant)[SYS]> COMMIT;
```

6. 退出登录。

```
obclient(SYS@oracletenant)[SYS]> exit;
```

7. 重新登录数据库。

```
obclient -h127.0.0.1 -us**@oracletenant -P2881 -p******
```

8. 查看表 T_INSERT 中的数据，发现表 T_INSERT 中数据的修改持久化成功。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T_INSERT;
```

返回结果如下：

```
+------+------+-------+------------+
| ID | NAME | VALUE | GMT_CREATE |
+------+------+-------+------------+
| 1 | CN | 10001 | 10-NOV-25 |
| 2 | US | 10002 | 10-NOV-25 |
| 3 | EN | 10003 | 10-NOV-25 |
| 4 | JP | 10004 | 10-NOV-25 |
+------+------+-------+------------+
4 rows in set
```

更多事务控制语句相关的说明，参见 事务管理概述。

4.8 回滚事务

使用 `ROLLBACK` 语句可以回滚事务。

回滚一个事务指将事务的修改全部撤销。可以回滚当前整个未提交的事务，也可以回滚到事务中任意一个保存点。如果要回滚到某个保存点，必须结合使用 `ROLLBACK` 和 `TO SAVEPOINT` 语句。

其中：

- 如果回滚整个事务，则：
  - 事务会结束
  - 所有的修改会被丢弃
  - 清除所有保存点
  - 释放事务持有的所有锁
- 如果回滚到某个保存点，则：
  - 事务不会结束
  - 保存点之前的修改被保留，保存点之后的修改被丢弃
  - 清除保存点之后的保存点（不包括保存点自身）
  - 释放保存点之后事务持有的所有锁

示例：回滚事务的全部修改。

1. 查看修改前表 `T_INSERT` 中的数据。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T_INSERT;
```

返回结果如下：

```
+------+------+-------+------------+
| ID | NAME | VALUE | GMT_CREATE |
+------+------+-------+------------+
| 1 | CN | 10001 | 10-NOV-25 |
| 2 | US | 10002 | 10-NOV-25 |
| 3 | EN | 10003 | 10-NOV-25 |
| 4 | JP | 10004 | 10-NOV-25 |
+------+------+-------+------------+
4 rows in set
```

2. 向表 `T_INSERT` 中插入两行数据。

```
obclient(SYS@oracletenant)[SYS]> INSERT INTO T_INSERT(ID, NAME, VALUE)
VALUES(5,'FR',10005),(6,'RU',10006);
```

3. 查看修改后表 `T_INSERT` 中的数据，确认插入数据成功。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T_INSERT;
```

返回结果如下：

```
+------+------+-------+------------+
| ID | NAME | VALUE | GMT_CREATE |
+------+------+-------+------------+
| 1 | CN | 10001 | 10-NOV-25 |
| 2 | US | 10002 | 10-NOV-25 |
| 3 | EN | 10003 | 10-NOV-25 |
| 4 | JP | 10004 | 10-NOV-25 |
| 5 | FR | 10005 | 10-NOV-25 |
| 6 | RU | 10006 | 10-NOV-25 |
```

```
+------+------+-------+------------+
6 rows in set
```

4. 回滚事务。

```
obclient(SYS@oracletenant)[SYS]> ROLLBACK;
```

5. 回滚后，再次查看表 T_INSERT 中的数据，发现插入的数据已回滚。

```
obclient(SYS@oracletenant)[SYS]> SELECT * FROM T_INSERT;
```

返回结果如下：

```
+------+------+-------+------------+
| ID | NAME | VALUE | GMT_CREATE |
+------+------+-------+------------+
| 1 | CN | 10001 | 10-NOV-25 |
| 2 | US | 10002 | 10-NOV-25 |
| 3 | EN | 10003 | 10-NOV-25 |
| 4 | JP | 10004 | 10-NOV-25 |
+------+------+-------+------------+
4 rows in set
```

更多事务控制语句相关的说明，参见 事务管理概述。

5 创建 Java 示例应用程序

本文介绍了 Java 应用程序如何使用 OceanBase Connector/J 驱动连接并使用 OceanBase 数据库。

5.0.0.1 功能适用性

该内容仅适用于 OceanBase 数据库企业版。OceanBase 数据库社区版仅提供 MySQL 模式。

5.1 前提条件

- 确保设置了基本的数据库开发环境。
- 确保计算机上的 Java 环境为 Java JDK 8 版本。
- 获取 OceanBase Connector/J 驱动程序安装包。请在 OceanBase 官方网站的 **资源 -> 下载中心 -> 企业版 -> 驱动和中间件** 下的 OceanBase JDBC 驱动程序 中单击对应的版本，填写信息后自助下载 OceanBase Connector/J 驱动程序安装包。

5.2 创建 Java 应用程序

5.2.1 步骤一：获取数据库连接串

联系 OceanBase 数据库部署人员或者管理员获取相应的数据库连接串，例如：

```
obclient -h100.88.xx.xx -usys@oracle -p****** -P2883
```

数据库连接串包含了访问数据库所需的参数信息，在创建应用程序前，可通过数据库连接串验证登录数据库，保证连接串参数信息正确。

参数说明：

- **-h**：OceanBase 数据库连接 IP，有时候是一个 ODP 地址。
- **-u**：租户的连接用户名，格式为 **用户@租户#集群名称**，租户的连接用户，Oracle 模式的管理员用户名默认是 `sys` 。直连数据库时不填集群名称，通过 ODP 连接时需要填写。
- **-p**：用户密码。
- **-P**：OceanBase 数据库连接端口，也是 ODP 的监听端口。

5.2.2 步骤二：安装 OceanBase Connector/J 驱动

将 OceanBase Connector/J 的 JAR 包解压后放入本地 `/usr/share/java` 路径中，并设置临时环境变量。

```
mv ./oceanbase-client-{version}.jar /usr/share/java
export CLASSPATH=/usr/share/java/oceanbase-client-{version}.jar:$CLASSPATH
```

5.2.2.1 说明

根据下载的实际文件版本进行相应操作。

5.2.3 步骤三：编写应用程序

编写 Java 示例文件 `Test.java` 。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.Statement;

public class Test {
public static void main(String[] args) {
try {

Class.forName("com.oceanbase.jdbc.Driver");
Connection connection = DriverManager.getConnection("jdbc:oceanbase://172.30.
xx.xx:2881/?pool=false&user=s**@oracle&password=******");
System.out.println(connection.getAutoCommit());
Statement sm = connection.createStatement();
//新建表 t_meta_form
sm.executeUpdate("CREATE TABLE t_meta_form (name varchar(36) , id int)");
//插入数据
sm.executeUpdate("insert into t_meta_form values ('an','1')");
```

```
//查询数据，并输出结果
ResultSet rs = sm.executeQuery("select * from t_meta_form");
while (rs.next()) {
String name = rs.getString("name");
String id = rs.getString("id");
System.out.println(name + ','+ id);
}
//删除表
sm.executeUpdate("drop table t_meta_form");
}catch (SQLException ex) {
System.out.println("error!");
ex.printStackTrace() ;
}catch (ClassNotFoundException e) {
e.printStackTrace();
}
}
}
```

修改代码中的数据库连接参数。参考如下字段，对应的值，则取自步骤一获取的数据库连接串。

- **url**：取自 `-h` 和 `-P` 参数，`jdbc:oceanbase://IP:port/?pool=false` 。OceanBase 数据库连接 IP，通常是一个 ODP 地址，以及访问所用的端口号。
- **user**：取自 `-u` 参数，租户的连接用户名，格式为 **用户@租户#集群名称**，Oracle 模式的管理员用户名默认是 `sys` 。直连数据库时不填集群名称，通过 ODP 连接时需要填写。
- **password**：取自 `-p` 参数，用户密码。

5.2.4 步骤四：执行应用程序

代码编辑完成后，可以通过如下命令进行编译：

```
javac Test.java
```

编译完成后，执行获得如下结果，说明数据库连接成功，示例语句正确执行：

```
java Test


true
an,1
```

5.3 更多信息

关于 OceanBase Connector/J 的详细使用信息，请参考官方文档《OceanBase Connector/J》。

6 创建 C 示例应用程序

本文介绍 C 应用程序如何通过 OBCI 连接并使用 OceanBase 数据库。

6.0.0.1 功能适用性

该内容仅适用于 OceanBase 数据库企业版。OceanBase 数据库社区版仅提供 MySQL 模式。

6.1 前提条件

- 确保设置了基本的数据库开发环境。
- 确保满足如下硬件环境：
  - 硬件平台：x86_64。
  - 操作系统：CentOS/Redhat 系 Linux 发行版 7.2。
  - 编译器：GCC 4.8。
- 请联系技术支持人员获取 OBCI 和 LibOBClient 的 RPM 安装包。

6.1.0.1 注意

前提条件还需注意以下事项：

- 从 OBCI V2.0.4 开始，RPM 包的 include 文件夹中不再包含 oci.h、ociap.h 等头文件，业务编译需要首先安装 Oracle clntsh 相关 RPM，直接使用其头文件。
- OceanBase：V2.2.76 及以上版本（推荐使用 OceanBase 数据库 V4.0 及以上版本，以体验当前版本的完整功能）。
- ODP（OBProxy）：推荐 V4.0 及以上版本。
- libobclient：V2.1.1 及以上版本。
- 安装 Oracle 相关驱动文件 oracle-instantclient（从 OBCI V2.0.4 开始需要依赖 Oracle 头文件）。
- 已安装 OBCI，并已提前部署 GCC 编译器。

6.1.1 步骤一：获取数据库连接串

联系 OceanBase 数据库部署人员或者管理员获取相应的数据库连接串，例如：

```
obclient -h100.88.xx.xx -usys@oracle -p****** -P2881
```

数据库连接串包含了访问数据库所需的参数信息，在创建应用程序前，可通过数据库连接串验证登录数据库，保证连接串参数信息正确。

参数说明：

- **-h**：OceanBase 数据库连接 IP，有时候是一个 ODP 地址。
- **-u**：租户的连接用户名，格式为 **用户@租户#集群名称**，租户的连接用户，Oracle 模式的管理员用户名默认是 `sys` 。直连数据库时不填集群名称，通过 ODP 连接时需要填写。
- **-p**：用户密码。
- **-P**：OceanBase 数据库连接端口，也是 ODP 的监听端口。

6.1.2 步骤二：安装 C 相关驱动

当您获取 RPM 包后，在命令行工具中以 root 用户权限执行如下命令进行 OBCI 驱动的安装：

```
rpm -ivh obci-<version>.x86_64.rpm
rpm -ivh libobclient-<version>.x86_64.rpm
```

在默认情况下，软件包中包含的程序与文件将安装在如下路径中：

- 头文件被安装在 `/u01/obclient/include` 路径下。
- 库文件被安装在 `/u01/obclient/lib` 路径下。

6.1.3 步骤三：编写应用程序

本文通过具体实例介绍在 OceanBase 数据库 Oracle 模式下，C 应用程序通过 OBCI 与数据库服务器 OBServer 节点交互的基本方式。

1. 初始化 OBCI 环境和线程。

   ```
   /*初始化 OBCI 程序环境*/
   OCIInitialize(OCI_DEFAULT, NULL, NULL, NULL, NULL)


   /*初始化环境句柄*/
   OCIEnvInit(&envhp, OCI_DEFAULT, 0, 0)
   ```

2. 分配必要的句柄与数据结构。

   ```
   /*服务上下文句柄*/
   OCIHandleAlloc(envhp, (dvoid **)&svchp, OCI_HTYPE_SVCCTX, 0, 0)
   ```

```
/*服务器句柄*/
OCIHandleAlloc(envhp, (dvoid **)&srvhp, OCI_HTYPE_SERVER, 0, 0)

/*会话句柄*/
OCIHandleAlloc(envhp, (dvoid **)&authp, OCI_HTYPE_SESSION, 0, 0)

/*错误句柄*/
OCIHandleAlloc(envhp, (dvoid **)&errhp, OCI_HTYPE_ERROR, 0, 0)

/*描述句柄*/
OCIHandleAlloc(envhp, (dvoid **)&dschp, OCI_HTYPE_DESCRIBE, 0, 0)
```

3. 建立与数据库的连接以及创建用户会话。

```
/*设置用户名和密码*/
OCIAttrSet(authp, OCI_HTYPE_SESSION, (text *)strUserName,
(ub4)strlen(strUserName), OCI_ATTR_USERNAME, errhp)
OCIAttrSet(authp, OCI_HTYPE_SESSION, (text *)strPassword,
(ub4)strlen(strPassword), OCI_ATTR_PASSWORD, errhp)

/*设置服务器环境句柄属性*/
OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX,(dvoid *)srvhp, (ub4)0,
OCI_ATTR_SERVER, errhp)
OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, (dvoid *)authp,0, OCI_ATTR_SESSION, errhp)

/*创建并开始一个用户会话*/
OCISessionBegin(svchp, errhp, authp, OCI_CRED_RDBMS, OCI_DEFAULT)
OCIHandleAlloc(envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0)
```

4. 通过 SQL 与 OceanBase 服务器交换数据，而后再做数据处理。一条 SQL 语句在 OBCI 应用程序中的执行步骤如下：

a. 通过调用函数 `OCIStmtPrepare()` 或者 `OCIStmtPrepare2()` 准备 SQL 语句。

```
OCIStmtPrepare(stmthp, errhp, (text *)sql, strlen(sql), OCI_NTV_SYNTAX,
OCI_DEFAULT)
```

b. 通过调用一个或者多个函数，如 `OCIBindByPos()` 或 `OCIBindByName()` 等把输入变量的地址绑定在 DML 语句中的占位符中。

```
OCIBindByPos(stmthp, &bidhp[0], errhp, 1, &szpersonid,
sizeof(szpersonid), SQLT_INT, NULL, NULL, NULL, 0, NULL, 0)


OCIBindByName(stmthp, &bidhp[0], errhp, (const OraText*)":personid", 9,
&szpersonid,
sizeof(szpersonid), SQLT_INT, NULL, NULL, NULL, 0, NULL, 0)
```

c. 调用 `OCIStmrExecute()` 函数执行 SQL 语句。

```
OCIStmtExecute(svchp, stmthp, errhp, (ub4)1, (ub4)0, (CONST OCISnapshot *)
0,
(OCISnapshot *)0, (ub4)OCI_DEFAULT)
```

d. 调用 `OCIDefineByPos()` 函数为 SQL 语句中的数据输出项定义输出变量。

```
OCIDefineByPos(stmthp, &defhp[0], errhp, 1, &szpersonid,
sizeof(szpersonid), SQLT_INT, &ind[0], 0, 0, OCI_DEFAULT)
```

e. 调用 `OCIStmtFetch()` 函数来获取查询的结果集。

```
OCIStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT)
```

5. 结束用户会话与断开与数据库的连接。

```
/*结束会话*/
OCISessionEnd(svchp, errhp, authp, (ub4)0)
```

```
/*断开与数据库的连接*/
OCIServerDetach(srvhp, errhp, OCI_DEFAULT)
```

6. 释放在程序中所分配的句柄。

```
OCIHandleFree((dvoid *)dschp, OCI_HTYPE_DESCRIBE)
OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT)
OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR)
OCIHandleFree((dvoid *)authp, OCI_HTYPE_SESSION)
OCIHandleFree((dvoid *)svchp, OCI_HTYPE_SVCCTX)
OCIHandleFree((dvoid *)srvhp, OCI_HTYPE_SERVER)
```

6.1.3.2 示例代码

示例文件 test.c 代码内容如下：

```
/************************************************************
* Copyright(C) 2014 - 2020 Alibaba Inc. All Rights Reserved.
*
* Filename: ob_oci_test.c
* Description: ----
* Create: 2020-07-07 10:14:59
* Last Modified: 2020-07-07 10:14:59
************************************************************/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include "oci.h"
/*声明句柄*/
OCIEnv *envhp; /*环境句柄*/
OCISvcCtx *svchp; /*服务环境句柄*/
```

```
OCIServer *srvhp; /*服务器句柄*/
OCISession *authp; /*会话句柄*/
OCIStmt *stmthp; /*语句句柄*/
OCIDescribe *dschp; /*描述句柄*/
OCIError *errhp; /*错误句柄*/
OCIDefine *defhp[3]; /*定义句柄*/
OCIBind *bidhp[4]; /*绑定句柄*/
sb2 ind[3]; /*指示符变量*/
/*绑定 select 结果集的参数*/
int szpersonid; /*存储 personid 列*/
text szname[51]; /*存储 name 列*/
text szemail[51]; /*存储 mail 列*/
char sql[256]; /*存储执行的 sql 语句*/

int main(int argc, char *argv[])
{
char strServerName[50];
char strUserName[50];
char strPassword[50];
/*设置服务器，用户名和密码*/
strcpy(strServerName, "172.30.xx.xx:2881");
strcpy(strUserName, "s**@oracle");
strcpy(strPassword, "******");
/*初始化 OCI 应用环境*/
OCIInitialize(OCI_DEFAULT, NULL, NULL, NULL, NULL);
/*初始化环境句柄*/
OCIEnvInit(&envhp, OCI_DEFAULT, 0, 0);
/*分配句柄*/
OCIHandleAlloc(envhp, (dvoid **)&svchp, OCI_HTYPE_SVCCTX, 0, 0);
```

