OCEANBASE | 海量记录 笔笔算数

数据库管理与运维 OceanBase 从入门到实践

OceanBase 研发团队

联系我们

欢迎 OceanBase 爱好者、用户和客户联系我们反馈问题：

- 社区版官网论坛：https://ask.oceanbase.com/
- 社区版项目网站提 Issue
  - MiniOB GitHub：https://github.com/oceanbase/miniob/issues
  - OceanBase GitHub：https://github.com/oceanbase/oceanbase/issues
  - OceanBase Gitee：https://gitee.com/oceanbase/oceanbase/issues
- 技术交流及关注动态：群号 33254054

钉钉技术交流群
（群号：33254054）

微信 OB 小助手

扫码加入群聊

目 录

第一章 OceanBase 数据库概述 5

1.1 OceanBase 数据库概述 6

1.2 OceanBase 生态工具介绍 14

第二章 OceanBase 数据库社区版的安装和部署 22

2.1 部署前准备 23

2.2 部署相关的生态组件介绍 42

2.3 部署个人实验环境 44

2.4 部署生产环境 69

2.5 查看 OceanBase 集群资源的使用情况 80

2.6 创建 MySQL 模式的用户租户 86

2.7 连接租户 103

2.8 设置参数和变量 109

第三章 测试 OceanBase 数据库 117

3.1 OceanBase 数据库测试概述 118

3.2 OceanBase 数据库性能的影响因素 120

3.3 进行 Sysbench 测试 128

3.4 进行 TPC-C 测试 137

3.5 进行 TPC-H 测试 154

3.6 使用 JMeter 运行业务场景测试 171

3.7 其他常见测试点 182

第四章 OceanBase 数据库的迁移和同步 187

4.1 OceanBase 数据库和 MySQL 兼容性介绍 188

4.2 迁移同步相关生态组件介绍 193

4.3 通过 OMS 进行数据迁移和同步 199

4.4 通过 oblogproxy 进行增量日志代理服务 210

4.5 使用导数工具进行数据迁移 213

4.6 使用 SQL 命令进行数据迁移 220

4.7 通过其他工具进行数据的迁移同步 241

第五章 运维 OceanBase 数据库 254
5.1 使用 OCP 进行运维 255
5.2 使用 obd 进行运维 280
5.3 使用 ob-operator 进行运维 299
5.4 使用命令行进行运维 313
第六章 使用 OceanBase 数据库进行业务开发 359
6.1 使用 MySQL 租户做常见数据库开发 360
6.2 通过 ODC 图形化开发工具进行 SQL 开发 418
6.3 使用 OceanBase 数据库分区表进行水平拆分 440
6.4 OceanBase 数据库在 MySQL 模式租户下的扩展功能 454
第七章 OceanBase 数据库的诊断和调优 485
7.1 诊断调优概述 486
7.2 ODP SQL 路由原理 487
7.3 管理 OceanBase 数据库连接 509
7.4 分析 SQL 监控视图 524
7.5 阅读和管理 OceanBase 数据库 SQL 执行计划 551
7.6 常见的 SQL 调优方式 609
7.7 SQL 性能问题的典 675
7.8 通过 SQL Diagnoser 工具进行 SQL 性能诊断和分析 700
7.9 通过 obdiag 工具进行诊断和分析 706
第八章 OceanBase 数据库故障排查和诊断 730
8.1 遇到问题如何在官网上进行自主排查 731
8.2 遇到问题如何向技术支持同学提问 735
8.3 常见的故障及其恢复手段 738
第九章 OceanBase 集群运维管理之用户实操 742
致谢 752

第一章 OceanBase 数据库概述

本章介绍影响 OceanBase 数据库性能的因素，以及如何对 OceanBase 数据库进行各种测试。

本章目录

1.1 OceanBase 数据库概述 6

1.2 OceanBase 生态工具介绍 14

1.1 OceanBase 数据库概述

本章简要概述 OceanBase 数据库产品核心特性、社区版和企业版的差异、重要版本如 4.x 版本和 3.x 版本的系统表和系统视图变更。

说明

本教程中所贴官网文档链接版本均为编写教程时的最新版本，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

OceanBase 数据库概述

OceanBase 数据库（OceanBase Database）是一款完全自研的原生分布式数据库，已连续 11 年稳定支撑双 11，创新推出“三地五中心”城市级容灾新标准，是目前全球唯一在 TPC-C 和 TPC-H 测试上都刷新了世界纪录的原生分布式数据库。产品采用自研的一体化架构，兼顾分布式架构的扩展性与集中式架构的性能优势，用一套引擎同时支持 TP 和 AP 的混合负载，具有数据强一致、高可用、高性能、在线扩展、高度兼容 Oracle/MySQL、对应用透明、高性价比等特点。14 年持续深耕海量核心场景，已助力金融、政务、运营商、零售、互联网等多个行业的 1000+ 客户实现关键业务系统升级。

核心特性

高可用

首创“三地五中心”容灾架构方案，建立金融行业无损容灾新标准。支持同城/异地容灾，可实现多地多活，满足金融行业 6 级容灾标准（RPO=0，RTO<8s），数据零丢失。

高兼容

社区版高度兼容 MySQL，覆盖绝大多数常见功能，支持过程语言、触发器等高级特性，提供自动迁移工具，支持迁移评估和反向同步以保障数据迁移安全，可支撑金融、政府、运营商、互联

网等行业关键业务场景。

水平扩展

实现透明水平扩展，支持业务快速的扩容缩容，同时通过准内存处理架构实现高性能。支持集群节点数千个，单集群最大数据量超过 3PB，最大单表行数达万亿级。

低成本

基于 LSM-Tree 的高压缩引擎，实现存储成本降低 70% ~ 90%。同时，由于原生支持多租户架构，同集群可为多个独立业务提供服务，且租户间数据隔离，降低部署和运维成本。

实时 HTAP

基于“同一份数据，同一个引擎”，同时支持在线实时交易及实时分析两种场景，“一份数据”的多个副本可以存储成多种形态，用于不同工作负载，从根本上保持数据一致性。

安全可靠

自 2010 年开始完全自主研发，代码级可控，自研单机分布式一体化架构，连续多年通过大规模金融核心场景的可靠性验证。

OceanBase 数据库社区版概述

OceanBase 数据库社区版使用 MulanPubL-2.0 许可证，您可以免费复制及使用源代码。当您修改或分发源代码时，请遵守木兰协议。

OceanBase 数据库社区版官方网站地址：https://open.oceanbase.com。

社区版核心功能

OceanBase 数据库社区版包含 OceanBase 数据库企业版的所有核心功能，具体如下：

- 多副本高可用、强同步能力。
- 多租户能力。

- 在线弹性伸缩能力。
- 异地容灾/多活能力（包括两地三中心、三地五中心等）。
- 分区表、复制表等分布式能力。
- HTAP 能力。
- MySQL 兼容性。
- 备份恢复能力。
- CDC 能力。

下载方法

- 官网下载：https://open.oceanbase.com/softwareCenter/community
- GitHub 下载：https://github.com/oceanbase/oceanbase/releases/
- 阿里云 Yum 源：https://mirrors.aliyun.com/oceanbase/OceanBase.repo

支持的操作系统

OceanBase 数据库社区版支持的操作系统详见《OceanBase 数据库》文档 部署数据库/部署 OceanBase 社区版/本地部署/软硬件要求 一文。

与 MySQL 数据库的不同

- OceanBase 数据库底层原理与 MySQL 无关，不依赖开源 MySQL 组件，不涉及 InnoDB 引擎等。但 OceanBase 数据库社区版兼容 MySQL 语法功能（兼容 MySQL 5.6、MySQL 5.7 的绝大部分语法，部分 MySQL 8.0 的新特性）。
- OceanBase 数据库自身的存储引擎与 MySQL 的存储引擎相比，空间压缩效果更明显，存储成本降低 70%~90%。
- OceanBase 数据库是分布式数据库集群产品，生产环境默认三副本，并且三副本之间的同步协议不是异步同步或半同步，而是使用 Paxos 协议同步事务日志。OceanBase 集群可以跨机房跨城市部署，机器或者机房故障时，集群内部多副本自动切换，不丢数据。因此 OceanBase 数据库天然适合两地三中心异地容灾和多活建设。

- OceanBase 集群支持多租户（也叫多实例），所有的租户按需分配，弹性伸缩，具备高可用能力，类似于云数据库服务。运维人员只需要维护少数几套集群，就可以提供很多实例给业务使用，易用性非常好。
- OceanBase 数据库支持水平拆分技术，具体体现为分区表，不需要分库分表，SQL 和事务对业务完全透明，功能上没有限制。分区表线性扩展性较好，目前已知案例最大单租户节点规模是 1500 台。
- OceanBase 数据库的 SQL 引擎比 MySQL 更加强大：支持对执行计划进行缓存，减少对相同 SQL 反复生成同样的执行计划带来的开销；支持通过使用 hint 和 outline 对 SQL 执行计划的形态进行干预；支持分布式场景和复杂场景下的计划生成和 SQL 计算等等。同时 OceanBase 数据库也支持 OLTP 和 OLAP 类型的混合场景需求，即通常说的 HTAP 能力。

适合社区版的业务场景

- 场景一：MySQL 5.6/5.7 实例规模较大

  MySQL 实例规模较大，需要自动化运维平台的场景下。自动化运维平台在处理 MySQL 异常宕机切换和主备不一致问题时很可能需要 DBA 介入。高可用和强一致问题是 MySQL 的风险隐患所在，OceanBase 数据库的多租户、高可用和强一致能力可以彻底解决这个痛点。

- 场景二：MySQL 5.6/5.7 数据量大、存储成本高

  MySQL 业务数据量增长到数 TB 时，查询和读写性能可能会下降，大表 DDL 时间变长，风险增加，单机磁盘容量可能到达扩容瓶颈。OceanBase 数据库 MySQL 租户的在线 DDL、数据存储高压缩比可以解决这些痛点。

- 场景三：业务访问压力大或者变化多

  基于 MySQL 改造的分布式数据库中间件产品能分担一定程度的业务压力和存储空间压力，但是缺乏跨节点的强一致性查询，并且需要分布式事务中间件协调事务，扩容的时候可能需要数据逻辑拆分（俗称拆库拆表），不仅运维成本高，风险也高。OceanBase 数据库 MySQL 租户提供分区表的水平拆分方案，提供原生的 SQL 和事务能力，对业务透明。并且 OceanBase 数据库支持在线扩容和缩容，内部数据迁移异步进行，具备高可用能力，不怕扩容和缩容过程中出现故障，可以解决上面这些痛点。

- 场景四：交易数据库上的复杂查询

交易数据库上有少量复杂的查询场景，涉及到的数据量很大，传统解决方案是通过数据同步到数据仓库进行查询。OceanBase 数据库的 SQL 引擎同时满足 OLTP 和 OLAP 场景，采用经过 Oracle 复杂业务场景检验的先进的 SQL 优化器技术，能支持复杂的 SQL 优化和高效执行。因此可以在交易数据库上直接做复杂查询，减少不必要的数据同步。此外，OceanBase 数据库还提供不同程度的读写分离技术来控制复杂查询对交易场景的影响。

- 更多的场景实践详见官网 企业案例。

OceanBase 数据库社区版和企业版的功能差异

OceanBase 数据库社区版跟企业版的差异在于企业版会包含更多高级功能，如商业特性兼容、操作审计、安全加密等。以 V4.2.2 为例，企业版和社区版支持的功能如下所示：

类目	功能	企业版	社区版
核心组件	一体化 SQL 引擎	支持	支持
	一体化事务引擎	支持	支持
	一体化存储引擎	支持	支持
	集群调度服务	支持	支持
	集群代理服务	支持	支持
	客户端与 C 驱动和 Java 驱动	支持	支持
高可用	支持多副本	支持	支持
	三地五中心部署	支持	支持
	透明水平扩展	支持	支持
	多租户管理	支持	支持
	数据备份恢复	支持	支持
	资源隔离	支持	支持
	物理备库	支持	支持
	仲裁服务	支持	不支持
兼容性	MySQL 语法和协议兼容	支持	支持
	数据类型与函数兼容	支持	支持
	存储过程与包	支持	支持

	复杂字符集	支持	支持
	Oracle 语法兼容	支持	不支持
	XA 事务	支持	不支持
	表锁	支持	不支持
	函数索引	支持	支持
高性能	基于代价的优化器	支持	支持
	复杂查询优化改写	支持	支持
	并行执行引擎	支持	支持
	向量化引擎	支持	支持
	高级执行计划管理（SPM）	支持	不支持
	小规格	支持	支持
	基于 Paxos 协议的日志传输	支持	支持
	分布式强一致事务、完整 ACID、支持多版本	支持	支持
	数据分区（Range/Hash/List）	支持	支持
	全局索引	支持	支持
	高级压缩能力	支持	支持
	动态采样	支持	支持
	Auto DOP	支持	支持
跨数据源	只读外表（CSV）	支持	支持
	DBLink	支持	支持
多模	TableAPI	支持	支持
	HbaseAPI	支持	支持
	JSON	支持	支持
	GIS	支持	支持
	XML	支持	不支持
安全	审计	支持	不支持
	权限管理	支持	支持

	通信加密	支持	支持
	高级安全扩展能力	支持	不支持 社区版本不支持行级标签、数据和日志加密存储（TDE）。
运维管理	全链路诊断	支持	支持
	运维组件（liboblog、ob_admin）	支持	支持
	导数工具（obloader/obdumper）	支持	支持
	图形化开发及管控工具	支持	支持 社区版本支持 OCP、OMS、ODC 等商业配套图形化开发和管控工具二进制免费下载使用，但不包含 OMA。
支持与服务	技术咨询（产品技术咨询服务）	支持	社区版本仅提供社区化的产品技术咨询服务，采用社区 issues 运作模式，不提供商业化专家团队技术咨询。
	服务获取（获取技术支持的渠道）	专业商业支持团队	社区版本仅支持在 OceanBase 社区官方网站或官方社区提供在线服务咨询，不提供商业化专家团队专属服务。
	专家服务（规划、实施、巡检、故障恢复、生产保障）	商业专家驻场服务	社区版本不提供专家保障服务。
	故障响应	7*24 服务	社区版本不提供故障应急处理服务。

社区版和企业版不同版本的具体功能差异详见官网《OceanBase 数据库》文档对应版本 OceanBase 简介/企业版和社区版的功能差异 一文。

OceanBase 数据库社区版产品动态

OceanBase 数据库社区版产品动态详见官网 版本发布记录。

OceanBase 数据库 4.x 版本和 3.x 版本的系统表和系统视图变更

相较于 OceanBase 数据库 3.x 版本，OceanBase 数据库 4.x 版本在稳定性、性能等方面有极大提升，并新增许多功能，其中，我们对内部表和虚拟表做了很多改造，同时提供了系统视图用于信息展示。简言之，4.x 版本最大的变化是内部信息的查询全面转向视图，统一定义的视图会

保持版本间兼容，信息也更清晰。

V4.x 和 V3.x 的具体系统视图变更详见官网《OceanBase 数据库》文档 参考指南/系统视图/3.x 与 4.x 视图变更 一文。

1.2 OceanBase 生态工具介绍

本节简单介绍 OceanBase 的各个生态工具，包括安装部署工具、监控工具、迁移同步工具等。

说明

本教程中所贴官网文档链接版本均为编写教程时的最新版本，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

数据库代理

ODP

OceanBase 数据库代理（OceanBase Database Proxy，简称 ODP，又称为 OBProxy）是 OceanBase 数据库专用的代理服务器，与本文中介绍的其他相对独立的生态工具不同，ODP 属于 OceanBase 数据库内核（OBServer+ODP）的一部分。OceanBase 数据库用户的数据会以多副本的形式存放在各个 OBServer 节点上，ODP 接收用户发出的 SQL 请求，并将 SQL 请求转发至最佳目标 OBServer 节点，最后将执行结果返回给用户。ODP 的核心特性包括：连接管理、最佳路由、高性能转发、易运维、高可用，具有专有协议。

ODP 的相关内容详见官网 OceanBase 数据库代理 文档。

安装部署工具

OCP

OceanBase 云平台（OceanBase Cloud Platform，简称 OCP）是一款为 OceanBase 数据库集群量身打造的企业级管理平台，兼容 OceanBase 数据库所有主流版本。OCP 提供对 OceanBase 集群的图形化管理能力，包括数据库组件及相关资源（主机、网络和软件包等）的全生命周期管理、故障恢复、性能诊断、监控告警等。OCP 旨在协助客户更加高效地管理 OceanBase 集群，降低企业的 IT 运维成本和用户的学习成本。

OCP 的系统架构和功能特性详见官网 OceanBase 云平台 文档。

OCP Express

OCP Express 是一个基于 Web 的 OceanBase 4.x 管理工具，它集成在 OceanBase 数据库集群中，支持数据库集群关键性能指标查看及基本数据库管理功能。

OCP Express 脱胎于 OceanBase 云平台（OceanBase Cloud Platform，OCP），在保留云平台核心能力的基础上，调整了整体的功能布局，给予用户全新的使用体验；同时也进一步调整了功能配置，使得 OCP Express 可以以极小的资源消耗部署在任意一台数据库节点上，使得 OCP Express 用户以最小的成本，获取全新的 OceanBase 4.x 管控体验。

OCP Express 面向轻量级运维管控，它与被管理的 OceanBase 集群深度融合，实行一对一专属管理，可以做到在满足基本运维管理需求的前提下尽量地降低资源消耗。OCP 面向大规模的复杂集群的运维场景，可以实现一个平台管理多套 OceanBase 集群，支持更丰富的管理能力，需要额外配置 OceanBase 集群作为 OCP 的数据存储，同时也需要更高资源配置来实现平台能力。

> **说明**
>
> OCP Express 是面向开发测试环境的 OceanBase 集群运维管理，适合单个集群环境在 20 台主机、3~5 个租户以下的规模时使用。生产环境推荐使用 OCP。

OCP Express 的系统架构和功能特性详见官网《OceanBase 数据库》文档中 参考指南/平台产品/OceanBase 云平台 Express（OCP Express） 章节。

obd

obd 全称为 OceanBase Deployer，是 OceanBase 安装部署工具，通过命令行部署或白屏界面部署的方式，将复杂配置流程标准化，降低集群部署难度。

其中，命令行支持编辑配置文件，可以更加灵活地进行配置调整，适用于深度了解 OceanBase 数据库的用户，有一定的使用门槛；白屏界面配置简单，通过页面的引导配置即可完成集群部署，适用于需要快速体验，构建标准环境的用户。

在集群部署之外，obd 还提供了包管理器、压测软件、集群管理等常用的运维能力，更好地支持用户使用 OceanBase 分布式数据库。

obd 的使用指南详见官网 OceanBase 安装部署工具 文档。

ob-operator

ob-operator 是一款基于 Kubernetes Operator 框架构建的工具，用于在 Kubernetes 中管理 OceanBase 集群。它提供了一种简单可靠的方式来实现 OceanBase 集群的容器化部署，可以简化 OceanBase 数据库的运维。ob-operator 定义了 OceanBase 数据库相关的各种资源并且实现相应的调协逻辑，因此，用户可以像管理 Kubernetes 的原生资源一样，通过声明式的方式管理 OceanBase 集群。

ob-operator 的功能特点和使用指导详见官网 OceanBase K8s 运维工具 文档。

监控工具

OCP/OCP Express

OCP 和 OCP Express 除了对集群、租户等进行创建和管理以外，还支持性能诊断、监控告警等功能。详细信息可参见上文 安装部署工具 中对 OCP 和 OCP Express 的介绍。

DOOBA

DOOBA 是 OceanBase 数据库内部采用 Python 语言开发的一个运维脚本，用于性能监控。DOOBA 的原理是使用 MySQL 命令连接到 OceanBase 数据库的 sys 租户中，实时展示租户 SQL 的 QPS（包括 `select`、`update`、`insert`、`delete`、`commit`）以及相应 SQL 的平均延时（RT），同时还可以展示各个节点 SQL 的 QPS 以及 RT 等信息。

DOOBA 的相关内容详见官网《OceanBase 数据库》文档中 参考指南/组件 & 工具/监控诊断/DOOBA 一文。

OBAgent

OBAgent 是一个监控采集和运维框架。

- 监控部分

OBAgent 支持推、拉两种数据采集模式，可以满足不同的应用场景。OBAgent 默认支持的插件包括主机数据采集、OceanBase 数据库指标采集、日志信息采集、监控数据标签处理和 Prometheus 协议的 HTTP 服务，支持推送数据到 pushgateway、vmagent、es、sls、alertmanager。您也可开发对应的插件，使 OBAgent 支持其他数据源的采集，或者自定义数据的处理流程。

- 运维部分

  OBAgent 支持配置热更新，同时提供了对 Agent 自身运维的接口，以及文件操作和 RPM 包操作的接口，您可根据自身需求来使用这些接口。

OBAgent 的相关内容详见 GitHub 中 obagent 仓库。

迁移同步工具

OMS

OceanBase 迁移服务（OceanBase Migration Service，OMS）是 OceanBase 提供的一种支持同构或异构数据源与 OceanBase 数据库之间进行数据交互的服务，具备在线迁移存量数据和实时同步增量数据的能力。

OMS 社区版提供可视化的集中管控平台，用户只需要进行简单的配置即可实时迁移数据。OMS 社区版旨在帮助用户低风险、低成本、高效率地实现同构或异构数据库向 OceanBase 数据库进行实时数据迁移和数据同步。

OMS 的相关内容详见官网 OceanBase 迁移服务 文档。

oblogproxy

oblogproxy 是 OceanBase 的增量日志代理服务，它可以与 OceanBase 数据库建立连接并进行增量日志读取，为下游服务提供了变更数据捕获（CDC）的能力。

oblogproxy 有如下两种模式。

- Binlog 模式为 OceanBase 数据库兼容 MySQL binlog 而推出，支持现有的 MySQL binlog 增量解析工具实时同步 OceanBase 数据库，使 MySQL binlog 增量解析工具可以平滑切换

到 OceanBase 数据库。

- CDC 模式用于解决数据同步，CDC 模式下 oblogproxy 可以订阅 OceanBase 数据库中的数据变更，并将这些数据变更实时同步至下游服务，实现数据的实时或准实时复制和同步。

oblogproxy 的相关内容详见官网 OceanBase 日志代理服务 文档。

obloader/obdumper

OceanBase 导数工具包括导入工具 obloader 和导出工具 obdumper。

obloader 提供了非常灵活的命令行选项，可以在多种复杂的场景下，将数据库对象定义和表数据导入到 OceanBase 数据库中。通常建议 obloader 与 obdumper 一起搭配使用。在外部的业务中，obloader 同时也支持将第三方工具（如：Navicat、Mydumper 和 SQLDeveloper 等）导出的 CSV 格式的文件导入 OceanBase 数据库。obloader 充分利用 OceanBase 数据库分布式系统的特性，重点优化了导入的性能。

obdumper 可以将 OceanBase 数据库中的表数据按照 SQL 或者 CSV 等格式导出到文件。同时，可以使用该工具将数据库中定义的对象导出到文件。

obloader/obdumper 的相关内容详见官网 OceanBase 导数工具 文档。

运维工具

OCP/OCP Express

OCP 除了对集群、租户等进行创建和管理以外，还支持故障恢复、性能诊断、监控告警等功能。

OCP Express 相比于 OCP，不支持一些高级得运维能力，例如备份恢复、版本升级、集群扩展等，以此来减少资源的占用。OCP Express 相比 OCP 更加轻量，目的是满足单个集群得基础运维监控需求。生产环境推荐使用 OCP。

详细信息可参见上文 安装部署工具 中对 OCP 和 OCP Express 的介绍。

obshell

obshell（OceanBase Shell）是 OceanBase 社区为运维人员和开发人员提供的免安装、开箱

即用的本地集群命令行工具。obshell 支持集群运维，同时基于 OceanBase 数据库对外提供运维管理 API，实现了不同生态产品对同一集群统一管理，从而方便第三方产品工具对接 OceanBase 数据库，同时降低了 OceanBase 集群管理难度和成本。

obshell 不需要额外安装，用户在通过任何方式安装 OceanBase-CE 数据库后，都可以在任何一个 OBServer 节点工作目录的 `bin` 目录下看到 obshell 可执行文件。

obshell 的相关内容详见官网《OceanBase 数据库》文档中 参考指南/组件 & 工具/运维管理/obshell 章节。

obdiag

obdiag 是一款适用于 OceanBase 数据库的黑屏诊断工具，现有功能包含了对 OceanBase 数据库日志、SQL Audit 以及 OceanBase 数据库进程堆栈等信息进行扫描、收集和分析，可以在 OceanBase 集群不同的部署模式下（OCP，obd 或根据文档手工部署）实现一键执行，完成诊断信息的收集和分析。

obdiag 支持黑屏命令行一键巡检，能够及时发现集群中已存在或潜在可能导致异常问题的情况，并对其进行分析，同时提供针对性的运维建议；支持黑屏命令行一键收集散落在各个节点上的诊断信息，打包回传到 obdiag 机器上；支持黑屏命令行一键对 OceanBase 数据库的日志进行分析，找出发生过的错误信息，同时还支持基于 trace.log 的一键全链路诊断等功能。

obdiag 的相关内容详见官网 OceanBase 诊断工具 文档。

ob_error

ob_error 是 OceanBase 数据库的一个错误码解析工具，ob_error 可以根据输入的错误码返回相对应的原因和解决方案。在 ob_error 的帮助下，您无需查找文档即可获取基本的错误信息。

ob_error 的相关内容详见官网《OceanBase 数据库》文档中 参考指南/组件 & 工具/运维管理/ob_error 一文。

ob_admin

ob_admin 是 OceanBase 数据库的配套运维工具之一，提供了 slog_tool、log_tool、dumpsst 和 dump_backup 功能，主要用于排查数据不一致、丢数据、错误数据等问题。

ob_admin 的相关内容详见官网《OceanBase 数据库》文档中 参考指南/组件 & 工具/运维管理/ob_admin 章节。

图形化开发工具

ODC

OceanBase 开发者中心（OceanBase Developer Center，ODC）是数据库图形化开发工具，也是数据研发和生产变更管控协同平台。ODC 有桌面版、Web 版两种产品形态：桌面版侧重数据库开发工具能力，支持 Windows、Mac、Linux 操作系统，具有轻量化和易部署的特性；Web 版在提供工具能力的同时还提供了管控、协同能力，侧重数据库变更的安全、合规和效率。