```
/*服务器环境句柄*/
OCIHandleAlloc(envhp, (dvoid **)&srvhp, OCI_HTYPE_SERVER, 0, 0);
/*服务器句柄*/
OCIHandleAlloc(envhp, (dvoid **)&authp, OCI_HTYPE_SESSION, 0, 0);
/*会话句柄*/
OCIHandleAlloc(envhp, (dvoid **)&errhp, OCI_HTYPE_ERROR, 0, 0);
/*错误句柄*/
OCIHandleAlloc(envhp, (dvoid **)&dschp, OCI_HTYPE_DESCRIBE, 0, 0);
/*描述符句柄*/
/*连接服务器*/
OCIServerAttach(srvhp, errhp, (text *)strServerName, (sb4)strlen(strServerName),
OCI_DEFAULT);
/*设置用户名和密码*/
OCIAttrSet(authp, OCI_HTYPE_SESSION, (text *)strUserName, (ub4)strlen
(strUserName), OCI_ATTR_USERNAME, errhp);
OCIAttrSet(authp, OCI_HTYPE_SESSION, (text *)strPassword, (ub4)strlen
(strPassword), OCI_ATTR_PASSWORD, errhp);
/*设置服务器环境句柄属性*/
OCIAttrSet((dvoid *)svchp, (ub4)OCI_HTYPE_SVCCTX, (dvoid *)srvhp, (ub4)0,
OCI_ATTR_SERVER, errhp);
OCIAttrSet(svchp, OCI_HTYPE_SVCCTX, (dvoid *)authp, 0, OCI_ATTR_SESSION, errhp);
/*创建并开始一个用户会话*/
OCISessionBegin(svchp, errhp, authp, OCI_CRED_RDBMS, OCI_DEFAULT);
OCIHandleAlloc(envhp, (dvoid **)&stmthp, OCI_HTYPE_STMT, 0, 0);
/*语句句柄*/

/*****************************************************************/
/*创建 person 表*/
/*****************************************************************/
```

```
static text* SQL_CREATE_TB = (text*)"create table person(personid number, name
varchar(256), email varchar(256))";
/*准备 SQL 语句*/
OCIStmtPrepare(stmthp, errhp, SQL_CREATE_TB, strlen((char *)SQL_CREATE_TB),
OCI_NTV_SYNTAX, OCI_DEFAULT);
/*执行 SQL 语句*/
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_DEFAULT);
/*提交到数据库*/
OCITransCommit(svchp, errhp, OCI_DEFAULT);

/*********************************************************************/
/*插入数据*/
/*********************************************************************/
memset(sql, 0, sizeof(sql));
strcpy(sql, "insert into person values(:personid,:name,:email)");
/*准备 SQL 语句*/
OCIStmtPrepare(stmthp, errhp, (text *)sql, strlen(sql),OCI_NTV_SYNTAX,
OCI_DEFAULT);
/*绑定输入列*/
OCIBindByName(stmthp, &bidhp[0], errhp, (const OraText*)":personid", 9,
&szpersonid, sizeof(szpersonid), SQLT_INT, NULL, NULL, NULL, 0, NULL, 0);
OCIBindByName(stmthp, &bidhp[2], errhp, (const OraText*)":name", 5, szname,
sizeof(szname), SQLT_STR, NULL, NULL, NULL, 0, NULL, 0);
OCIBindByName(stmthp, &bidhp[3], errhp, (const OraText*)":email", 6, szemail, sizeof
(szemail), SQLT_STR, NULL, NULL, NULL, 0, NULL, 0);
/*设置输入参数*/
szpersonid = 1;
memset(szname, 0, sizeof(szname));
strcpy((char*)szname, "obtest");
```

```
memset(szemail, 0, sizeof(szemail));
strcpy((char*)szemail, "t***@ob.com");
/*执行 SQL 语句*/
OCIStmtExecute(svchp, stmthp, errhp, (ub4)1, (ub4)0, (CONST OCISnapshot *)0,
(OCISnapshot *)0, (ub4)OCI_DEFAULT);
/*提交到数据库*/
OCITransCommit(svchp, errhp, OCI_DEFAULT);

/*****************************************************************/
/*查询 person 表*/
/*****************************************************************/
strcpy(sql, "select personid ,name,email from person;");
/*准备 SQL 语句*/
OCIStmtPrepare(stmthp, errhp, (text *)sql, strlen(sql), OCI_NTV_SYNTAX,
OCI_DEFAULT);
/*绑定输出列*/
OCIDefineByPos(stmthp, &defhp[0], errhp, 1, &szpersonid, sizeof(szpersonid),
SQLT_STR, &ind[0], 0, 0, OCI_DEFAULT);
OCIDefineByPos(stmthp, &defhp[1], errhp, 2, (ub1 *)szname, sizeof(szname),
SQLT_STR, &ind[1], 0, 0, OCI_DEFAULT);
OCIDefineByPos(stmthp, &defhp[2], errhp, 3, (ub1 *)szemail, sizeof(szemail),
SQLT_STR, &ind[2], 0, 0, OCI_DEFAULT);
/*执行 SQL 语句*/
OCIStmtExecute(svchp, stmthp, errhp, (ub4)0, 0, NULL, NULL,
OCI_DEFAULT);
printf("%-10s%-10s%-10s\n", "PERSONID", "NAME", "email");
while ((OCIStmtFetch(stmthp, errhp, 1, OCI_FETCH_NEXT, OCI_DEFAULT)) !=
OCI_NO_DATA)
{
```

```
printf("%-10d", szpersonid);
printf("%-10s", szname);
printf("%-10s\n", szemail);
break;
}
/*提交到数据库*/
OCITransCommit(svchp, errhp, OCI_DEFAULT);

/*****************************************************************/
/*删除 person 表*/
/*****************************************************************/
static text* SQL_DROP_TB = (text*)"drop table person";
/*准备 SQL 语句*/
OCIStmtPrepare(stmthp, errhp, (text*)SQL_DROP_TB, strlen((char *)SQL_DROP_TB),
OCI_NTV_SYNTAX, OCI_DEFAULT);
/*执行 SQL 语句*/
OCIStmtExecute(svchp, stmthp, errhp, 1, 0, 0, 0, OCI_COMMIT_ON_SUCCESS);
/*提交到数据库*/
OCITransCommit(svchp, errhp, OCI_DEFAULT);

//结束会话
OCISessionEnd(svchp, errhp, authp, (ub4)0);
//断开与数据库的连接
OCIServerDetach(srvhp, errhp, OCI_DEFAULT);
//释放OCI句柄
OCIHandleFree((dvoid *)dschp, OCI_HTYPE_DESCRIBE);
OCIHandleFree((dvoid *)stmthp, OCI_HTYPE_STMT);
OCIHandleFree((dvoid *)errhp, OCI_HTYPE_ERROR);
OCIHandleFree((dvoid *)authp, OCI_HTYPE_SESSION);
```

```
OCIHandleFree((dvoid *)svchp, OCI_HTYPE_SVCCTX);
OCIHandleFree((dvoid *)srvhp, OCI_HTYPE_SERVER);
return 0;
}
```

修改代码中的数据库连接参数。参考如下字段，对应的值，则取自步骤一获取的数据库连接串。

```
strcpy(strServerName, "172.30.xx.xx:2881");
strcpy(strUserName, "s**@oracle");
strcpy(strPassword, "******");
```

- **strServerName**：取自 `-h` 和 `-P` 参数，`IP:port`。OceanBase 数据库连接 IP，通常是一个 ODP 地址，以及访问所用的端口号。
- **strUserName**：取自 `-u` 参数，租户的连接用户名，格式为 **用户@租户#集群名称**，Oracle 模式的管理员用户名默认是 `sys`。直连数据库时不填集群名称，通过 ODP 连接时需要填写。
- **strPassword**：取自 `-p` 参数，用户密码。

6.1.4 步骤四：执行应用程序

代码编辑完成后，可以通过如下命令进行编译：

```
//编译
gcc test.c -I/u01/obclient/include /u01/obclient/lib/libobci.a -L/usr/local/lib64 -
lstdc++ -lpthread -ldl -lm -g -o test
```

编译完成后，运行得到如下结果，说明数据库连接成功且语句执行正常：

```
./test

PERSONID NAME email
1 obtest t***@ob.com
```

6.2 更多信息

关于 OBCI 的详细安装和使用信息，请参考官网文档《OceanBase C 语言调用接口》。

7 创建 Python 示例应用程序

本文介绍如何通过 Python 驱动连接和使用 OceanBase 数据库。不同版本的 Python 环境需要使用不同的驱动，Python 3.x 系列需要使用 PyMySQL 驱动，Python 2.x 系列需要使用 MySQL-python 驱动。

7.1 前提条件

确保本地已部署 Python 语言运行环境。

7.2 Python 3.x 创建应用程序

Python 3.x 创建应用程序时，需要 PyMySQL 驱动进行数据库连接及使用。

7.2.1 步骤一：获取数据库连接串

联系 OceanBase 数据库部署人员或者管理员获取相应的数据库连接串，例如：

```
obclient -h100.88.xx.xx -uroot@test -p****** -P2881 -Doceanbase
```

数据库连接串包含了访问数据库所需的参数信息，在创建应用程序前，可通过数据库连接串验证登录数据库，保证连接串参数信息正确。

参数说明：

- **-h**：OceanBase 数据库连接 IP，有时候是一个 ODP 地址。
- **-u**：租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直连数据库时不填写集群名称，通过 ODP 连接时需要填写。
- **-p**：用户密码。
- **-P**：OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **-D**：需要访问的数据库名称。

7.2.2 步骤二：安装 PyMySQL 驱动

PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库。PyMySQL 遵循 Python 数据库 API v2.0 规范，并包含了 pure-Python MySQL 客户端库。

有关 PyMySQL 的详细信息，请参考 PyMySQL 官网 和 相关 API 参考文档。

PyMySQL 有以下两种安装方式：

- 使用命令行安装

```
python3 -m pip install PyMySQL
```

- 源码编译安装

```
git clone https://github.com/PyMySQL/PyMySQL
cd PyMySQL/
python3 setup.py install
```

7.2.3 步骤三：编写应用程序

编辑运行示例 `test.py`，代码如下：

```
#!/usr/bin/python3

import pymysql

conn = pymysql.connect(host="localhost", port=2881,
user="root", passwd="", db="test")

cur = conn.cursor()

try:

#创建表 cities
sql = 'create table cities (id int, name varchar(24))'
cur.execute(sql)

#往 cities 表中插入两组数据
sql = "insert into cities values(1,'hangzhou'),(2,'shanghai')"
cur.execute(sql)

#查询 cities 表中的所有数据
sql = 'select * from cities'
cur.execute(sql)
```

```
ans = cur.fetchall()
print(ans)

#删除表 cities
sql = 'drop table cities'
cur.execute(sql)

finally:
cur.close()
conn.close()
```

修改代码中的数据库连接参数。参考如下字段，对应的值，则取自步骤一获取的数据库连接串。

- **user**：取自 `-u` 参数，租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直连数据库时不填写集群名称，通过 ODP 连接时需要填写。
- **password**：取自 `-p` 参数，用户密码。
- **host**：取自 `-h` 参数，OceanBase 数据库连接地址，有时候是 ODP 地址。
- **port**：取自 `-P` 参数，OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **db**：取自 `-D` 参数，需要访问的数据库名称。

7.2.4 步骤四：运行应用程序

代码编辑完成后，运行 `test.py` 。

```
python3 test.py
#返回以下结果，说明数据库连接成功，示例语句正确执行
((1, 'hangzhou'), (2, 'shanghai'))
```

7.3 Python 2.x 创建应用程序

Python 2.x 创建应用程序时，需要 MySQL-python 驱动进行数据库连接及使用。MySQL-python 是 Python2.X 版本中用于连接数据库的一个库。

7.3.5 步骤一：获取数据库连接串

联系 OceanBase 数据库部署人员或者管理员获取相应的数据库连接串，例如：

```
obclient -h100.88.xx.xx -uroot@test -p****** -P2881 -Doceanbase
```

数据库连接串包含了访问数据库所需的参数信息，在创建应用程序前，可通过数据库连接串验证登录数据库，保证连接串参数信息正确。

参数说明：

- **-h**：OceanBase 数据库连接 IP，有时候是一个 ODP 地址。
- **-u**：租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直连数据库时不填写集群名称，通过 ODP 连接时需要填写。
- **-p**：用户密码。
- **-P**：OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **-D**：需要访问的数据库名称。

7.3.6 步骤二：安装 MySQL-python 驱动

MySQL-python 是 Python 连接 MySQL 数据库的接口，它实现了 Python 数据库 API 规范 V2.0，基于 MySQL C API 建立。

有关 MySQL-python 的详细信息，您可参考 MySQL-python 官网 和 Github 文档.

MySQL-python 驱动可以通过 yum 安装，命令如下：

```
yum install MySQL-python
```

7.3.7 步骤三：编写应用程序

编辑运行示例 `test2.py`，代码如下：

```
#!/usr/bin/python2

import MySQLdb

conn= MySQLdb.connect(
host='127.0.0.1',
port = 2881,
```

```
user='root',
passwd='',
db ='test'
)


cur = conn.cursor()

try:

#创建表 cities
sql = 'create table cities (id int, name varchar(24))'
cur.execute(sql)

#往 cities 表中插入两组数据
sql = "insert into cities values(1,'hangzhou'),(2,'shanghai')"
cur.execute(sql)

#查询 cities 表中的所有数据
sql = 'select * from cities'
cur.execute(sql)
ans = cur.fetchall()
print(ans)

#删除表 cities
sql = 'drop table cities'
cur.execute(sql)

finally:
```

```
cur.close()
conn.close()
```

修改代码中的数据库连接参数。参考如下字段，对应的值，则取自步骤一获取的数据库连接串。

- **host**：取自 `-h` 参数，OceanBase 数据库连接地址，有时候是 ODP 地址。
- **user**：取自 `-u` 参数，租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直连数据库时不填写集群名称，通过 ODP 连接时需要填写。
- **passwd**：取自 `-p` 参数，用户密码。
- **port**：取自 `-P` 参数，OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **db**：取自 `-D` 参数，需要访问的数据库名称。

7.3.8 步骤四：运行应用程序

代码编辑完成后，运行 `test.py` 。

```
python test.py
#返回以下结果，说明数据库连接成功，示例语句正确执行
((1L, 'hangzhou'), (2L, 'shanghai'))
```

7.4 更多信息

创建 Python 3.x 示例应用程序在 OceanBase 数据库开源社区中也有相关的完整示例，详情请参考 Python3 示例应用程序。

8 创建 Java 示例应用程序

OceanBase 数据库支持通过 MySQL 官方 JDBC 驱动连接。本文介绍了如何通过 MySQL Connector/J 连接并使用 OceanBase 数据库。

8.1 前提条件

- 确保计算机上的 Java 环境为 Java JDK 8 及以上版本。
- 安装 MySQL Connector/J，并配置运行环境。

  推荐使用 MySQL Connector/J 5.1.47 版本。详细的下载及安装方法，请参考 Connector/J 下载、Connector/J 安装。

8.2 创建 Java 应用程序

8.2.1 步骤一：获取数据库连接串

联系 OceanBase 数据库部署人员或者管理员获取相应的数据库连接串，例如：

```
obclient -h100.88.xx.xx -uroot@test -p****** -P2883 -Doceanbase
```

数据库连接串包含了访问数据库所需的参数信息，在创建应用程序前，可通过数据库连接串验证登录数据库，保证连接串参数信息正确。

参数说明：

- **-h**：OceanBase 数据库连接 IP，有时候是一个 ODP 地址。
- **-u**：租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直接连接数据库时不填集群名称，通过 ODP 连接时需要填写。
- **-p**：用户密码。
- **-P**：OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **-D**：需要访问的数据库名称。

8.2.2 步骤二：编写应用程序

下文以 Linux 中通过 Java 驱动 `Connector/J 5.1.47` 连接数据库为例。

在正确安装 MySQL Connector/J 5.1.47 驱动并配置环境之后，可以通过以下 `Test.java` 文件的示例代码进行数据库连接及使用。

8.2.2.1 注意

如果是 MySQL Connector/J 8.x 版本，`Class.forName("com.mysql.jdbc.Driver")` 中的 `com.mysql.jdbc.Driver` 需要替换成 `com.mysql.cj.jdbc.Driver` 。

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class Test {
public static void main(String[] args) {
try {

Class.forName("com.mysql.jdbc.Driver").newInstance();

try{

Connection connection = DriverManager.getConnection("jdbc:mysql://127.0.0.1:
2881/test?user=r***&password=");
System.out.println(connection.getAutoCommit());
Statement sm = connection.createStatement();
//新建表 t_meta_form
sm.executeUpdate("CREATE TABLE t_meta_form (name varchar(36) , id int)");
//插入数据
sm.executeUpdate("insert into t_meta_form values ('an','1')");
//查询数据，并输出结果
ResultSet rs = sm.executeQuery("select * from t_meta_form");
while (rs.next()) {
String name = rs.getString("name");
String id = rs.getString("id");
```

```
System.out.println(name + ','+ id);
}
//删除表
sm.executeUpdate("drop table t_meta_form");

}catch(SQLException se){
System.out.println("error!");
se.printStackTrace() ;
}
}catch (Exception ex) {
ex.printStackTrace();
}
}
}
```

修改代码中的数据库连接参数。参考如下字段及拼接方法，对应的值，则取自步骤一获取的数据库连接串。

```
connection = DriverManager.getConnection("jdbc:mysql://{host}:{port}/{dbname}?
user={username}&password={******}")

//示例
jdbc:mysql://100.88.xx.xx:2881/test?user=r***&password=******`
```

- **host**：取自 `-h` 参数，OceanBase 数据库连接地址，有时候是 ODP 地址。
- **port**：取自 `-P` 参数，OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **dbname**：取自 `-D` 参数，需要访问的数据库名称。
- **username**：取自 `-u` 参数，租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直连数据库时不填写集群名称，通过 ODP 连接时需要填写。
- **password**：取自 `-p` 参数，用户密码。

8.2.3 步骤三：运行应用程序

代码编辑完成后，可以通过如下命令进行编译：

```
#配置临时环境配置，根据 mysql-connector-java-5.1.47.jar 实际安装路径填写
export CLASSPATH=/usr/share/java/mysql-connector-java-5.1.47.jar:$CLASSPATH
#编译
javac Test.java
```

编译完成后，通过如下命令运行 `Test`：

```
java Test
#输出以下结果说明数据库连接成功，示例语句正确执行
true
an,1
```

8.3 更多信息

创建 Java 示例应用程序在 OceanBase 数据库开源社区中也有相关的完整示例，详情请参考 Java 示例应用程序。

9 创建 C 示例应用程序

本文介绍了如何通过 MySQL Connector/C (libmysqlclient) 驱动连接并使用 OceanBase 数据库。

9.1 前提条件

在安装使用 MySQL Connector/C (libmysqlclient) 前请确保设置了基本的数据库开发环境，要求如下：

- GCC 版本为 3.4.6 及以上，推荐使用 4.8.5 版本。
- CMake 版本为 2.8.12 及以上。

9.2 创建 C 应用程序

9.2.1 步骤一：获取数据库连接串

联系 OceanBase 数据库部署人员或者管理员获取相应的数据库连接串，例如：

```
obclient -h100.88.xx.xx -uroot@test -p****** -P2881 -Doceanbase
```

数据库连接串包含了访问数据库所需的参数信息，在创建应用程序前，可通过数据库连接串验证登录数据库，保证连接串参数信息正确。

参数说明：

- **-h**：OceanBase 数据库连接 IP，有时候是一个 ODP 地址。
- **-u**：租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 sys，租户的默认管理员用户是 root。直接连接数据库时不填集群名称，通过 ODP 连接时需要填写。
- **-p**：用户密码。
- **-P**：OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **-D**：需要访问的数据库名称。

9.2.2 步骤二：安装 MySQL Connector/C 驱动

9.2.2.1 通过 yum 安装 mariadb client

1. 安装 mariadb client。

   ```
   sudo yum install mariadb-devel
   ```

9.2.3 步骤三：编写应用程序

应用程序通过 MySQL Connector/C 与数据库服务器 OBServer 节点交互的基本方式如下：

1. 调用 `mysql_library_init()` 初始化 MySQL 库。

   ```
   mysql_library_init(0, NULL, NULL);
   ```

2. 调用 `mysql_init()` 初始化一个连接句柄。

   ```
   MYSQL *mysql = mysql_init(NULL);
   ```

3. 调用 `mysql_real_connect()` 连接到 OBServer 节点。

   ```
   mysql_real_connect (mysql, host_name, user_name, password,
   db_name, port_num, socket_name, CLIENT_MULTI_STATEMENTS)
   ```

4. 调用 `mysql_real_query()` 或 `mysql_query()` 向 OBServer 节点发送 SQL 语句。

   ```
   mysql_query(mysql,"sql_statement");
   ```

5. 调用 `mysql_store_result()` 或 `mysql_use_result()` 处理其结果。

   ```
   result=mysql_store_result(mysql);
   ```

6. 调用 `mysql_free_result()` 释放内存。

   ```
   mysql_free_result(result);
   ```

7. 调用 `mysql_close()` 关闭与 OBServer 节点的连接。

   ```
   mysql_close(mysql);
   ```

8. 调用 `mysql_library_end()` 结束 MariaDB client 的使用。

   ```
   mysql_library_end();
   ```

9.2.3.2 示例代码

以 `mysql_test.c` 文件为例，代码如下：

```
#include "mysql.h"
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv)
{
mysql_library_init(0, NULL, NULL);
MYSQL *mysql = mysql_init(NULL);
char* host_name = "xxx.xxx.xxx.xxx";//set your mysql host
char* user_name = "******"; //set your user_name
char* password = "******"; //set your password
char* db_name = "test"; //set your databasename
int port_num = 2883; //set your mysql port
char* socket_name = NULL;
MYSQL_RES* result;
MYSQL_FIELD* fields;
MYSQL_ROW row;
int status = 0;
/* connect to server with the CLIENT_MULTI_STATEMENTS option */
if (mysql_real_connect (mysql, host_name, user_name, password,
db_name, port_num, socket_name, CLIENT_MULTI_STATEMENTS) == NULL)
{
printf("mysql_real_connect() failed\n");
mysql_close(mysql);
exit(1);
}

/* execute multiple statements */
status = mysql_query(mysql, "DROP TABLE IF EXISTS test_table;");
```

```
if (status)
{
printf("Could not execute statement(s)");
mysql_close(mysql);
exit(0);
}

status = mysql_query(mysql, "CREATE TABLE test_table(id INT,name varchar(24));");
status = mysql_query(mysql, "INSERT INTO test_table VALUES(10,'hangzhou'),
(20,'shanghai');");
status = mysql_query(mysql, "UPDATE test_table SET id=20 WHERE id=10;");
status = mysql_query(mysql, "SELECT * FROM test_table;");

/* did current statement return data? */
result = mysql_store_result(mysql);

if (result)
{
/* yes; process rows and free the result set */
//process_result_set(mysql, result);

int num_fields = mysql_num_fields(result);
int num_rows = mysql_num_rows(result);

printf("result: %d rows %d fields\n", num_rows, num_fields);
printf("--------------------\n");

fields = mysql_fetch_fields(result);
```

```
for (int i = 0; i < num_fields; ++i)
{
printf("%s\t", fields[i].name);
}

printf("\n----------------------\n");

while ((row = mysql_fetch_row(result)))
{
for (int i = 0; i < num_fields; ++i)
{
printf("%s\t", row[i] ? row[i] : "NULL");
}

printf("\n");
}

printf("----------------------\n");

mysql_free_result(result);
}
else /* no result set or error */
{
if (mysql_field_count(mysql) == 0)
{
printf("%lld rows affected\n",
mysql_affected_rows(mysql));
}
```

```
else /* some error occurred */
{
printf("Could not retrieve result set\n");
}
}

status = mysql_query(mysql, "DROP TABLE test_table;");

mysql_close(mysql);
return 0;
}
```

修改代码中的数据库连接参数。参考如下字段，对应的值，则取自步骤一获取的数据库连接串。

- **host_name**：取自 `-h` 参数，OceanBase 数据库连接地址，有时候是 ODP 地址。
- **user_name**：取自 `-u` 参数，租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 `root`。直连数据库时不填写集群名称，通过 ODP 连接时需要填写。
- **password**：取自 `-p` 参数，用户密码。
- **db_name**：取自 `-D` 参数，需要访问的数据库名称。
- **port_num**：取自 `-P` 参数，OceanBase 数据库连接端口，也是 ODP 的监听端口。

9.2.4 步骤四：运行应用程序

1. 代码编辑完成后，可以通过如下命令进行编译。

   ```
   g++ -I/usr/include/mysql/ -L/usr/lib64/mysql/ -lmysqlclient mysql_test.c -o mysql_test
   ```

   选项说明：

   - `-I` 选项：指定编译器的搜索路径，以便让编译器能够找到头文件。例如将 `mysql.h` 头文件所在的目录 `/usr/include/mysql` 添加到编译器的搜索路径中，使编译器能够找到

mysql.h 头文件。可以使用 find / -name mysql.h 2>/dev/null 这个命令查找 mysql.h 文件路径。

- -L 选项：指定动态链接库的搜索路径。
- -l 选项：指定需要链接的库文件。使用 -l 选项时，库文件名应该去掉 lib 前缀和 .so 后缀，例如上面的命令中库文件名为 libmysqlclient.so，但是使用 -l 选项时只需要指定 mysqlclient。

2. 指定运行路径。

```
export LD_LIBRARY_PATH=/usr/lib64/mysql
```

3. 通过如下命令运行应用程序。

```
./mysql_test
```

输出结果如下，输出如下结果，说明数据库连接成功，示例语句正确执行。

```
----------------------
id name
----------------------
20 hangzhou
20 shanghai
----------------------
```

10 创建 Go 示例应用程序

本文介绍 Go 应用程序示例如何通过驱动 Go-SQL-Driver/MySQL 连接并使用 OceanBase 数据库。

10.1 前提条件

确保本地已部署 Go 语言环境。

10.2 创建 Go 应用程序

10.2.1 步骤一：获取数据库连接串

联系 OceanBase 数据库部署人员或者管理员获取相应的数据库连接串，例如：

```
obclient -h100.88.xx.xx -uroot@test -p****** -P2881 -Doceanbase
```

数据库连接串包含了访问数据库所需的参数信息，在创建应用程序前，可通过数据库连接串验证登录数据库，保证连接串参数信息正确。

参数说明：

- **-h**：OceanBase 数据库连接 IP，有时候是一个 ODP 地址。
- **-u**：租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直接连接数据库时不填集群名称，通过 ODP 连接时需要填写。
- **-p**：用户密码。
- **-P**：OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **-D**：需要访问的数据库名称。

10.2.2 步骤二：安装 Go-SQL-Driver/MySQL

根据 Go 语言的不同版本，可以选择不同的安装方式。

10.2.2.1 通过 go get 安装（适用于Go V1.13 - V1.16）

安装命令如下：

```
go get -u github.com/go-sql-driver/mysql
```

关于 `Go-SQL-Driver/MySQL` 的详细信息，您可参考 Github。

10.2.2.2 通过 go install 安装

如果由于版本或网络的原因，无法通过 `go get` 命令安装时，可通过如下方法进行 `go-sql-driver/mysql` 安装。

1. 在 `go/src` 目录克隆 github 中的 `go-sql-driver/mysql` 仓库。

   ```
   cd /usr/local/go/src
   git clone https://github.com/go-sql-driver/mysql.git
   ```

   # 10.2.2.3 注意

   `/usr/local/go/src` 需要替换成 Go 实际安装目录操作。

2. 通过 `go install` 进行安装。

   ```
   go install mysql
   ```

   # 10.2.2.4 注意

   部分版本 `go install` 的默认执行目录可能不是 `/src`，可以通过 `go install` 执行后的报错判断实际目录。例如，报错 `cannot find package "mysql" in: /usr/local/go/src/vendor/mysql`，则应该将 mysql 文件夹放在 `/src/vendor` 目录下再执行安装命令。

10.2.3 步骤三：编写应用程序

将下文编写示例文件 `test.go`，代码如下：

```
package main

import (
"database/sql"
"fmt"
"log"

_ "mysql"
```

```
//填写 go-sql-driver/mysql 安装的准确路径。如果安装在 src 目录下，可以直接填
"mysql"。
)

type Str struct {
Name string
}

func main() {
select_all()
}

func select_all() {
conn := "root:@tcp(127.0.0.1:2881)/test"
db, err := sql.Open("mysql", conn)
if err != nil {
log.Fatal(err)
}

defer db.Close()

if err != nil {
log.Fatal(err)
}
fmt.Printf("success to connect OceanBase with go_mysql driver\n")
//创建表 t1
db.Query("create table t1(str varchar(256))")
//插入数据
db.Query("insert into t1 values ('Hello OceanBase')")
```

```
//查询数据
res, err := db.Query("SELECT * FROM t1")
//删除表 t1
db.Query("drop table t1")
if err != nil {
log.Fatal(err)
}

defer res.Close()

if err != nil {
log.Fatal(err)
}

for res.Next() {

var str Str
res.Scan(&str.Name)
fmt.Printf("%s\n", str.Name)
}
}
```

修改代码中的数据库连接参数。参考如下字段及拼接方法，对应的值，则取自步骤一获取的数据库连接串。

```
//格式
conn := "{username}:{password}@tcp({hostname}:{port})/{dbname}"
//示例
conn := "root:@tcp(127.0.0.1:2881)/test"
```

参数说明：

- **username**：取自 `-u` 参数，租户的连接用户名，格式为 **用户@租户#集群名称**，集群的默认租户是 'sys'，租户的默认管理员用户是 'root'。直连数据库时不填写集群名称，通过 ODP 连接时需要填写。
- **password**：取自 `-p` 参数，用户密码。
- **hostname**：取自 `-h` 参数，OceanBase 数据库连接地址，有时候是 ODP 地址。
- **port**：取自 `-P` 参数，OceanBase 数据库连接端口，也是 ODP 的监听端口。
- **dbname**：取自 `-D` 参数，需要访问的数据库名称。

10.2.4 步骤四：执行应用程序

代码编辑完成后，可以通过如下命令运行：

```
//配置临时环境变量，根据 Go 语言实际安装路径填写
export PATH=$PATH:/usr/local/go/bin

//通过 go run 直接运行 go 文件
go run test.go

//或者通过 go build 生成二进制文件后运行
go build test.go
./test
```

运行后返回如下内容，说明数据库连接成功，示例语句正确执行：

```
success to connect OceanBase with go_mysql driver
Hello OceanBase
```

10.3 更多信息

创建 Go 示例应用程序在 OceanBase 数据库开源社区中也有相关的完整示例，详情请参考 Go 示例应用程序。

11 在 OceanBase 数据库上进行 TPC-C 测试

TPC-C 是 TPC 组织（Transaction Processing Performance Council）推出的一系列性能测试标准中的一款，自推出以来，一直是数据库业界性能测试的重要参考，全世界各大数据库厂商都向 TPC 委员会提交了测试结果，以期望在 TPC-C 测试的排行榜上取得更好的成绩。OceanBase 数据库在 2019 年和 2020 年两次刷新了 TPC-C 的世界纪录，成为榜单上首个分布式关系型数据库。

在本篇文章中，通过在 OceanBase 数据库上运行 TPC-C 测试的方式，体验 OceanBase 数据库的 OLTP 能力。

11.1 关于 TPC-C

11.1.1 数据库模型

TPC-C Benchmark 规定了数据库的初始状态，其中 ITEM 表中包含固定的 10 万种商品，而仓库的数量可根据测试规模进行调整，假设 WAREHOUSE 表中有 W 条记录，那么：

- STOCK 表中应有 W×10 万条记录（每个仓库对应 10 万种商品的库存数据）。
- DISTRICT 表中应有 W×10 条记录（每个仓库为 10 个地区提供服务）。
- CUSTOMER 表中应有 W×10×3000 条记录（每个地区有 3000 个客户）。
- HISTORY 表中应有 W×10×3000 条记录（每个客户一条交易历史）。
- ORDER 表中应有 W×10×3000 条记录（每个地区 3000 个订单），并且最后生成的 900 个订单将被添加到 NEW-ORDER 表中，每个订单随机生成 5~15 条 ORDER-LINE 记录。

在测试过程中，每一个地区（ DISTRICT ）都有一个对应的终端（ Terminal ），模拟为用户提供服务。在每个终端的生命周期内，要循环往复地执行各类事务，当终端执行完一个事务的周期后，就进入下一个事务的周期。

客户下单后，包含若干个订单明细（ ORDER-LINE ）的订单（ ORDER ）被生成，并被加入新订单（ NEW-ORDER ）列表。

客户支付订单会产生交易历史（ HISTORY ）。每个订单（ ORDER ）平均包含 10 条订单项（ ORDER-LINE ），其中 1% 需要从远程仓库中获取，这些就是 TPC-C 模型中的 9 个数据表。

11.1.2 事务类型

该 Benchmark 包含 5 类事务：

- NewOrder：新订单请求从某一仓库中随机选取 5~15 件商品，创建新订单。其中 1% 的事务需要回滚（即 err）。一般地，新订单请求不可能超出全部事务请求的 45%。
- Payment：订单付款更新客户账户余额，反映其支付情况。在全部事务请求中占比 43%。
- OrderStatus：最近订单查询随机选择一个用户，查询其最近一条订单，显示该订单内的每个商品状态。在全部事务请求中占比 4%。
- Delivery：配送模拟批处理交易，更新该订单用户的余额，把发货单从 NewOrder 中删除。在全部事务请求中占比 4%。
- StockLevel：库存缺货状态分析，在全部事务请求中占比 4%。

11.2 环境准备

11.2.3 OceanBase 集群

根据部署的 OceanBase 集群类型的差异以不同的方式观察 OceanBase 数据库的表现，如果是按照快速开始章节中创建的单节点 OceanBase 集群，那么本文中可以看到 OceanBase 数据库在单机形态下的运行情况。如果希望体验 OceanBase 数据库分布式架构的 Scalable OLTP 能力，那么建议采用至少三节点的 OceanBase 集群进行测试。

本例中使用的租户模式为 MySQL 模式，租户名为 test，您可以创建自己的租户，具体步骤详见 体验多租户特性。

11.2.4 安装 BenchmarkSQL

TPC 组织为 TPC-C 定义了严格、详细的测试标准。一般情况下开发者如果想要模拟 TPC-C 的场景进行测试，可以使用目前常用的开源测试工具。例如本文中使用的 BenchmarkSQL，下载请访问 BenchmarkSQL 官方下载地址。

11.2.4.1 注意

测试环境需要有 Java 运行环境，且版本不低于 V1.8.0。

11.2.5 适配 Benchmark SQL5

由于 Benchmark SQL5 不支持 OceanBase 数据库的 TPC-C 测试，本节将详细介绍如何通过修改 BenchMarkSQL5 部分源码支持 OceanBase 数据库。

1. 修改 benchmarksql-5.0/src/client/jTPCC.java 文件，增加 OceanBase 数据库相关内容。

```
if (iDB.equals("firebird"))
dbType = DB_FIREBIRD;
else if (iDB.equals("oracle"))
dbType = DB_ORACLE;
else if (iDB.equals("postgres"))
dbType = DB_POSTGRES;
else if (iDB.equals("oceanbase"))
dbType = DB_OCEANBASE;
else
{
log.error("unknown database type '" + iDB + "'");
return;
}
```

修改 `benchmarksql-5.0/src/client/jTPCCConfig.java` 文件，增加 OceanBase 数据库类型。

```
public final static int
DB_UNKNOWN = 0,
DB_FIREBIRD = 1,
DB_ORACLE = 2,
DB_POSTGRES = 3,
DB_OCEANBASE = 4;
```

2. 修改 `benchmarksql-5.0/src/client/jTPCCConnection.java` 文件，在 SQL 子查询增加 `AS L` 别名。

```
default:
stmtStockLevelSelectLow = dbConn.prepareStatement(
"SELECT count(*) AS low_stock FROM (" +
" SELECT s_w_id, s_i_id, s_quantity " +
```

```
" FROM bmsql_stock " +
" WHERE s_w_id = ? AND s_quantity < ? AND s_i_id IN (" +
" SELECT ol_i_id " +
" FROM bmsql_district " +
" JOIN bmsql_order_line ON ol_w_id = d_w_id " +
" AND ol_d_id = d_id " +
" AND ol_o_id >= d_next_o_id - 20 " +
" AND ol_o_id < d_next_o_id " +
" WHERE d_w_id = ? AND d_id = ? " +
" ) " +
" )AS L");
break;
```

3. 重新编译修改后的源码。

```
[oceanbase@testdrier test]# cd benchmarksql-5.0
[oceanbase@testdrier benchmarksql-5.0]# ant
```