ODC Web 版提供了个人空间和团队空间两种工作模式。个人空间适合个人开发者使用，为您在浏览器端提供桌面版的体验，您可以自由访问，新建数据源并使用平台提供的各种窗口、工具对数据库进行开发。团队空间适合开发者和 DBA 协同使用，既是开发工具也是管控协同平台，ODC 通过项目协同、稳定变更、数据安全、冷热数据分离等一系列功能，为您提供数据库开发管控一站式服务。

ODC 的相关内容详见官网 OceanBase 开发者中心 文档。

> **说明**
>
> 除 ODC 外，您也可以使用第三方的图形化开发工具连接 OceanBase 数据库，例如 Navicat、DBeaver。

数据库驱动

OceanBase Connector/J

OceanBase Connector/J 是一种实现 JDBC API 的驱动程序，为基于 Java 开发的应用程序提供与 OceanBase 数据库的连接。该驱动程序属于 JDBC Type 4 驱动类型，可以通过本地协议直接与数据库引擎通信。

OceanBase 数据库支持 OceanBase Connector/J 驱动，同时完全兼容 MySQL 原生的 JDBC 驱动（MySQL Connector Java）。OceanBase Connector/J 完全兼容 MySQL JDBC 的使用方式，可以自动识别 OceanBase 数据库的运行模式是 MySQL 还是 Oracle，并在协议层同时兼容这两种模式。

OceanBase Connector/J 的相关内容详见官网 OceanBase JDBC 驱动 文档。

OceanBase Connector/C

OceanBase Connector/C 是一个基于 C/C++ 的 OceanBase 客户端开发组件，支持 C API Lib 库。该组件允许 C/C++ 程序以一种较为底层的方式访问 OceanBase 分布式数据库集群，以进行数据库连接、数据访问、错误处理和 Prepared Statement 处理等操作。

OceanBase Connector/C 也称为 libobclient，可用于应用程序中，使应用程序能够作为独立的服务器进程通过网络连接与 OceanBase 数据库服务器进行通信。客户端程序在编译时会引用 C API 头文件，同时可以连接到 C API 库文件。

OceanBase Connector/C 的相关内容详见官网 OceanBase C API 库 文档。

OceanBase Connector/ODBC

Connector/ODBC 开放数据库连接（Open Database Connectivity，ODBC）是为解决异构数据库间的数据共享而创建，现已成为 Windows 开放系统体系结构（The Windows Open System Architecture，WOSA）的主要部分和基于 Windows 环境的一种数据库访问接口标准。

OceanBase Connector/ODBC的相关内容详见官网 OceanBase ODBC 驱动程序 文档。

第二章 OceanBase 数据库社区版的安装和部署

本章介绍如何手动和自动部署 OceanBase 社区版集群，包括单副本和三副本集群。

本章目录

2.1 部署前准备 23

2.2 部署相关的生态组件介绍 42

2.3 部署个人实验环境 44

2.4 部署生产环境 69

2.5 查看 OceanBase 集群资源的使用情况 80

2.6 创建 MySQL 模式的用户租户 86

2.7 连接租户 103

2.8 设置参数和变量 109

2.1 部署前准备

OceanBase 数据库作为原生分布式数据库，在学习环境中可部署单机版本，在生产环境中至少要求三台机器。OceanBase 数据库的部署与传统数据库的部署存在许多相通之处，都需要对操作系统硬件、软件设置、文件系统等进行调整和优化，使 OceanBase 数据库能够发挥其稳定运行、高性能的特性。

软硬件资源

OceanBase 数据库在以下配置进行了系统性测试验证，推荐您使用下表展示的配置：

项目	要求	说明
服务器	支持主流服务器，适配常用国内软件。	目前适配情况如下： • 已适配基于硬件整机中科可控 H620 系列、华为 TaiShan 200 系列、长城擎天 DF720 等整机。 • 已适配支持海光 7185/7280、鲲鹏 920、飞腾 2000+ 等 CPU。 • 已适配支持麒麟 V4、V10 和 UOS V20 等国产操作系统，并适配上层中间件东方通 TongWeb V7.0、金蝶 Apusic 应用服务器软件 V9.0 等。
系统	支持 x86、ARM 架构。	支持如下系统： • Alibaba Cloud Linux 2/3 版本（内核 Linux 3.10.0 版本及以上） • Anolis OS 8.X 版本（内核 Linux 3.10.0 版本及以上） • Red Hat Enterprise Linux Server 7.X 版本、8.X 版本（内核 Linux 3.10.0 版本及以上） • CentOS Linux 7.X 版本、8.X 版本（内核 Linux 3.10.0 版本及以上） • Debian 9.X 版本及以上版本（内核 Linux 3.10.0 版本及以上） • Ubuntu 20.X 版本及以上版本（内核 Linux 3.10.0 版本及以上） • SUSE/openSUSE 15.X 版本及以上版本（内核

		Linux 3.10.0 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	• 测试环境最低要求 2 核。 • 生产环境最低要求 4 核，推荐 32 核及以上。 • 性能测试场景推荐 24 核及以上。	此处介绍的是可分配给 `OceanBase` 数据库的最低核数，非服务器总核数。
内存	• 测试环境最低要求 6 GB。 • 生产环境最低要求 16 GB，长期使用要求不低于 32 GB，推荐设置在 256 GB 至 1024 GB 范围内。 • 性能测试场景推荐设置在 128 GB 至 1024 GB 范围内。	此为可分配给 `OceanBase` 数据库的最低内存，非服务器总内存大小。 说明 当部署多个集群时，推荐使用 OCP 进行统一运维管理；当部署集群比较少时，建议使用 obd 进行安装部署。
SWAP	禁止使用 SWAP 功能。	SWAP 交换分区会影响整个集群性能。
磁盘类型	使用 SSD 存储。	不推荐生产环境或性能测试环境使用机械磁盘。
文件系统	支持 xfs、ext4 文件系统。	不支持其他文件系统，数据大于 16 TB 时，推荐使用 xfs 文件系统。
磁盘容量	总磁盘容量需为内存的 6 倍以上。	无
磁盘挂载	数据盘空间大小最低为 20 GB。个人实验环境下，日志盘空间大小最低为 24 GB；生产环境中日志盘空间大小最低为 48 GB。	数据盘和日志盘的配置说明如下： • 日志盘和数据盘总体容量推荐是分配给 OceanBase 数据库内存大小的 6 倍以上（即 `memory_limit` * 6）。 • 日志盘大小推荐根据分配给 OceanBase 数据库总内存的 3 倍进行规划（即 `memory_limit` * 3）。 • 数据盘大小可根据业务数据量评估。 • 生产环境下数据盘和日志盘一定要分盘，否则会影响性能。

RAID	支持	RAID 阵列缓存需要使用 write through 模式。
网卡	最低千兆，推荐万兆	无

说明

- 更多软硬件平台适配正在测试中。
- 如果您使用多台机器部署 OceanBase 集群，请确保集群内的所有机器配置相同。

对于 Red Hat 操作系统 RHEL 9 之前版本，需要运行以下命令，手动关闭透明大页：

```
echo never > /sys/kernel/mm/redhat_transparent_hugepage/enabled
```

对于 Red Hat 操作系统 RHEL 9 版本或 CentOS 操作系统，需要运行以下命令，手动关闭透明大页：

```
echo never > /sys/kernel/mm/transparent_hugepage/enabled
```

命令依赖要求

环境包	要求	原因
nc	强依赖	OCP 部署 OceanBase 数据库依赖 nc 命令
net-tools	强依赖	OCP 部署 OceanBase 数据库依赖 netstat 命令
python	python2.7 及其以上	OCP 添加主机时依赖，并且不能是 python2 命令
jdk	java-1.8.0-openjdk	obd 部署 OCP Express 和 OCP 依赖
ntp/Chrony	非强依赖	OceanBase 集群要求时差小于 2s
nfs	非强依赖	OceanBase 数据库远程物理备份介质依赖

机器初始化

说明

- 本节内容在部署个人实验环境时为可选操作，在部署生产环境时需参照本节内容进行配置。
- 本节内容以 x86 架构的 CentOS Linux 7.9 镜像作为环境，其他环境可能略有不同。

创建用户和组

以创建操作系统普通用户 `admin` 为例，操作如下。

说明

- 您需在每一个 OBServer 节点执行如下操作。
- OCP V4.2.0-CE 之前版本仅支持使用 admin 用户安装部署 OceanBase 数据库和 OBProxy；OCP V4.2.0-CE 开始支持使用自定义用户安装部署 OceanBase 数据库和 OBProxy。建议使用 OCP 最新版本进行部署。
- 使用 obd 安装 OceanBase 数据库和 OBProxy 时，对使用的操作系统用户没有限制。

1. 创建 admin 用户组

```
[root@test001 ~]# groupadd -g 6001 admin
```

2. 创建 admin 用户

```
[root@test001 ~]# useradd -u 6001 -g admin admin
```

注意

所有 OBServer 节点的用户 uid 需要保持一致，否则使用 NFS 作为备份介质时会出现权限问题导致备份失败。

3. 配置 admin 用户密码

```
[root@test001 ~]# passwd admin
```

执行后在命令行界面输入想要配置的密码即可。

配置 sudo 免密

1. 添加 `/etc/sudoers` 文件的修改权限

```
[root@test001 ~]# chmod u+w /etc/sudoers
```

2. 为 admin 用户设置 sudo 权限

```
[root@test001 ~]# vim /etc/sudoers
```

在 `/etc/sudoers` 文件末尾添加如下内容：

```
## Same thing without a password
# %wheel        ALL=(ALL)       NOPASSWD: ALL
admin       ALL=(ALL)       NOPASSWD: ALL
```

3. 删去 `/etc/sudoers` 文件的修改权限

```
[root@test001 ~]# chmod u-w  /etc/sudoers
```

配置用户 SSH 互信

1. 在 obd 所在机器上执行以下命令查看密钥是否存在

```
[admin@test001 ~]# ls ~/.ssh/id_rsa.pub
```

若结果显示密钥已存在，则无需生成新的密钥。

2. （可选）执行以下命令生成 SSH 公钥和私钥

```
[admin@test001 ~]# ssh-keygen -t rsa
```

3. 在 obd 所在机器上执行如下命令将生成的公钥复制到目标机器的 `authorized_keys` 文件中

```
[admin@test001 ~]# ssh-copy-id -i ~/.ssh/id_rsa.pub <user>@<server_ip>
```

若您所用机器可执行 yum 命令，可在生成密钥后通过执行脚本配置 SSH 互信，通过脚本配置 SSH 互信的步骤如下。

说明

通过脚本配置 SSH 互信时需确保所用用户拥有 sudo 权限。

1. 创建脚本

```
[admin@test001 ~]# vim ssh.sh
```

此处以脚本名为 `ssh.sh` 为例进行介绍，您可自定义脚本名。脚本内容如下：

```
#!/usr/bin/bash

SERVERS=("<user>@<server_ip1>" "<user>@<server_ip2>" "<user>@<server_ip3>")
PASSWORD="******"
keygen() {
sudo yum -y install expect
expect -c "
   spawn ssh-keygen -t rsa
   expect {
      *(~/.ssh/id_rsa):* { send -- \r;exp_continue}
      *(y/n)* { send -- y\r;exp_continue}
      *Enter* { send -- \r;exp_continue}
      *(y/n)* { send -- y\r;exp_continue}
      *Enter* { send -- \r;exp_continue}
      eof {exit 0}
   }
   expect eof
"
}
copy(){
expect -c "
   set timeout -1
   spawn ssh-copy-id $1
   expect {
      *(yes/no)* { send -- yes\r; exp_continue }
      *password:* { send -- $PASSWORD\r; exp_continue}
      eof {exit 0}
   }
   expect eof
"
}
ssh_copy_id_to_all(){
keygen ;
for host in ${SERVERS[@]}
do
      copy $host
done
}
ssh_copy_id_to_all
```

> **注意**
>
> - 您须将脚本前两行的 SERVERS 列表和 PASSWORD 替换成您自己的实际机器列表和密码。
> - 创建的用户需要在 obd、OCP 节点和部署服务目标节点实现 SSH 互信。

2. 您须将脚本前两行的 `SERVERS` 列表和 `PASSWORD` 替换成您自己的实际机器列表和密码。
3. 创建的用户需要在 obd、OCP 节点和部署服务目标节点实现 SSH 互信。
4. 为脚本添加执行权限

```
[admin@test001 ~]# chmod u+x your_script.sh
```

5. 执行脚本

```
[admin@test001 ~]# ./ssh.sh
```

磁盘和文件系统规划

挂载点	大小	说明	文件系统格式
/home	100 GB ~ 300 GB	安装目录	数据小于 16 TB 时，xfs 和 ext4 两种文件系统都可以，建议使用 ext4 文件系统。数据大于 16 TB 时必须为 xfs 文件系统。
/data	取决于所需存储的数据大小	OceanBase 数据库的数据目录	
/redo	数据库内存大小的 3 ~ 4 倍	OceanBase 数据库的事务日志目录	

> **说明**
>
> 在生产环境，强烈建议数据目录 /data 和 事务日志目录 /redo 分盘部署，即使用独立挂载目录。

上文表格中提到的各个目录的详细介绍如下：

- 安装目录

  OceanBase 数据库安装目录主要用于存放数据库的二进制文件、配置文件、系统日志文件等非数据相关的文件，主要读写特点是顺序写。其中，系统日志类型包括 `observer.log`、

election.log、rootservice.log、trace.log，占用磁盘空间比较大，OceanBase 数据库单个系统日志文件大小为 256 MB，一般 3 ~ 8 分钟会生成新的系统日志，因此需要保证有足够的空间来存放这些文件，并留有一定的余量以适应软件升级和维护操作。

生产环境建议剩余空间不低于 200G，如果有更大的系统日志保留周期，需要根据现场实际业务产生的系统日志数量进行评估测算。

- 数据目录

  OceanBase 数据库的数据目录用于存放 OceanBase 数据库中的基线数据，主要读写特点是随机读、顺序写，偶尔密集的随机写。数据目录为预占用目录，一旦设置目录大小并部署完成后会直接占用数据目录磁盘空间，且不支持通过修改参数调小占用空间。数据目录的空间需求取决于业务数据量的大小以及预估的数据增长趋势，在同盘场景建议使用固定磁盘容量，通过 datafile_size 参数控制。

- 事务日志目录

  OceanBase 数据库的事务日志目录用于存放 OceanBase 数据库的所有事务操作日志，主要读写特点是顺序写。事务日志目录为预占用目录，一旦设置目录大小并部署完成后会直接占用事务日志目录磁盘空间，支持动态修改相关参数以调整占用空间。

  OceanBase 数据库采用 WAL（Write-Ahead Logging）策略，保证事务的原子性和持久性，需要有足够的日志空间来应对高并发事务场景的日志生成速率时。为了保证系统的稳定运行，避免日志溢出导致的服务中断，事务日志目录需要有足够的预留空间，建议不低于 OceanBase 数据库内存大小的 3 ~ 4 倍，同盘场景建议使用固定容量，通过 log_disk_size 参数控制。

正常情况下安装目录默认在部署用户的家目录下，比如：/home/admin 目录。数据目录和事务日志目录需要使用不同磁盘挂载的目录，且目录需要有部署用户所属权限，比如：/data、/redo 目录。目录如下所示：

```
[admin@test001 ~]$ ls -ld /data
drwxr-xr-x 2 admin admin 4096 Jan 25 17:34 /data

[admin@test001 ~]$ ls -ld /redo
drwxr-xr-x 2 admin admin 4096 Jan 25 17:35 /redo

[admin@test001 ~]$ df -h
Filesystem      Size  Used Avail Use% Mounted on
devtmpfs         16G     0   16G   0% /dev
```

```
tmpfs            16G     0   16G   0% /dev/shm
tmpfs            16G  676K   16G   1% /run
tmpfs            16G     0   16G   0% /sys/fs/cgroup
/dev/vda1       500G   50G  450G  90% /
/dev/vdb        500G    0   500G   0% /data
/dev/vdc        500G    0   500G   0% /redo
```

配置磁盘逻辑卷（可选）

在生产使用中，经常会遇到部署时使用服务器磁盘容量不足以存储更多数据的情况，但扩容服务器成本太高，此时需要对磁盘进行扩容来缓解存储容量问题，建议部署前可以考虑对磁盘做逻辑卷。

以下演示如何使用 4 块盘来创建我们部署 OceanBase 数据库环境需要的盘/文件系统。

1. 初始化物理卷

```
[admin@test001 ~]$ pvcreate /dev/vdb
[admin@test001 ~]$ pvcreate /dev/vdc
[admin@test001 ~]$ pvcreate /dev/vdd
[admin@test001 ~]$ pvcreate /dev/vde
```

2. 创建卷组

```
[admin@test001 ~]$ vgcreate vg_obdata /dev/vdb /dev/vdc /dev/vdd
[admin@test001 ~]$ vgcreate vg_obredo /dev/vde
```

3. 创建逻辑卷

```
[admin@test001 ~]$ lvcreate -l 100%FREE vg_obdata -n lv_obdata
[admin@test001 ~]$ lvcreate -l 100%FREE vg_obredo -n lv_obredo
```

4. 格式化逻辑卷

```
[admin@test001 ~]$ mkfs.ext4 /dev/vg_obdata/lv_obdata
[admin@test001 ~]$ mkfs.ext4 /dev/vg_obredo/lv_obredo
```

目前支持的文件系统有： ext4 和 xfs ，具体参考上文文件格式的说明。

5. 创建挂载点

```
[admin@test001 ~]$ sudo mkdir /data
[admin@test001 ~]$ sudo mkdir /redo
```

6. 添加开机自动挂载

   在 `/etc/fstab` 文件中添加如下内容。

```
/dev/vg_obdata/lv_obdata          /data         ext4            defaults,noatime,n
odiratime,nodelalloc,barrier=0        0 0
/dev/vg_obredo/lv_obredo          /redo         ext4            defaults,noatime,n
odiratime,nodelalloc,barrier=0        0 0
```

7. 执行挂载并确认

```
[admin@test001 ~]$ mount -a
[admin@test001 ~]$ df -h
```

修改用户资源限制

在 `/etc/security/limits.conf` 文件中添加如下内容：

```
root soft nofile 655350
root hard nofile 655350

* soft nofile 655350
* hard nofile 655350
* soft stack 20480
* hard stack 20480
* soft nproc 655360
* hard nproc 655360
* soft core unlimited
* hard core unlimited
```

说明

如存在 `/etc/security/limits.d/20-nproc.conf` 文件，并且该文件中含有 nproc 的配置，需同步修改该文件中 nproc 的值为 655360。

修改内核参数

在 `/etc/sysctl.conf` 文件中添加如下内容：

说明

容器化部署 OceanBase 数据库时，需要在宿主机上修改对应的内核配置，容器会继承宿主机的内核配置。

```
# for oceanbase
# 修改内核异步 I/O 限制
fs.aio-max-nr=1048576

# 网络优化
net.core.somaxconn = 2048
net.core.netdev_max_backlog = 10000
net.core.rmem_default = 16777216
net.core.wmem_default = 16777216
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216

net.ipv4.ip_local_port_range = 3500 65535
net.ipv4.ip_forward = 0
net.ipv4.conf.default.rp_filter = 1
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_rmem = 4096 87380 16777216
net.ipv4.tcp_wmem = 4096 65536 16777216
net.ipv4.tcp_max_syn_backlog = 16384
net.ipv4.tcp_fin_timeout = 15
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
net.ipv4.tcp_slow_start_after_idle=0

vm.swappiness = 0
vm.min_free_kbytes = 2097152
vm.overcommit_memory = 0

fs.file-max = 6573688

# 修改进程可以拥有的虚拟内存区域数量
vm.max_map_count = 655360

# 此处为 OceanBase 数据库的 data 目录
kernel.core_pattern = /data/core-%e-%p-%t
```

说明

```
kernel.core_pattern 对应的路径不建议放在根盘，避免出现 core 时产生的文件过大导致根分区沾满。
```

如果是 arm 架构的系统，需要同时在 `/etc/sysctl.conf` 文件里添加下述内容：

```
# 关闭 NUMA Balancing，避免平衡过程中性能发生抖动
rnel.numa_balancing = 0

# 禁用内存回收和重新分配功能
vm.zone_reclaim_mode = 0
```

添加后可执行 `sysctl -p` 命令加载配置文件并使之生效。

关闭防火墙和 SELinux

- 关闭防火墙并确认

```
[root@test001 ~]$ systemctl disable firewalld
[root@test001 ~]$ systemctl stop firewalld
[root@test001 ~]$ systemctl status firewalld
```

  输出中包含 `Active: inactive (dead)` 表示已关闭防火墙。

- 关闭 SELinux 并确认

```
[root@test001 ~]$ sed -i 's/SELINUX=enforcing/SELINUX=disabled/g' /etc/sysconfig/s
elinux
```

  selinux 参数生效需要重启服务器，虽然也有临时生效方式，建议服务器初始化完成后，重启服务器生效。

  重启服务器后执行如下命令确认 SELinux 是否关闭。

```
[root@test001 ~]$ getenforce
```

  getenforce 返回为 `Disabled` 表示已关闭 SELinux。

配置时钟源

OceanBase 数据库作为一款原生的分布式数据库，如果部署是集群形态，需要保证集群内各机器的时间同步，否则集群无法启动。OceanBase 数据库 4.x 版本允许集群中节点的时钟偏差不

能超过 2s，当超过 2s 时，会出现无主的情况。恢复时钟同步后，重启 OceanBase 集群，可以恢复正常。

CentOS 或 RedHat 7.x 版本推荐使用 Chrony 服务做时间源。Chrony 是 NTP（即 Network Time Protocol，网络时间协议，服务器时间同步的一种协议）的另一种实现。与 ntpd 不同的是，Chrony 可以更快且更准确地同步系统时钟，较大程度地减少时间和频率误差。

您可参考本节内容安装并被指 Chrony，使用 NTP 作为时钟源的操作步骤可参见官网《OceanBase 数据库》文档 部署数据库/部署 OceanBase 社区版/本地部署/部署前配置/（可选）配置时钟源。

1. 安装 Chrony 服务端

```
[root@test001 ~]$ yum -y install chrony
```

2. 配置 Chrony 服务端

```
[root@test001 ~]$ vim /etc/chrony.conf
```

内容如下：

```
# server 后面跟时间同步服务器

# 使用 pool.ntp.org 项目中的公共服务器。按 server 配置，理论上您想添加多少时间服务器都可以

# 或者使用阿里云的 ntp 服务器

# Please consider joining the pool (<http://www.pool.ntp.org/join.html>)

server ntp.cloud.aliyuncs.com minpoll 4 maxpoll 10 iburst
server ntp.aliyun.com minpoll 4 maxpoll 10 iburst
server ntp1.aliyun.com minpoll 4 maxpoll 10 iburst
server ntp1.cloud.aliyuncs.com minpoll 4 maxpoll 10 iburst
server ntp10.cloud.aliyuncs.com minpoll 4 maxpoll 10 iburst

# 如果是测试环境，没有时间同步服务器，那就选取一台配置为时间同步服务器

# 如果选中的是本机，则取消下面 server 注释

# server 127.127.1.0

# 根据实际时间计算出服务器增减时间的比率，然后记录到一个文件中，在系统重启后为系统做出最佳时间补偿
调整
```

```
driftfile /var/lib/chrony/drift

# chronyd 根据需求减慢或加速时间调整

# 在某些情况下系统时钟可能漂移过快，导致时间调整用时过长

# 该指令强制 chronyd 调整时期，大于某个阀值时步进调整系统时钟

# 只有在因 chronyd 启动时间超过指定的限制时（可使用负值来禁用限制）没有更多时钟更新时才生效

makestep 1.0 3

# 将启用一个内核模式，在该模式中，系统时间每 11 分钟会拷贝到实时时钟（RTC）

rtcsync

# Enable hardware timestamping on all interfaces that support it

# 通过使用 hwtimestamp 指令启用硬件时间戳

# hwtimestamp eth0
# hwtimestamp eth1
# hwtimestamp *

# Increase the minimum number of selectable sources required to adjust

# the system clock

# minsources 2

# 指定一台主机、子网，或者网络以允许或拒绝 NTP 连接到扮演时钟服务器的机器

# allow 192.168.0.0/16
# deny 192.168/16

# 即使没有同步到时间源，也要服务时间

local stratum 10

# 指定包含 NTP 验证密钥的文件

# keyfile /etc/chrony.keys

# 指定日志文件的目录

logdir /var/log/chrony

# Select which information is logged

# log measurements statistics tracking
```

```