4. 修改文件：`benchmarksql-5.0/run/funcs.sh`，添加 OceanBase 数据库类型。

```
function setCP()
{
case "$(getProp db)" in
firebird)
cp="../lib/firebird/*:../lib/*"
;;
oracle)
cp="../lib/oracle/*"
if [ ! -z "${ORACLE_HOME}" -a -d ${ORACLE_HOME}/lib ] ; then
cp="${cp}:${ORACLE_HOME}/lib/*"
fi
cp="${cp}:../lib/*"
```

```
;;
postgres)
cp="../lib/postgres/*:../lib/*"
;;
oceanbase)
cp="../lib/oceanbase/*:../lib/*"
;;
esac
myCP=".:${cp}:../dist/*"
export myCP
}

...省略

case "$(getProp db)" in
firebird|oracle|postgres|oceanbase)
;;
"") echo "ERROR: missing db= config option in ${PROPS}" >&2
exit 1
;;
*) echo "ERROR: unsupported database type 'db=$(getProp db)' in ${PROPS}" >&2
exit 1
;;
esac
```

5. 修改 benchmarksql-5.0/run/runDatabaseBuild.sh 。

```
AFTER_LOAD="indexCreates foreignKeys extraHistID buildFinish"
# 修改为:
AFTER_LOAD="indexCreates buildFinish"
```

11.2.6 修改配置文件

配置文件 `props.ob` 在 BenchmarkSQL 的 `run/` 目录下：

```
db=oceanbase
driver=com.mysql.jdbc.Driver
conn=jdbc:mysql://127.0.0.1:2881/tpccdb?
useUnicode=true&characterEncoding=utf-
8&rewriteBatchedStatements=true&allowMultiQueries=true
user=root@test
password=****

warehouses=10
loadWorkers=2
//fileLocation=/data/temp/

terminals=10
//To run specified transactions per terminal- runMins must equal zero
runTxnsPerTerminal=0
//To run for specified minutes- runTxnsPerTerminal must equal zero
runMins=10
//Number of total transactions per minute
limitTxnsPerMin=0

//Set to true to run in 4.x compatible mode. Set to false to use the
//entire configured database evenly.
terminalWarehouseFixed=true

//The following five values must add up to 100
newOrderWeight=45
```

```
paymentWeight=43
orderStatusWeight=4
deliveryWeight=4
stockLevelWeight=4

// Directory name to create for collecting detailed result data.
// Comment this out to suppress.
resultDirectory=my_result_%tY-%tm-%td_%tH%tM%tS
osCollectorScript=./misc/os_collector_linux.py
osCollectorInterval=1
//osCollectorSSHAddr=user@dbhost
//osCollectorDevices=net_eth0 blk_sda
```

11.2.6.2 说明

- db：指定数据库类型。此处保持和模板一致即可。
- driver：驱动程序文件，推荐使用 MySQL 的 JDBC 驱动：mysql-connector-java-5.1.47，驱动下载地址。
- conn：此处的 IP 建议填写 OceanBase Server 的 IP，端口为 OceanBase Server 部署端口，其他部分保持和模板一致。
- user & password：根据环境中使用的用户名、租户名以及密码即可。如果环境中有多个 OceanBase 集群，则 user 的格式建议为 {user_name}@{tenant_name}#{cluster_name}。
- warehouses：指定仓库数，仓库数决定性能测试的成绩。如果希望针对多节点的 OceanBase 集群进行测试，建议选择 1000 仓以上。如果机器配置有限，可以选择 100 仓进行测试。
- loadWorkers：指定仓库数据加载时的并发。如果机器配置较高，该值可以设置大一些，例如 100。如果机器配置有限，该值需要设置小一些，如 10 并发。过高的并发可能会导致内存消耗太快，出现报错，导致数据加载需要重新进行。

- terminals：指定性能压测时的并发数。建议并发数不要高于仓库数 * 10。否则，会有不必要的锁等待。在生产环境中，建议将此参数设置为最多 1000。在测试环境中，建议从 100 开始。
- runMins：指定性能测试持续的时间。时间越久，越能考验数据库的性能和稳定性。建议不要少于 10 分钟，生产环境中机器建议不少于 1 小时。

11.3 数据准备

11.3.7 创建 tpccdb 数据库

在测试租户 test 中，创建本次测试的数据库 tpccdb：

```
CREATE DATABASE tpccdb;
```

11.3.8 创建表

建表脚本通常放在 benchmarskSQL 的 run/sql.common 下或者其他指定目录下。建表脚本如下，采用分区表方式创建，大部分表按照仓库 ID 做 HASH 分区。分区数取决于要测试的数据规模和机器数。 如果集群只有 1 台或 3 台机器，分区数设置 9 个即可。如果是 5000 仓以上，或者集群中节点数较多，则分区数可以调整到 99。

```
[root@obce-0000 run]# cat sql.common/tableCreates_parts.sql
CREATE TABLE bmsql_config (
cfg_name varchar(30) primary key,
cfg_value varchar(50)
);

-- drop tablegroup tpcc_group;
CREATE TABLEGROUP tpcc_group binding true partition by hash partitions 9;

CREATE TABLE bmsql_warehouse (
w_id integer not null,
w_ytd decimal(12,2),
w_tax decimal(4,4),
```

```
w_name varchar(10),
w_street_1 varchar(20),
w_street_2 varchar(20),
w_city varchar(20),
w_state char(2),
w_zip char(9),
primary key(w_id)
)tablegroup='tpcc_group' partition by hash(w_id) partitions 9;

CREATE TABLE bmsql_district (
d_w_id integer not null,
d_id integer not null,
d_ytd decimal(12,2),
d_tax decimal(4,4),
d_next_o_id integer,
d_name varchar(10),
d_street_1 varchar(20),
d_street_2 varchar(20),
d_city varchar(20),
d_state char(2),
d_zip char(9),
PRIMARY KEY (d_w_id, d_id)
)tablegroup='tpcc_group' partition by hash(d_w_id) partitions 9;

CREATE TABLE bmsql_customer (
c_w_id integer not null,
c_d_id integer not null,
c_id integer not null,
c_discount decimal(4,4),
```

```
c_credit char(2),
c_last varchar(16),
c_first varchar(16),
c_credit_lim decimal(12,2),
c_balance decimal(12,2),
c_ytd_payment decimal(12,2),
c_payment_cnt integer,
c_delivery_cnt integer,
c_street_1 varchar(20),
c_street_2 varchar(20),
c_city varchar(20),
c_state char(2),
c_zip char(9),
c_phone char(16),
c_since timestamp,
c_middle char(2),
c_data varchar(500),
PRIMARY KEY (c_w_id, c_d_id, c_id)
)tablegroup='tpcc_group' partition by hash(c_w_id) partitions 9;



CREATE TABLE bmsql_history (
hist_id integer,
h_c_id integer,
h_c_d_id integer,
h_c_w_id integer,
h_d_id integer,
h_w_id integer,
h_date timestamp,
```

```
h_amount decimal(6,2),
h_data varchar(24)
)tablegroup='tpcc_group' partition by hash(h_w_id) partitions 9;

CREATE TABLE bmsql_new_order (
no_w_id integer not null ,
no_d_id integer not null,
no_o_id integer not null,
PRIMARY KEY (no_w_id, no_d_id, no_o_id)
)tablegroup='tpcc_group' partition by hash(no_w_id) partitions 9;

CREATE TABLE bmsql_oorder (
o_w_id integer not null,
o_d_id integer not null,
o_id integer not null,
o_c_id integer,
o_carrier_id integer,
o_ol_cnt integer,
o_all_local integer,
o_entry_d timestamp,
PRIMARY KEY (o_w_id, o_d_id, o_id)
)tablegroup='tpcc_group' partition by hash(o_w_id) partitions 9;

CREATE TABLE bmsql_order_line (
ol_w_id integer not null,
ol_d_id integer not null,
ol_o_id integer not null,
ol_number integer not null,
ol_i_id integer not null,
```

```
ol_delivery_d timestamp,
ol_amount decimal(6,2),
ol_supply_w_id integer,
ol_quantity integer,
ol_dist_info char(24),
PRIMARY KEY (ol_w_id, ol_d_id, ol_o_id, ol_number)
)tablegroup='tpcc_group' partition by hash(ol_w_id) partitions 9;

CREATE TABLE bmsql_item (
i_id integer not null,
i_name varchar(24),
i_price decimal(5,2),
i_data varchar(50),
i_im_id integer,
PRIMARY KEY (i_id)
);

CREATE TABLE bmsql_stock (
s_w_id integer not null,
s_i_id integer not null,
s_quantity integer,
s_ytd integer,
s_order_cnt integer,
s_remote_cnt integer,
s_data varchar(50),
s_dist_01 char(24),
s_dist_02 char(24),
s_dist_03 char(24),
s_dist_04 char(24),
```

```
s_dist_05 char(24),
s_dist_06 char(24),
s_dist_07 char(24),
s_dist_08 char(24),
s_dist_09 char(24),
s_dist_10 char(24),
PRIMARY KEY (s_w_id, s_i_id)
)tablegroup='tpcc_group' use_bloom_filter=true partition by hash(s_w_id)
partitions 9;
```

运行如下命令建表。

```
./runSQL.sh props.ob sql.common/tableCreates_parts.sql
```

11.3.9 加载数据

加载数据即数据初始化，加载数据的速度，取决于机器配置，配置越高的机器，加载数据的速度越快。

```
./runLoader.sh props.ob
```

加载数据的 `INSERT` SQL 使用了 Batch Insert 特性，这点是在 `props.ob` 里的 JDBC URL 里指定的。开启该特性的写入性能会有明显提升。

11.3.10 创建索引

当数据初始化完成后，登录到集群的 `test` 租户，在 `tpccdb` 中补充创建如下两个索引。

```
[root@obce-0000 run]# cat sql.common/indexCreates.sql
CREATE INDEX bmsql_customer_idx1
on bmsql_customer (c_w_id, c_d_id, c_last, c_first) local;

CREATE INDEX bmsql_oorder_idx1
on bmsql_oorder (o_w_id, o_d_id, o_carrier_id, o_id) local;
```

11.4 开始测试

在开始性能测试之前，建议您先登录到对应租户做一次集群合并（major freeze），获得更好的测试结果。您可以通过如下的方式手动触发合并，这个过程并不是必须的。

```
obclient[oceanbase]> ALTER SYSTEM MAJOR FREEZE;
```

当看到如下查询返回 IDLE 时，表示合并完成。

```
MySQL [oceanbase]> SELECT * FROM oceanbase.
CDB_OB_ZONE_MAJOR_COMPACTION;
+-----------+-------+--------------------+--------------------
+---------------------------+---------------------------+--------+
| TENANT_ID | ZONE | BROADCAST_SCN | LAST_SCN | LAST_FINISH_TIME | START_TIME |
STATUS |
+-----------+-------+--------------------+--------------------
+---------------------------+---------------------------+--------+
| 1 | zone1 | 1664503499339325817 | 1664503499339325817 | 2022-09-30 10:05:
19.442115 | 2022-09-30 10:04:59.369976 | IDLE |
| 1002 | zone1 | 1 | 1 | 1970-01-01 08:00:00.000000 | 1970-01-01 08:00:00.000000 |
IDLE |
+-----------+-------+--------------------+--------------------
+---------------------------+---------------------------+--------+
2 rows in set
```

合并完成后，开始执行测试：

```
./runBenchmark.sh props.ob
```

TPC-C 用 tpmC 值（Transactions per Minute）来衡量系统最大有效吞吐量。其中 Transactions 以 NewOrder Transaction 为准，即最终衡量单位为每分钟处理的订单数。

11.5 体验 OceanBase 数据库 Scalable OLTP

上面的测试中，不管是单节点集群测试，还是多节点集群，默认情况下参与事务处理的只有一个副本 Zone，及租户 Leader 所在的副本。这是因为默认设置下，租户的 Leader 是按照 Zone 维度以一定优先级分布的。

OceanBase 作为分布式数据库，用户可以选择将一个租户的所有数据分区的多个 Leader 打散 Shuffle 到多个副本上，实现计算处理能力的多机线性扩展。

如果您的集群环境是三节点或更多节点的配置，您只需以管理员身份执行以下命令，然后再次启动测试，即可体验分布式集群在扩展模式下的处理能力。

```
obclient> ALTER TENANT test SET PRIMARY_ZONE='zone1';
```

12 体验 OceanBase 数据库热点行更新能力

随着在线交易、电商行业的发展，业务系统的热点并发压力逐渐成为一种挑战。热点账户短时间内余额大量更新，或者热门商品在营销活动中限时抢购，都是这种场景的直接体现。热点更新的本质是短时间内对数据库中的同一行数据的某些字段值进行高并发的修改（余额，库存等），这其中的瓶颈主要在于关系型数据库为了保持事务一致性，对数据行的更新都需要经过 “加锁->更新->写日志提交->释放锁” 的过程，而这个过程实质上是串行的。所以，提高热点行更新能力的关键在于如何尽可能缩短持有锁的时间。

虽然学术界很早就提出了 “提前解行锁（Early Lock Release）” 的方案（即 ELR），但是因为 ELR 的异常处理场景非常复杂，业界很少有成熟的工业实现。OceanBase 数据库在这个问题上通过持续的探索，提出了一种基于分布式架构的实现方式，提升类似业务场景中单行并发更新的能力，作为 OceanBase 数据库 “可扩展的 OLTP” 中的关键能力之一。

本篇文章中，我们将通过构造一个多并发单行更新的场景，介绍 OceanBase 数据库 ELR 特性的使用方法和效果对比。因为是在多并发压力场景下验证，我们建议至少使用和本例中相同的节点规格进行体验和验证，达到更好的效果。关于 OceanBase 数据库 ELR 的设计和原理实现，因其较为复杂，本文暂不做详细讨论。

本例中我们使用一台 16C-128GB 配置的节点，下面我们来分步骤体验 OceanBase 数据库 ELR 特性。

12.1 步骤一：创建测试表，插入测试数据

首先我们在测试库中创建一张表，并插入测试数据。

```
CREATE TABLE `sbtest1` (
`id` int(11) NOT NULL AUTO_INCREMENT,
`k` int(11) NOT NULL DEFAULT '0',
`c` char(120) NOT NULL DEFAULT '',
`pad` char(60) NOT NULL DEFAULT '',
PRIMARY KEY (`id`)
);

INSERT INTO sbtest1 VALUES(1,0,'aa','aa');
```

本例中我们通过类似 `UPDATE sbtest1 SET k=k+1 WHERE id=1` 的语句，以主键查询方式针对 `k` 列进行并发更新，您也可以插入更多数据进行测试，但因为是针对单行并发的压测，对整体结果基本没有影响。

12.2 步骤二：构造并发更新场景

本例中，我们使用 Python 多线程的方式来模拟并发更新，同时启动 50 个 Thread，每个 Thread 并发的对 id=1 的行数据将 k 字段值 +1。您在自己搭建的环境中也可以直接使用下面的脚本 ob_elr.py 进行测试，只需要将脚本中的数据库连接信息修改一下即可使用。

```
#!/usr/bin/env python3


from concurrent.futures import ThreadPoolExecutor
import pymysql
import time
import threading


# database connection info
config = {
'user': 'root@test',
'password': '****',
'host': 'xxx.xxx.xxx.xxx',
'port': 2881,
'database': 'test'
}


# parallel thread and updates in each thread
parallel = 50
batch_num = 2000


# update query
def update_elr():
update_hot_row = ("update sbtest1 set k=k+1 where id=1")
cnx = pymysql.connect(**config)
cursor = cnx.cursor()
```

```
for i in range(0,batch_num):
cursor.execute(update_hot_row)
cursor.close()
cnx.close()

start=time.time()

with ThreadPoolExecutor(max_workers=parallel) as pool:
for i in range(parallel):
pool.submit(update_elr)

end = time.time()
elapse_time = round((end-start),2)

print('Parallel Degree:',parallel)
print('Total Updates:',parallel*batch_num)
print('Elapse Time:',elapse_time,'s')
print('TPS on Hot Row:' ,round(parallel*batch_num/elapse_time,2),'/s')
```

12.3 步骤三：默认配置下执行测试

作为对比参照，我们先在默认不开启 ELR 的情况下测试，在测试机器上直接执行 ob_elr.py 脚本。

本例中我们采用 50 并发数，总计更新 100000 次。

```
./ob_elr.py
```

执行完成，测试脚本输出执行时间和 TPS：

```
[root@obce00 ~]# ./ob_elr.py
Parallel Degree: 50
```

```
Total Updates: 100000
Elapse Time: 54.5 s
TPS on Hot Row: 1834.86 /s
```

测试结果如下：

在不开启 ELR 的默认配置下，本例测试环境中，单行并发更新 TPS 为1834.86/s。

12.4 步骤四：打开 OceanBase 数据库 ELR 配置

接下来我们开启 OceanBase 数据库的热点行功能，首先需要使用 `root` 用户登录集群的 `sys` 租户。

```
[root@obce00 ~]# obclient -h127.0.0.1 -P2881 -uroot@sys -Doceanbase -A -p -c
```

然后进行如下两个参数的设置。其中 `enable_early_lock_release` 参数的生效范围，既可以指定具体租户，也可以指定全部租户，即 `tenant=all` 。

```
ALTER SYSTEM SET _max_elr_dependent_trx_count = 1000;
ALTER SYSTEM SET enable_early_lock_release=true tenant= test;
```

12.5 步骤五：开启 OceanBase 数据库 ELR 进行测试

开启热点行功能后，我们再次执行测试。在执行前我们先查看表 `sbtest` 中 `id=1` 的记录，`k` 字段值为 100000，这是因为刚刚执行了一轮默认配置下的 100000 次的更新。

```
SELECT * FROM sbtest1 WHERE id=1;
+----+--------+----+-----+
| id | k | c | pad |
+----+--------+----+-----+
| 1 | 100000 | aa | aa |
+----+--------+----+-----+
1 row in set
```

接下来我们在测试机器上再次执行 `ob_elr.py` 。仍然采用 50 并发数，总计更新 100000 次。

```
./ob_elr.py
```

执行完成，测试脚本会输出执行时间和 TPS：

```
[root@obce00 ~]# ./ob_elr.py
Parallel Degree: 50
Total Updates: 100000
Elapse Time: 12.16 s
TPS on Hot Row: 8223.68 /s
```