```

此处给一个最简单的配置示例：

```
server 10.10.10.1
allow 10.10.10.0/24
local stratum 10
```

3. 启动和确认 Chrony 服务

```
[root@test001 ~]$ systemctl start chronyd
[root@test001 ~]$ systemctl enable chronyd
[root@test001 ~]$ systemctl status chronyd
```

4. 安装 Chrony 客户端

```
[root@test002 ~]$ yum -y install chrony
```

5. 配置 Chrony 客户端

```
[root@test002 ~]$ server 10.10.10.1 minpoll 4 maxpoll 10 iburst
```

6. 确认时钟是否同步

```
[root@test002 ~]$chronyc tracking
```

返回结果如下，`Leap status` 值为 `Normal` 表示同步过程正常。若输出中 `Leap status` 值为 `Not synchronised`，表示同步过程出错。

```
Reference ID    : AC18FF60 (10.10.10.1)
Stratum         : 3
Ref time (UTC)  : Wed Jan 31 07:49:10 2024
System time     : 0.000003143 seconds slow of NTP time
Last offset     : -0.000003572 seconds
RMS offset      : 0.000003572 seconds
Frequency       : 7.534 ppm slow
Residual freq   : +14.035 ppm
Skew            : 0.192 ppm
Root delay      : 0.003962355 seconds
Root dispersion : 0.010611359 seconds
Update interval : 2.0 seconds
Leap status     : Normal
```

OceanBase 数据库常用资源参数简介及计算方式

说明

本节提到的配置项均可访问官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/配置项总览 搜索查看。

	参数	解释	特性	小课堂
CPU 资源参数	cpu_count	OceanBase 数据库可使用的 CPU 核数，参数为整数类型，例如：16。默认值为 0，设置为 0 时，系统将自动检测并设置 CPU 核数。	支持动态调整	集群下的其他配置可能会影响该配置项的生效方式。建议在修改该配置项前，通过 SELECT 语句，在 GV$OB_SERVERS 内部表中查看 CPU_CAPACITY 字段实际生效的值。若未动态生效，需要重启整个集群。 注意 调小该配置项时，配置的值不能小于已分配出去的 CPU 核数。
内存资源参数	memory_limit	OceanBase 数据库可使用的内存大小，参数配置时需标注单位，例如：32G。	支持动态调整	配置时需注意如下几点： • memory_limit 没有上限边界，建议按实际内存（free -m 命令输出中的 free 列）剩余大小进行规划设置。 • 支持动态增大和缩小，但配置的值不能低于 OBServer 节点已分配出去的内存大小。 • memory_limit 的优先级高于 memory_limit_percentage，同时设置时，优先使用 memory_limit 配置项的值。
	memory_limit_percentage	OceanBase 数据库可用内存占总内存的百分比，参数	支持动态调整	与 memory_limit 同时配置时，优先使用 memory_limit 配置项的值。

		为整数类型，例如：80（表示 80%）。		

	system_memory	OceanBase 数据库中租户 ID 为500的租户的内存容量，即 OceanBase 数据库系统内部运行内存，参数配置时需标注单位，例如：30G。	支持动态调整	配置时需注意如下几点：
				• system_memory 的建议取值范围跟随 memory_limit 配置项变动，具体如下：
				  • memory_limit 配置在 [16G, 32G] 范围时，system_memory 取值范围为 [3G, 5G]。
				  • memory_limit 配置在 [32G, 64G] 范围时，system_memory 取值范围为 [5G, 10G]。
				  • memory_limit 配置为大于 64 GB 的值时，system_memory 可按照公式（ $system_memory = 3(\sqrt{memory_limit} -3)$ ）计算，并取整数部分。
				• system_memory 配置项设置为 0 时会自适应去申请内存。
				• system_memory 配置和 sys 租户没关系，sys 租户是 OceanBase 数据库部署完成由系统自建的自适应资源租户，租户 ID 为 1，而 system_memory 配置对应的租户 ID 为 500。

磁盘资源参数	datafile_size	OceanBase 数据库数据文件预占用的大小，参数配置时需标注单位，例如：32G。	预占用，不支持调小	配置时需注意如下几点： • 预占用会提前申请磁盘空间，部署完成后查看磁盘占用大，属于正常现象。 • datafile_size 的优先级高于 datafile_disk_percentage，同时设置时，优先使用 datafile_size 配置项的值。
	datafile_disk_percentage	OceanBase 数据库数据文件预占用其所在目录的百分	预占用，不支持调	与 datafile_size 同时配置时，优先使用 datafile_size 配置项的值。

	比，参数为整数类型，例如：80（表示 80%）。	小	
datafile_next	OceanBase 数据磁盘文件自动扩容的增长步长，参数配置时需标注单位，例如：100G。	支持动态调整	需要和 `datafile_max` 配置项搭配使用。
datafile_maxsize	OceanBase 数据磁盘文件自动扩容的空间上限。参数配置时需标注单位，例如：100G。	支持动态调整	配置时需注意如下几点： • 支持动态调整，但调小时不能小于已占用的数据磁盘空间。 • 需要和 `datafile_max` 配置项搭配使用。 • 当 `datafile_next`、`datafile_maxsize` 配置为大于 0 的值时，表示开启按需分配。此时，`datafile_size` 或 `datafile_disk_percentage` 影响的是磁盘初始数据文件的大小。
log_disk_size	OceanBase 数据库 Redo 日志文件预占用大小，参数配置时需标注单位，例如：20G。	预占用，支持动态调整	配置时需注意如下几点： • `log_disk_size` 取值至少需为 `memory_limit` 的三倍，即 `log_disk_size` >= `memory_limit` * 3。 • 预占用会提前申请磁盘空间，部署完成后查看磁盘占用大，属于正常现象。 • `log_disk_size` 的优先级高于 `log_disk_percentage`，同时设置时，优先使用 `log_disk_size` 配置项的值。
log_disk_percentage	OceanBase 数据库 Redo 日志文件预占用其所在目	预占用，支持动态	与 `log_disk_size` 同时配置时，优先使用 `log_disk_size` 配置项的值。

		录的百分比，参数为整数类型，例如：80（表示 80%）。	调整	
系统日志参数	enable_syslog_recycle	控制是否将重启前的日志文件纳入回收范围，建议开启。	支持动态调整	需要和 `max_syslog_file_count` 配置项搭配使用。该配置项设置为 `false` 时，回收日志前可容纳的日志文件数量中不包含重启前的日志。
	max_syslog_file_count	在回收 OceanBase 数据库系统日志文件之前可保留的系统日志文件数量。	支持动态调整	需要和 `enable_syslog_recycle` 配置项搭配使用，单个系统日志文件最大可占用 256 MB 的磁盘空间，建议生产环境按需要日志保留天数需要来设置系统日志个数。 注意 OceanBase 数据库系统日志文件打印较频繁，需要关注文件个数和磁盘大小关系，防止磁盘被占满。

2.2 部署相关的生态组件介绍

部署 OceanBase 数据库一般用到 obd、OCP、ob-operator 三种部署工具，不同的部署工具有不同的使用场景：

- 对于企业用户推荐使用 OCP，因为 OCP 拥有丰富的运维功能，方便未来进行进一步的运维管理。
- 对于个人用户或者资源非常少的企业用户，推荐使用 obd，因为 obd 为纯黑屏操作，对资源消耗最少。
- 对于 k8s 的用户，推荐使用 ob-operator，可以更好地沿用已有的使用习惯来使用 OceanBase 数据库。

以下是 obd、OCP、ob-operator 三种部署工具支持情况的具体对比：

比对方向	obd	OCP	ob-operator
支持安装的版本	3.1.x 和 4.x	3.1.x 和 4.x	4.x
离线/在线部署	均支持	仅支持离线部署	均支持
是否支持自编译的安装包	支持	支持	支持
是否支持高可用 **说明** 这里指工具自身是否支持高可用，而非部署的 OceanBase 集群是否支持高可用。	不支持	支持	支持（依赖 k8s）
是否支持单机多节点部署	支持	不支持	支持
安全性	依赖操作系统权限	高（提供了基于用户角色的权限隔离机制，保障了资源的使用安全）	依赖 k8s 的权限体系
生态对接	开放源码	开放 API	开放源码
上手难度	低	中	取决于对 k8s 环境熟悉程度
资源	轻量	高	中

下表为功能点对比，是指通过部署工具（平台）是否可以操作（执行）这些功能。

说明

下表内容基于 obd V2.8.0、OCP V4.3.0、ob-operator V2.2.0（dashboard V0.3.0）整理，不同版本间功能可能存在差异。为确保获取最准确的功能信息与支持详情，请参照您正在使用的具体版本查阅各工具的官方文档。

功能点对比	obd	OCP	ob-operator
租户创建	支持	支持	支持
租户查看	支持	支持	支持
数据库管理	不支持	支持	不支持
用户权限管理	不支持	支持	不支持
资源管理	不支持	支持	支持
资源隔离	不支持	支持	不支持
合并管理	不支持	支持	不支持
备份恢复	不支持	支持	支持
监控告警	支持监控，不支持告警	支持	支持
扩缩容	支持扩容，暂不支持缩容	支持	支持
topsql/slowsql	不支持	支持	不支持
事务诊断	不支持	支持	不支持
死锁诊断	不支持	支持	不支持
会话管理	不支持	支持	不支持
主备库	支持	支持	支持

经过如上两个表格的对比之后，我们就可以结合自己的需要选择合适的工具来部署 OceanBase 数据库环境了。

2.3 部署个人实验环境

在部署 OceanBase 社区版之前，您可通过小规格主机快速体验部署 OceanBase 社区版环境。同时我们提供了一个 OceanBase 社区版 Docker 镜像，您也可使用 Docker 技术快速部署并启动 OceanBase 社区版的 Docker 容器。

本节我们提供三种部署形态可供个人实验环境选择：演示环境、集群环境、容器环境。本文中的方法仅适用于 OceanBase 数据库快速上手体验，不适用生产环境。

部署生产环境所需的部署环境要求可以参见本章 部署前准备 内容。

部署 OceanBase 数据库演示环境

当您仅拥有一台可用机器时，可参考本节内容使用 `obd demo` 命令快速部署单机 OceanBase 数据库。

公网环境快捷部署

若您的机器可以连接外部网络，可参照本节内容进行部署。

1. 远程拉取执行安装脚本

```
[admin@test001 ~] bash -c "$(curl -s https://obbusiness-private.oss-cn-shanghai.aliyuncs.com/download-center/opensource/oceanbase-all-in-one/installer.sh)"
```

2. 加载环境变量

```
[admin@test001 ~] source ~/.oceanbase-all-in-one/bin/env.sh
```

3. （可选）开启远程仓库

通过 all-in-one 方式进行安装时，远程仓库默认为关闭状态。若想部署非 all-in-one 提供的组件版本，可执行如下命令开启远程仓库。

> **说明**
>
> 若未开启远程仓库，执行 obd demo 命令部署时会使用 all-in-one 提供的组件安装包。开启远程仓库后，

执行 obd demo 命令时会自动从远程仓库拉取组件最新版本安装包。

```
[admin@test001 ~]$ obd  mirror enable remote
```

4. 部署 demo 演示环境

```
[admin@test001 ~] obd demo
```

输出中会显示部署的组件信息，示例如下：

```
+---------------------------------------------+
|                   observer                  |
+-----------+---------+------+-------+--------+
| ip        | version | port | zone  | status |
+-----------+---------+------+-------+--------+
| 127.0.0.1 | 4.2.1.3 | 2881 | zone1 | ACTIVE |
+-----------+---------+------+-------+--------+
obclient -h127.0.0.1 -P2881 -uroot -Doceanbase -A

+-----------------------------------------------------------------+
|                             obagent                             |
+---------------+--------------------+--------------------+--------+
| ip            | mgragent_http_port | monagent_http_port | status |
+---------------+--------------------+--------------------+--------+
| 10.10.10.1    | 8089               | 8088               | active |
+---------------+--------------------+--------------------+--------+
+---------------------------------------------+
|                   obproxy                   |
+-----------+------+-----------------+--------+
| ip        | port | prometheus_port | status |
+-----------+------+-----------------+--------+
| 127.0.0.1 | 2883 | 2884            | active |
+-----------+------+-----------------+--------+
obclient -h127.0.0.1 -P2883 -uroot -Doceanbase -A

+-----------------------------------------------------+
|                      prometheus                     |
+---------------------------+------+----------+--------+
| url                       | user | password | status |
+---------------------------+------+----------+--------+
| http://10.10.10.1:9090    |      |          | active |
+---------------------------+------+----------+--------+
+--------------------------------------------------------------------+
|                               grafana                              |
+------------------------------------+-------+------------+--------+
| url                                | user  | password   | status |
+------------------------------------+-------+------------+--------+
```

```
| http://10.10.10.1:3000/d/oceanbase     | admin | ******      | active |
+--------------------------------------+-------+------------+---------+
demo running
```

执行 `obd demo` 命令默认会在当前用户家目录下以最小规格部署并启动 OceanBase 数据库及相关组件（包括 OBProxy、OBAgent、Grafana 和 Prometheus），固定部署名为 `demo`。如需更多定制化的部署形式，可参见官网《OceanBase 安装部署工具》文档 obd 命令/快速部署命令。您也可以使用 obd 命令管理通过 `obd demo` 命令部署的 OceanBase 数据库，详细命令介绍请参见官网《OceanBase 安装部署工具》文档 obd 命令/集群命令组。

> **说明**
>
> 当安装 Grafana 或 Prometheus 时，会输出 Grafana 或 Prometheus 的访问地址。在阿里云或其他云环境下，可能出现因无法获取公网 IP 而输出内网地址的情况，此 IP 非公网地址，您需要使用正确的地址。

离线环境部署

1. 访问官网 OceanBase 软件下载中心 下载并安装 **OceanBase社区版一键安装包 (OceanBase All in One)**，并将其上传到机器任一目录下，建议下载最新版本。
2. 在安装包所在目录下执行如下命令解压安装包。

```
[admin@test001 ~]$ sudo tar -xzf oceanbase-all-in-one-*.tar.gz
```

3. 进入解压后的目录，执行安装初始化

```
[admin@test001 ~]$ cd oceanbase-all-in-one/bin/

# 执行安装初始化
[admin@test001 bin]$ ./install.sh
```

执行成功后会返回如下信息：

```
##########################################################################################
#######
 Install Finished
==========================================================================================
=======
Setup Environment:              source ~/.oceanbase-all-in-one/bin/env.sh
```

```
Quick Start:                    obd demo
Use Web Service to install:     obd web
Use Web Service to upgrade:     obd web upgrade
More Details:                   obd -h
==================================================================================
=======
```

命令解释：

- `source ~/.oceanbase-all-in-one/bin/env.sh`：设置 obd 环境变量。
- `obd demo`：快捷部署小型化 OceanBase 数据库。
- `obd web`：obd 的图形化部署操作界面。
- `obd web upgrade`：obd 的图形化升级操作界面。
- `obd -h`：obd 命令帮助。

4. `source ~/.oceanbase-all-in-one/bin/env.sh`：设置 obd 环境变量。
5. `obd demo`：快捷部署小型化 OceanBase 数据库。
6. `obd web`：obd 的图形化部署操作界面。
7. `obd web upgrade`：obd 的图形化升级操作界面。
8. `obd -h`：obd 命令帮助。
9. 按提示执行环境变量脚本。

```
[admin@test001 ~] source ~/.oceanbase-all-in-one/bin/env.sh
```

10. 检查远程仓库状态。

离线部署时需保证远程仓库为关闭状态，否则 obd 会尝试连接远程仓库获取安装包，进而报错。

```
[admin@test001 ~]$ obd  mirror list
```

输出如下，`oceanbase.community.stable` 和 `oceanbase.development-kit` 远程仓库状态均为 `False` 表示已关闭，部署会使用本地 local 仓库的安装包。

```
+------------------------------------------------------------------------------+
|                          Mirror Repository List                              |
+----------------------------+--------+---------+----------+-------------------+
| SectionName                | Type   | Enabled | Avaiable | Update Time       |
+----------------------------+--------+---------+----------+-------------------+
| oceanbase.community.stable | remote | False   | False    | 1970-01-01 08:00 |
| oceanbase.development-kit  | remote | False   | False    | 1970-01-01 08:00 |
| local                      | local  | -       | True     | 2024-01-30 15:31 |
+----------------------------+--------+---------+----------+-------------------+
```

说明

通过 all-in-one 方式进行安装时，远程仓库默认为关闭状态，若输出中远程仓库为开启状态（True），您需执行 `obd mirror disable remote` 命令关闭远程仓库。

11. 部署 demo 演示环境。

```
[admin@test001 ~] obd demo
```

输出中会显示部署的组件信息，示例如下：

```
+----------------------------------------------+
|                   observer                   |
+-----------+---------+------+-------+--------+
| ip        | version | port | zone  | status |
+-----------+---------+------+-------+--------+
| 127.0.0.1 | 4.2.1.3 | 2881 | zone1 | ACTIVE |
+-----------+---------+------+-------+--------+
obclient -h127.0.0.1 -P2881 -uroot -Doceanbase -A

+-------------------------------------------------------------------+
|                             obagent                               |
+---------------+---------------------+---------------------+--------+
| ip            | mgragent_http_port  | monagent_http_port  | status |
+---------------+---------------------+---------------------+--------+
| 10.10.10.1    | 8089                | 8088                | active |
+---------------+---------------------+---------------------+--------+
+----------------------------------------------+
|                   obproxy                    |
+-----------+------+-----------------+--------+
| ip        | port | prometheus_port | status |
+-----------+------+-----------------+--------+
```

```
| 127.0.0.1 | 2883 | 2884            | active |
+-----------+------+-----------------+--------+
obclient -h127.0.0.1 -P2883 -uroot -Doceanbase -A

+----------------------------------------------------------+
|                     prometheus                           |
+----------------------------+------+----------+--------+
| url                        | user | password | status |
+----------------------------+------+----------+--------+
| http://10.10.10.1:9090     |      |          | active |
+----------------------------+------+----------+--------+
+----------------------------------------------------------------------+
|                              grafana                                 |
+----------------------------------------+-------+-------------+--------+
| url                                    | user  | password    | status |
+----------------------------------------+-------+-------------+--------+
| http://10.10.10.1:3000/d/oceanbase     | admin | ******      | active |
+----------------------------------------+-------+-------------+--------+
demo running
```

执行 `obd demo` 命令默认会在当前用户家目录下以最小规格部署并启动 OceanBase 数据库及相关组件（包括 OBProxy、OBAgent、Grafana 和 Prometheus），固定部署名为 `demo`。如需更多定制化的部署形式，可参见官网《OceanBase 安装部署工具》文档 obd 命令/快速部署命令。您也可以使用 obd 命令管理通过 `obd demo` 命令部署的 OceanBase 数据库，详细命令介绍请参见官网《OceanBase 安装部署工具》文档 obd 命令/集群命令组。

> **说明**
>
> 当安装 Grafana 或 Prometheus 时，会输出 Grafana 或 Prometheus 的访问地址。在阿里云或其他云环境下，可能出现因无法获取公网 IP 而输出内网地址的情况，此 IP 非公网地址，您需要使用正确的地址。

相关操作

为 root@sys 用户配置密码

`obd demo` 命令创建的集群默认管理员密码为空，建议下述步骤为 root@sys 用户设置密码。

1. 修改配置文件

```
[admin@test001 ~] obd cluster edit-config demo
```

执行上述命令打开配置文件后，在配置文件中 `oceanbase-ce` 组件下配置 `root_password`，完成后 `:x` 保存退出，并按提示执行 reload 命令使配置生效。配置文件示例如下：

```
oceanbase-ce:
  servers:
    - 127.0.0.1
  global:
    home_path: /home/admin/oceanbase-ce
    ... # 省略部分配置项
    log_disk_size: 13G
    root_password: ****** # 配置 root@sys 用户密码
```

2. 重载集群

修改并保存配置文件后，obd 会输出待执行的 reload 重载命令，直接复制执行即可，输出示例如下。

```
Search param plugin and load ok
Search param plugin and load ok
Parameter check ok
Save deploy "demo" configuration
Use `obd cluster reload demo` to make changes take effect.
Trace ID: 4c977d02-bf47-11ee-bc44-00163e039b49
If you want to view detailed obd logs, please run: obd display-trace     4c977d02
-bf47-11ee-bc44-00163e039b49
```

直接执行输出中的 reload 命令。

```
[admin@test001 ~]$ obd cluster reload demo
```

输出如下：

```
Get local repositories and plugins ok
Load cluster param plugin ok
Open ssh connection ok
Cluster status check ok
Connect to observer 127.0.0.1:2881 ok
Connect to Obagent ok
Reload obagent ok
Connect to obproxy ok
Connect to Prometheus ok
Reload prometheus ok
```

```
Connect to grafana ok
Reload Grafana ok
demo reload
Trace ID: 393d5bb8-bf48-11ee-bcb8-00163e039b49
If you want to view detailed obd logs, please run: obd display-trace 393d5bb8-bf48
-11ee-bcb8-00163e039b49
```

注意

不可以直接登录数据库后通过 `ALTER USER` 或者 `SET PASSWORD` 命令修改 `root@sys` 用户密码。如果 OceanBase 数据库中的 `root@sys` 用户密码和配置文件中的 `root_password` 配置项不一致，会导致 obd 管理命令无法连接到数据库，影响实际操作，例如使用 obd 重启集群会失败。

创建用户租户

强烈推荐创建并使用用户租户，禁止使用管理租户（sys 租户）做测试或业务使用。可执行如下命令创建用户租户：

```
[admin@test001 ~]$ obd cluster tenant create demo -n test
```

输出如下：

```
Get local repositories and plugins ok
Open ssh connection ok
Connect to observer 127.0.0.1:2881 ok
Create tenant test ok
Trace ID: 7c73104c-bf45-11ee-91d0-00163e039b49
If you want to view detailed obd logs, please run: obd display-trace    7c73104c-b
f45-11ee-91d0-00163e039b49
```

执行上述命令，obd 默认会使用剩余的所有资源创建一个租户，您也可以通过配置一些参数控制租户的情况。`obd cluster tenant create` 命令的详细介绍可参见官网《OceanBase 安装部署工具》文档 obd 命令/集群命令组 中 **obd cluster tenant create**。

连接数据库

`obd demo` 命令成功执行后会输出通过 OBClient 连接 OceanBase 数据库的命令，可执行输出中的连接命令连接 OceanBase 数据库。示例如下：

```
# 直连 OBServer 节点连接 OceanBase 数据库 sys 租户
obclient -h127.0.0.1 -P2881 -uroot@sys -Doceanbase -A

# 通过 ODP 连接 OceanBase 数据库 sys 租户
obclient -h127.0.0.1 -P2883 -uroot@sys -Doceanbase -A
```

使用 OBClient 客户端连接 OceanBase 集群的详细操作可参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 OBClient 连接 OceanBase 租户。更多连接 OceanBase 数据库的方法请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/连接方式概述。

销毁集群

若部署的集群需要销毁，且数据库数据可以清理，可以使用 `obd cluster destory` 命令销毁部署集群。

> **注意**
>
> 销毁操作为高危操作，此操作会卸载对应部署名称下的部署组件并删除数据目录，请谨慎使用。

```
[admin@test001 ~]$ obd cluster destroy demo
```

输出如下：

```
Get local repositories ok
Open ssh connection ok
Check for standby tenant ok
Search plugins ok
Stop observer ok
Stop prometheus ok
Stop obagent ok
Stop grafana ok
Stop obproxy ok
demo stopped
Search plugins ok
Cluster status check ok
observer work dir cleaning ok
prometheus work dir cleaning ok
obagent work dir cleaning ok
grafana work dir cleaning ok
obproxy work dir cleaning ok
demo destroyed
```

```
Trace ID: 500c3140-bf4b-11ee-b85e-00163e039b49
If you want to view detailed obd logs, please run: obd display-trace 500c3140-bf4b
-11ee-b85e-00163e039b49
```

部署 OceanBase 集群环境

说明

此处以 obd V2.8.0 为例介绍操作步骤，其他版本的图形化页面可能略有不同。不同版本的操作步骤可参见官网 OceanBase 安装部署工具 文档中对应版本内容。