测试结果如下：

OceanBase 数据库在开启 ELR 提前解行锁能力后，单行更新的 TPS 达到 8223.68/s，相比默认配置下提升 4.5 倍左右。

同时我们可以看到表 `sbtest1` 的 `k` 值为 200000，即本次也更新了 100000 次。

```
SELECT * FROM sbtest1 WHERE id=1;
+----+--------+----+-----+
| id | k | c | pad |
+----+--------+----+-----+
| 1 | 200000 | aa | aa |
+----+--------+----+-----+
1 row in set
```

本例中仅介绍了单行并发更新的场景，OceanBase 数据库的 ELR 能力还支持多语句事务的并发更新，根据语句数量和场景差异，同样可以获得显著的性能提升。

此外 OceanBase 数据库的 ELR 还可以应用在多地部署高网络延迟的场景中，例如单个事务在默认场景下 30 ms，并发下开启 ELR 可以获得近百倍的 TPS 吞吐量提升。

由于 OceanBase 数据库的日志协议基于 Multi-Paxos 构建，并且优化了 2PC 提交过程，OceanBase 数据库在开启 ELR 后，如果发生节点宕机重启、Leader 切换，仍然可以保证事务的一致性。您可以尝试构造这些实验进行体验。

13 体验 Operational OLAP

OceanBase 数据库可以处理混合负载类型的场景。由于 OceanBase 数据库是基于对等节点的分布式架构，使得它既可以承载高并发和可扩展的 OLTP 任务，还可以在同一套数据引擎中基于 MPP 架构进行 OLAP 的并行计算，无需维护两套数据。

在 OceanBase 数据库中，您不但可以在大量在线业务数据上直接进行并行分析，还可以通过 PDML 能力（Parallel DML）将批量写入数据的大事务以并发的方式快速安全的执行。并且，这一切都是在严格保证事务一致性的前提下做到的。

下面让我们手动进行 TPC-H 测试，演示 OceanBase 数据库在 Operational OLAP 场景的特点和用法。TPC-H 是一个业界常用的基于决策支持业务的 Benchmark，通过一系列在大量数据集上面执行的复杂查询请求，检验数据库系统的分析以及决策支持能力。详细信息，可参见 TPC 组织官方网站。

13.0.0.1 说明

2021 年 5 月 20 日，OceanBase 数据库以 1526 万 QphH 的成绩刷新了 TPC-H 世界纪录，并且是唯一一个同时刷新了TPC-C 以及 TPC-H 纪录的数据库，证明了其能够同时处理在线交易和实时分析两类业务场景能力。详细信息请参见 TPC-H Result。

13.1 手动进行 TPC-H 测试

以下内容为基于 TPC 官方 TPC-H 工具进行手动 Step-by-Step 进行 TPC-H 测试。手动测试可以帮助更好的学习和了解 OceanBase 数据库，尤其是一些参数的设置。

13.1.1 步骤一：创建测试租户

13.1.1.1 说明

本次测试的 OceanBase 集群环境部署模式为 1:1:1。

在系统租户（ `sys` 租户）下执行的命令创建测试租户：

1. 创建资源单元 `mysql_box` 。

```
CREATE RESOURCE UNIT mysql_box
MAX_CPU 28,
MEMORY_SIZE '200G',
MIN_IOPS 200000,
```

```
MAX_IOPS 12800000,
LOG_DISK_SIZE '300G';
```

2. 创建资源池 `mysql_pool` 。

```
CREATE RESOURCE POOL mysql_pool
UNIT = 'mysql_box',
UNIT_NUM = 1,
ZONE_LIST = ('z1','z2','z3');
```

3. 创建 MySQL 模式租户 `mysql_tenant` 。

```
CREATE TENANT mysql_tenant
RESOURCE_POOL_LIST = ('mysql_pool'),
PRIMARY_ZONE = RANDOM,
LOCALITY = 'F@z1,F@z2,F@z3'
SET VARIABLES ob_compatibility_mode='mysql', ob_tcp_invited_nodes='%',
secure_file_priv = "/";
```

13.1.2 步骤二：进行环境调优

1. OceanBase 数据库调优。

   请在系统租户（ `sys` 租户）下执行以下语句配置相关参数。

```
ALTER SYSTEM FLUSH PLAN CACHE GLOBAL;
ALTER SYSTEM SET enable_sql_audit = false;
SELECT sleep(5);
ALTER SYSTEM SET enable_perf_event = false;
ALTER SYSTEM SET syslog_level = 'PERF';
ALTER SYSTEM SET enable_record_trace_log = false;
ALTER SYSTEM SET data_storage_warning_tolerance_time = '300s';
ALTER SYSTEM SET _data_storage_io_timeout = '600s';
ALTER SYSTEM SET trace_log_slow_query_watermark = '7d';
```

```
ALTER SYSTEM SET large_query_threshold = '0ms';
ALTER SYSTEM SET enable_syslog_recycle = 1;
ALTER SYSTEM SET max_syslog_file_count = 300;
```

2. 租户调优。

   请在测试租户（用户租户）下执行以下语句配置相关参数。

```
SET GLOBAL NLS_DATE_FORMAT = 'YYYY-MM-DD HH24:MI:SS';
SET GLOBAL NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF';
SET GLOBAL NLS_TIMESTAMP_TZ_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF TZR
TZD';

SET GLOBAL ob_query_timeout = 10800000000;
SET GLOBAL ob_trx_timeout = 10000000000;

SET GLOBAL ob_sql_work_area_percentage = 50;
ALTER SYSTEM SET default_table_store_format = 'column' ;
ALTER SYSTEM SET ob_enable_batched_multi_statement = 'true';
ALTER SYSTEM SET _io_read_batch_size = '128k';
ALTER SYSTEM SET _io_read_redundant_limit_percentage = 50;
SET GLOBAL parallel_degree_policy = AUTO;
SET GLOBAL parallel_servers_target = 10000;

SET GLOBAL collation_connection = utf8mb4_bin;
SET GLOBAL collation_database = utf8mb4_bin;
SET GLOBAL collation_server = utf8mb4_bin;

SET GLOBAL autocommit = 1;

ALTER SYSTEM SET ob_enable_batched_multi_statement = 'true';
```

13.1.3 步骤三：安装 TPC-H Tool

1. 下载 TPC-H Tool。详细信息请参考 TPC-H Tool 下载页面。
2. 下载完成后解压文件，进入 TPC-H 解压后的目录。

```
[wieck@localhost ~] $ unzip 7e965ead-8844-4efa-a275-34e35f8ab89b-tpc-h-tool.zip
[wieck@localhost ~] $ cd TPC-H_Tools_v3.0.0
```

3. 复制 `Makefile.suite` 。

```
[wieck@localhost TPC-H_Tools_v3.0.0] $ cd dbgen/
[wieck@localhost dbgen] $ cp Makefile.suite Makefile
```

4. 修改 `Makefile` 文件中的 `CC` 、 `DATABASE` 、 `MACHINE` 、 `WORKLOAD` 等参数定义。

```
[wieck@localhost dbgen] $ vim Makefile
```

```
CC = gcc
# Current values for DATABASE are: INFORMIX, DB2, TDAT (Teradata)
# SQLSERVER, SYBASE, ORACLE, VECTORWISE
# Current values for MACHINE are: ATT, DOS, HP, IBM, ICL, MVS,
# SGI, SUN, U2200, VMS, LINUX, WIN32
# Current values for WORKLOAD are: TPCH
DATABASE= MYSQL
MACHINE = LINUX
WORKLOAD = TPCH
```

5. 修改 `tpcd.h` 文件，并添加新的宏定义。

```
[wieck@localhost dbgen] $ vim tpcd.h
```

```
#ifdef MYSQL
#define GEN_QUERY_PLAN ""
#define START_TRAN "START TRANSACTION"
```

```
#define END_TRAN "COMMIT"
#define SET_OUTPUT ""
#define SET_ROWCOUNT "limit %d;\n"
#define SET_DBASE "use %s;\n"
#endif
```

6.编译文件。

```
make
```

返回结果如下：

```
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o build.o build.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o driver.o driver.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o bm_utils.o bm_utils.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o rnd.o rnd.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o print.o print.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o load_stub.o load_stub.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o bcd2.o bcd2.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o speed_seed.o speed_seed.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o text.o text.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o permute.o permute.c
```

```
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o rng64.o rng64.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -O -o dbgen build.o driver.o bm_utils.o rnd.o print.o
load_stub.o bcd2.o speed_seed.o text.o permute.o rng64.o -lm
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o qgen.o qgen.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -c -o varsub.o varsub.c
gcc -g -DDBNAME=\"dss\" -DLINUX -DMYSQL -DTPCH -DRNG_TEST -
D_FILE_OFFSET_BITS=64 -O -o qgen build.o bm_utils.o qgen.o rnd.o varsub.o text.o
bcd2.o permute.o speed_seed.o rng64.o -lm
```

会生成后续生成数据的 dbgen 文件和生成 sql 的 qgen、dists.dss 文件。

13.1.4 步骤四：生成数据

您可以根据实际环境生成 TCP-H 10G、100G 或者 1T 数据。本文以生成 100G 数据为例。

```
./dbgen -s 100
mkdir tpch100
mv *.tbl tpch100
```

多线程生成 1T 数据，Oceanbase 支持旁路导入数据，可以同时将多个文件的数据导入表中：

```
#!/bin/bash


SCALE_FACTOR=1000
CHUNK_COUNT=20
for ((i=1; i<=CHUNK_COUNT; i++))
do
CMD="./dbgen -s ${SCALE_FACTOR} -C ${CHUNK_COUNT} -S ${i} -vf"
$CMD &
```

```
done
wait
echo "All data generation tasks completed."
```

13.1.5 步骤五：生成查询 SQL

13.1.5.2 说明

您可参考本节中的下述步骤生成查询 SQL 后进行调整，也可直接使用 GitHub 中给出的查询 SQL。若您选择使用 GitHub 中的查询 SQL，您需将 SQL 语句中的 `cpu_num` 修改为实际并发数。

使用 tpch 自带工具生成，步骤如下：

1. 将 `dbgen/qgen` 和 `dbgen/dists.dss` 拷贝到 `mysql_sql` 文件夹下。
2. 在 `mysql_sql` 文件夹下创建 `gen.sh` 脚本生成查询 SQL。

   ```
   vim gen.sh
   ```

   ```
   #!/usr/bin/bash
   for i in {1..22}
   do
   ./qgen -d $i -s 100 > db"$i".sql
   done
   ```

3. 按照实际并发数修改查询 SQL。

   您可在 `sys` 租户下使用如下命令查看租户的可用 CPU 总数。

   ```
   select sum(max_cpu) from DBA_OB_UNITS;
   ```

   以 `Q1` 为例，修改后的 SQL 语句如下：

   ```
   SELECT /*+ parallel(96) */ ---增加 parallel 并发执行
   l_returnflag,
   l_linestatus,
   sum(l_quantity) as sum_qty,
   ```

```
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
FROM
lineitem
WHERE
l_shipdate <= date '1998-12-01' - interval '90' day
GROUP BY
l_returnflag,
l_linestatus
ORDER BY
l_returnflag,
l_linestatus;
```

13.1.6 步骤六：新建表

- 100G 数据，创建表结构文件 `create_tpch_mysql_table_part.ddl` 。

```
drop tablegroup IF EXISTS tpch_tg_SF_TPC_USER_lineitem_order_group;
drop tablegroup IF EXISTS tpch_tg_SF_TPC_USER_partsupp_part;
create tablegroup tpch_tg_SF_TPC_USER_lineitem_order_group binding true
partition by key 1 partitions 256;
create tablegroup tpch_tg_SF_TPC_USER_partsupp_part binding true partition by
key 1 partitions 256;



DROP TABLE IF EXISTS LINEITEM;
```

```
CREATE TABLE lineitem (
l_orderkey int(11) NOT NULL,
l_partkey int(11) NOT NULL,
l_suppkey int(11) NOT NULL,
l_linenumber int(11) NOT NULL,
l_quantity decimal(15,2) NOT NULL,
l_extendedprice decimal(15,2) NOT NULL,
l_discount decimal(15,2) NOT NULL,
l_tax decimal(15,2) NOT NULL,
l_returnflag char(1) DEFAULT NULL,
l_linestatus char(1) DEFAULT NULL,
l_shipdate date NOT NULL,
l_commitdate date DEFAULT NULL,
l_receiptdate date DEFAULT NULL,
l_shipinstruct varchar(25) DEFAULT NULL,
l_shipmode varchar(10) DEFAULT NULL,
l_comment varchar(44) DEFAULT NULL,
primary key(l_shipdate, l_orderkey, l_linenumber)
)row_format = condensed
tablegroup = tpch_tg_SF_TPC_USER_lineitem_order_group
partition by key (l_orderkey) partitions 256 with column group(each column);
alter table lineitem CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS ORDERS;
CREATE TABLE orders (
o_orderkey int(11) NOT NULL,
o_custkey int(11) NOT NULL,
o_orderstatus varchar(1) DEFAULT NULL,
o_totalprice decimal(15,2) DEFAULT NULL,
```

```
o_orderdate date NOT NULL,
o_orderpriority varchar(15) DEFAULT NULL,
o_clerk varchar(15) DEFAULT NULL,
o_shippriority int(11) DEFAULT NULL,
o_comment varchar(79) DEFAULT NULL,
PRIMARY KEY (o_orderkey, o_orderdate)
) row_format = condensed
tablegroup = tpch_tg_SF_TPC_USER_lineitem_order_group
partition by key(o_orderkey) partitions 256 with column group(each column);
alter table orders CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS PARTSUPP;
CREATE TABLE partsupp (
ps_partkey int(11) NOT NULL,
ps_suppkey int(11) NOT NULL,
ps_availqty int(11) DEFAULT NULL,
ps_supplycost decimal(15,2) DEFAULT NULL,
ps_comment varchar(199) DEFAULT NULL,
PRIMARY KEY (ps_partkey, ps_suppkey)) row_format = condensed
tablegroup tpch_tg_SF_TPC_USER_partsupp_part
partition by key(ps_partkey) partitions 256 with column group(each column);
alter table partsupp CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS PART;
CREATE TABLE part (
p_partkey int(11) NOT NULL,
p_name varchar(55) DEFAULT NULL,
p_mfgr varchar(25) DEFAULT NULL,
p_brand varchar(10) DEFAULT NULL,
```

```
p_type varchar(25) DEFAULT NULL,
p_size int(11) DEFAULT NULL,
p_container varchar(10) DEFAULT NULL,
p_retailprice decimal(12,2) DEFAULT NULL,
p_comment varchar(23) DEFAULT NULL,
PRIMARY KEY (p_partkey)) row_format = condensed
tablegroup tpch_tg_SF_TPC_USER_partsupp_part
partition by key(p_partkey) partitions 256 with column group(each column);
alter table part CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE customer (
c_custkey int(11) NOT NULL,
c_name varchar(25) DEFAULT NULL,
c_address varchar(40) DEFAULT NULL,
c_nationkey int(11) DEFAULT NULL,
c_phone varchar(15) DEFAULT NULL,
c_acctbal decimal(15,2) DEFAULT NULL,
c_mktsegment char(10) DEFAULT NULL,
c_comment varchar(117) DEFAULT NULL,
PRIMARY KEY (c_custkey)) row_format = condensed
partition by key(c_custkey) partitions 256 with column group(each column);
alter table customer CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS SUPPLIER;
CREATE TABLE supplier (
s_suppkey int(11) NOT NULL,
s_name varchar(25) DEFAULT NULL,
s_address varchar(40) DEFAULT NULL,
```

```
s_nationkey int(11) DEFAULT NULL,
s_phone varchar(15) DEFAULT NULL,
s_acctbal decimal(15,2) DEFAULT NULL,
s_comment varchar(101) DEFAULT NULL,
PRIMARY KEY (s_suppkey)
) row_format = condensed partition by key(s_suppkey) partitions 256 with column
group(each column);
alter table supplier CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS NATION;
CREATE TABLE nation (
n_nationkey int(11) NOT NULL,
n_name varchar(25) DEFAULT NULL,
n_regionkey int(11) DEFAULT NULL,
n_comment varchar(152) DEFAULT NULL,
PRIMARY KEY (n_nationkey)
) row_format = condensed with column group(each column);
alter table nation CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS REGION;
CREATE TABLE region (
r_regionkey int(11) NOT NULL,
r_name varchar(25) DEFAULT NULL,
r_comment varchar(152) DEFAULT NULL,
PRIMARY KEY (r_regionkey)
) row_format = condensed with column group(each column);
alter table region CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

CREATE VIEW revenue0 AS
```

```
SELECT l_suppkey as supplier_no,
SUM(l_extendedprice * ( 1 - l_discount )) as total_revenue
FROM lineitem
WHERE l_shipdate >= DATE '1996-01-01'
AND l_shipdate < DATE '1996-04-01'
GROUP BY l_suppkey;
```

- 1T 数据，创建表结构文件 `create_tpch_mysql_table_part_1000G.ddl` 。

```
drop tablegroup IF EXISTS tpch_tg_SF_TPC_USER_lineitem_order_group_1000;
drop tablegroup IF EXISTS tpch_tg_SF_TPC_USER_partsupp_part_1000;
create tablegroup tpch_tg_SF_TPC_USER_lineitem_order_group_1000 binding true
partition by key 1 partitions 256;
create tablegroup tpch_tg_SF_TPC_USER_partsupp_part_1000 binding true
partition by key 1 partitions 256;



DROP TABLE IF EXISTS LINEITEM;
CREATE TABLE lineitem (
l_orderkey bigint NOT NULL,
l_partkey int(32) NOT NULL,
l_suppkey int(32) NOT NULL,
l_linenumber int(32) NOT NULL,
l_quantity decimal(32,2) NOT NULL,
l_extendedprice decimal(32,2) NOT NULL,
l_discount decimal(15,2) NOT NULL,
l_tax decimal(15,2) NOT NULL,
l_returnflag varchar(64) DEFAULT NULL,
l_linestatus varchar(64) DEFAULT NULL,
l_shipdate date NOT NULL,
```

```
l_commitdate date DEFAULT NULL,
l_receiptdate date DEFAULT NULL,
l_shipinstruct varchar(64) DEFAULT NULL,
l_shipmode varchar(64) DEFAULT NULL,
l_comment varchar(64) DEFAULT NULL,
primary key(l_shipdate, l_orderkey, l_linenumber)
)row_format = condensed
tablegroup = tpch_tg_SF_TPC_USER_lineitem_order_group_1000
partition by key (l_orderkey) partitions 256 with column group(each column);
alter table lineitem CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS ORDERS;
CREATE TABLE orders (
o_orderkey bigint NOT NULL,
o_custkey int(32) NOT NULL,
o_orderstatus varchar(64) DEFAULT NULL,
o_totalprice decimal(15,2) DEFAULT NULL,
o_orderdate date NOT NULL,
o_orderpriority varchar(15) DEFAULT NULL,
o_clerk varchar(15) DEFAULT NULL,
o_shippriority int(32) DEFAULT NULL,
o_comment varchar(128) DEFAULT NULL,
PRIMARY KEY (o_orderkey, o_orderdate)
) row_format = condensed
tablegroup = tpch_tg_SF_TPC_USER_lineitem_order_group_1000
partition by key(o_orderkey) partitions 256 with column group(each column);
alter table orders CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS PARTSUPP;
```

```
CREATE TABLE partsupp (
ps_partkey int(11) NOT NULL,
ps_suppkey int(11) NOT NULL,
ps_availqty int(11) DEFAULT NULL,
ps_supplycost decimal(15,2) DEFAULT NULL,
ps_comment varchar(199) DEFAULT NULL,
PRIMARY KEY (ps_partkey, ps_suppkey)) row_format = condensed
tablegroup tpch_tg_SF_TPC_USER_partsupp_part_1000
partition by key(ps_partkey) partitions 256 with column group(each column);
alter table partsupp CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS PART;
CREATE TABLE part (
p_partkey int(11) NOT NULL,
p_name varchar(55) DEFAULT NULL,
p_mfgr varchar(25) DEFAULT NULL,
p_brand varchar(10) DEFAULT NULL,
p_type varchar(25) DEFAULT NULL,
p_size int(11) DEFAULT NULL,
p_container varchar(10) DEFAULT NULL,
p_retailprice decimal(12,2) DEFAULT NULL,
p_comment varchar(23) DEFAULT NULL,
PRIMARY KEY (p_partkey)) row_format = condensed
tablegroup tpch_tg_SF_TPC_USER_partsupp_part_1000
partition by key(p_partkey) partitions 256 with column group(each column);
alter table part CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS CUSTOMER;
CREATE TABLE customer (
```

```
c_custkey int(11) NOT NULL,
c_name varchar(25) DEFAULT NULL,
c_address varchar(40) DEFAULT NULL,
c_nationkey int(11) DEFAULT NULL,
c_phone varchar(15) DEFAULT NULL,
c_acctbal decimal(15,2) DEFAULT NULL,
c_mktsegment char(10) DEFAULT NULL,
c_comment varchar(117) DEFAULT NULL,
PRIMARY KEY (c_custkey)) row_format = condensed
partition by key(c_custkey) partitions 256 with column group(each column);
alter table customer CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS SUPPLIER;
CREATE TABLE supplier (
s_suppkey int(11) NOT NULL,
s_name varchar(25) DEFAULT NULL,
s_address varchar(40) DEFAULT NULL,
s_nationkey int(11) DEFAULT NULL,
s_phone varchar(15) DEFAULT NULL,
s_acctbal decimal(15,2) DEFAULT NULL,
s_comment varchar(101) DEFAULT NULL,
PRIMARY KEY (s_suppkey)
) row_format = condensed partition by key(s_suppkey) partitions 256 with column
group(each column);
alter table supplier CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS NATION;
CREATE TABLE nation (
n_nationkey int(11) NOT NULL,
```

```
n_name varchar(25) DEFAULT NULL,
n_regionkey int(11) DEFAULT NULL,
n_comment varchar(152) DEFAULT NULL,
PRIMARY KEY (n_nationkey)
) row_format = condensed with column group(each column);
alter table nation CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

DROP TABLE IF EXISTS REGION;
CREATE TABLE region (
r_regionkey int(11) NOT NULL,
r_name varchar(25) DEFAULT NULL,
r_comment varchar(152) DEFAULT NULL,
PRIMARY KEY (r_regionkey)
) row_format = condensed with column group(each column);
alter table region CONVERT TO CHARACTER SET utf8mb4 COLLATE utf8mb4_bin;

CREATE VIEW revenue0 AS
SELECT l_suppkey as supplier_no,
SUM(l_extendedprice * ( 1 - l_discount )) as total_revenue
FROM lineitem
WHERE l_shipdate >= DATE '1996-01-01'
AND l_shipdate < DATE '1996-04-01'
GROUP BY l_suppkey;
```

13.1.7 步骤七：加载数据

您可以根据上述步骤生成的数据和 SQL 自行编写脚本。加载数据示例操作如下：

1. 创建加载数据的脚本 `load_data.sh` 。

```
#!/bin/bash
host='$host_ip' # 注意！！ 请填写某个 observer，如 observer A 所在服务器的 IP 地址，
```

```
最好将数据文件也放在该服务器下
port='$host_port' # observer A 的端口号
user='$user' # 用户名
tenant='$tenant_name' # 租户名
password='$password' # 密码
database='$db_name' # 数据库名
data_path='$data_file' # 注意！！ 请填写某个 observer，如 observer A 下在生成数据步骤中生成的数据文件.tbl路径

function load_data
{
remote_user="$user" # 存放数据的 observer 节点用户名
table_name=${1}
if [[ ${password} == "" ]];then
obclient_conn="obclient -h${host} -P${port} -u${user} -D${database} -A -c"
else
obclient_conn="obclient -h${host} -P${port} -u${user} -D${database} -
p${password} -A -c"
fi
table_list=$(ssh "${remote_user}@${host}" "ls ${data_path}/${table_name}.tbl* 2>
/dev/null")
echo "$table_list"

IFS=$'\n' read -d '' -r -a table_files <<< "$table_list"
table_files_comma_separated=$(IFS=,; echo "${table_files[*]}")
echo "${table_files_comma_separated}"
echo `date "+[%Y-%m-%d %H:%M:%S]"` "----------------------正在导
入${table_name}表的数据文件----------------------"
```

```
# 使用旁路导入方式导入数据，也可自行修改为其他方式
echo "load data /*+ parallel(80) direct(true,0) */ infile
'${table_files_comma_separated}' into table ${table_name} fields terminated by
'|';" | ${obclient_conn}

}

starttime=`date +%s%N`
for table in "nation" "region" "customer" "lineitem" "orders" "partsupp" "part"
"supplier"
do
load_data "${table}"
done
end_time=`date +%s%N`
totaltime=`echo ${end_time} ${starttime} | awk '{printf "%0.2f\n", ($1 - $2) /
1000000000}'`
echo `date "+[%Y-%m-%d %H:%M:%S]"` "load data cost ${totaltime}s"
```

加载完数据后需进行合并和统计信息。

2. 执行合并。

   在测试租户执行以下语句进行合并。

   ```
   ALTER SYSTEM MAJOR FREEZE;
   ```

3. 查看合并是否完成。

   可以在 `sys` 租户下查看合并是否完成

   ```
   SELECT dt.TENANT_NAME, cc.FROZEN_SCN, cc.LAST_SCN
   FROM oceanbase.DBA_OB_TENANTS dt, oceanbase.CDB_OB_MAJOR_COMPACTION
   cc
   WHERE dt.TENANT_ID = cc.TENANT_ID
   AND dt.TENANT_NAME = 'mysql_tenant';
   ```

13.1.7.3 说明

所有的 FROZEN_SCN 和 LAST_SCN 的值相等即表示合并完成。

4. 执行收集统计信息。

   创建收集统计信息文件 analyze_table.sql 。

```
call dbms_stats.gather_table_stats(NULL, 'part', degree=>128,
granularity=>'AUTO', method_opt=>'FOR ALL COLUMNS SIZE 128');
call dbms_stats.gather_table_stats(NULL, 'lineitem', degree=>128,
granularity=>'AUTO', method_opt=>'FOR ALL COLUMNS SIZE 128');
call dbms_stats.gather_table_stats(NULL, 'customer', degree=>128,
granularity=>'AUTO', method_opt=>'FOR ALL COLUMNS SIZE 128');
call dbms_stats.gather_table_stats(NULL, 'orders', degree=>128,
granularity=>'AUTO', method_opt=>'FOR ALL COLUMNS SIZE 128');
call dbms_stats.gather_table_stats(NULL, 'partsupp', degree=>128,
granularity=>'AUTO', method_opt=>'FOR ALL COLUMNS SIZE 128');
call dbms_stats.gather_table_stats(NULL, 'supplier', degree=>128,
granularity=>'AUTO', method_opt=>'FOR ALL COLUMNS SIZE 128');
```

   登录测试租户，执行以下语句进行收集统计信息：

```
source analyze_table.sql
```

13.1.8 步骤八：执行测试

您可以根据上述步骤生成的数据和 SQL 自行编写脚本。执行测试示例操作如下：

1. 编写测试脚本 tpch.sh 。

```
#!/bin/bash
host='$host_ip' # 注意！！ 请填写某个 observer，如 observer A 所在服务器的 IP 地址
port='$host_port' # observer A 的端口号
user='$user' # 用户名
tenant='$tenant_name' # 租户名
```

```
password='$password' # 密码
database='$db_name' # 数据库名
if [[ ${password} == "" ]];then
TPCH_TEST="obclient -h${host} -P${port} -u${user}@{$tenant} -D${database} -A -c"
else
TPCH_TEST="obclient -h${host} -P${port} -p${password} -u${user}@{$tenant} -
D${database} -A -c"
fi


function clear_kvcache
{
if [[ ${password_sys} == "" ]];then
obclient_sys="obclient -h${host} -P${port} -uroot@sys -Doceanbase -A -c"
else
obclient_sys="obclient -h${host} -P${port} -uroot@sys -Doceanbase -
p${password_sys} -A -c"
fi
tenant_name=${user#*@}
echo "alter system flush kvcache ;" | ${obclient_sys}
echo "alter system flush kvcache tenant '${tenant_name}' cache
'user_row_cache';" | ${obclient_sys}
sleep 3s
}

function do_explain
{
#执行计划
echo `date '+[%Y-%m-%d %H:%M:%S]'` "BEGIN EXPLAIN ALL TPCH PLAN"
```

```
for i in {1..22}
do
sql_explain="source explain_mysql/${i}.sql"
echo `date '+[%Y-%m-%d %H:%M:%S]'` "BEGIN EXPLAIN Q${i}:"
echo ${sql_explain} | ${TPCH_TEST} | sed 's/\\n/\n/g' |tee explain_log/${i}.exp
echo `date '+[%Y-%m-%d %H:%M:%S]'` "Q${i} END"
done
}

function do_warmup
{
#warmup预热
totaltime=0
for i in {1..22}
do
starttime=`date +%s%N`
echo `date '+[%Y-%m-%d %H:%M:%S]'` "BEGIN prewarm Q${i}"
sql1="source mysql_sql/${i}.sql"
echo ${sql1}| ${TPCH_TEST} > mysql_log/${i}_prewarm.log || ret=1
stoptime=`date +%s%N`
costtime=`echo ${stoptime} ${starttime} | awk '{printf "%0.2f\n", ($1 - $2) /
1000000000}'`
first_array[$i]=$(echo "scale=2; ${first_array[$i]} + $costtime" | bc)
echo `date '+[%Y-%m-%d %H:%M:%S]'` "END,COST ${costtime}s"
totaltime=`echo ${totaltime} ${costtime} | awk '{printf "%0.2f\n", ($1 + $2)}'`
done
echo "total cost:${totaltime}s"
}
```

```
function hot_run
{
#正式执行
for j in {1..10}
do
totaltime=0
for i in {1..22}
do
starttime=`date +%s%N`
echo `date '+[%Y-%m-%d %H:%M:%S]'` "BEGIN BEST Q${i} (hot run)"
sql1="source mysql_sql/${i}.sql"
echo ${sql1}| ${TPCH_TEST} > mysql_log/${i}.log || ret=1
stoptime=`date +%s%N`
costtime=`echo ${stoptime} ${starttime} | awk '{printf "%0.2f\n", ($1 - $2) /
1000000000}'`
hot_array[$i]=$(echo "scale=2; ${hot_array[$i]} + $costtime" | bc)
echo `date '+[%Y-%m-%d %H:%M:%S]'` "END,COST ${costtime}s"
totaltime=`echo ${totaltime} ${costtime} | awk '{printf "%0.2f\n", ($1 + $2)}'`
done
echo "total cost:${totaltime}s"
done
}

function cold_run
{
#正式执行
for j in {1..3}
do
totaltime=0
```

```
for i in {1..22}
do
clear_kvcache
starttime=`date +%s%N`
echo `date '+[%Y-%m-%d %H:%M:%S]'` "BEGIN BEST Q${i} (cold run)"
sql1="source mysql_sql/${i}.sql"
echo $sql1| $TPCH_TEST > mysql_log/${i}_cold.log || ret=1
stoptime=`date +%s%N`
costtime=`echo $stoptime $starttime | awk '{printf "%0.2f\n", ($1 - $2) /
1000000000}'`
cold_array[$i]=$(echo "scale=2; ${cold_array[$i]} + $costtime" | bc)
echo `date '+[%Y-%m-%d %H:%M:%S]'` "END,COST ${costtime}s"
totaltime=`echo ${totaltime} ${costtime} | awk '{printf "%0.2f\n", ($1 + $2)}'`
done
echo "total cost:${totaltime}s"
done
}

do_explain

do_warmup

hot_run

cold_run
```

2. 执行测试脚本。

```
sh tpch.sh
```

13.1.9 FAQ

- 导入数据失败。报错信息如下：

  ```
  ERROR 1017 (HY000) at line 1: File not exist
  ```

  `tbl` 文件必须放在所连接的 OceanBase 数据库所在机器的某个目录下，因为加载数据必须本地导入。
- 查看数据报错。报错信息如下：

  ```
  ERROR 4624 (HY000): No memory or reach tenant memory limit
  ```

  内存不足，建议增大租户内存。
- 导入数据报错。报错信息如下：

  ```
  ERROR 1227 (42501) at line 1: Access denied
  ```

  需要授予用户访问权限。运行以下命令，授予权限：

  ```
  grant file on *.* to tpch_100g_part;
  ```

13.2 手动体验 Operational OLAP

通过上一步的操作，我们已经获得了一个 TPCH 的测试环境，下面让我们通过手动执行来看看，OceanBase 数据库在 OLAP 方面的能力和特性。 我们先使用 OBClient 登录到数据库中，如果您没有安装 OBClient，使用 `mysql` 客户端也是可以的。

```
obclient -h127.0.0.1 -P2881 -uroot@test -Dtest -A -p -c
```

在开始之前，您需要根据 OceanBase 集群和租户的配置，进行并行度的设置，具体大小建议不超过当前租户配置的 CPU 核数的 2 倍。例如您的租户 CPU 最大配置为8，那么此处建议并行度设置为16：

```
MySQL [test]> SET GLOBAL parallel_servers_target=16;
Query OK, 0 rows affected


MySQL [test]> SET GLOBAL parallel_max_servers=16;
Query OK, 0 rows affected
```

OceanBase 数据库兼容大多数 MySQL 的内部视图，我们可以通过如下查询查看当前环境中表的大小：

```
```

```
MySQL [test]> SELECT table_name, table_rows, CONCAT(ROUND(data_length/
(1024*1024*1024),2),' GB') table_size FROM information_schema.TABLES WHERE
table_schema = 'test' order by table_rows desc;
+------------+------------+------------+
| table_name | table_rows | table_size |
+------------+------------+------------+
| lineitem | 6001215 | 0.37 GB |
| orders | 1500000 | 0.08 GB |
| partsupp | 800000 | 0.04 GB |
| part | 200000 | 0.01 GB |
| customer | 150000 | 0.01 GB |
| supplier | 10000 | 0.00 GB |
| nation | 25 | 0.00 GB |
| region | 5 | 0.00 GB |
+------------+------------+------------+
8 rows in set
```

下面我们通过 TPC-H 测试中的 Q1 来体验 OceanBase 数据库查询能力，Q1 查询会在最大的 `lineitem` 表上，汇总分析指定时间内各类商品的价格、折扣、发货、数量等信息。这个查询对全表数据都会进行读取、并进行分区、排序、聚合等计算。

13.2.10 不开启并发查询

首先，我们在默认不开启并发的情况下执行该查询：

```
select
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
```

```
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01' - interval '90' day
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
```

在本例的测试环境中，执行结果如下：

```
+--------------+--------------+----------+----------------+----------------+--------------
+---------+------------+----------+-------------+
| l_returnflag | l_linestatus | sum_qty | sum_base_price | sum_disc_price | sum_charge |
avg_qty | avg_price | avg_disc | count_order |
+--------------+--------------+----------+----------------+----------------+--------------
+---------+------------+----------+-------------+
| A | F | 37734107 | 56586577106 | 56586577106 | 56586577106 | 25.5220 | 38273.1451 |
0.0000 | 1478493 |
| N | F | 991417 | 1487505208 | 1487505208 | 1487505208 | 25.5165 | 38284.4806 |
0.0000 | 38854 |
| N | O | 74476040 | 111701776272 | 111701776272 | 111701776272 | 25.5022 |
38249.1339 | 0.0000 | 2920374 |
| R | F | 37719753 | 56568064200 | 56568064200 | 56568064200 | 25.5058 | 38250.8701 |
0.0000 | 1478870 |
```

```
+--------------+--------------+----------+---------------+---------------+-------------
+---------+------------+----------+-------------+
4 rows in set (6.791 sec)
```

13.2.11 开启并发查询

OceanBase 数据库的 Operational OLAP 能力基于一套数据以及执行引擎，无需进行异构的数据同步和维护。下面我们通过添加一个 `parallel` Hint，以并行度为8的方式再次执行这条语句：

```
select /*+parallel(8) */
l_returnflag,
l_linestatus,
sum(l_quantity) as sum_qty,
sum(l_extendedprice) as sum_base_price,
sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
avg(l_quantity) as avg_qty,
avg(l_extendedprice) as avg_price,
avg(l_discount) as avg_disc,
count(*) as count_order
from
lineitem
where
l_shipdate <= date '1998-12-01' - interval '90' day
group by
l_returnflag,
l_linestatus
order by
l_returnflag,
l_linestatus;
```