当您拥有多台可用机器时，可参考本节内容使用 `obd web` 命令启动白屏，通过白屏界面部署分布式 OceanBase 集群。

1. 命令行执行 `obd web` 命令启动白屏界面，根据输出地址登录白屏界面。

```
[admin@obtest001 ~]$ obd web
```

输出如下：

```
start OBD WEB in 0.0.0.0:8680
please open <http://10.10.10.1:8680>
```

说明

- 白屏界面默认使用 8680 端口，您可使用 `obd web -p` 命令指定端口。
- 在云环境下，可能出现程序无法获取公网 IP，从而输出内网地址的情况，此 IP 非公网地址，您需要使用正确的地址访问白屏界面。
- `obd web` 命令绑定在 `0.0.0.0` 上，在多网卡部署的情况下，您可通过任意一个可访问的 IP 访问白屏界面。
- 部署 OCP Express 服务依赖 JAVA 环境，obd V2.7.0 开始支持部署时检测并安装 JDK。若所用 obd 低于 V2.7.0，需要自行安装 JDK1.8，且构建版本不低于 161，同时 JAVA 命令需要在 `/usr/bin/java` 下。

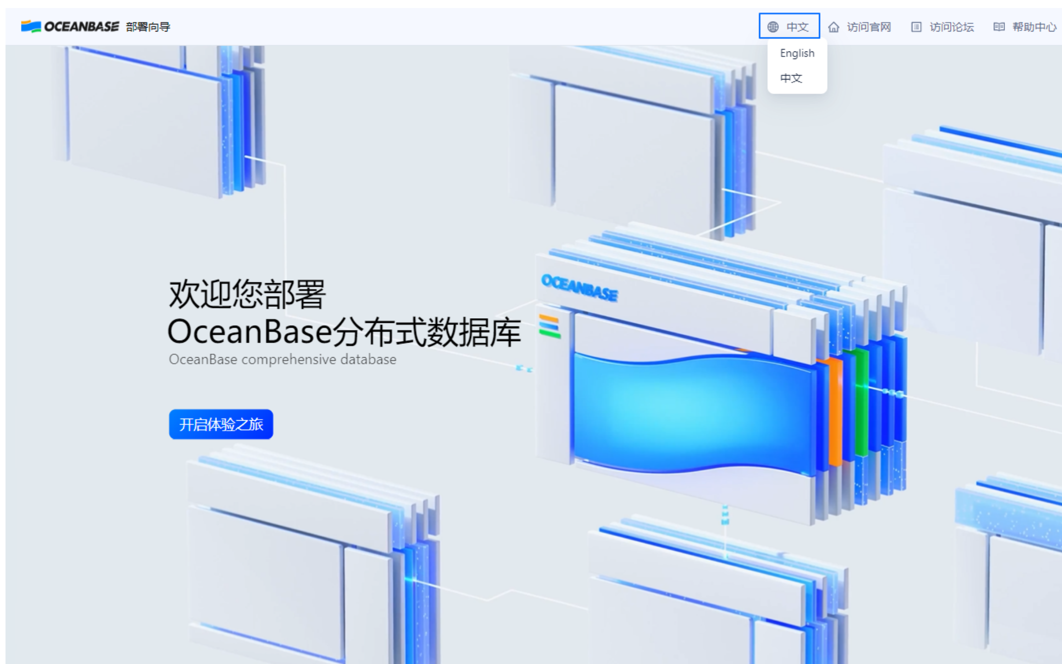
2. 白屏界面默认使用 `8680` 端口，您可使用 `obd web -p` 命令指定端口。

3. 在云环境下，可能出现程序无法获取公网 IP，从而输出内网地址的情况，此 IP 非公网地址，您需要使用正确的地址访问白屏界面。

4. `obd web` 命令绑定在 `0.0.0.0` 上，在多网卡部署的情况下，您可通过任意一个可访问的 IP 访问白屏界面。

5. 部署 OCP Express 服务依赖 JAVA 环境，obd V2.7.0 开始支持部署时检测并安装 JDK。若所用 obd 低于 V2.7.0，需要自行安装 JDK1.8，且构建版本不低于 161，同时 JAVA 命令需要在 `/usr/bin/java` 下。

6. 单击 **开启体验之旅** 开始部署流程。



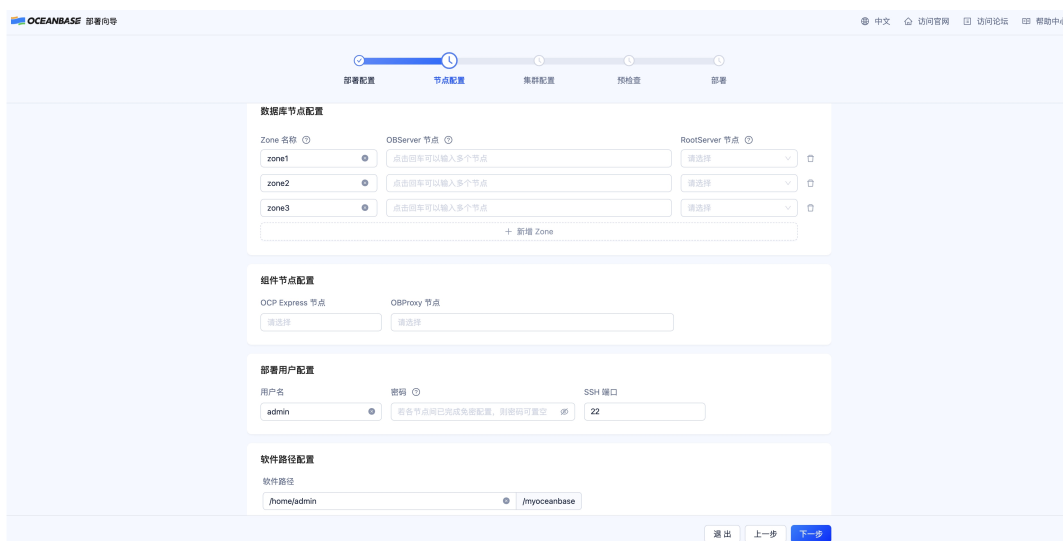
> **说明**
>
> 鼠标放置到图形化操作界面右上角的 **中文** 字符处可根据显示的语种单击切换中英文界面。

7. 选择 **OceanBase 及配套工具** 栏，点击 **确定** 进行下一步。

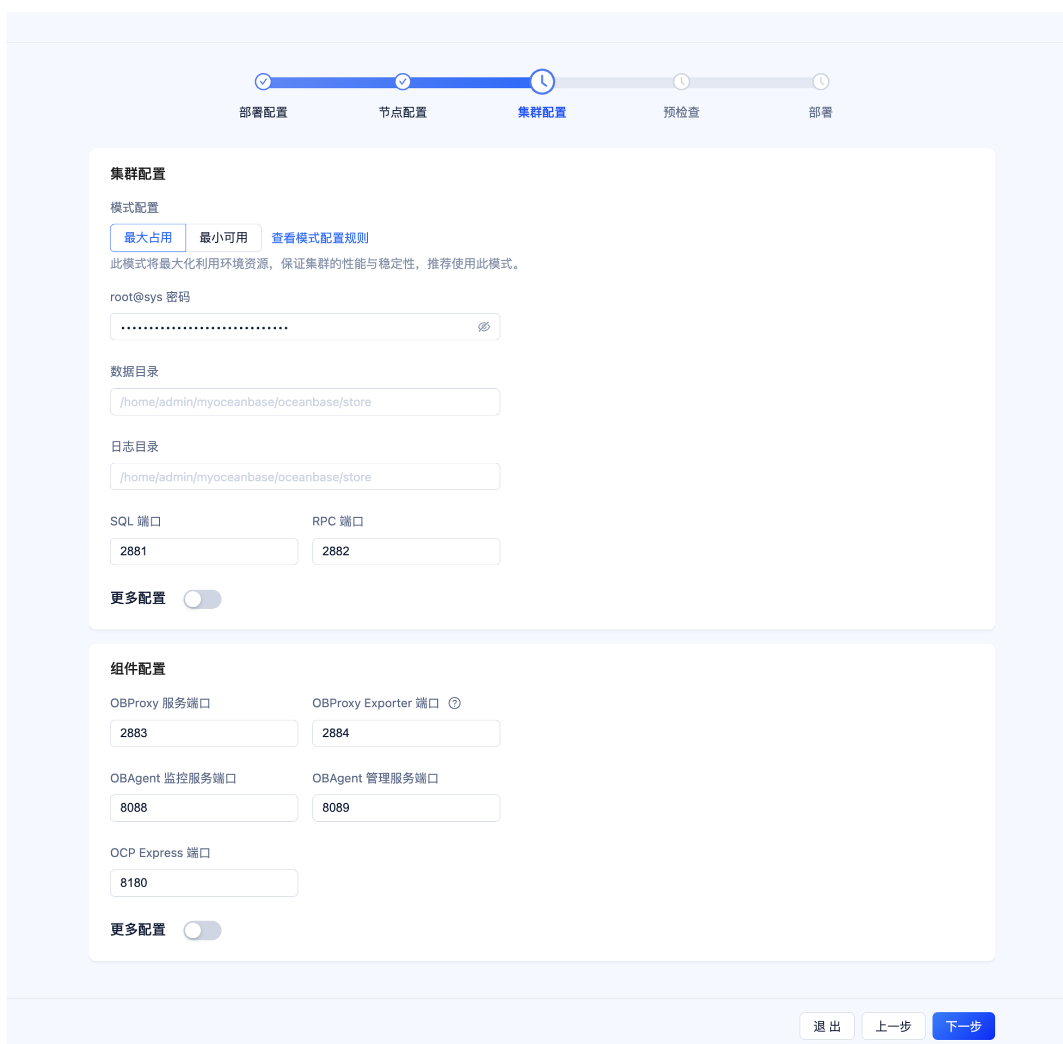
8. 在 **部署配置** 页面修改集群名称并勾选待部署组件，也可不做修改，使用默认配置（默认部署所有组件）。

配置完成后，单击 **下一步** 进入 **节点配置** 页面。

9. 在 **节点配置** 页面输入节点 IP 和用户密码，单击 **下一步** 进入 **集群配置** 页面。



10. 在 **集群配置** 页面配置集群的部署模式、密码、目录、端口以及更多配置，也可不做修改，使用默认配置。



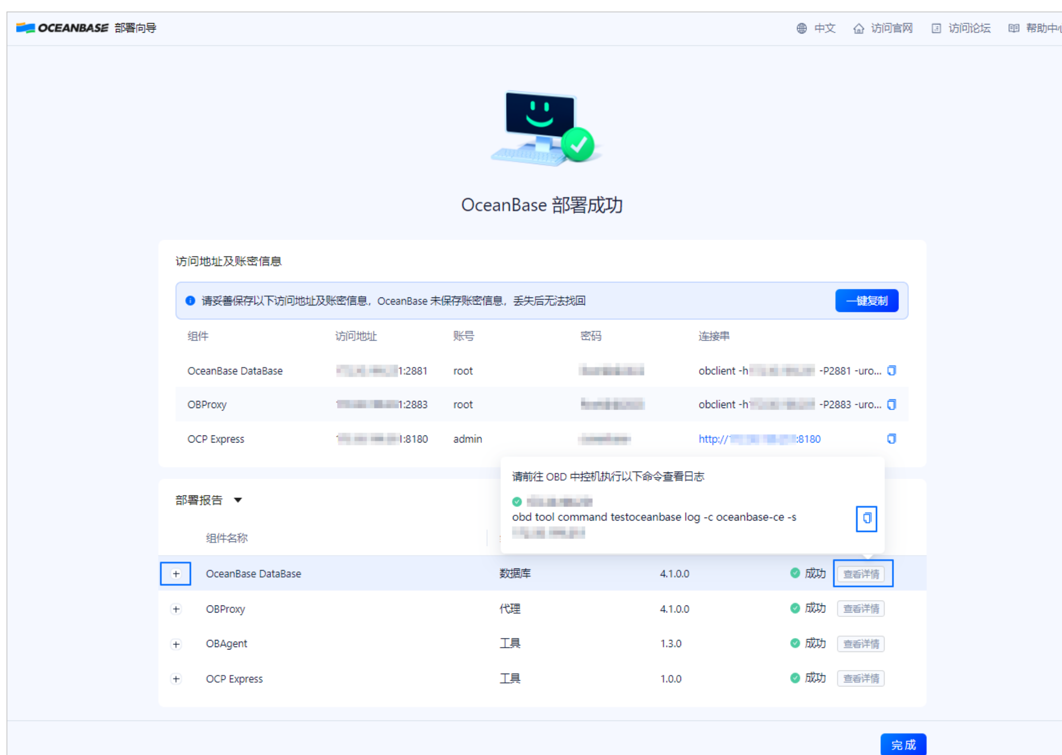
配置完成后，单击 **下一步** 进入 **预检查** 页面。

11. 在 **预检查** 页面查看配置信息，确认无误后，单击 **预检查** 进行检查。

若预检查报错，您可根据页面建议选择 **自动修复** 或者单击 **了解更多方案** 跳转至错误码文档，参考文档自行修改。所有报错修改后，可单击 **重新检查** 再次进行预检查。

12. 预检查通过后，单击 **部署** 开始 OceanBase 集群的部署。

部署成功会输出各个组件的连接方式，可复制进行访问。



13. 单击 **完成**，结束部署流程。

14. 登录 OCP Express 白屏界面创建用户租户。

    如果忘记或未保存 OCP Express 白屏登录信息，可以使用 `obd cluster display <deploy name>` 命令查看。

```
# 查看集群部署列表
obd cluster list

# 查看 myoceanbase 集群详情
obd cluster display myoceanbase
```

    在 **租户管理** 界面中，选择 **新建租户**，按要求填写所需信息，创建 `test` 用户租户。

15. 使用 OBClient 客户端连接 OceanBase 数据库，或者登录 OCP Express 白屏界面租户管理查看对应租户连接串信息。

```
# 通过 2881 端口直连数据库 sys 租户
[admin@test001 ~]$ obclient -h10.10.10.1 -P2881 -uroot@sys -p -Doceanbase -A

# 通过 2881 端口直连数据库 test 租户
[admin@test001 ~]$ obclient -h10.10.10.1 -P2881 -uroot@test -p -Doceanbase -A

# 通过 ODP 代理访问数据库 sys 租户
[admin@test001 ~]$ obclient -h10.10.10.1 -P2883 -uroot@sys -p -Doceanbase -A

# 通过 ODP 代理访问数据库 test 租户
[admin@test001 ~]$ obclient -h10.10.10.1 -P2883 -uroot@test -p -Doceanbase -A
```

    使用 OBClient 客户端连接 OceanBase 集群的详细操作可参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 OBClient 连接 OceanBase 租户。连接 OceanBase 数据库的更多方法请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/连接方式概述。

说明

同样的，如果部署的集群需要销毁，且数据库数据可以清理，可以使用 `obd cluster destory` 命令销毁部署集群，此操作会卸载对应部署名称下的部署组件并删除数据目录。

部署 OceanBase 容器环境

部署 OceanBase 容器环境前需确保满足如下条件：

- 机器中已安装 Docker，并启动 Docker 服务，详细操作请参考 Docker 文档。
- 容器可用内存不少于 6 G。此处的可用内存指剩余可用内存。
- 容器可用 CPU 建议至少有 2 个逻辑 CPU。

Docker 常用命令

您可参考如下常见 Docker 命令管理 Docker。

```
# 查看 Docker 版本
docker version
# 显示 Docker 系统的信息
docker info
# 查看当前正在运行的容器
docker ps
# 故障检查
service docker status
# 启动关闭 Docker
service docker start | stop
# 查看容器日志
docker logs -f <容器名 or ID>
# 清理不用的容器
docker container prune
# 清理不用的镜像
docker image prune
# 清理不用的卷
docker volume prune
```

下载镜像并启动

若您有兴趣可点击链接下面查看详情。

- OceanBase Docker 镜像地址：https://hub.docker.com/r/oceanbase/oceanbase-ce
- 镜像在 Github 上的源码地址：https://github.com/oceanbase/docker-images/tree/main/oceanbase-ce

1. 搜索 oceanbase 相关镜像

```
[admin@test001 ~]$ sudo docker search oceanbase
```

2. 拉取 oceanbase 最新镜像

```
[admin@test001 ~]$ sudo docker pull oceanbase/oceanbase-ce
```

输出如下：

```
Using default tag: latest
latest: Pulling from oceanbase/oceanbase-ce
bf5ec1942180: Pull complete
e80e84f8272f: Pull complete
f72149611c5a: Pull complete
be8bfe13b526: Pull complete
e689549a77e0: Pull complete
96b878f840c3: Pull complete
4f4fb700ef54: Pull complete
Digest: sha256:547ce27e204149f7491f13b3af503299fc25ee8e4161101724357f716b787c84
Status: Downloaded newer image for oceanbase/oceanbase-ce:latest
docker.io/oceanbase/oceanbase-ce:latest
```

说明

- 若拉取 Docker 镜像失败，您也可从 quay.io 或者 ghcr.io 仓库中拉取镜像，只需将上述拉取命令中的 `oceanbase/oceanbase-ce` 对应替换为 `quay.io/oceanbase/oceanbase-ce` 或 `ghcr.io/oceanbase/oceanbase-ce`，如执行 `sudo docker pull quay.io/oceanbase/oceanbase-ce` 从 quay.io 中拉取镜像。
- 本步骤命令中的仓库地址更换后，启动步骤中的命令需同步替换仓库地址，两条命令所用仓库需保持统一。
- 命令默认拉取最新版本，可根据实际需求在 dockerhub、quay.io 或者 ghcr.io 中选择版本。

3. 若拉取 Docker 镜像失败，您也可从 quay.io 或者 ghcr.io 仓库中拉取镜像，只需将上述拉取命令中的 `oceanbase/oceanbase-ce` 对应替换为 `quay.io/oceanbase/oceanbase-ce` 或 `ghcr.io/oceanbase/oceanbase-ce`，如执行 `sudo docker pull quay.io/oceanbase/oceanbase-ce` 从 quay.io 中拉取镜像。
4. 本步骤命令中的仓库地址更换后，启动步骤中的命令需同步替换仓库地址，两条命令所用仓

库需保持统一。

5. 命令默认拉取最新版本，可根据实际需求在 dockerhub、quay.io 或者 ghcr.io 中选择版本。

6. 启动 OceanBase Docker 容器

```
# 根据当前容器部署最大规格的实例
[admin@test001 ~]$ sudo docker run -p 2881:2881 --name obstandalone -e MODE=normal -e OB_TENANT_PASSWORD=***** -d oceanbase/oceanbase-ce

## 部署 mini 的独立实例
[admin@test001 ~]$ sudo docker run -p 2881:2881 --name obstandalone -e MODE=mini -e OB_TENANT_PASSWORD=***** -d oceanbase/oceanbase-ce
```

其中：

- `--name` 用于设置 Docker 容器名称，比如示例中创建一个名为 `obstandalone` 的 Docker 容器。

- `-e` 用于设置环境变量，其中 `MODE` 用于设置 OceanBase 数据库的部署规格，`OB_TENANT_PASSWORD` 用于设置 OceanBase 数据库中 root@sys 用户密码。

7. `--name` 用于设置 Docker 容器名称，比如示例中创建一个名为 `obstandalone` 的 Docker 容器。

8. `-e` 用于设置环境变量，其中 `MODE` 用于设置 OceanBase 数据库的部署规格，`OB_TENANT_PASSWORD` 用于设置 OceanBase 数据库中 root@sys 用户密码。

9. 查看容器启动日志

刚启动的 OceanBase 数据库需要 2 ~ 5 分钟初始化集群。您可多次运行如下命令查看容器启动进度日志，如果最终返回 `boot success!`，则表示启动成功。

```
[admin@test001 ~]$ sudo docker logs obstandalone | tail -1
```

您也可直接执行 `sudo docker logs obstandalone` 查看全部日志，从日志中可以查看到如下信息：

1. OceanBase 数据库安装时需获取 oceanbase-ce-lib、oceanbase-ce 和 obagent

包。

2. 在启动 OceanBase 数据库前会先初始化集群目录。

3. 安装过程中会创建用户租户（test 租户）。

> **说明**
>
> 日志中会打印一些 [WARN] 信息，这些信息可以忽略，若安装失败，仅需关注 [ERROR] 信息。

10. OceanBase 数据库安装时需获取 oceanbase-ce-lib、oceanbase-ce 和 obagent 包。
11. 在启动 OceanBase 数据库前会先初始化集群目录。
12. 安装过程中会创建用户租户（test 租户）。

连接 OceanBase 数据库实例

OceanBase 镜像包含 obd（OceanBase Deployer，OceanBase 安装部署工具）和 OBClient（OceanBase 命令行客户端）。您可选择进入容器，使用 obd 命令管理和 OBClient 客户端连接实例，也可使用宿主机本地 OBClient 或 MySQL 客户端连接到 OceanBase 数据库实例。

进入容器后连接

1. 进入 Docker 容器

```
[admin@test001 ~]$ sudo docker exec -it obstandalone bash
```

2. 查看集群详情

```
# 查看集群列表
obd cluster list
# 查看 obcluster 集群详情
obd cluster display obcluster
```

3. 连接集群

```
obclient -h127.0.0.1 -uroot@sys -A -Doceanbase -P2881 -p
```

使用宿主机本地客户端连接

您可选择使用宿主机本地 OBClient 或 MySQL 客户端连接到 OceanBase 数据库实例，示例如下：

```
[admin@test001 ~]# obclient -uroot@sys -h127.1 -P2881 -p
```

输出如下：

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 3221506989
Server version: 5.7.25 OceanBase_CE 4.2.1.3   (r103000032023122818-8fe69c2056b0715
4bbd1ebd2c26e818ee0d5c56f) (Built Dec 28 2023 19:07:26)

Copyright (c) 2000, 2018, OceanBase and/or its affiliates. All rights reserved.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

obclient [(none)]>
```

OceanBase 数据库进程特点

此处以 Docker 容器部署 OceanBase 数据库为例，执行 `sudo docker exec -it obstandalone bash` 命令登录到 Docker 内查看 OceanBase 数据库的进程及目录信息。

注意

此处示例把 OceanBase 数据库安装在 root 用户目录下，并以 root 用户运行，仅是用作学习。生产环境中请勿以 root 用户部署和运行 OceanBase 数据库。

- 查看 observer 进程特点，分析 OceanBase 集群节点进程

  可通过如下命令确定其启动位置、启动文件和启动参数等。

  查看进行信息，确认启动位置：

```
[root@7bfd1eb06ada ~]# ps -ef|grep observer |grep -v grep
```

输出如下，进程启动目录在 `/root/ob` 下：

```
root       263     1 36 08:16 ?        00:46:30 /root/ob/bin/observer -r 127.0.0.1
:2882:2881 -p 2881 -P 2882 -z zone1 -n obcluster -c 1 -d /root/ob/store -l INFO -
I 127.0.0.1 -o __min_full_resource_pool_memory=2147483648,memory_limit=6G,system_m
emory=1G,datafile_size=5G,log_disk_size=5G,cpu_count=16,enable_syslog_wf=False,ena
ble_syslog_recycle=True,max_syslog_file_count=4,enable_rich_error_msg=True
```

查看启动文件：

```
[root@7bfd1eb06ada ~]# ls -l  /proc/`pidof observer`/{cwd,exe,cmdline}
```

输出如下：

```
-r--r--r-- 1 root root 0 Jan 24 08:16 /proc/263/cmdline
lrwxrwxrwx 1 root root 0 Jan 24 08:16 /proc/263/cwd -> /root/ob
lrwxrwxrwx 1 root root 0 Jan 24 08:16 /proc/263/exe -> /root/ob/bin/observer
```

查看启动参数：

```
[root@7bfd1eb06ada ~]# cat /proc/`pidof observer`/cmdline
```

输出如下，对应参数的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/配置项总览。

```
/root/ob/bin/observer-r127.0.0.1:2882:2881-p2881-P2882-zzone1-nobcluster-c1-d/root
/ob/store-lINFO-I127.0.0.1-o__min_full_resource_pool_memory=2147483648,memory_limi
t=6G,system_memory=1G,datafile_size=5G,log_disk_size=5G,cpu_count=16,enable_syslog
_wf=False,enable_syslog_recycle=True,max_syslog_file_count=4,enable_rich_error_msg
=True
```

- 查看进程监听端口。

> **说明**
>
> 若容器中未安装 netstat 命令，可执行 `yum install -y net-tools` 命令进行安装。

```
[root@7bfd1eb06ada ~]# netstat -ntlp
```

输出如下，可以看到 observer 进程会监听连接端口 `2881` 和 RPC 通信端口 `2882`。

```
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PI
D/Program name
tcp        0      0 0.0.0.0:2881            0.0.0.0:*LISTEN      263/observer
tcp        0      0 0.0.0.0:2882            0.0.0.0:*               LISTEN      26
3/observer
tcp6       0      0 :::8088                 :::*LISTEN      797/ob_monagent
tcp6       0      0 :::8089                 :::*                    LISTEN      79
6/ob_mgragent
```

- 查看 OceanBase 数据库工作目录结构

> **说明**
>
> 若容器中未安装 tree 命令，可执行 `yum install -y tree` 命令进行安装。