在相同的环境和数据集中，执行结果如下：

```
+--------------+--------------+----------+---------------+---------------+-------------
+---------+------------+----------+-------------+
| l_returnflag | l_linestatus | sum_qty | sum_base_price | sum_disc_price | sum_charge |
avg_qty | avg_price | avg_disc | count_order |
+--------------+--------------+----------+---------------+---------------+-------------
+---------+------------+----------+-------------+
| A | F | 37734107 | 56586577106 | 56586577106 | 56586577106 | 25.5220 | 38273.1451 |
0.0000 | 1478493 |
| N | F | 991417 | 1487505208 | 1487505208 | 1487505208 | 25.5165 | 38284.4806 |
0.0000 | 38854 |
| N | O | 74476040 | 111701776272 | 111701776272 | 111701776272 | 25.5022 |
38249.1339 | 0.0000 | 2920374 |
| R | F | 37719753 | 56568064200 | 56568064200 | 56568064200 | 25.5058 | 38250.8701 |
0.0000 | 1478870 |
+--------------+--------------+----------+---------------+---------------+-------------
+---------+------------+----------+-------------+
4 rows in set (1.197 sec)
```

可以看到，对比默认无并发的执行耗时，并行查询下速度提升了将近 6 倍。如果我们通过 `EXPLAIN` 命令查看执行计划，也可以看到并行度的展示（第 18 行，1 号算子，dop=8）：

```
============================================================
|ID|OPERATOR |NAME |EST. ROWS|COST |
------------------------------------------------------------
|0 |PX COORDINATOR MERGE SORT | |6 |13507125|
|1 | EXCHANGE OUT DISTR |:EX10001|6 |13507124|
|2 | SORT | |6 |13507124|
|3 | HASH GROUP BY | |6 |13507107|
|4 | EXCHANGE IN DISTR | |6 |8379337 |
```

```
|5 | EXCHANGE OUT DISTR (HASH)|:EX10000|6 |8379335 |
|6 | HASH GROUP BY | |6 |8379335 |
|7 | PX BLOCK ITERATOR | |5939712 |3251565 |
|8 | TABLE SCAN |lineitem|5939712 |3251565 |
==============================================================

Outputs & filters:
-------------------------------------
0 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [T_FUN_SUM(T_FUN_SUM
(lineitem.l_quantity))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice))],
[T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount))],
[T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 +
lineitem.l_tax))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_quantity)) / cast
(T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_quantity)), DECIMAL(20, 0))],
[T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice)) / cast(T_FUN_COUNT_SUM
(T_FUN_COUNT(lineitem.l_extendedprice)), DECIMAL(20, 0))], [T_FUN_SUM
(T_FUN_SUM(lineitem.l_discount)) / cast(T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.
l_discount)), DECIMAL(20, 0))], [T_FUN_COUNT_SUM(T_FUN_COUNT(*))]), filter(nil),
sort_keys([lineitem.l_returnflag, ASC], [lineitem.l_linestatus, ASC])
1 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [T_FUN_SUM(T_FUN_SUM
(lineitem.l_quantity))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice))],
[T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount))],
[T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 +
lineitem.l_tax))], [T_FUN_COUNT_SUM(T_FUN_COUNT(*))], [T_FUN_SUM(T_FUN_SUM
(lineitem.l_quantity)) / cast(T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_quantity)),
DECIMAL(20, 0))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice)) / cast
(T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_extendedprice)), DECIMAL(20, 0))],
[T_FUN_SUM(T_FUN_SUM(lineitem.l_discount)) / cast(T_FUN_COUNT_SUM
(T_FUN_COUNT(lineitem.l_discount)), DECIMAL(20, 0))]), filter(nil), dop=8
```

2 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [T_FUN_SUM(T_FUN_SUM (lineitem.l_quantity))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.l_tax))], [T_FUN_COUNT_SUM(T_FUN_COUNT(*))], [T_FUN_SUM(T_FUN_SUM (lineitem.l_quantity)) / cast(T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_quantity)), DECIMAL(20, 0))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice)) / cast (T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_extendedprice)), DECIMAL(20, 0))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_discount)) / cast(T_FUN_COUNT_SUM (T_FUN_COUNT(lineitem.l_discount)), DECIMAL(20, 0))]), filter(nil), sort_keys([lineitem. l_returnflag, ASC], [lineitem.l_linestatus, ASC])
3 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [T_FUN_SUM(T_FUN_SUM (lineitem.l_quantity))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.l_tax))], [T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_quantity))], [T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_extendedprice))], [T_FUN_SUM (T_FUN_SUM(lineitem.l_discount))], [T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem. l_discount))], [T_FUN_COUNT_SUM(T_FUN_COUNT(*))]), filter(nil), group([lineitem.l_returnflag], [lineitem.l_linestatus]), agg_func([T_FUN_SUM (T_FUN_SUM(lineitem.l_quantity))], [T_FUN_SUM(T_FUN_SUM(lineitem. l_extendedprice))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem. l_discount))], [T_FUN_SUM(T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem. l_discount * 1 + lineitem.l_tax))], [T_FUN_COUNT_SUM(T_FUN_COUNT(*))], [T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_quantity))], [T_FUN_COUNT_SUM (T_FUN_COUNT(lineitem.l_extendedprice))], [T_FUN_SUM(T_FUN_SUM(lineitem. l_discount))], [T_FUN_COUNT_SUM(T_FUN_COUNT(lineitem.l_discount))])
4 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [T_FUN_SUM(lineitem. l_quantity)], [T_FUN_SUM(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.

l_extendedprice * 1 - lineitem.l_discount)], [T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.l_tax)], [T_FUN_COUNT(lineitem.l_quantity)], [T_FUN_COUNT(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.l_discount)], [T_FUN_COUNT(lineitem.l_discount)], [T_FUN_COUNT(*)]), filter(nil)
5 - (#keys=2, [lineitem.l_returnflag], [lineitem.l_linestatus]), output([lineitem.l_returnflag], [lineitem.l_linestatus], [T_FUN_SUM(lineitem.l_quantity)], [T_FUN_SUM(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount)], [T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.l_tax)], [T_FUN_COUNT(lineitem.l_quantity)], [T_FUN_COUNT(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.l_discount)], [T_FUN_COUNT(lineitem.l_discount)], [T_FUN_COUNT(*)]), filter(nil), dop=8
6 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [T_FUN_SUM(lineitem.l_quantity)], [T_FUN_SUM(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount)], [T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.l_tax)], [T_FUN_COUNT(lineitem.l_quantity)], [T_FUN_COUNT(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.l_discount)], [T_FUN_COUNT(lineitem.l_discount)], [T_FUN_COUNT(*)]), filter(nil), group([lineitem.l_returnflag], [lineitem.l_linestatus]), agg_func([T_FUN_SUM(lineitem.l_quantity)], [T_FUN_SUM(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount)], [T_FUN_SUM(lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.l_tax)], [T_FUN_COUNT(*)], [T_FUN_COUNT(lineitem.l_quantity)], [T_FUN_COUNT(lineitem.l_extendedprice)], [T_FUN_SUM(lineitem.l_discount)], [T_FUN_COUNT(lineitem.l_discount)])
7 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [lineitem.l_quantity], [lineitem.l_extendedprice], [lineitem.l_discount], [lineitem.l_extendedprice * 1 - lineitem.l_discount], [lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.l_tax]), filter(nil)
8 - output([lineitem.l_returnflag], [lineitem.l_linestatus], [lineitem.l_quantity], [lineitem.l_extendedprice], [lineitem.l_discount], [lineitem.l_extendedprice * 1 -

```
lineitem.l_discount], [lineitem.l_extendedprice * 1 - lineitem.l_discount * 1 + lineitem.
l_tax]), filter([lineitem.l_shipdate <= ?]),
access([lineitem.l_shipdate], [lineitem.l_returnflag], [lineitem.l_linestatus], [lineitem.
l_quantity], [lineitem.l_extendedprice], [lineitem.l_discount], [lineitem.l_tax]),
partitions(p[0-15])
```

本文中的例子使用单节点环境部署，值得特别说明的是，OceanBase 数据库的并行执行框架最大的特点是还可以将大量数据的分析查询以多节点并发执行的方式进行分析。例如一张表包含上亿行数据，分布在多个 OceanBase 数据库节点上，当进行分析查询时，OceanBase 数据库的分布式执行框架可以生成一个分布式并行执行计划，利用多个节点的资源进行分析。因此具备很好的扩展性，同时针对并行的设置还可以在 SQL、会话、表上多个维度进行设置。

13.3 使用 obd 工具自动进行 TPCH 测试

13.3.11.1 功能适用性

该内容仅适用于 OceanBase 数据库社区版。OceanBase 数据库企业版暂不支持 obd 使用。

进行 TPC-H 测试除了可以参考 TPC 官方网站提供的数据集生成工具，我们也可以使用 obd 方便的进行数据集生成、建表、数据导入等工作，并且自动完成 22 个 SQL 的执行。 在使用 obd 进行 TPC-H 测试前，您需要先在部署了 OceanBase 和 obd 的节点上安装 obtpch 组件：

```
sudo yum install obtpch
```

完成后，我们通过如下命令，就可以启动一个数据集规模为 1 GB 的 TPCH 测试了，整个过程包括数据集生成、schema 导入、以及自动运行测试。本文中假设您的测试环境部署和 快速体验 OceanBase 数据库 中的步骤一致，如有差别，例如集群名称、密码 安装目录等，请根据具体情况进行调整。

13.3.11.2 注意

请确保您的磁盘空间足够放置数据集文件，以免将空间占满导致系统异常。

本例中我们使用 /tmp 目录进行测试。

```
cd /tmp
obd test tpch obtest --tenant=test -s 1 --password='******' --remote-tbl-dir=/tmp
/tpch1
```

执行上述命令后，obd 开始执行，可以看到执行过程中的每个步骤：

```
[root@obce00 ~]# obd test tpch obtest  --tenant=test -s 1 --password=root --remote-tbl-dir=/oceanbase/tmp/tpch1
Get local repositories and plugins ok
Open ssh connection ok
Cluster status check ok
Connect observer(server1(127.0.0.1):2881) ok
Send tbl to remote (server1(127.0.0.1)) ok
Format DDL ok
Create table ok
Load data ok
Merge ok
Format SQL ok
Warmup ok
```

```
[2022-07-16 07:47:25]: start /root/tmp/db21.sql
[2022-07-16 07:47:26]: end /root/tmp/db21.sql, cost 1.1s
[2022-07-16 07:47:26]: start /root/tmp/db22.sql
[2022-07-16 07:47:26]: end /root/tmp/db22.sql, cost 0.3s
Total Cost: 11.5s
See https://open.oceanbase.com/docs/obd-cn/V1.3.0/10000000000099584.
```

数据导入完成后，obd 自动进行 22 个 SQL 的执行，并打印每个 SQL 的耗时以及总耗时。

14 体验并行导入和数据压缩

本文介绍了并行导入和数据压缩相关的使用及说明。

常见的应用场景如下：

- **数据迁移**：当需要将大量数据从一个系统迁移到另一个系统时，使用并行导入可以显著加快数据迁移的速度，而数据压缩则可以减少所需的传输带宽和存储资源。
- **备份和恢复**：在进行数据备份时，采用数据压缩可以减少备份文件的大小，节省存储空间。在恢复数据时，采用并行导入可以加快数据恢复的速度。
- **列存表**：在列存表中进行查询时，通常只需要访问查询所涉及的列，而不是整行数据。因此，压缩后的列仍可在压缩状态下进行扫描和处理，这可以减少I/O操作，加快查询速度，提高整体的查询性能。对于列存表，批量导入数据后，做一次数据压缩，可以使读性能更优。需要注意的是，列存表的合并速度较慢。

14.1 并行导入

除了分析查询，Operational OLAP 中还有很重要的一个部分，那就是大量数据的并行导入，也就是数据批处理能力。OceanBase 数据库的并行执行框架能够将 DML 语句也通过并发的方式进行执行（Parallel DML），对于多节点的数据库，实现多机并发写入，并且保证大事务的一致性。结合异步转储机制，还能在很大程度上优化 LSM-Tree 存储引擎在内存紧张的情况下对大事务的支持。

我们通过这样一个例子来体验 PDML：仍然以 TPC-H 的 `lineitem` 表为基础，创建一张相同表结构的空表 `lineitem2`。然后以 `INSERT INTO ...SELECT` 的方式，用一条 SQL 语句将 `lineitem` 的全部 600 万行数据插入到新表 `lineitem2` 中。下面我们分别用关闭和开启 PDML 的方式执行，观察其效果和区别。

首先，复制 `lineitem` 的表结构，创建 `lineitem2`。注意，在 OceanBase 数据库中我们使用分区表进行数据扩展，此处的例子中我们使用 16 个分区，那么对应的 `lineitem2` 也应完全相同：

```
obclient [test]> SHOW CREATE TABLE lineitem\G
*************************** 1. row ***************************
Table: lineitem

Create Table: CREATE TABLE `lineitem` (
`l_orderkey` bigint(20) NOT NULL,
`l_partkey` bigint(20) NOT NULL,
```

```
`l_suppkey` bigint(20) NOT NULL,
`l_linenumber` bigint(20) NOT NULL,
`l_quantity` bigint(20) NOT NULL,
`l_extendedprice` bigint(20) NOT NULL,
`l_discount` bigint(20) NOT NULL,
`l_tax` bigint(20) NOT NULL,
`l_returnflag` char(1) DEFAULT NULL,
`l_linestatus` char(1) DEFAULT NULL,
`l_shipdate` date NOT NULL,
`l_commitdate` date DEFAULT NULL,
`l_receiptdate` date DEFAULT NULL,
`l_shipinstruct` char(25) DEFAULT NULL,
`l_shipmode` char(10) DEFAULT NULL,
`l_comment` varchar(44) DEFAULT NULL,
PRIMARY KEY (`l_orderkey`, `l_linenumber`),
KEY `I_L_ORDERKEY` (`l_orderkey`) BLOCK_SIZE 16384 LOCAL,
KEY `I_L_SHIPDATE` (`l_shipdate`) BLOCK_SIZE 16384 LOCAL
) DEFAULT CHARSET = utf8mb4 ROW_FORMAT = COMPACT COMPRESSION = 'zstd_1.
3.8' REPLICA_NUM = 1 BLOCK_SIZE = 16384 USE_BLOOM_FILTER = FALSE TABLET_SIZE
= 134217728 PCTFREE = 0 TABLEGROUP = 'x_tpch_tg_lineitem_order_group'
partition by key(l_orderkey)
(partition p0,
partition p1,
partition p2,
partition p3,
partition p4,
partition p5,
partition p6,
partition p7,
```

```
partition p8,
partition p9,
partition p10,
partition p11,
partition p12,
partition p13,
partition p14,
partition p15)
1 row in set
```

14.1.1 默认方式执行，不开启 PDML

创建好 lineitem2 后，我们先以默认配置不开启并行的方式插入，因为这是一个 600 万行的大事务，我们需要将 OceanBase 数据库默认的事务超时时间调整到更大的值（单位为 μs）：

```
# SET ob_query_timeout = 1000000000;
# SET ob_trx_timeout = 1000000000;
```

插入数据，执行结果如下：

```
obclient [test]> INSERT INTO lineitem2 SELECT * FROM lineitem;
Query OK, 6001215 rows affected (1 min 47.312 sec)
Records: 6001215 Duplicates: 0 Warnings: 0
```

可以看到，不开启并行的情况下，单个事务插入 600 万行数据，OceanBase 的耗时为 107 秒。

14.1.2 开启 PDML 执行

下面我们通过添加一个 Hint，开启 PDML 的执行选项。注意再次插入前，我们先将上次插入的数据清空。

```
obclient [test]> TRUNCATE TABLE lineitem2;
obclient [test]> INSERT /*+ parallel(16) enable_parallel_dml */ INTO lineitem2 SELECT
* FROM lineitem;
```

来看这次的执行耗时：

```
obclient> TRUNCATE TABLE lineitem2;
Query OK, 0 rows affected (0.108 sec)

obclient> INSERT /*+ parallel(16) enable_parallel_dml */ INTO lineitem2 SELECT *
FROM lineitem;
Query OK, 6001215 rows affected (22.117 sec)
Records: 6001215 Duplicates: 0 Warnings: 0
```

可以看到开启 PDML 后，相同的表插入 600 万行数据，OceanBase 数据库的耗时缩短为 22 秒左右。PDML 特性带来的性能提升大约为 5 倍。这一特性可以在用户在需要批量数据处理的场景提供帮助。

14.2 数据压缩

OceanBase 数据库基于 LSM-Tree 结构开发了自己的存储引擎。其中数据大致被分为基线数据（SSTable）和增量数据（MemTable）两部分，基线数据被保存在磁盘中，增量修改在内存中进行。这使得一方面数据在磁盘中能够以更紧凑的方式存储。除此之外由于在磁盘中的基线数据不会频繁更新，OceanBase 数据库又基于通用压缩算法对基线数据进行了再次压缩，使得数据存储在 OceanBase 数据库中可以获得非常好的压缩比。同时这种数据压缩并未带来查询和写入性能的下降。下面我们介绍 OceanBase 数据库导入大量外部数据并且观察数据压缩比的方法。

14.2.3 数据准备

首先我们使用数据准备工具 CreateData.jar 生成 5 千万行模拟数据到 `/home/soft` 目录下，生成数据大概需要十几分钟时间，您也可以使用其他工具生成测试数据。

```
#mkdir /home/soft/
#java -jar CreateData.jar /home/soft/ 50000000
filePath is：/home/soft/
Start Time：2022-07-10 15:33:00
End Time：2022-07-10 15:52:53
#du -sh *
10G t_bigdata_loader.unl
```