```
[root@7bfd1eb06ada ~]# tree /root/ob/
```

输出如下：

```
/root/ob/
|-- admin
|    |-- __dbms_upgrade_body_mysql.sql
|    |--__dbms_upgrade_mysql.sql
|    |-- dbms_application_body_mysql.sql
|    |-- dbms_application_mysql.sql
|    |-- dbms_ash_internal.sql
|    |-- dbms_ash_internal_body.sql
|    |-- dbms_monitor_body_mysql.sql
|    |-- dbms_monitor_mysql.sql
|    |-- dbms_resource_manager_body_mysql.sql
|    |-- dbms_resource_manager_mysql.sql
|    |-- dbms_rls.sql
|    |-- dbms_rls_body.sql
|    |-- dbms_scheduler_mysql.sql
|    |-- dbms_scheduler_mysql_body.sql
|    |-- dbms_session_body_mysql.sql
|    |-- dbms_session_mysql.sql
|    |-- dbms_stats_body_mysql.sql
|    |-- dbms_stats_mysql.sql
|    |-- dbms_udr.sql
|    |-- dbms_udr_body.sql
|    |-- dbms_udr_body_mysql.sql
|    |-- dbms_udr_mysql.sql
```

```
|    |-- dbms_workload_repository.sql
|    |-- dbms_workload_repository_body.sql
|    |-- dbms_workload_repository_body_mysql.sql
|    |-- dbms_workload_repository_mysql.sql
|    |-- dbms_xplan_mysql.sql
|    |-- dbms_xplan_mysql_body.sql
|    |-- json_element_type.sql
|    |-- json_element_type_body.sql
|    |-- json_object_type.sql
|    `-- json_object_type_body.sql
|-- audit
|`-- observer_262_20240124081603704889888.aud
|-- bin
|    |-- import_srs_data.py
|    |-- import_time_zone_info.py
|    `-- observer
|-- boot.yaml
|-- etc
|    |-- default_srs_data_mysql.sql
|    |-- fill_help_tables-ob.sql
|    |-- observer.config.bin
|    |-- observer.config.bin.history
|    |-- oceanbase_upgrade_dep.yml
|    |-- timezone_V1.log
|    |-- upgrade_checker.py
|    |-- upgrade_health_checker.py
|    |-- upgrade_post.py
|`-- upgrade_pre.py
|-- etc2
|    |-- observer.conf.bin
|    `-- observer.conf.bin.history
|-- etc3
|    |-- observer.conf.bin
|`-- observer.conf.bin.history
|-- lib
|    |-- libaio.so
|    |-- libaio.so.1
|    |-- libaio.so.1.0.1
|    |-- libmariadb.so
|    `-- libmariadb.so.3
|-- log
|    |-- election.log
|    |-- election.log.wf
|    |-- observer.log
|    |-- observer.log.20240124092657594
|    |-- observer.log.20240124094706195
|    |-- observer.log.20240124100732088
|    |-- observer.log.20240124102758021
|    |-- observer.log.wf
|    |-- rootservice.log
```

```
|   |-- rootservice.log.wf
|`-- trace.log
|-- run
|   |-- lua.sock
|   |-- observer.pid
|   `-- sql.sock
`-- store
    |-- clog
    |   |-- log_pool
    |   |   |-- 1
    |   |   |-- 2
    ... 此处省略
    |   |   |-- 80

    |   |   `-- meta
    |   |-- tenant_1
    |   |   `-- 1
    |   |       |-- log
    |   |       |   `-- 0
    |   |       `-- meta
    |   |           `-- 0
    |   |-- tenant_1001
    |   |   `-- 1
    |   |       |-- log
    |   |       |   |-- 0
    |   |       |   `-- 1
    |   |       `-- meta
    |   |           `-- 0
    |   `-- tenant_1002
    |       |-- 1
    |       |   |-- log
    |       |   |   `-- 0
    |       |   `-- meta
    |       |       `-- 0
    |       `-- 1001
    |           |-- log
    |           |   `-- 0
    |           `-- meta
    |               `-- 0
    |-- slog
    |   |-- server
    |   |   `-- 1
    |   |-- tenant_1
    |   |   `-- 1
    |   |-- tenant_1001
    |   |   `-- 1
    |   `-- tenant_1002
    |       `-- 1
    `-- sstable
        `-- block_file
```

各目录介绍如下：

目录路径（相对于工作目录）	说明
bin	observer 主进程文件
boot.yaml	部署所使用的配置文件
etc、etc2、etc3	配置文件所在目录
log	运行日志目录
run	运行输出目录，输出 pid 文件
store	数据（包括日志）所在总目录
store/clog	commit log 所在目录
store/ilog	ilog 所在目录
store/slog	slog 所在目录
store/sstable	数据文件 block file 所在目录

说明

- 若您通过手动安装 OceanBase 数据库进行部署，该工作目录下的子目录结构需手动维护。否则，OceanBase 数据库可能会启动失败。
- 若您通过自动化部署软件 obd 部署 OceanBase 数据库，obd 会自动创建相应目录。

- 若您通过手动安装 OceanBase 数据库进行部署，该工作目录下的子目录结构需手动维护。否则，OceanBase 数据库可能会启动失败。
- 若您通过自动化部署软件 obd 部署 OceanBase 数据库，obd 会自动创建相应目录。

2.4 部署生产环境

OceanBase 数据库高可用部署方案

OceanBase 数据库社区版采用基于无共享（Shared-Nothing）的多副本架构，整个系统没有任何单点故障，保证系统的持续可用。OceanBase 数据库社区版支持单机（单机房部署 OceanBase 集群）、机房（同城多机房部署 OceanBase 集群，机房以下统称：IDC）、城市（多城市部署 OceanBase 集群）级别的高可用和容灾，可以进行单机房、双机房、两地三中心、三地五中心部署。

目前 OceanBase 数据库提供七种高可用部署方案，其中社区版适用如下四种方案。高可用部署方案的详细介绍可参见官网《OceanBase 数据库》文档 部署数据库/OceanBase 集群高可用部署方案简介

- 同城三机房三副本部署
- 三地五中心五副本部署
- 同城两机房“主-备”部署
- 两地三中心“主-备”部署

OceanBase 数据库自 4.1.0 版本开始，物理备库的产品形态变更为租户级主备，即主或备的角色信息属于租户，分为主租户和备租户，集群不再有主备角色的概念，而只是承载租户的容器。详细介绍可参见官网《OceanBase 数据库》文档 管理数据库/高可用/物理备库容灾 章节。

> **注意**
>
> OceanBase 数据库社区版不支持仲裁服务，仲裁服务相关的高可用部署方案无法使用。

规划 OceanBase 集群部署

在本章第一节 部署前准备 中已经详细介绍了部署生产环境所需的部署环境要求，本节介绍如何规划 OceanBase 集群的部署。

OceanBase 数据库以集群形态运行，生产环境中最小规模为 3 台服务器（节点）。即集群中业务数据有三份，所以也叫三副本。

生产环境中，每台机器上启动一个 `observer` 进程，所以一台机器就对应一个节点，每个节点的监听端口（默认是 `2881` 和 `2882`）、数据目录和事务日志目录是独立磁盘，每个节点 `observer` 进程启动的最小内存是 6 GB，生产场景建议不低于 16 GB，推荐 32 GB 以上。

注意

最小内存指的是机器执行 `free -g` 命令输出中剩余内存（`free` 列）的值，而非服务器总内存（`total` 列）的值。

在部署 OceanBase 数据库后还需要部署 OBProxy。OBProxy 也是单进程软件，是访问 OceanBase 数据库的反向代理。虽然 OBServer 节点可以直接访问，但是生产环境中还是建议通过 OBProxy 访问 OceanBase 集群。

OBProxy 对于部署位置没有要求，您可以部署在应用服务器上，也可以部署在独立的机器上，或者部署在 OceanBase 数据库机器上。OBProxy 可以部署多个，生产环境中建议至少部署两个。

通过 OCP 部署集群

常见 OCP 部署集群架构：

四节点环境	部署组件	说明
节点 A	OCP、MetaDB	OCP 和 MetaDB 共同使用一个节点，不推荐 MetaDB 和业务集群混用。
节点 B	OceanBase 数据库、OBProxy、OCP-Agent	作为 OceanBase 集群节点、OBProxy 服务，OCP-Agent 服务节点。
节点 C	OceanBase 数据库、OBProxy、OCP-Agent	作为 OceanBase 集群节点、OBProxy 服务，OCP-Agent 服务节点。
节点 D	OceanBase 数据库、OBProxy、OCP-Agent	作为 OceanBase 集群节点、OBProxy 服务，OCP-Agent 服务节点。

为什么推荐使用 OCP 部署和管理集群?

- 企业级集群管理

  OCP 是专为 OceanBase 数据库设计的企业级集群管理平台，提供了全面的集群管理功能，如安装、运维、性能监控、配置、升级、删除以及主机和租户的全生命周期管理，有助于提高运维效率并降低 IT 成本。

- 租户与资源管理

  OCP 支持租户级别的管理，包括创建、结构拓扑、性能监控、会话管理和参数管理等，同时还能管理与 OceanBase 数据库相关的资源，如主机、网络和软件包，确保资源的有效利用和优化。

- 监控告警

  OCP 实现了多维度的监控告警机制，包括针对集群、租户和主机的实时监控和定制化告警策略，帮助及时发现和处理潜在问题。

- 自动化运维

  使用 OCP 可以实现一键安装、升级、扩容和卸载 OceanBase 数据库集群，简化运维流程，减少人为错误。

- 安全性增强

  OCP 提供用户和角色管理功能，允许精细控制数据库访问权限，并且支持用户个人设置、密码和告警订阅的管理。

- 可视化界面与用户体验:

  OCP 采用面向对象的架构设计，提供清晰的功能模块和流畅的操作作路径，通过可视化的方式展示集群和租户的拓扑结构及资源使用情况，使得运维人员能更直观、高效地管理数据库集群。

- 可接管其他集群

  OCP 支持接管通过 obd、obshell 和 OCP 部署的集群。

综上所述，部署 OCP 可以显著提升 OceanBase 数据库集群的管理水平和运维效率，保障系统的稳定性和安全性，从而更好地支撑业务的发展需求。

安装 OCP

安装 OCP 需要了解如下几个问题：

- 什么是 MetaDB？

  MetaDB 指的是专门用于存储 OCP 的元数据和监控数据的数据库，目前仅支持 OceanBase 数据库作为 MetaDB。OceanBase 数据库可以是单机或者集群形态，也可以和 OCP 服务部署在同一服务器或分开部署。

  > **注意**
  >
  > 生产环境不建议将 MetaDB 作为业务数据库使用。

- 什么是 MetaDB 元租户？

  MetaDB 元租户包含 meta 租户和 monitor 租户，租户名称可自定义，分别用于元数据和监控数据管理。

  > **注意**
  >
  > 禁止使用一个租户的两个不同用户来替代 meta 租户和 monitor 租户进行管理。即：meta 租户和 monitor 租户必须使用两个不同租户分开管理。

- 怎么实现 OCP 高可用？

  OCP 的高可用需要 OCP 服务和 MetaDB 均实现高可用，即：OCP 服务至少为两节点，MetaDB 至少为三节点，通常以 3 台服务器部署三节点 OCP 和三节点 MetaDB 集群，即可实现 OCP 高可用。

- OCP 服务器资源要求是什么？

  以 OCP 管理的 OBServer 节点数量在 10 台以内为例，MetaDB 和 OCP 共用同台服务器时，这台物理机需要有 17 个 CPU 和 60 GB 内存。详细的资源要求可参见官网《OceanBase 云平台》文档 部署 OceanBase 云平台/部署社区版 OceanBase 云平台/安装规划/主机规划。

模块	CPU	内存
OCP-Server 服务	4	8G
MetaDB 的系统租户	5	28G

MetaDB 的 meta 元租户	4	8G
MetaDB 的 monitor 元租户	4	16G

> **注意**
>
> 部署 MetaDB 的其他资源要求可以参考 OceanBase 数据库的部署要求。

推荐图形化界面方式部署最新版本 OCP，具体部署流程可参看官网《OceanBase 云平台》文档 部署 OceanBase 云平台/部署社区版 OceanBase 云平台/安装流程。

使用 OCP 部署集群

使用 OCP 部署集群时，有如下几个常见问题。

- OCP 部署集群需要上传哪些软件包？

  可在登录 OCP 后，单击页面中 **系统管理 > 软件包管理**，查看是否包含如下表所示的安装包，如果缺少安装包会导致部署集群过程中任务失败。

组件	包名	描述
OCP-Agent	• ocp-agent-ce-*.x86_64.rpm • ocp-agent-ce-*.aarch64.rpm	OCP 客户端代理，OCP 部署完成默认上传
OceanBase	• oceanbase-ce-*.rpm • oceanbase-ce-libs-*.rpm • oceanbase-ce-utils-*.rpm	包名分别对应： • OceanBase 数据库安装包 • 依赖库 (OceanBase Libs) • 工具集成包（OceanBase Utils）
OBProxy	obproxy-ce-*.rpm	OceanBase 数据库代理

- ocp-agent-ce-*.x86_64.rpm
- ocp-agent-ce-*.aarch64.rpm
- oceanbase-ce-*.rpm
- oceanbase-ce-libs-*.rpm
- oceanbase-ce-utils-*.rpm
- OceanBase 数据库安装包
- 依赖库 (OceanBase Libs)
- 工具集成包（OceanBase Utils）

- OCP 添加主机需要注意什么？
    - 主机节点需要关闭防火墙、Selinux。
    - 主机节点需要使用静态 IP。
    - 主机节点需要和 OCP 节点做时钟源同步。
    - 主机节点需要提前创建凭据操作系统用户。
    - 主机节点的部署（凭据）用户需要有 `sudo` 权限。
    - 主机节点和 OCP 节点的部署（凭据）用户需要有 clockdiff 命令操作权限。如果没有权限可以使用 root 用户执行赋权 `setcap cap_net_raw+ep /usr/sbin/clockdiff`。

- 主机节点需要关闭防火墙、Selinux。
- 主机节点需要使用静态 IP。
- 主机节点需要和 OCP 节点做时钟源同步。
- 主机节点需要提前创建凭据操作系统用户。
- 主机节点的部署（凭据）用户需要有 `sudo` 权限。
- 主机节点和 OCP 节点的部署（凭据）用户需要有 clockdiff 命令操作权限。如果没有权限可以使用 root 用户执行赋权 `setcap cap_net_raw+ep /usr/sbin/clockdiff`。

- OCP 新建集群需要注意什么？

  如果需要对系统参数做定制，例如：系统日志文件保留个数，磁盘预占用大小等，可以参考 部署前准备 中 **OceanBase 数据库常用资源参数简介及计算方式** 内容对涉及的系统参数进行自定义。

- 部署完成后需要注意什么？
    - 新建用户租户，默认的 sys 租户资源较小且仅用于集群管理使用，需要新建租户为业务使用。

    - 创建 OBProxy 集群，建议最少部署两节点的 OBProxy 集群，如果业务量很大，推荐独立服务器部署 OBProxy 集群。

- 新建用户租户，默认的 sys 租户资源较小且仅用于集群管理使用，需要新建租户为业务使用。
- 创建 OBProxy 集群，建议最少部署两节点的 OBProxy 集群，如果业务量很大，推荐独立服务器部署 OBProxy 集群。

使用 OCP 部署集群以及后续创建租户和 OBProxy 集群的具体流程可参见官网《OceanBase 云平台》文档 快速入门 章节。

通过 obd 部署集群

obd 作为 OceanBase 的安装部署工具，目前已经具备多个生态组件的部署和管理，后续也会持续纳入其他生态组件。obd 在支持白屏化界面部署，降低自动化部署难度的同时，还提供了包管理器、压测软件、集群管理等常用的运维能力。

常见 obd 部署集群架构如下表所示：

四节点环境	部署组件	说明
节点 A	obd、OCP Express	作为中控节点，主要用于 obd 部署管理和运维监控服务。OCP Express 没有单独的 MetaDB，会在业务集群创建 OCP Express 元租户。
节点 B	OceanBase 数据库、OBProxy、OBAgent	作为 OceanBase 集群节点、OBProxy 服务 和 OBAgent 服务节点。
节点 C	OceanBase 数据库、OBProxy、OBAgent	作为 OceanBase 集群节点、OBProxy 服务 和 OBAgent 服务节点。
节点 D	OceanBase 数据库、OBProxy、OBAgent	作为 OceanBase 集群节点、OBProxy 服务 和 OBAgent 服务节点。

说明

obd 中控节点对服务器配置要求不高，建议不低于 4C8G 即可。

安装 obd

obd 除了通过 all-in-one 安装包使用脚本快捷部署，也可以使用 RPM 安装包本地部署或者配置 YUM 源在线部署。

安装 obd 的三种方式可参见官网《OceanBase 安装部署工具》文档 快速上手/安装 obd。

使用 obd 部署集群

使用 obd 部署集群时，有如下几个常见问题。

- obd 部署集群时，什么情况下选择 OCP Express 服务？

  OCP Express 服务是 OCP 轻量版，仅具备面向轻量级运维管控的能力，特别适合开发测试环境和中小规模生产环境（单机或者三节点 OceanBase 集群）。OCP Express 目前暂不支持集群部署、集群升级、集群扩缩容、集群备份恢复管理、监控告警等核心运维管控功能。

- obd 部署集群需要注意哪些问题？

  - obd 是通过 SSH 远程执行安装部署，OCP Express 服务依赖 `java-1.8.0-openjdk`，所以需通过 SSH 验证 Java 环境是否可用。

  - 当 obd 所在节点无法连接网络，使用 obd 部署组件服务时需提前下载好所需服务的安装包，并将其添加到 obd 的本地仓库（`obd mirror clone *.rpm`），并关闭远程仓库（`obd mirror disable remote`）

- obd 是通过 SSH 远程执行安装部署，OCP Express 服务依赖 `java-1.8.0-openjdk`，所以需通过 SSH 验证 Java 环境是否可用。

- 当 obd 所在节点无法连接网络，使用 obd 部署组件服务时需提前下载好所需服务的安装包，并将其添加到 obd 的本地仓库（`obd mirror clone *.rpm`），并关闭远程仓库（`obd mirror disable remote`）

使用 obd 部署集群具体步骤可参见文档，根据选择的部署方法不同有如下两篇参考文档。

- 白屏部署：可参见官网《OceanBase 安装部署工具》文档 快速上手/通过白屏部署 OceanBase 集群。

- 命令行部署：可参见官网《OceanBase 数据库》文档 部署数据库/部署 OceanBase 社区版/本地部署/使用命令行部署 OceanBase 数据库生产环境。

> **说明**
>
> 若在使用 obd 部署集群过程中遇到报错，可访问官网 错误码 文档查看解决方法。

常用 obd 命令介绍

obd 的命令非常丰富，包括快捷部署命令、集群命令组、镜像和仓库命令组、测试命令组、工具命令组、诊断工具命令组等。接下来会介绍一些常用 obd 命令和注意事项。

```
#查看仓库列表
obd mirror list

#关闭远程仓库访问
obd mirror disable remote

#开启远程仓库访问
obd mirror enable remote

#查看部署服务列表
obd cluster list

#查看部署服务详情
obd cluster display <deploy name>

#指定配置文件安装服务
obd cluster deploy <deploy name> -c <deploy file>

#销毁部署服务，高危操作，会清理指定部署服务的所有数据。
obd cluster destroy <deploy name>

#编辑或查看部署配置文件
obd cluster edit-config <deploy name>

#启动/停止/重启服务
obd cluster start/stop/restart <deploy name>

#启动/停止/重启指定服务，服务名称可以通过 obd cluster edit-config 方式查看模块名
obd cluster start/stop/restart <deploy name> -c <component name>

#启动/停止/重启指定服务指定节点，服务名称可以通过 obd cluster edit-config 方式查看模块名
obd cluster start/stop/restart <deploy name> -c <component name> -s IP1,IP2
```

```
#重载配置，在通过 obd cluster edit-config 修改配置后，黑屏会输出执行提醒
obd cluster reload <deploy name>

#创建一个最小化 2C 6G 资源的单机 OceanBase 环境
obd demo

#demo 环境部署指定服务名称、端口。
obd demo -c oceanbase-ce --oceanbase-ce.mysql_port=3881 --oceanbase-ce.rpc_port=38
82

#创建一个用户租户，会最大化使用剩余资源。
obd cluster tenant create <deploy name> -n <tenant name>

#一键性能测试，目前支持 mysqltest、sysbench、tpch、tpcc
obd test mysqltest/sysbench/tpch/tpcc

#一键集群巡检
obd obdiag check --cases=<deploy name>

#一键收集部署名称所属 OceanBase 集群的诊断信息
obd obdiag gather all <deploy name>

#一键诊断分析部署名称所属 OceanBase 集群指定时间段的系统日志
obd obdiag analyze log <deploy name> --from 2024-02-06 18:00:00 --to 2024-02-06 18
:10:00

#一键对部署名称所属 OceanBase 集群进行 trace 日志全链路诊断
obd obdiag analyze flt_trace <deploy name>

#一键根因分析，支持 OBProxy 断连/集群合并卡住/锁冲突诊断
obd obdiag rca --scene=disconnection/major_hold/lock_conflict
```

详细 obd 命令集介绍请参见官网《OceanBase 安装部署工具》文档 obd 命令 章节。

通过 ob-operator 部署 OceanBase 数据库

生产环境部署前提条件

在开始之前，请确保您已满足以下条件：

- 您有可用的 Kubernetes 集群且至少有 9 个可用 CPU，33 GB 可用内存 和 360 GB 的可用存储空间。
- 您已在 Kubernetes 集群中安装 cert-manager。cert-manager 的安装方法请参考对应的

安装文档。

- 您的 Kubernetes 集群中有至少一个可用的存储类。如果没有可用存储类，可安装 local-path-provisioner 并确认其配置的目标地点有足够的存储空间，它将提供 `local-path` 存储类，使用集群节点的本地存储。local-path-provisioner 的安装方法请参考对应的 安装文档。
- 您已部署 ob-operator，详细步骤可参见 ob-operator 部署。

部署要求

ob-operator 部署时需要注意：

服务	要求	说明
ob-operator	需使用 2.x 版本。	建议使用最新版本，功能更为完善。
OceanBase 数据库	• 需为 V4.0.0 或之后版本，推荐使用 LTS 版本（如 V4.2.1）。 • 推荐部署 3 节点集群，且按物理机推荐资源要求设置。	当 OceanBase 数据库所需资源为 16C32G 时，容器最低资源配置为 16C36GB（内存默认认按 90% 分配给 OcaeanBase 数据库）。
k8s	ob-operator 对 Calico 网络插件进行了适配，如若希望普通模式的集群也能保持 pod ip 特性，则需要 k8s 使用 Calico 作为网络插件。	ob-operator 支持 pod 的亲和性设定，推荐使用 nodeSelector 将 OBServer 分配到不同的宿主机节点。

使用 ob-operator 部署 OceanBase 数据库的详细介绍可参见 集群创建。

2.5 查看 OceanBase 集群资源的使用情况

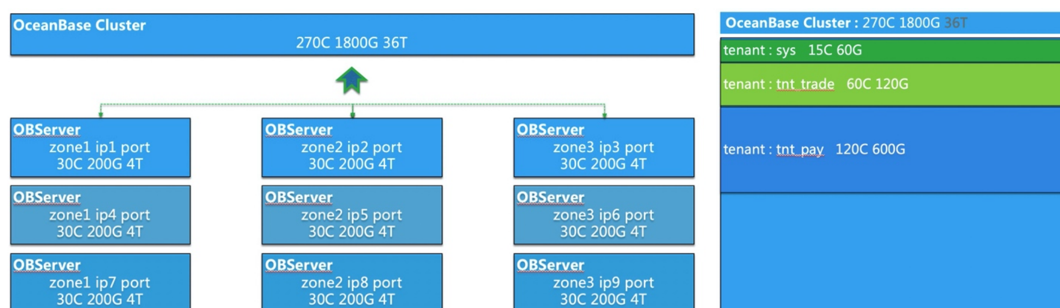
在业务开发之前，DBA 需要在 OceanBase 集群中创建数据库实例，即 OceanBase 租户。OceanBase 集群可以分配出多个租户，这个能力称为多租户。

多租户原理简介

OceanBase 数据库以集群形态部署运行，将集群形态中的租户提供给业务使用。租户在一定程度上相当于传统数据库的“实例”概念。租户之间是完全隔离的，在数据安全方面，OceanBase 数据库不允许跨租户的数据访问，可以确保用户的数据资产没有被其他租户窃取的风险；在资源使用方面，OceanBase 数据库表现为租户“独占”其资源配额。总体上来说，租户（tenant）既是各类数据库对象的容器，又是资源（CPU、Memory、I/O 等）的容器。

OceanBase 数据库是单进程软件，进程名为 `observer`。进程启动后，默认会将操作系统的大部分 CPU、内存和磁盘资源据为己有。当然，资源的使用情况也可以通过集群启动参数进行设置。

如下图所示，OceanBase 集群能把所有节点进程取得的资源集中管理，然后从集群中分配出多个租户，每个租户对应一定的资源。资源的大小可以自行定义，并且资源可以在线调整，该功能也是弹性伸缩能力的体现。



OceanBase 数据库的租户资源定义包含 CPU、内存、IOPS。目前 OceanBase 数据库实现了 CPU 、内存、IOPS 的资源隔离。建议创建资源时根据实际情况设置这些参数，尤其是空间资源，不要超出机器磁盘实际可用空间过多，否则将影响后期负载均衡。

说明

`observer` 进程取得的资源中，CPU 个数是声明式的，内存资源是独占的，磁盘空间是独占的（预分配）。

查看集群可用资源

计算集群剩余可用资源是为了避免创建用户租户时发生资源不足的情况。OceanBase 集群默认有个内部租户（sys），可以查看和管理集群的资源。您可使用 root 用户登录 OceanBase 集群的 sys 租户，执行如下 SQL 查看资源。

查看集群可用资源

```
SELECT ZONE,SVR_IP,SVR_PORT,
  CPU_CAPACITY,
  CPU_ASSIGNED,
  CPU_CAPACITY-CPU_ASSIGNED AS CPU_MIN_FREE,
  CPU_CAPACITY_MAX,
  CPU_ASSIGNED_MAX,
  CPU_CAPACITY_MAX-CPU_ASSIGNED_MAX AS CPU_MAX_FREE,
  ROUND(MEMORY_LIMIT/1024/1024/1024,2) AS MEMORY_TOTAL_GB,
  ROUND((MEMORY_LIMIT-MEM_CAPACITY)/1024/1024/1024,2) AS SYSTEM_MEMORY_GB,
  ROUND(MEM_ASSIGNED/1024/1024/1024,2) AS MEM_ASSIGNED_GB,
  ROUND((MEM_CAPACITY-MEM_ASSIGNED)/1024/1024/1024,2) AS MEMORY_FREE_GB,
  ROUND(LOG_DISK_CAPACITY/1024/1024/1024,2) AS LOG_DISK_CAPACITY_GB,
  ROUND(LOG_DISK_ASSIGNED/1024/1024/1024,2) AS LOG_DISK_ASSIGNED_GB,
  ROUND((LOG_DISK_CAPACITY-LOG_DISK_ASSIGNED)/1024/1024/1024,2) AS LOG_DISK_FREE_G
B,
  ROUND((DATA_DISK_CAPACITY/1024/1024/1024),2) AS DATA_DISK_GB,
  ROUND((DATA_DISK_IN_USE/1024/1024/1024),2) AS DATA_DISK_USED_GB,
  ROUND((DATA_DISK_CAPACITY-DATA_DISK_IN_USE)/1024/1024/1024,2) AS DATA_DISK_FREE_
GB
  FROM oceanbase.GV$OB_SERVERS;
```

输出如下：

```
+--------+----------------+----------+--------------+--------------+---------------+
------------------+-------------------+--------------+-----------------+-----------
-------+-----------------+----------------+----------------------+----------------
------+------------------+--------------+-------------------+-------------------+
| ZONE   | SVR_IP         | SVR_PORT | CPU_CAPACITY | CPU_ASSIGNED | CPU_MIN_FREE
| CPU_CAPACITY_MAX | CPU_ASSIGNED_MAX | CPU_MAX_FREE | MEMORY_TOTAL_GB | SYSTEM_ME
MORY_GB | MEM_ASSIGNED_GB | MEMORY_FREE_GB | LOG_DISK_CAPACITY_GB | LOG_DISK_ASSIG
NED_GB | LOG_DISK_FREE_GB | DATA_DISK_GB | DATA_DISK_USED_GB | DATA_DISK_FREE_GB |
+--------+----------------+----------+--------------+--------------+---------------+
------------------+-------------------+--------------+-----------------+-----------
-------+-----------------+----------------+----------------------+----------------
------+------------------+--------------+-------------------+-------------------+
| zone1 | 10.10.10.1      |     2882 |            8 |            5 |             3
|                16 |                    5 |           11 |           20.00
```

```
|            2.00 |            6.00 |         12.00 |                 48.00
|               15.00 |            33.00 |        30.00 |              6.82
|            23.18 |
| zone2 | 10.10.10.2      |    2882   |            8 |             5 |             3
|              16 |               5 |           11 |        20.00
|            2.00 |            6.00 |         12.00 |                 48.00
|               15.00 |            33.00 |        30.00 |              8.91
|            21.09 |
| zone3 | 10.10.10.3      |    2882   |            8 |             5 |             3
|              16 |               5 |           11 |        20.00
|            2.00 |            6.00 |         12.00 |                 48.00
|               15.00 |            33.00 |        30.00 |              8.91
|            21.09 |
+-------+-----------------+----------+--------------+--------------+--------------+
------------------+------------------+--------------+-----------------+------------
-------+-----------------+----------------+----------------------+---------------
------+------------------+--------------+-------------------+-------------------+
```

字段说明：

字段名称	类型	是否可以为 NULL	描述
ZONE	varchar(128)	NO	Zone 名称
SVR_IP	varchar(46)	NO	服务器 IP 地址
SVR_PORT	bigint(20)	NO	服务器端口号
CPU_CAPACITY	bigint(20)	NO	节点 CPU 总容量
CPU_CAPACITY_MAX	double	NO	节点 CPU 总容量的超卖值。它由 resource_hard_limit 配置项控制：CPU_CAPACITY_MAX = CPU_CAPACITY * resource_hard_limit
CPU_ASSIGNED	double	NO	OBServer 节点已经分配的 CPU 数量，它是 OBServer 节点上所有 Unit 的MIN_CPU 规格总和。有如下约束：CPU_ASSIGNED <= CPU_CAPACITY
CPU_ASSIGNED_MAX	double	NO	OBServer 节点已经分配的 CPU 上界值，它是 OBServer 节点上所有 Unit 的MAX_CPU 规格总和。有如下约束：CPU_ASSIGNED_MAX <= CPU_CAPACITY_MAX

MEM_CAPACITY	bigint(20)	NO	observer 进程可用的内存大小
MEM_ASSIGNED	bigint(20)	NO	OBServer 节点已分配的内存大小，它是 OBServer 节点上所有 Unit 的MEMORY_SIZE 规格总和。有如下约束：MEM_ASSIGNED <= MEM_CAPACITY
LOG_DISK_CAPACITY	bigint(20)	NO	日志盘空间总大小
LOG_DISK_ASSIGNED	bigint(20)	NO	日志盘已分配大小，它是 OBServer 节点上所有 Unit 的 MAX_DISK_SIZE 规格总和
LOG_DISK_IN_USE	bigint(20)	NO	日志盘已使用大小
DATA_DISK_CAPACITY	bigint(20)	NO	数据盘空间总大小
DATA_DISK_IN_USE	bigint(20)	NO	数据盘已使用大小

查看资源单元规格

```
SELECT * FROM oceanbase.DBA_OB_UNIT_CONFIGS;
```

输出如下：

```
+----------------+-----------------+----------------------------+------------------
-----------+---------+---------+-------------+---------------+--------------------
-+---------------------+-------------+
| UNIT_CONFIG_ID | NAME            | CREATE_TIME                | MODIFY_TIM
E                | MAX_CPU | MIN_CPU | MEMORY_SIZE | LOG_DISK_SIZE | MAX_IOP
S            | MIN_IOPS            | IOPS_WEIGHT |
+----------------+-----------------+----------------------------+------------------
-----------+---------+---------+-------------+---------------+--------------------
-+---------------------+-------------+
|              1 | sys_unit_config | 2024-02-19 15:33:47.524052 | 2024-02-19 15:33
:47.524052 |       2 |       2 |  1073741824 |    3221225472 | 922337203685477580
7 | 9223372036854775807 |           2 |
+----------------+-----------------+----------------------------+------------------
-----------+---------+---------+-------------+---------------+--------------------
-+---------------------+-------------+
```

字段说明：

字段名称	类型	是否可以为 NULL	描述
UNIT_CONFIG_ID	bigint(20)	NO	Unit 规格 ID
NAME	varchar(128)	NO	Unit 规格名称
CREATE_TIME	timestamp(6)	YES	Unit 规格创建时间
MODIFY_TIME	timestamp(6)	YES	信息更新时间
MAX_CPU	double	NO	CPU 规格上限
MIN_CPU	double	NO	CPU 规格下限
MEMORY_SIZE	bigint(20)	NO	内存规格，单位：字节
LOG_DISK_SIZE	bigint(20)	NO	日志盘规格，单位：字节
MAX_IOPS	bigint(20)	NO	磁盘 IOPS 规格上限
MIN_IOPS	bigint(20)	NO	磁盘 IOPS 规格下限
IOPS_WEIGHT	bigint(20)	NO	IOPS 权重

查看资源分配细节

```
SELECT T4.TENANT_ID,T4.TENANT_NAME,
  T1.NAME RESOURCE_POOL_NAME, T1.UNIT_COUNT,
  T2.`NAME` UNIT_CONFIG_NAME,
  T2.MAX_CPU, T2.MIN_CPU,
  ROUND(T2.MEMORY_SIZE/1024/1024/1024,2) MEM_SIZE_GB,
  ROUND(T2.LOG_DISK_SIZE/1024/1024/1024,2) LOG_DISK_SIZE_GB, T2.MAX_IOPS,
  T2.MIN_IOPS, T3.UNIT_ID, T3.ZONE, CONCAT(T3.SVR_IP,':',T3.`SVR_PORT`) OBSERVER
  FROM oceanbase.DBA_OB_RESOURCE_POOLS T1
  JOIN oceanbase.DBA_OB_UNIT_CONFIGS T2 ON (T1.UNIT_CONFIG_ID=T2.UNIT_CONFIG_ID)
  JOIN oceanbase.DBA_OB_UNITS T3 ON (T1.`RESOURCE_POOL_ID` = T3.`RESOURCE_POOL_ID`
)
  LEFT JOIN oceanbase.DBA_OB_TENANTS T4 ON (T1.TENANT_ID=T4.TENANT_ID)
  ORDER BY T4.TENANT_NAME,T3.ZONE;
```

输出如下，命令中使用的视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统参考/系统视图总览。

```
+-----------+-------------+--------------------+------------+--------------------+--
-------+---------+-------------+-------------------+---------------------+---------
------------+---------+-------+--------------------+
| TENANT_ID | TENANT_NAME | RESOURCE_POOL_NAME | UNIT_COUNT | UNIT_CONFIG_NAME | M
```

```
AX_CPU | MIN_CPU | MEM_SIZE_GB | LOG_DISK_SIZE_GB | MAX_IOPS            | MIN_IOP
S             | UNIT_ID | ZONE  | OBSERVER           |
+------------+-------------+--------------------+------------+------------------+--
-------+---------+-------------+------------------+----------------------+---------
------------+---------+-------+---------------------+
|         1 | sys         | sys_pool           |          1 | sys_unit_config
|       2 |         2 |        1.00 |              3.00 | 9223372036854775807 | 92233
72036854775807 |        1 | zone1 | 10.10.10.1 :2882    |
|         1 | sys         | sys_pool           |          1 | sys_unit_config
|       2 |         2 |        1.00 |              3.00 | 9223372036854775807 | 92233
72036854775807 |        2 | zone2 | 10.10.10.1 :2882    |
|         1 | sys         | sys_pool           |          1 | sys_unit_config
|       2 |         2 |        1.00 |              3.00 | 9223372036854775807 | 92233
72036854775807 |        3 | zone3 | 10.10.10.1 :2882    |
+------------+-------------+--------------------+------------+------------------+--
-------+---------+-------------+------------------+----------------------+---------
------------+---------+-------+---------------------+
```

该运行结果显示 sys 租户的资源池（`resource pool`），由每个 Zone 里的一个节点上的资源单元（`resource unit`）组成，每个资源单元使用同一规格（`sys_unit_config`）。

2.6 创建 MySQL 模式的用户租户

强烈推荐创建并使用用户租户，禁止使用管理租户（sys 租户）做测试或业务使用，推荐参考本节内容选择合适的方法创建并使用用户租户。

> **注意**
>
> OceanBase 数据库社区版仅支持创建 MySQL 兼容模式租户。

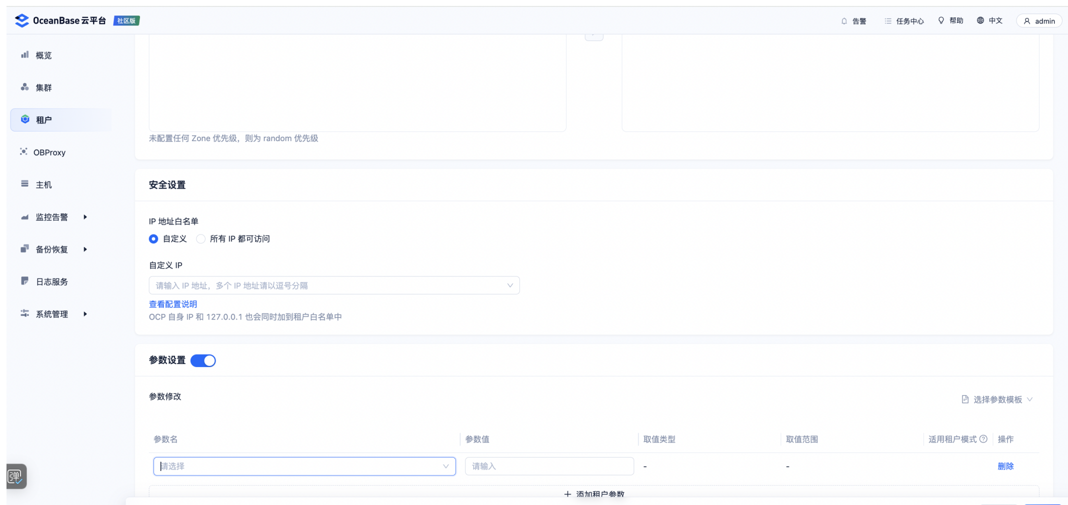
通过 OCP 白屏创建租户

> **说明**
>
> 本节仅简单介绍如何使用 OCP 创建租户，详细的操作步骤可参见官网《OceanBase 云平台》文档 快速入门/新建租户。

1. 参照下图配置待创建租户的信息

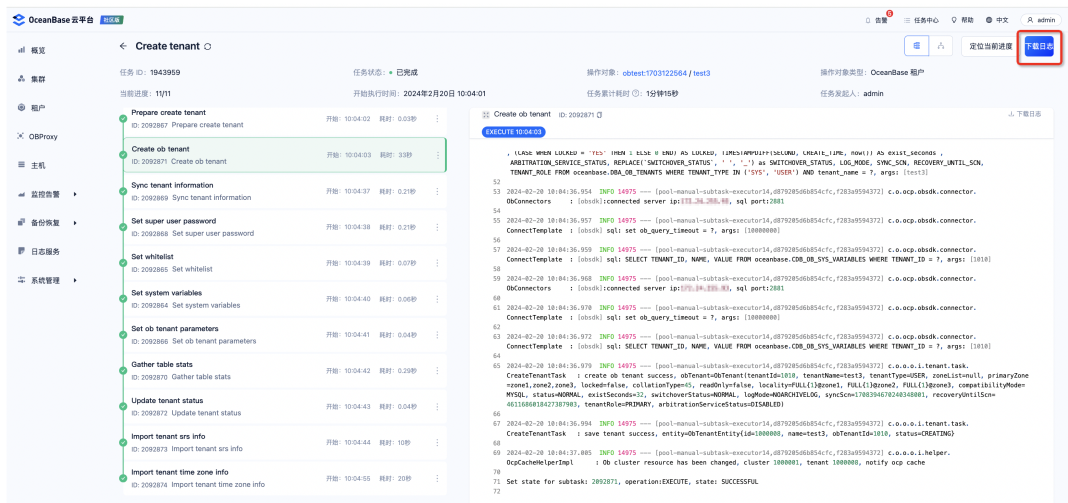
   创建租户时，如果没有合适的 Unit 规格，可以新建一个。在 OCP 中新建 Unit 后，并没有在 OceanBase 数据库的内部视图 `DBA_OB_UNIT_CONFIGS` 中真正生成对应的数据，对应的数据在 OCP 的 `meta_database.ob_unit_spec` 表中会有记录，只有在创建租户时才会真正去创建 Unit。

2. 按需配置租户 IP 白名单和租户变量



3. 查看 OCP 创建租户的任务

如果想了解具体的执行过程，可以下载日志。



从输出的日志中我们不难发现，OCP 在创建租户的时候和 obd 有所区别：OCP 给每个 Unit 和 resource_pool 分别命名，这样带来的好处主要有：

- Unit 不复用，调整一个租户的 Unit 规格不会影响其他租户。
- resource_pool 不复用，删除一个节点时，租户的 resource_pool 不需要执行 split 操作。

4. Unit 不复用，调整一个租户的 Unit 规格不会影响其他租户。
5. resource_pool 不复用，删除一个节点时，租户的 resource_pool 不需要执行 split 操作。

通过 obd 命令创建租户

若待新建租户的集群可使用 obd 管理，可直接使用 obd 命令来创建租户。若待新建租户的集群非 obd 管理的集群，且集群所用 OceanBase 数据库为社区版 V4.2.1 BP4 或之后版本，且已启动过 obshell，您可先执行接管命令后再使用如下 obd 命令创建租户，接管操作可参见官网《OceanBase 安装部署工具》文档 使用指南/命令行/使用 obd 接管集群。

```
obd cluster tenant create obtest -n test2 \
--max-cpu=2 --memory-size=2G --log-disk-size=3G --max-iops=10000 \
--unit-num=1 --charset=utf8 -s 'ob_tcp_invited_nodes="%"'
```

`obd cluster tenant create` 命令的详细介绍可参见官网《OceanBase 安装部署工具》文档 obd 命令/集群命令组 中 **obd cluster tenant create**。

- 命令里的 `obtest` 是示例的集群名（`deploy name`），可执行 `obd cluster list` 命令，根据输出中 `Name` 列值进行替换。
- `test2` 是示例中待创建租户的租户名。
- 使用 obd 命令创建的租户使用的资源单元名称为 `${tenant_name}_unit`，资源池名称为 `${tenant_name}_pool`。

输出如下：

```
Get local repositories and plugins ok
Open ssh connection ok
Connect to observer 10.10.10.1:2881 ok
Create tenant test2 ok
Trace ID: 4af55084-cf17-11ee-b825-00163e04608d
If you want to view detailed obd logs, please run: obd display-trace 4af55084-cf17
-11ee-b825-00163e04608d
```

可复制执行输出中的 `obd display` 命令，查看详细的创建过程日志：

```
obd display-trace 4af55084-cf17-11ee-b825-00163e04608d
```

日志输出如下：

```
[2024-02-19 19:09:04.385] [DEBUG] - cmd: ['obtest']
[2024-02-19 19:09:04.385] [DEBUG] - opts: {'tenant_name': 'test2', 'max_cpu': 2.0
, 'min_cpu': None, 'max_memory': None, 'min_memory': None, 'memory_size': '2G', 'm
ax_disk_size': None, 'log_disk_size': '3G', 'max_iops': 10000, 'min_iops': None, '
```

```
iops_weight': None, 'max_session_num': None, 'unit_num': 1, 'zone_list': None, 'mo
de': 'mysql', 'charset': 'utf8', 'collate': None, 'replica_num': None, 'logonly_re
plica_num': None, 'tablegroup': None, 'primary_zone': 'RANDOM', 'locality': None,
'variables': 'ob_tcp_invited_nodes="%"'}
[2024-02-19 19:09:04.385] [DEBUG] - mkdir /home/admin/.obd/lock/
[2024-02-19 19:09:04.385] [DEBUG] - unknown lock mode
[2024-02-19 19:09:04.386] [DEBUG] - try to get share lock /home/admin/.obd/lock/gl
obal
[2024-02-19 19:09:04.386] [DEBUG] - share lock `/home/admin/.obd/lock/global`, cou
nt 1
[2024-02-19 19:09:04.386] [DEBUG] - Get Deploy by name
[2024-02-19 19:09:04.386] [DEBUG] - mkdir /home/admin/.obd/cluster/
[2024-02-19 19:09:04.386] [DEBUG] - mkdir /home/admin/.obd/config_parser/
[2024-02-19 19:09:04.386] [DEBUG] - try to get exclusive lock /home/admin/.obd/loc
k/deploy_obtest
[2024-02-19 19:09:04.386] [DEBUG] - exclusive lock `/home/admin/.obd/lock/deploy_o
btest`, count 1
[2024-02-19 19:09:04.390] [DEBUG] - Deploy status judge
[2024-02-19 19:09:04.390] [DEBUG] - Get deploy config
[2024-02-19 19:09:04.407] [INFO] Get local repositories and plugins
[2024-02-19 19:09:04.407] [DEBUG] - mkdir /home/admin/.obd/repository
[2024-02-19 19:09:04.407] [DEBUG] - Get local repository oceanbase-ce-4.2.2.0-aa30
53da7370a6685a2ef457cd202d50e5ab75d3
[2024-02-19 19:09:04.407] [DEBUG] - try to get share lock /home/admin/.obd/lock/mi
rror_and_repo
[2024-02-19 19:09:04.408] [DEBUG] - share lock `/home/admin/.obd/lock/mirror_and_r
epo`, count 1
[2024-02-19 19:09:04.409] [DEBUG] - Get local repository obproxy-ce-4.2.1.0-0aed4b
782120e4248b749f67be3d2cc82cdcb70d
[2024-02-19 19:09:04.409] [DEBUG] - share lock `/home/admin/.obd/lock/mirror_and_r
epo`, count 2
[2024-02-19 19:09:04.410] [DEBUG] - Searching param plugin for components ...
[2024-02-19 19:09:04.410] [DEBUG] - Search param plugin for oceanbase-ce
[2024-02-19 19:09:04.410] [DEBUG] - mkdir /home/admin/.obd/plugins
[2024-02-19 19:09:04.411] [DEBUG] - Found for oceanbase-ce-param-4.2.2.0 for ocean
base-ce-4.2.2.0
[2024-02-19 19:09:04.411] [DEBUG] - Applying oceanbase-ce-param-4.2.2.0 for oceanb
ase-ce-4.2.2.0-100000192024011915.el7-aa3053da7370a6685a2ef457cd202d50e5ab75d3
[2024-02-19 19:09:04.762] [DEBUG] - Search param plugin for obproxy-ce
[2024-02-19 19:09:04.763] [DEBUG] - Found for obproxy-ce-param-3.1.0 for obproxy-c
e-4.2.1.0
[2024-02-19 19:09:04.763] [DEBUG] - Applying obproxy-ce-param-3.1.0 for obproxy-ce
-4.2.1.0-11.el7-0aed4b782120e4248b749f67be3d2cc82cdcb70d
[2024-02-19 19:09:04.846] [DEBUG] - Searching connect plugin for components ...
[2024-02-19 19:09:04.846] [DEBUG] - Searching connect plugin for oceanbase-ce-4.2.
2.0-100000192024011915.el7-aa3053da7370a6685a2ef457cd202d50e5ab75d3
[2024-02-19 19:09:04.846] [DEBUG] - Found for oceanbase-ce-py_script_connect-4.2.2
.0 for oceanbase-ce-4.2.2.0
[2024-02-19 19:09:04.846] [DEBUG] - Searching connect plugin for obproxy-ce-4.2.1.
0-11.el7-0aed4b782120e4248b749f67be3d2cc82cdcb70d
[2024-02-19 19:09:04.846] [DEBUG] - Found for obproxy-ce-py_script_connect-3.1.0 f
```