OceanBase 数据库支持多种方式将 csv 格式的数据导入到 OceanBase 数据库中，本文我们介绍通过 Load Data 命令执行。

1. 首先您需对生成的文件进行命名，并确认实际大小。

```
# mv t_bigdata_loader.unl t_f1.csv
# du -sh t_f1.csv
10G t_f1.csv
```

2. 对 `t_f1.csv` 文件内容查看可知，预先生成好的 csv 文件，通过随机算法，获取了 8 列数据，可对应不同的数据类型。因此在体验 OceanBase 数据压缩特性时，需要在租户下先创建一张表，将 csv 文件中的记录，导入到表中。

```
1|1896404182|1980-06-01|2004-10-25 13:30:39|7470.689|33062564.9527|nOLqnBY
tnp|BzWYjZjeodtBNzXSMyBduMNzwDPSiVmhVgPJMeEkeAwKBCorzblwovIHDKBsQh
bVjQnIdoeTsiLXTNwyuAcuneuNaol|
2|572083440|2018-11-09|1998-07-11 01:23:28|6891.054|66028434.4013|UzqteeMa
HP|vQWbWBXEWgUqUTzqsOSciiOuvWVcZSrlEOQDwDVGmvGRQYWmhCFdEkpsUsq
rWEpKtmxSwURHIHxvmlXHUIxmfelYboeGEuScKKqzpuNLryFsStaFTTRqSsVlCngFFjH
nEnpaCnWsdwztbiHJyoGkaxrFmyPAmVregfydArrUZsgRqBpQ|
3|1139841892|2006-10-07|1999-06-26 17:02:22|286.43692|51306547.
5055|KJJtylgxkv|BuBdFTBIIFsEPVxsVBRqAnFXSBdtZDgfumUhIx|
4|1777342512|1982-12-18|2017-11-19 07:56:35|2986.242|85860387.8696|rTkUBWh
dPt|JSazOTAmvtCBrINttDwublNJNRFDIiWkHtWZXmWgKHoZCKGqmmETkIcYLXiSgKk
oaATNgjvPxVGjeCOODLEWqrQHqowbMjOLOKrtirWEOpUSxiUudZduTCUvZElKzZfgg
vCBNthwzKJc|
....
```

3. 在 `test` 租户下的 `test` 数据库中创建一张表，表名为 `t_f1` 。详细的租户创建过程，请参考管理租户内容。

```
# obclient -hxxx.xxx.xxx.xxx -P2881 -uroot@test -Dtest -A -p -c
```

```
obclient [test]> CREATE TABLE t_f1(id DECIMAL(10,0),id2 DECIMAL(10,0),id3 DATE,
id4 DATE,id5 FLOAT,id6 FLOAT,id7 VARCHAR(30),id8 VARCHAR(300));
```

14.2.4 数据导入

我们可以使用 OceanBase 数据库内置的 Load Data 命令导入数据，Load Data 同样支持并行导入。开始导入前进行如下设置。Load Data 命令仅支持数据文件在 OBServer 节点本地执行，如果您希望远程进行数据导入，可以参考使用 OceanBase 数据库的 obloader 工具。

```
obclient [test]> SET GLOBAL secure_file_priv = "/";
obclient [test]> GRANT FILE ON *.* to username;
```

14.2.4.1 注意

由于安全原因，只能使用通过本地连接的 Client 执行修改 `secure_file_priv` 的 SQL 语句。详细信息请参见 secure_file_priv

设置完成后，重连会话使设置生效。再设置下会话的事务超时时间，保证执行过程中不会因为超时退出。

```
obclient [test]> SET ob_query_timeout=1000000000;
obclient [test]> SET ob_trx_timeout=1000000000;
```

然后运行导入语句：

```
obclient [test]> LOAD DATA /*+ parallel(16) */ INFILE '/home/soft/t_f1.csv' INTO table
t_f1 fields TERMINATED BY '\|' LINES TERMINATED BY '\n';
```

可以看到，开启并行导入后，10 GB 数据耗时大约 4 分钟。本文中租户的 CPU 并行度配置为 16，可以根据您的具体配置设置合适的并行度，配置越高导入速度越快。

导入后，进入数据库对该表记录条数及占用空间大小进行查看。

1. 查看表记录数有 5 千万条。

```
obclient [test]> SELECT COUNT(*) FROM t_f1;
+----------+
| count(*) |
+----------+
```

```
| 50000000 |
+----------+
```

2. 对数据库合并。

   为了查看基线数据的压缩效果，使用 `root` 用户登录集群的 `sys` 租户，主动触发对数据库进行合并，使增量数据可以和基线数据进行合并与压缩。您可以通过如下的方式手动触发合并。

```
# obclient -h127.0.0.1 -P2881 -uroot@sys -Doceanbase -A -p -c
obclient[oceanbase]> ALTER SYSTEM MAJOR FREEZE;
```

3. 当看到如下查询返回 IDLE 时，表示合并完成。

```
obclient [oceanbase]> SELECT * FROM oceanbase.CDB_OB_MAJOR_COMPACTION;
+-----------+--------------------+---------------------------+----------------------
+--------------------+---------------------------+---------------------------
+--------+----------+--------------+------+
| TENANT_ID | FROZEN_SCN | FROZEN_TIME | GLOBAL_BROADCAST_SCN | LAST_SCN |
LAST_FINISH_TIME | START_TIME | STATUS | IS_ERROR | IS_SUSPENDED | INFO |
+-----------+--------------------+---------------------------+----------------------
+--------------------+---------------------------+---------------------------
+--------+----------+--------------+------+
| 1 | 1679248800404017149 | 2023-03-20 02:00:00.404017 | 1679248800404017149
| 1679248800404017149 | 2023-03-20 02:00:44.035785 | 2023-03-20 02:00:
00.442481 | IDLE | NO | NO | |
| 1001 | 1679248804191204504 | 2023-03-20 02:00:04.191205 |
1679248804191204504 | 1679248804191204504 | 2023-03-20 02:00:46.094551 |
2023-03-20 02:00:04.218608 | IDLE | NO | NO | |
| 1002 | 1679248802450394471 | 2023-03-20 02:00:02.450394 |
1679248802450394471 | 1679248802450394471 | 2023-03-20 02:00:33.818514 |
2023-03-20 02:00:02.484814 | IDLE | NO | NO | |
1 row in set
```

4. 使用 sys 租户查询如下语句，可查看导入至 OceanBase 后的数据存储占用情况。

```
obclient [oceanbase]> select b.table_name,a.svr_ip,data_size/1024/1024/1024
from CDB_OB_TABLET_REPLICAS a,CDB_OB_TABLE_LOCATIONS b where a.
tablet_id=b.tablet_id and b.table_name='T_F1';
+------------+--------------+---------------------------+
| table_name | svr_ip | a.data_size/1024/1024/1024 |
+------------+--------------+---------------------------+
| t_f1 | xxx.xx.xxx.xx | 6.144531250000 |
+------------+--------------+---------------------------+
```

压缩后的表大小约为 6.145 G，压缩比为 = 10/6.145 = 1.62。

15 体验多租户特性

OceanBase 数据库具有多租户的特性，在集群层面实现了实例资源的池化。在 OceanBase 数据库中，每一个租户即一个实例（类比 MySQL Instance）。租户与租户之间数据、权限、资源隔离，每个租户拥有自己独立的访问端口及 CPU、内存访问资源。

15.1 背景信息

OceanBase 数据库可以灵活的调整租户资源分配情况（CPU、内存），并且整个过程对上层业务透明。通过多租户机制，OceanBase 集群可以帮助用户高效的利用资源，在保证可用性和性能的前提下，优化成本，并且做到按照需求弹性扩容。

为了帮助您更容易的理解 OceanBase 数据库多租户的概念。下面通过创建一个 OceanBase 数据库租户的例子分步骤进行说明。

15.2 创建资源规格 Unit Config

在 OceanBase 数据库中，资源单元 Unit 是一个租户使用 CPU、内存的最小逻辑单元，也是集群扩展和负载均衡的一个基本单位，在集群节点上下线，扩容、缩容时会动态调整资源单元在节点上的分布进而达到资源的使用均衡。而 Unit Config 则规定了一个 Unit 需要使用的计算存储资源（包含内存、CPU 和 IO 等）的规格，是一个配置信息。

因为其分布式的架构，一个 OceanBase 数据库租户可以在资源规格、资源池、副本类型，副本分布几个不同维度灵活定义，所以创建租户时，也需要按照 “unit config -> resource pool -> tenant” 的顺序进行创建和定义。

OceanBase 数据库在创建租户前，需要先确定租户的 Unit Config。您需要使用 `sys` 租户管理员，通过如下 SQL 语句来创建。

下面创建两个 Unit Config，分别为 `unit1` 和 `unit2` 并指定 CPU、内存、使用的最大及最小阈值。

- 创建 `unit1` 资源单元的 CPU、内存使用大小为 3 C、6 G。

```
obclient [oceanbase]> CREATE RESOURCE UNIT UNIT1 MAX_CPU =3,MIN_CPU =3 ,
MEMORY_SIZE ='6G';
```

- 创建 `unit2` 资源单元的 CPU、内存使用大小为 4 C、8 G。

```
obclient [oceanbase]> CREATE RESOURCE UNIT UNIT2 MAX_CPU =4,MIN_CPU =4 ,
MEMORY_SIZE ='8G';
```

15.3 创建资源池和关联 Unit Config

Unit Config 是一组租户的配置规格信息，而 Resource Pool 则是租户的资源实体，所以在这一步我们需要创建一个 Resource Pool，并且与 Unit Config 关联起来。

创建两个不同的资源池 `pool1` 、`pool2`，并为其分别指定到 `unit1` 、`unit2` 上。实现资源单元与资源池的对应。`UNIT_NUM` 表示在一个副本中指定多少个 `unit` 单元（同一个租户中，一个节点上最多只能有一个 `unit`），`ZONE_LIST` 则指定租户在当前集群中在哪几个副本进行部署。

本例中考虑到是单节点集群，我们都指定单个 Unit 和 Zone List。创建 Resource Pool 需要保证有足够的资源剩余，如果资源不足，您可以先尝试删除已有的 `test` 租户，或者将已有租户的 Unit Config 调整到更小。如何调整请参见 调整租户资源大小。

```
obclient [oceanbase]> CREATE RESOURCE POOL pool1 UNIT='UNIT1',UNIT_NUM=1,
ZONE_LIST=('zone1');
obclient [oceanbase]> CREATE RESOURCE POOL pool2 UNIT='UNIT2',UNIT_NUM=1,
ZONE_LIST=('zone1');
```

注意

上面的例子针对的是单节点的集群环境，如果您的集群有 3 个节点，那么 `ZONE_LIST` 的值应该为 `('zone1','zone2','zone3')`，其中 Zone 的名称需要根据您创建的情况具体填写。

15.4 根据创建的 Resource Pool 创建租户

在完成 Unit Config、Resource Pool 的创建，完成资源单元和资源池的对应后，就可以正常开始租户创建了。

本例中由于是单节点集群，所以我们只能创建单副本的租户。如果希望创建 3 副本的租户，OceanBase 集群至少需要 3 个节点。

定义一个名为 `tenant1` 的单副本租户，并规定字符集为 `utf8mb4`，使用 `pool1` 的资源池，`ob_tcp_invited_nodes` 是租户白名单定义，初始可以设置为 '%'，表示任意 IP 地址均可以访问，后期可以修改。

```
obclient [oceanbase]> CREATE TENANT IF NOT EXISTS tenant1 CHARSET='utf8mb4',
PRIMARY_ZONE='zone1', RESOURCE_POOL_LIST=('pool1') SET
ob_tcp_invited_nodes='%';
```

注意

上面的例子针对的是单节点的集群环境，只能创建单副本的租户。如果您的集群有 3 个节点，那么 PRIMARY_ZONE 则填写 'zone1;zone2;zone3'，表示租户的 Leader 优先分布在 zone1，其次为 zone2 。

类似的，定义一个名为 tenant2 的一个 单副本的租户，并规定字符集为 utf8mb4，使用 pool2 的资源池。

```
obclient [oceanbase]> CREATE TENANT IF NOT EXISTS tenant2 CHARSET='utf8mb4',
PRIMARY_ZONE='zone1', RESOURCE_POOL_LIST=('pool2') SET
ob_tcp_invited_nodes='%';
```

在完成 tenant1 、tenant2 租户创建后，您可以通过查询 oceanbase.DBA_OB_TENANTS 视图来确认租户是否创建成功。

```
obclient [oceanbase]> SELECT * FROM DBA_OB_TENANTS;
```

上述操作确认完成后，您已完成在同一个集群下实现两个租户的创建过程。接下来便可以在租户里正常的进行数据库操作了。

使用 root 用户登录集群的 tenant1 租户，并创建一张测试表。

```
obclient -hxxx.xxx.xxx.xxx -P2883 -uroot@tenant1#ob_test -p******


obclient [(none)]> SHOW DATABASES;
+--------------------+
| Database |
+--------------------+
| oceanbase |
| information_schema |
| mysql |
| test |
+--------------------+
4 rows in set
```

```
obclient [(none)]> USE test;
Database changed
obclient [test]> CREATE TABLE t_f1(id DECIMAL(10,0),id2 DECIMAL(10,0),id3 DATE,id4
DATE,id5 FLOAT,id6 FLOAT,id7 VARCHAR(30),id8 VARCHAR(300));
Query OK, 0 rows affected

obclient [test]> SHOW TABLES;
+----------------+
| Tables_in_test |
+----------------+
| t_f1 |
+----------------+
1 row in set
```

使用 `root` 用户登录集群的 `tenant2` 租户，查看 `test` 库情况。

```
obclient -hxxx.xxx.xxx.xxx -P2883 -uroot@tenant2#ob_test -p******

obclient [(none)]> SHOW DATABASES;
+--------------------+
| Database |
+--------------------+
| oceanbase |
| information_schema |
| mysql |
| test |
+--------------------+
4 rows in set
```

```
obclient [(none)]> USE test;
Database changed
obclient [test]> SHOW TABLES;
Empty set
```

可以看到集群下两个租户的资源、数据、权限都是隔离的，您可以进行更多测试，体验 OceanBase 数据库的租户隔离特性。

15.5 修改租户配置和调整实例资源规格

OceanBase 数据库可以灵活的对租户资源的 CPU，内存资源进行调整，并且在线生效，对业务透明。修改租户所占用的 CPU、内存大小，您仅需要调整租户所对应的 Unit Config，不需要对资源池或者租户进行调整。

租户资源单元可通过系统视图 `oceanbase.DBA_OB_UNIT_CONFIGS` 进行查看。

```
obclient [oceanbase]> SELECT * FROM oceanbase.DBA_OB_UNIT_CONFIGS;
+----------------+-----------------+---------------------------
+----------------------------+---------+---------+-------------+---------------+----------
+----------+-------------+
| UNIT_CONFIG_ID | NAME | CREATE_TIME | MODIFY_TIME | MAX_CPU | MIN_CPU |
MEMORY_SIZE | LOG_DISK_SIZE | MAX_IOPS | MIN_IOPS | IOPS_WEIGHT |
+----------------+-----------------+---------------------------
+----------------------------+---------+---------+-------------+---------------+----------
+----------+-------------+
| 1 | sys_unit_config | 2022-11-17 18:00:47.297945 | 2022-11-17 18:00:47.297945 | 1 |
1 | 8053063680 | 8053063680 | 10000 | 10000 | 1 |
| 1010 | unit1 | 2022-12-15 11:40:17.811095 | 2022-12-15 11:40:17.811095 | 3 | 3 |
6442450944 | 2147483648 | 128 | 128 | 0 |
| 1011 | unit2 | 2022-12-16 16:16:27.053872 | 2022-12-16 16:16:27.053872 | 4 | 4 |
8589934592 | 8589934592 | 128 | 128 | 0 |
+----------------+-----------------+---------------------------
+----------------------------+---------+---------+-------------+---------------+----------
```

```
+----------+-------------+
6 rows in set (0.003 sec)
```

查询结果显示，`unit1` 资源单元 CPU、内存使用的分别是 3 C，6 G。

通过以下命令调整 `unit1` 资源单元 CPU、内存的使用大小为 5 c，10 G。

```
obclient [oceanbase]> ALTER resource unit unit1 max_cpu =5,min_cpu =5 ,
memory_size ='10G';
Query OK, 0 rows affected

obclient [oceanbase]> SELECT * FROM oceanbase.DBA_OB_UNIT_CONFIGS;
+---------------+-----------------+---------------------------
+---------------------------+---------+---------+-------------+---------------+----------
+----------+-------------+
| UNIT_CONFIG_ID | NAME | CREATE_TIME | MODIFY_TIME | MAX_CPU | MIN_CPU |
MEMORY_SIZE | LOG_DISK_SIZE | MAX_IOPS | MIN_IOPS | IOPS_WEIGHT |
+---------------+-----------------+---------------------------
+---------------------------+---------+---------+-------------+---------------+----------
+----------+-------------+
| 1 | sys_unit_config | 2022-11-17 18:00:47.297945 | 2022-11-17 18:00:47.297945 | 1 |
1 | 8053063680 | 8053063680 | 10000 | 10000 | 1 |
| 1010 | unit1 | 2022-12-15 11:40:17.811095 | 2023-01-05 16:24:17.287801 | 5 | 5 |
10737418240 | 2147483648 | 128 | 128 | 0 |
| 1011 | unit2 | 2022-12-16 16:16:27.053872 | 2022-12-16 16:16:27.053872 | 4 | 4 |
8589934592 | 8589934592 | 128 | 128 | 0 |
+---------------+-----------------+---------------------------
+---------------------------+---------+---------+-------------+---------------+----------
+----------+-------------+
6 rows in set (0.037 sec)
```

如上所示，租户配置的调整是在线立即生效的，调整成功后，`unit1` 的 CPU、内存为 5 c，10 G。OceanBase 数据库通过内核的虚拟化技术，在变更配置后租户的 CPU 和内存资源可以立即生效，无需数据迁移或者切换，对业务无感知。

通过查看 `DBA_OB_UNITS` 信息，您可以获取到资源单元、资源池、租户在集群里的对应信息及 CPU、内存信息。

```
obclient [oceanbase]> SELECT * FROM DBA_OB_UNITS;
```