```
or obproxy-ce-4.2.1.0
[2024-02-19 19:09:04.846] [DEBUG] - Searching create_tenant plugin for components
...
[2024-02-19 19:09:04.847] [DEBUG] - Searching create_tenant plugin for oceanbase-c
e-4.2.2.0-100000192024011915.el7-aa3053da7370a6685a2ef457cd202d50e5ab75d3
[2024-02-19 19:09:04.847] [DEBUG] - Found for oceanbase-ce-py_script_create_tenant
-4.2.0.0 for oceanbase-ce-4.2.2.0
[2024-02-19 19:09:04.847] [DEBUG] - Searching create_tenant plugin for obproxy-ce-
4.2.1.0-11.el7-0aed4b782120e4248b749f67be3d2cc82cdcb70d
[2024-02-19 19:09:04.847] [DEBUG] - No such create_tenant plugin for obproxy-ce-4.
2.1.0
[2024-02-19 19:09:04.960] [INFO] Open ssh connection
[2024-02-19 19:09:04.960] [DEBUG] - host: 10.10.10.1, port: 22, user: admin, passw
ord: None
[2024-02-19 19:09:05.019] [DEBUG] - host: 10.10.10.2, port: 22, user: admin, passw
ord: None
[2024-02-19 19:09:05.076] [DEBUG] - host: 10.10.10.3, port: 22, user: admin, passw
ord: None
[2024-02-19 19:09:05.132] [DEBUG] - host: 10.10.10.2, port: 22, user: admin, passw
ord: None
[2024-02-19 19:09:05.221] [DEBUG] - Call oceanbase-ce-py_script_connect-4.2.2.0 fo
r oceanbase-ce-4.2.2.0-100000192024011915.el7-aa3053da7370a6685a2ef457cd202d50e5ab
75d3
[2024-02-19 19:09:05.221] [DEBUG] - import connect
[2024-02-19 19:09:05.281] [DEBUG] - add connect ref count to 1
[2024-02-19 19:09:05.282] [DEBUG] -- connect obshell (10.10.10.1:2886)
[2024-02-19 19:09:05.282] [DEBUG] -- connect obshell (10.10.10.2:2886)
[2024-02-19 19:09:05.282] [DEBUG] -- connect obshell (10.10.10.3:2886)
[2024-02-19 19:09:05.282] [INFO] Connect to observer
[2024-02-19 19:09:05.283] [DEBUG] -- connect 10.10.10.1 -P12881 -uroot -pRoot2023@
@Root2023
[2024-02-19 19:09:05.284] [DEBUG] -- execute sql: select 1. args: None
[2024-02-19 19:09:05.414] [DEBUG] - sub connect ref count to 0
[2024-02-19 19:09:05.414] [DEBUG] - export connect
[2024-02-19 19:09:05.414] [DEBUG] - Call oceanbase-ce-py_script_create_tenant-4.2.
0.0 for oceanbase-ce-4.2.2.0-100000192024011915.el7-aa3053da7370a6685a2ef457cd202d
50e5ab75d3
[2024-02-19 19:09:05.414] [DEBUG] - import create_tenant
[2024-02-19 19:09:05.417] [DEBUG] - add create_tenant ref count to 1
[2024-02-19 19:09:05.418] [DEBUG] -- execute sql: select * from oceanbase.DBA_OB_U
NIT_CONFIGS where name like "test2_unit%" order by unit_config_id desc limit 1. ar
gs: None
[2024-02-19 19:09:05.420] [DEBUG] -- execute sql: select * from oceanbase.DBA_OB_T
ENANTS where TENANT_NAME = %s. args: ('test2',)
[2024-02-19 19:09:05.439] [INFO] Create tenant test2
[2024-02-19 19:09:05.440] [DEBUG] -- execute sql: select zone, count(*) num from o
ceanbase.__all_server where status = 'active' group by zone. args: None
[2024-02-19 19:09:05.441] [DEBUG] -- execute sql: select count(*) num from oceanba
se.__all_server where status = 'active' and start_service_time > 0. args: None
[2024-02-19 19:09:05.442] [DEBUG] -- execute sql: SELECT * FROM oceanbase.GV$OB_SE
RVERS where zone in ('zone1','zone2','zone3'). args: None
```

```
[2024-02-19 19:09:05.447] [DEBUG] -- execute sql: create resource unit test2_unit
max_cpu 2.0, memory_size 2147483648, min_cpu 2.0, max_iops 10000, log_disk_size 32
21225472. args: None
[2024-02-19 19:09:05.453] [DEBUG] -- execute sql: create resource pool test2_pool
unit='test2_unit', unit_num=1, zone_list=('zone1','zone2','zone3'). args: None
[2024-02-19 19:09:05.472] [DEBUG] -- execute sql: create tenant test2 replica_num=
3,zone_list=('zone1','zone2','zone3'),primary_zone='RANDOM',resource_pool_list=('t
est2_pool'), charset = 'utf8'set ob_tcp_invited_nodes="%", ob_compatibility_mode
= 'mysql'. args: None
[2024-02-19 19:09:38.645] [DEBUG] - sub create_tenant ref count to 0
[2024-02-19 19:09:38.645] [DEBUG] - export create_tenant
[2024-02-19 19:09:38.645] [INFO] Trace ID: 4af55084-cf17-11ee-b825-00163e04608d
[2024-02-19 19:09:38.645] [INFO] If you want to view detailed obd logs, please run
: obd display-trace 4af55084-cf17-11ee-b825-00163e04608d
[2024-02-19 19:09:38.645] [DEBUG] - share lock /home/admin/.obd/lock/mirror_and_re
po release, count 1
[2024-02-19 19:09:38.645] [DEBUG] - share lock /home/admin/.obd/lock/mirror_and_re
po release, count 0
[2024-02-19 19:09:38.645] [DEBUG] - unlock /home/admin/.obd/lock/mirror_and_repo
[2024-02-19 19:09:38.645] [DEBUG] - exclusive lock /home/admin/.obd/lock/deploy_ob
test release, count 0
[2024-02-19 19:09:38.645] [DEBUG] - unlock /home/admin/.obd/lock/deploy_obtest
[2024-02-19 19:09:38.646] [DEBUG] - share lock /home/admin/.obd/lock/global releas
e, count 0
[2024-02-19 19:09:38.646] [DEBUG] - unlock /home/admin/.obd/lock/global
```

成功创建租户后，可使用 obd 命令查看创建的租户。

```
obd cluster tenant show obtest -t test2
```

输出如下：

```
Get local repositories and plugins ok
Get deployment connections ok
Connect to observer 10.10.10.1:2881 ok
Select tenant ok
+---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
+
|                                                                          tenant
base info
|
+-------------+-------------+--------------------+--------------+---------+-------
--+-------------+----------+----------+--------------+-------------+-------------
+
| tenant_name | tenant_type | compatibility_mode | primary_zone | max_cpu | min_cp
u | memory_size | max_iops | min_iops | log_disk_size | iops_weight | tenant_role
|
```

```
+-------------+-------------+-------------------+--------------+---------+-------
--+-------------+---------+---------+--------------+------------+-------------
+
| test2       | USER        | MYSQL             | RANDOM       | 2.0     | 2.
0     | 2.0G        | 10000    | 10000    | 3.0G          | 0           | PRIMAR
Y     |
+-------------+-------------+-------------------+--------------+---------+-------
--+-------------+---------+---------+--------------+------------+-------------
+
Trace ID: 00f5e506-cf18-11ee-a74f-00163e04608d
If you want to view detailed obd logs, please run: obd display-trace 00f5e506-cf18
-11ee-a74f-00163e04608d
```

通过 SQL 命令创建租户

说明

以下操作需使用 root 用户登录到 OceanBase 数据库的 sys 租户执行。

通过 SQL 命令创建租户分为以下三步:

1. 创建资源单元规格：该步骤为可选步骤，如果有合适的规格可以跳过此步骤，直接进行复用。
2. 创建资源池：可以每个 `zone` 一个资源池，使用独立的资源单元规格，也可以所有 `zone` 使用同一个资源单元规格，都在一个资源池下。
3. 创建租户：创建租户时需关联到第二步中创建的资源池。

（可选）步骤一：创建资源单元规格

资源单元规格是描述资源池的配置信息，用来描述资源池中每个资源单元可用的 CPU、内存、日志盘存储空间和 IOPS 的规格，创建资源单元规格后不会实际分配资源。

1. 创建资源单元规格

```
create resource unit u0 min_cpu=2,max_cpu=2,memory_size='2g', log_disk_size='6g',m
ax_iops=10000;
```

2. 查看资源单元规格详情

```
select * from oceanbase.DBA_OB_UNIT_CONFIGS;
```

输出如下，DBA_OB_UNIT_CONFIGS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_UNIT_CONFIGS。

```
+----------------+-----------------+----------------------------+-----------------
-----------+---------+---------+-------------+---------------+--------------------
-+---------------------+-------------+
| UNIT_CONFIG_ID | NAME            | CREATE_TIME                | MODIFY_TIM
E                | MAX_CPU | MIN_CPU | MEMORY_SIZE | LOG_DISK_SIZE | MAX_IOP
S            | MIN_IOPS            | IOPS_WEIGHT |
+----------------+-----------------+----------------------------+-----------------
-----------+---------+---------+-------------+---------------+--------------------
-+---------------------+-------------+
|              1 | sys_unit_config | 2024-02-19 15:33:47.524052 | 2024-02-19 15:33
:47.524052 |       2 |       2 |  1073741824 |    3221225472 | 922337203685477580
7 | 9223372036854775807 |           2 |
|           1001 | u0              | 2024-02-19 15:50:23.848604 | 2024-02-19 15:50
:23.848604 |       2 |       2 |  2147483648 |    6442450944 |               1000
0 |               10000 |           0 |
+----------------+-----------------+----------------------------+-----------------
-----------+---------+---------+-------------+---------------+--------------------
-+---------------------+-------------+
```

步骤二：创建资源池

创建资源池时会实际创建 Unit，按照定义的规格分配资源，若对应节点预留资源不够将会创建失败，通过 `GV$OB_SERVERS` 视图可以查看所有节点资源分配信息。若创建成功可以通过 `DBA_OB_RESOURCE_POOLS` 视图和 `DBA_OB_UNITS` 视图查看资源池及其对应 Unit。

注意

资源池不能复用，成功创建租户后指定资源池将会分配给租户。

1. 创建资源池

```
create resource pool p1 unit='u0', zone_list=('zone1','zone2','zone3'),unit_num=1;
```

2. 使用 root 用户登录 OceanBase 数据库的 sys 租户，查看资源池详情

```
select t4.tenant_id,t4.tenant_name,
    t1.name resource_pool_name, t1.unit_count,
    t2.`name` unit_config_name,
    t2.max_cpu, t2.min_cpu,
    ROUND(t2.memory_size/1024/1024/1024,2) mem_size_gb,
    ROUND(t2.log_disk_size/1024/1024/1024,2) log_disk_size_gb, t2.max_iops,
    t2.min_iops, t3.unit_id, t3.zone, concat(t3.svr_ip,':',t3.`svr_port`) observer
    from oceanbase.dba_ob_resource_pools t1
    join oceanbase.dba_ob_unit_configs t2 on (t1.unit_config_id=t2.unit_config_id)
    join oceanbase.dba_ob_units t3 on (t1.`resource_pool_id` = t3.`resource_pool_i
d`)
    left join oceanbase.dba_ob_tenants t4 on (t1.tenant_id=t4.tenant_id)
    order by t4.tenant_name,t3.zone;
```

输出如下：

```
+-----------+-------------+--------------------+------------+------------------+--
-------+---------+-------------+------------------+---------------------+---------
-----------+---------+-------+-----------------+
| tenant_id | tenant_name | resource_pool_name | unit_count | unit_config_name | m
ax_cpu | min_cpu | mem_size_gb | log_disk_size_gb | max_iops            | min_iop
s           | unit_id | zone  | observer        |
+-----------+-------------+--------------------+------------+------------------+--
-------+---------+-------------+------------------+---------------------+---------
-----------+---------+-------+-----------------+
|      NULL | NULL        | p1                 |          1 | u0
|       2 |       2 |        2.00 |             6.00 |               10000
|                10000 |    1007 | zone1 | 10.10.10.1:2882 |
|      NULL | NULL        | p1                 |          1 | u0
|       2 |       2 |        2.00 |             6.00 |               10000
|                10000 |    1008 | zone2 | 10.10.10.2:2882 |
|      NULL | NULL        | p1                 |          1 | u0
|       2 |       2 |        2.00 |             6.00 |               10000
|                10000 |    1009 | zone3 | 10.10.10.3:2882 |
|         1 | sys         | sys_pool           |          1 | sys_unit_config
|       2 |       2 |        1.00 |             3.00 | 9223372036854775807 | 92233
72036854775807 |       1 | zone1 | 10.10.10.1:2882 |
|         1 | sys         | sys_pool           |          1 | sys_unit_config
|       2 |       2 |        1.00 |             3.00 | 9223372036854775807 | 92233
72036854775807 |       2 | zone2 | 10.10.10.2:2882 |
|         1 | sys         | sys_pool           |          1 | sys_unit_config
|       2 |       2 |        1.00 |             3.00 | 9223372036854775807 | 92233
72036854775807 |       3 | zone3 | 10.10.10.3:2882 |
+-----------+-------------+--------------------+------------+------------------+--
-------+---------+-------------+------------------+---------------------+---------
-----------+---------+-------+-----------------+
```

说明

创建的 p1 资源池还没有关联到具体租户，所以查询展示 tenant_id 和 tenant_name 为 NU LL，无法被业务使用。

DBA_OB_RESOURCE_POOLS 视图相关字段说明见下表，详细字段说明可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_RESOURCE_POOLS。

字段名称	类型	是否可以为 NULL	描述
NAME	varchar(128)	NO	Resource Pool 名称
UNIT_COUNT	bigint(20)	NO	Unit 数量

DBA_OB_UNIT_CONFIGS 视图相关字段说明见下表，详细字段说明可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_UNIT_CONFIGS。

字段名称	类型	是否可以为 NULL	描述
UNIT_CONFIG_ID	bigint(20)	NO	Unit 规格 ID
NAME	varchar(128)	NO	Unit 规格名称
MAX_CPU	double	NO	CPU 规格上限
MIN_CPU	double	NO	CPU 规格下限
MEMORY_SIZE	bigint(20)	NO	内存规格，单位：字节
LOG_DISK_SIZE	bigint(20)	NO	日志盘规格，单位：字节
MAX_IOPS	bigint(20)	NO	磁盘 IOPS 规格上限
MIN_IOPS	bigint(20)	NO	磁盘 IOPS 规格下限

DBA_OB_UNITS 视图相关字段说明见下表，详细字段说明可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_UNITS。

字段名称	类型	是否可以为 NULL	描述
UNIT_ID	bigint(20)	NO	Unit ID
RESOURCE_POOL_ID	bigint(20)	NO	Unit 所属的资源池 ID
ZONE	varchar(128)	NO	Zone 名称

SVR_IP	varchar(46)	NO	Unit 所属的 OBServer 节点 IP
SVR_PORT	bigint(20)	NO	Unit 所属的 OBServer 端口号

`DBA_OB_TENANTS` 视图相关字段说明见下表，详细字段说明可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_TENANTS。

字段	类型	是否可以为 NULL	描述
TENANT_ID	bigint(20)	NO	租户 ID： • 1：系统租户 ID • 其他值：用户租户或者 Meta 租户 ID
TENANT_NAME	varchar(128)	NO	租户名

3. 1：系统租户 ID
4. 其他值：用户租户或者 Meta 租户 ID
5. 查看集群剩余可用资源

```
SELECT ZONE,SVR_IP,SVR_PORT,
  CPU_CAPACITY,CPU_ASSIGNED_MAX,CPU_CAPACITY-CPU_ASSIGNED_MAX as CPU_FREE,
  ROUND(MEMORY_LIMIT/1024/1024/1024,2) as MEMORY_TOTAL_GB,
  ROUND((MEMORY_LIMIT-MEM_CAPACITY)/1024/1024/1024,2) as SYSTEM_MEMORY_GB,
  ROUND(MEM_ASSIGNED/1024/1024/1024,2) as MEM_ASSIGNED_GB,
  ROUND((MEM_CAPACITY-MEM_ASSIGNED)/1024/1024/1024,2) as MEMORY_FREE_GB,
  ROUND(LOG_DISK_CAPACITY/1024/1024/1024,2) as LOG_DISK_CAPACITY_GB,
  ROUND(LOG_DISK_ASSIGNED/1024/1024/1024,2) as LOG_DISK_ASSIGNED_GB,
  ROUND((LOG_DISK_CAPACITY-LOG_DISK_ASSIGNED)/1024/1024/1024,2) as LOG_DISK_FREE_G
B,
  ROUND((DATA_DISK_CAPACITY/1024/1024/1024),2) as DATA_DISK_GB,
  ROUND((DATA_DISK_IN_USE/1024/1024/1024),2) as DATA_DISK_USED_GB,
  ROUND((DATA_DISK_CAPACITY-DATA_DISK_IN_USE)/1024/1024/1024,2) as DATA_DISK_FREE_
GB
  FROM oceanbase.GV$OB_SERVERS;
```

输出如下，资源池成功创建后，集群的可用资源就减少了。GV$OB_SERVERS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_SERVERS。

```
+--------+---------------+----------+--------------+-------------------+----------+-
----------------+-------------------+-----------------+----------------+-----------
-----------+----------------------+-------------------+---------------+-------------
------+--------------------+
| ZONE   | SVR_IP          | SVR_PORT | CPU_CAPACITY | CPU_ASSIGNED_MAX | CPU_FREE |
```

```
MEMORY_TOTAL_GB | SYSTEM_MEMORY_GB | MEM_ASSIGNED_GB | MEMORY_FREE_GB | LOG_DISK_C
APACITY_GB | LOG_DISK_ASSIGNED_GB | LOG_DISK_FREE_GB | DATA_DISK_GB | DATA_DISK_US
ED_GB | DATA_DISK_FREE_GB |
+-------+--------------+---------+--------------+------------------+----------+-
----------------+------------------+------------------+----------------+----------
-----------+---------------------+-------------------+--------------+------------
------+-------------------+
| zone2 | 10.10.10.2    |     2882 |          16 |                2 |       14
|          12.00 |            5.00 |            1.00 |               6.00
|               50.00 |                  3.00 |                47.00 |         50.00
|             0.05 |               49.95 |
| zone1 | 10.10.10.1    |     2882 |          16 |                2 |       14
|          12.00 |            5.00 |            1.00 |               6.00
|               50.00 |                  3.00 |                47.00 |         50.00
|             0.05 |               49.95 |
| zone3 | 10.10.10.3    |     2882 |          16 |                2 |       14
|          12.00 |            5.00 |            1.00 |               6.00
|               50.00 |                  3.00 |                47.00 |         50.00
|             0.05 |               49.95 |
+-------+--------------+---------+--------------+------------------+----------+-
----------------+------------------+------------------+----------------+----------
-----------+---------------------+-------------------+--------------+------------
------+-------------------+
```

步骤三：创建租户

创建租户时通过指定 `RESOURCE_POOL_LIST` 将资源池分配给租户，指定资源池时，可以每个 Zone 单独指定一个资源池，使用独立的资源规格；也可以所有 Zone 使用同一个资源池，从而所有 Zone 使用同一个资源规格。除了资源池列表，还有 Primary Zone、Locality、连接白名单等其他重要属性和系统变量，其中资源池列表为创建租户时的必填项。

```
CREATE TENANT IF NOT EXISTS test1 CHARSET='utf8mb4',
ZONE_LIST=('zone1','zone2','zone3'), PRIMARY_ZONE='RANDOM',
RESOURCE_POOL_LIST=('p1') SET ob_tcp_invited_nodes='%';
```

创建完成后可以通过 `DBA_OB_TENANTS` 视图查看所有租户：

```
SELECT * FROM oceanbase.DBA_OB_TENANTS \G
```

输出如下，DBA_OB_TENANTS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_TENANTS。

```
*************************** 1. row ***************************
```

```
                  TENANT_ID: 1
                TENANT_NAME: sys
                TENANT_TYPE: SYS
                CREATE_TIME: 2024-02-19 15:33:47.549569
                MODIFY_TIME: 2024-02-19 15:33:47.549569
               PRIMARY_ZONE: RANDOM
                   LOCALITY: FULL{1}@zone1, FULL{1}@zone2, FULL{1}@zone3
          PREVIOUS_LOCALITY: NULL
         COMPATIBILITY_MODE: MYSQL
                     STATUS: NORMAL
              IN_RECYCLEBIN: NO
                     LOCKED: NO
                TENANT_ROLE: PRIMARY
          SWITCHOVER_STATUS: NORMAL
           SWITCHOVER_EPOCH: 0
                   SYNC_SCN: NULL
             REPLAYABLE_SCN: NULL
               READABLE_SCN: NULL
         RECOVERY_UNTIL_SCN: NULL
                   LOG_MODE: NOARCHIVELOG
ARBITRATION_SERVICE_STATUS: DISABLED
                   UNIT_NUM: 1
                 COMPATIBLE: 4.2.1.3
                  MAX_LS_ID: 1
*************************** 2. row ***************************
                  TENANT_ID: 1005
                TENANT_NAME: META$1006
                TENANT_TYPE: META
                CREATE_TIME: 2024-02-19 19:03:03.358119
                MODIFY_TIME: 2024-02-19 19:03:23.572876
               PRIMARY_ZONE: RANDOM
                   LOCALITY: FULL{1}@zone1, FULL{1}@zone2, FULL{1}@zone3
          PREVIOUS_LOCALITY: NULL
         COMPATIBILITY_MODE: MYSQL
                     STATUS: NORMAL
              IN_RECYCLEBIN: NO
                     LOCKED: NO
                TENANT_ROLE: PRIMARY
          SWITCHOVER_STATUS: NORMAL
           SWITCHOVER_EPOCH: 0
                   SYNC_SCN: NULL
             REPLAYABLE_SCN: NULL
               READABLE_SCN: NULL
         RECOVERY_UNTIL_SCN: NULL
                   LOG_MODE: NOARCHIVELOG
ARBITRATION_SERVICE_STATUS: DISABLED
                   UNIT_NUM: 1
                 COMPATIBLE: 4.2.1.3
                  MAX_LS_ID: 1
*************************** 3. row ***************************
```

```
                 TENANT_ID: 1006
               TENANT_NAME: test1
               TENANT_TYPE: USER
               CREATE_TIME: 2024-02-19 19:03:03.360234
               MODIFY_TIME: 2024-02-19 19:03:23.624946
              PRIMARY_ZONE: RANDOM
                  LOCALITY: FULL{1}@zone1, FULL{1}@zone2, FULL{1}@zone3
         PREVIOUS_LOCALITY: NULL
        COMPATIBILITY_MODE: MYSQL
                    STATUS: NORMAL
             IN_RECYCLEBIN: NO
                    LOCKED: NO
               TENANT_ROLE: PRIMARY
         SWITCHOVER_STATUS: NORMAL
          SWITCHOVER_EPOCH: 0
                  SYNC_SCN: 1708340630218168002
            REPLAYABLE_SCN: 1708340630218168002
              READABLE_SCN: 1708340630218168001
        RECOVERY_UNTIL_SCN: 4611686018427387903
                  LOG_MODE: NOARCHIVELOG
ARBITRATION_SERVICE_STATUS: DISABLED
                  UNIT_NUM: 1
                COMPATIBLE: 4.2.1.3
                 MAX_LS_ID: 1003
3 rows in set (0.06 sec)
```

新建租户的管理员（`root`）密码默认为空，建议在租户创建完成后为 `root` 用户设置密码，

1. 生成随机字符串

```
[admin@test001 ~]$ strings /dev/urandom |tr -dc A-Za-z0-9 | head -c8; echo
```

输出如下：

```
b******t
```

2. 登录 OceanBase 数据库，将字符串设置为 root 用户密码

```
MySQL [oceanbase]> ALTER USER root IDENTIFIED BY 'b******t' ;
Query OK, 0 rows affected (0.118 sec)
```

`ALTER USER` 命令的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/ALTER USER。

通过 ob-operator 创建租户

说明

本节仅简单介绍如何使用 ob-operator 创建租户，详细的操作步骤可参见 创建租户。

前提条件

创建租户前，您需要确保：

- ob-operator 为 V2.1.0 或以上版本。
- OceanBase 集群部署完成且正常运行。

操作步骤

1. 创建配置文件

   配置文件内容可参见 GitHub 中 tenant.yaml 示例，各个字段的详细介绍可参见 创建租户示例。

2. 创建租户

   执行如下命令创建租户，该命令会在当前 Kubernetes 集群中创建一个 OBTenant 租户的资源。此处以配置文件名为 `tenant.yaml` 为例，您需根据实际名称修改配置文件名称。

   ```
   kubectl apply -f tenant.yaml
   ```

3. 确认租户是否创建成功

   执行如下命令，查看当前 Kubernetes 集群中是否有新创建的租户的 OBTenant 资源。

   ```
   kubectl describe obtenants.oceanbase.oceanbase.com -n oceanbase t1
   ```

   输出如下，该 OBTenant 资源的 Status.status 为 `running`，相关配置都会在 `Status` 中展示。

   ```
   Name:         t1
   Namespace:    oceanbase
   ```

```
Labels:       <none>
Annotations:  <none>
API Version:  oceanbase.oceanbase.com/v1alpha1
Kind:         OBTenant
Metadata:
  Creation Timestamp:  2023-11-13T07:28:31Z
  Finalizers:
    finalizers.oceanbase.com.deleteobtenant
  Generation:        2
  Resource Version:  940236
  UID:               34036a49-26bf-47cf-8201-444b3850aaa2
Spec:
  Charset:             utf8mb4
  Connect White List:  %
  Credentials:
    Root:         t1-ro
    Standby Ro:  t1-ro
  Force Delete:  true
  Obcluster:     obcluster
  Pools:
    Priority:  1
    Resource:
      Iops Weight:    2
      Log Disk Size:  12Gi
      Max CPU:        1
      Max Iops:       1024
      Memory Size:    5Gi
      Min CPU:        1
      Min Iops:       1024
    Type:
      Is Active:  true
      Name:       Full
      Replica:    1
    Zone:         zone1
    ...    # 省略部分输出
Status:
  Credentials:
    Root:         t1-ro
    Standby Ro:  t1-ro
  Resource Pool:
    Priority:  1
    Type:
      Is Active:  true
      Name:       FULL
      Replica:    1
    Unit Config:
      Iops Weight:    2
      Log Disk Size:  12884901888
      Max CPU:        1
      Max Iops:       1024
```

```
      Memory Size:    5368709120
      Min CPU:        1
      Min Iops:       1024
    Unit Num:         1
    Units:
      Migrate:
        Server IP:
        Server Port:  0
      Server IP:      10.10.10.1
      Server Port:    2882
      Status:         ACTIVE
      Unit Id:        1006
    Zone List:        zone1
    ...  # 省略部分输出
  Status:                running
  Tenant Record Info:
    Charset:              utf8mb4
    Connect White List:   %
    Locality:             FULL{1}@zone1, FULL{1}@zone2, FULL{1}@zone3
    Pool List:            pool_t1_zone1,pool_t1_zone2,pool_t1_zone3
    Primary Zone:         zone3;zone1,zone2
    Tenant ID:            1006
    Unit Num:             1
    Zone List:            zone1,zone2,zone3
  Tenant Role:            PRIMARY
Events:
  Type    Reason  Age                      From                   Message
  ----    ------  ----                     ----                   -------
  Normal          2m58s                    obtenant-controller    start creating
  Normal          115s                     obtenant-controller    create OBTenant succ
essfully
```

2.7 连接租户

OceanBase 数据库开源版的租户只兼容 MySQL 模式，连接协议兼容 MySQL 5.7。因此 MySQL 命令行客户端或者图形化工具理论上也能连接 OceanBase 数据库的 MySQL 兼容模式租户。此外，OceanBase 数据库也提供专属的命令行客户端工具 OBClient 和图形化客户端工具 ODC。

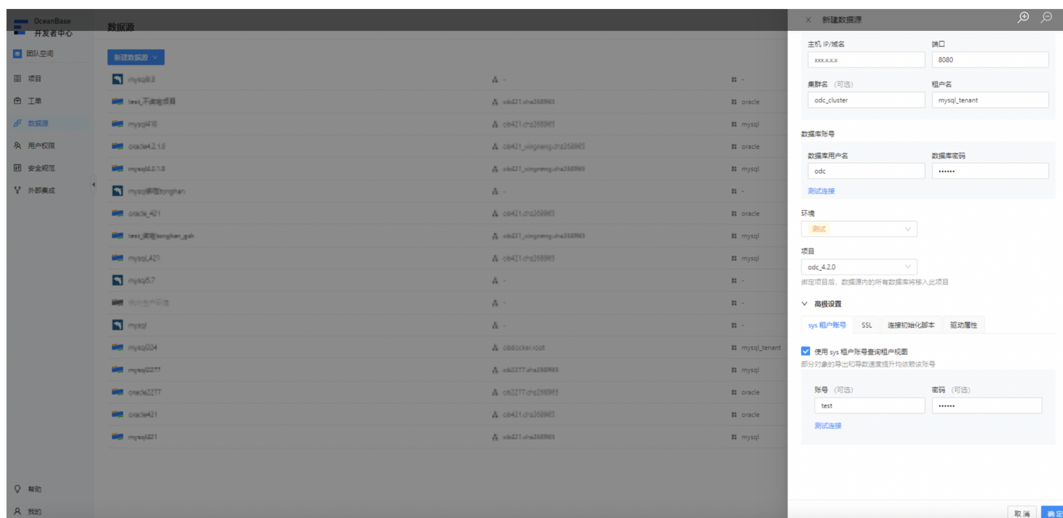
ODC 客户端连接

OceanBase 提供官方图形化客户端工具 ODC（OceanBase Developer Center）。该工具的详细信息请参见官网 OceanBase 开发者中心 文档。

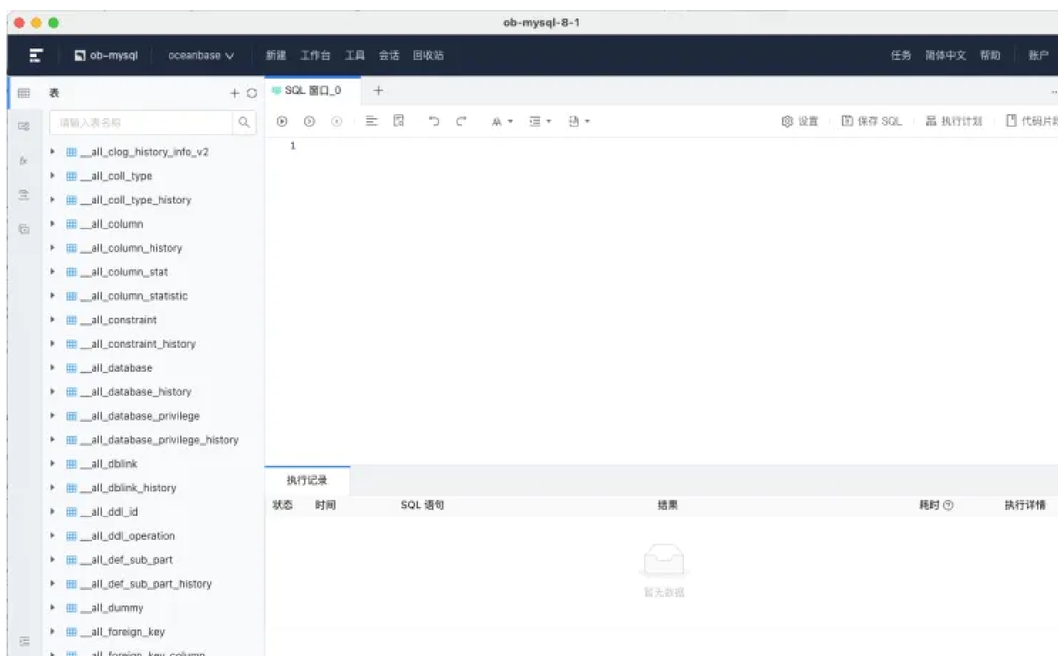
您可访问官网 软件下载中心，搜索 **OceanBase 开发者中心**，根据实际环境下载 ODC。

使用 ODC 连接 OceanBase 数据库的方法如下，详细的操作介绍可参见官网《OceanBase 开发者中心》文档 数据源管理/创建数据源。

1. 新建连接



2. 保存连接后，打开连接，即进入工作界面



MySQL 客户端连接

说明

本节仅做简单介绍，使用 MySQL 客户端连接 OceanBase 租户的详细介绍可参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 MySQL 客户端连接 OceanBase 租户。

OceanBase 数据库的 MySQL 兼容模式租户支持传统 MySQL 客户端连接。与传统 MySQL 相比，除用户名格式外，其他连接方式不变。连接示例如下：

```
mysql -h10.10.10.1 -uroot@sys#obtest -P2883 -p -c -A oceanbase
```

输出如下：

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 900573
Server version: 5.6.25 OceanBase_CE 4.2.1.7 (r107000162024060611-69b64b84b656a4cfa
126dab60b4e66dc1bc156ca) (Built Jun 6 2024 11:51:48)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

MySQL [oceanbase]>
```

参数说明：

- `-h`：提供 OceanBase 数据库连接 IP，直连时为 OBServer 节点地址，通过 ODP 连接时为 ODP 地址。
- `-u`：提供租户的连接账户。通过 OBProxy 连接时的常用格式有两种：`用户名@租户名#集群名` 或者 `集群名:租户名:用户名`；通过 OBServer 节点直连时，无需指定集群名。MySQL 租户的管理员用户名默认是 `root`，若仅指定用户名，则默认登录 sys 租户。

> **说明**
>
> 当使用 OBProxy 连接 OceanBase 集群时，可以通过以下方式获取集群名：
>
> 1. 通过直连的方式连接 OceanBase 数据库。
> 2. 使用 `SHOW PARAMETERS LIKE 'cluster';` 命令获取集群的名称，该命令返回结果中 VALUE 的值对应 OceanBase 集群的名称。

- 通过直连的方式连接 OceanBase 数据库。
- 使用 `SHOW PARAMETERS LIKE 'cluster';` 命令获取集群的名称，该命令返回结果中 `VALUE` 的值对应 OceanBase 集群的名称。
- `-P`：提供 OceanBase 数据库连接端口，直连时为 `mysql_port` 配置项的值，通过 ODP 连接时为 `listen_port` 配置项的值。
- `-p`：提供账户密码，出于安全考虑，建议在提示符下输入，此时密码文本不可见。
- `-c`：表示在 MySQL 运行环境中不要忽略注释。

> **说明**
>
> Hint 是特殊的注释，不受 -c 影响。

- `-A`：表示在 MySQL 连接数据库时不自动获取统计信息。
- `oceanbase`：访问的数据库名，可以改为业务数据库。

OBClient 客户端连接

说明

本节仅做简单介绍，使用 OBClient 客户端连接 OceanBase 租户的详细介绍可参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 OBClient 客户端连接 OceanBase 租户。

OceanBase 提供了专用的命令行客户端工具 OBClient，可访问官网 OceanBase 软件下载中心，搜索 **OceanBase 命令行客户端** 进行下载。OBClient 的使用方法和 MySQL 客户端一样。示例如下：

```
obclient -h10.10.10.1 -uroot@test1#obtest -P2883 -p -c -A oceanbase
```

输出如下：

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MySQL connection id is 900551
Server version: 5.6.25 OceanBase_CE 4.2.1.7 (r107000162024060611-69b64b84b656a4cfa
126dab60b4e66dc1bc156ca) (Built Jun 6 2024 11:51:48)

Copyright (c) 2000, 2018, OceanBase and/or its affiliates. All rights reserved.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

obclient [oceanbase]>
```

参数说明：

- `-h`：提供 OceanBase 数据库连接 IP，直连时为 OBServer 节点地址，通过 ODP 连接时为 ODP 地址。
- `-u`：提供租户的连接账户。通过 OBProxy 连接时的常用格式有两种：`用户名@租户名#集群名` 或者 `集群名:租户名:用户名`；通过 OBServer 节点直连时，无需指定集群名。MySQL 租户的管理员用户名默认是 `root`，若仅指定用户名，则默认登录 sys 租户。

> **说明**
>
> 当使用 OBProxy 连接 OceanBase 集群时，可以通过以下方式获取集群名：
>
> 1. 通过直连的方式连接 OceanBase 数据库。

> 2. 使用 SHOW PARAMETERS LIKE 'cluster'; 命令获取集群的名称，该命令返回结果中 VALUE 的值对应 OceanBase 集群的名称。

- 通过直连的方式连接 OceanBase 数据库。
- 使用 `SHOW PARAMETERS LIKE 'cluster';` 命令获取集群的名称，该命令返回结果中 `VALUE` 的值对应 OceanBase 集群的名称。
- `-P`：提供 OceanBase 数据库连接端口，直连时为 `mysql_port` 配置项的值，通过 ODP 连接时为 `listen_port` 配置项的值。
- `-p`：提供账户密码，出于安全考虑，建议在提示符下输入，此时密码文本不可见。
- `-c`：表示在 OBClient 运行环境中不要忽略注释。

> **说明**
>
> Hint 是特殊的注释，不受 -c 影响。

- `-A`：表示在 OBClient 连接数据库时不自动获取统计信息。
- `oceanbase`：访问的数据库名，可以改为业务数据库。

OceanBase 连接驱动（JDBC）

OceanBase 数据库目前支持的应用主要有 Java、C/C++、Python、GO，详细介绍可参见官网《OceanBase 数据库》对应文档。

Java 语言

- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Java/SpringBoot 连接 OceanBase 数据库
- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Java/MyBatis 连接 OceanBase 数据库
- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Java/SpringBatch 连接 OceanBase 数据库

- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Java/Spring JDBC 连接 OceanBase 数据库示例程序
- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Java/SpringJPA 连接 OceanBase 数据库
- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Java/Hibernate 连接 OceanBase 数据库

C/C++ 语言

快速上手/基于 MySQL 模式创建示例应用程序/创建 C 示例应用程序

Python 语言

- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Python/mysqlclient 连接 OceanBase 数据库示例程序
- 应用开发/基于 MySQL 模式进行应用开发/示例程序/Python/PyMySQL 连接 OceanBase 数据库示例程序

GO 语言

- 应用开发/基于 MySQL 模式进行应用开发/示例程序/GO/Go-SQL-Driver/MySQL 连接 OceanBase 数据库示例程序
- 应用开发/基于 MySQL 模式进行应用开发/示例程序/GO/GORM 连接 OceanBase 数据库示例程序

2.8 设置参数和变量

参数

本节介绍参数的分类，以及如何查看和修改参数。

参数的分类

OceanBase 数据库的参数分为集群级参数和租户级参数。

- 集群级参数指适用于整个 OceanBase 数据库集群的配置选项，它们具有全局性质，用于配置整个集群的基本信息、性能参数、安全选项等等。
- 租户级参数指适用于租户级别的配置选项，它们是针对单个租户或多个租户的配置选项。用于对单个租户或多个租户进行特定的配置和优化。这些参数通常包括存储引擎参数、SQL 执行策略、访问控制等方面的配置选项。租户级参数通常可以在租户创建和管理时进行配置，可以随时根据需要进行修改。

查看参数的方式

目前参数支持通过 OCP、obd 或 SQL 命令行查看，您可根据实际情况选择合适的查看方法。

通过 OCP 查看参数

此处以查看集群参数为例简单介绍查看方法，详细介绍可参见官网《OceanBase 云平台》文档 集群管理/管理集群参数/查看参数列表。

1. 登录 OCP
2. 在左侧导航栏选择 **集群**，进入集群页面
3. 在 **集群列表** 区域，选择待操作的集群并单击其集群名
4. 在显示的页面的左侧导航栏上，单击 **参数管理**

通过 obd 查看参数

您可执行如下命令查看参数，该命令仅显示配置文件中配置的参数，无法查看到 OceanBase 数据库的所有参数。obd 管理的参数详情可参见 GitHub 仓库中 parameter.yaml 页面。

```
obd cluster edit-config obtest
```

此处以集群名为 `obtest` 为例，您可执行 `obd cluster list` 命令查看 obd 管理的集群名，并根据实际情况替换 `obtest`。

通过 SQL 命令行查看参数

通过 SQL 命令行有两种方法可查看参数。命令的输出中，需关注如下字段。

- `SCOPE`：表示参数是集群级别还是租户级别。对应的值为 `CLUSTER` 表示该参数为集群级别；对应的值为 `TENANT` 表示该参数为租户级别。
- `EDIT_LEVEL`：表示参数是否允许修改，以及若允许修改，修改后是动态生效还是需要重启才能生效。
  - `READONLY`：只读参数不允许修改。
  - `STATIC_EFFECTIVE`：可以修改，单需要重启才能生效。
  - `DYNAMIC_EFFECTIVE`：可以修改，且动态生效。

> **注意**
>
> 由于历史原因，部分参数写的是 DYNAMIC_EFFECTIVE，其实是不允许修改的，修改时需要格外留意。

- `READONLY`：只读参数不允许修改。
- `STATIC_EFFECTIVE`：可以修改，单需要重启才能生效。
- `DYNAMIC_EFFECTIVE`：可以修改，且动态生效。
- 执行 `show parameters` 命令查看，示例如下：

```
show parameters like '%memory%';
show parameters like 'enable_rebalance' tenant='test3';
show parameters where name like 'cpu_count';
```

```
show parameters where name in ('memory_limit','cpu_count');
```

说明

通过 `show parameters` 命令查看参数时，部分参数所有租户都可以查询，但是还有部分参数仅 sys 租户可以查询。

- 查询 `GV$OB_PARAMETERS` 表，示例如下：

```
SELECT * FROM oceanbase.GV$OB_PARAMETERS WHERE NAME LIKE '%memstore%';
```

输出如下：

```
+---------------+----------+-------+---------+-----------+-----------------------
---+-----------+-------+--------------------------------------------------------
--------------------------------------------------------------------------------
-----------------------------------------------------------------+---------+-----
--------------+
| SVR_IP        | SVR_PORT | ZONE  | SCOPE   | TENANT_ID | NAM
E                      | DATA_TYPE | VALUE | INF
O

                                       | SECTION | EDIT_LEVEL        |
+---------------+----------+-------+---------+-----------+-----------------------
---+-----------+-------+--------------------------------------------------------
--------------------------------------------------------------------------------
-----------------------------------------------------------------+---------+-----
--------------+
| 10.10.10.1    |     2882 | zone1 | CLUSTER |      NULL | memstore_limit_percenta
ge | NULL      | 50    | used in calculating the value of MEMSTORE_LIMIT parameter
: memstore_limit_percentage = memstore_limit / memory_size,memory_size, where MEMO
RY_SIZE is determined when the tenant is created. Range: (0, 100) | TENANT  | DYNA
MIC_EFFECTIVE |
+---------------+----------+-------+---------+-----------+-----------------------
---+-----------+-------+--------------------------------------------------------
--------------------------------------------------------------------------------
-----------------------------------------------------------------+---------+-----
--------------+
```

该方法的过滤条件更灵活，支持所有租户查询。输出字段的详细说明可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_PARAMETERS。

修改参数

目前参数支持通过 OCP、obd 或 SQL 命令行修改，您可根据实际情况选择合适的修改方法。

通过 OCP 修改参数

此处以修改集群参数为例简单介绍修改方法，详细介绍可参见官网《OceanBase 云平台》文档 集群管理/管理集群参数/修改参数。

1. 登录 OCP
2. 在左侧导航栏选择 **集群**，进入集群页面
3. 在 **集群列表** 区域，选择待操作的集群并单击其集群名
4. 在显示的页面的左侧导航栏上，单击 **参数管理**
5. （可选）在 **参数列表** 页面上方的搜索框中，输入参数名相关信息进行模糊搜索
6. 找到待修改的参数，在对应的 **操作** 列中，单击 **修改值**
7. 在弹出的对话框中，修改参数的值、生效范围及生效对象，单击 **确定**

通过 obd 修改参数

obd 支持修改的参数可参见 GitHub 中 parameter.yaml 页面。

> **注意**
>
> 有些参数不支持修改（need_redeploy 为 true），修改后需执行 `obd cluster redeploy` 命令才可生效，`obd cluster redeploy` 命令会销毁集群并重新部署，请谨慎操作。

1. 执行如下命令进入配置文件

   ```
   obd cluster edit-config obtest
   ```

   此处以集群名为 `obtest` 为例，您可执行 `obd cluster list` 命令查看 obd 管理的集群名，并根据实际情况替换 `obtest`。

2. 修改对应参数并保存退出后，执行输出中对应的重启命令

   保存并退出配置文件后的输出如下，表示需要执行 `obd cluster reload obtest` 命令使修改

生效。

```
Search param plugin and load ok
Search param plugin and load ok
Parameter check ok
Save deploy "obtest" configuration
Use `obd cluster reload obtest` to make changes take effect.
```

注意

若修改某个参数后，obd 提示需要执行 `obd cluster redeploy`，一定要联系官方进行确认，以防执行 `obd cluster redeploy` 后原有的集群被重新部署导致数据丢失。

通过 SQL 命令修改参数

通过 SQL 命令修改参数的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统管理/配置管理/设置参数。

```
alter system set 参数='参数值' [tenant='xxx'];
```

注意

若当前环境是通过 obd 部署的，通过 obd 修改参数后，通过 SQL 命令查看对应参数，值为修改后的内容。但若先通过 SQL 命令修改配置文件中存在的参数，再通过 obd 查看参数，此时对应参数仍然是修改前的。

变量

通过系统变量的设置可以控制数据库系统的各种行为，如缓存大小、并发连接数、CPU 使用率、内存使用率等等。同时，系统变量也可以用于配置数据库系统的各种功能。

系统变量分类

OceanBase 数据库的系统变量分为全局（Global）变量和 Session 变量。

- Global 变量表示 Global 级别的修改，数据库同一租户内的不同用户共享全局变量。全局变

量的修改不会随会话的退出而失效。此外，全局变量修改后，对当前已打开的 Session 不生效，需要重新建立 Session 才能生效。

- Session 变量表示 Session 级别的修改。当客户端连接到数据库后，数据库会复制全局变量来自动生成 Session 变量。Session 变量的修改仅对当前 Session 生效。

目前没有相关的表或者视图可以查看一个变量是否是只读的，需要从对应版本的源码包中获取。可执行如下命令从 `src/share/system_variable/ob_system_variable_init.json` 文件中查看哪些变量为只读变量。

```
cat ob_system_variable_init.json  | jq ".[] | {name,flags}" | grep -C 2 "READONLY"
| grep -v "ORACLE_ONLY"
```

查看变量

在 OCP 中，租户的参数就是指变量，若您选择通过 OCP 查看变量，具体操作可参见官网《OceanBase 云平台》文档 租户管理/管理租户参数/查看参数列表。

若您选择通过 SQL 命令查看变量，有如下三种查看方法：

- 查询 CDB_OB_SYS_VARIABLES 视图

  该方法仅可使用 sys 租户查询，支持查询所有租户的变量，CDB_OB_SYS_VARIABLES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_SYS_VARIABLES。

  ```
  select * from oceanbase.CDB_OB_SYS_VARIABLES where name='ob_query_timeout' and ten
  ant_id=x;
  ```

- 执行 `SHOW VARIABLES` 命令查询

  `SHOW VARIABLES` 命令对执行的租户没有要求，但是仅支持查看当前租户的变量。

  ```
  SHOW VARIABLES LIKE 'ob_query_timeout';
  ```

- 查询 DBA_OB_SYS_VARIABLES 视图

  该方法对执行的租户没有要求，但是仅支持查看当前租户的变量。

  DBA_OB_SYS_VARIABLES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指

南/系统视图/MySQL 租户系统视图/字典视图/oceanbase.DBA_OB_SYS_VARIABLES。

说明

该视图自 OceanBase 数据库 V4.2.2 开始引入。

```
select * from oceanbase.DBA_OB_SYS_VARIABLES where name='ob_query_timeout';
```

修改变量

在 OCP 中，租户的参数就是指变量，若您选择通过 OCP 修改变量，具体操作可参见官网《OceanBase 云平台》文档 租户管理/管理租户参数/修改参数。

若您选择通过 SQL 命令修改变量，有如下两种情况，详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统管理/配置管理/设置变量。

- 设置全局级别变量，设置后当前 session 不生效，新 session 生效。

```
set global ob_query_timeout=10000000;
```

- 设置会话级别变量，设置后当前 session 生效，其他 session 不生效。

```
set session ob_query_timeout=10000000;
```

参数和变量对比

对比项	参数	系统变量
生效范围	分为集群、Zone、机器和租户。	分为租户的 Global 或 Session 级别。
生效	• 动态生效：`edit_level` 为 `dynamic_effective` • 重启生效：`edit_level` 为 `static_effective`	• 设置 Session 级别的变量仅对当前 Session 有效，对其他

方式		Session 无效。 • 设置 Global 级别的变量对当前 Session 无效，需要重新登录建立新的 Session 才会生效。

修改方式	• 支持通过 SQL 语句修改，示例：
	`ALTER SYSTEM SET schema_history_expire_time='1h';`
	• 支持通过启动参数修改，示例：
	`cd /home/admin/oceanbase && ./bin/observer -o "schema_history_expire_time='1h'";`
		仅支持通过 SQL 语句修改，示例：
		• `SET ob_query_timeout = 20000000;`
		• `SET GLOBAL ob_query_timeout = 20000000;`

查询方式	可以使用 `SHOW PARAMETERS` 语句查询。示例：`SHOW PARAMETERS LIKE 'schema_history_expire_time';`	可以使用 `SHOW [GLOBAL] VARIABLES` 语句查询。示例： • `SHOW VARIABLES LIKE 'ob_query_timeout';` • `SHOW GLOBAL VARIABLES LIKE 'ob_query_timeout';`
持久化	持久化到内部表与配置文件，可以在 `/home/admin/oceanbase/etc/observer.config.bin` 与 `/home/admin/oceanbase/etc/observer.config.bin.history` 文件中查询该参数。	仅 Global 级别的变量会持久化，Session 级别的变量不会进行持久化。
生命周期	长，从进程启动到退出。	短，需要租户的 Schema 创建成功以后才生效。

第三章 测试 OceanBase 数据库

本章介绍影响 OceanBase 数据库性能的因素，以及如何对 OceanBase 数据库进行各种测试。

本章目录

3.1 OceanBase 数据库测试概述 118

3.2 OceanBase 数据库性能的影响因素 120

3.3 进行 Sysbench 测试 128

3.4 进行 TPC-C 测试 137

3.5 进行 TPC-H 测试 154

3.6 使用 JMeter 运行业务场景测试 171

3.7 其他常见测试点 182

3.1 OceanBase 数据库测试概述

在这个数据驱动的时代，数据库行业的快速发展引领了全新的技术创新和解决方案。为了确保数据库选型能够适应不断扩大的数据规模和日益增长的处理需求，企业和技术人员会通过 PoC 测试考察不同数据库系统的能力。在这一过程中，通常大家会重点关注几个关键点，如性能、并发处理能力、存储成本、高可用，以确保所选技术能够满足特定的业务要求和性能标准。

性能测试

数据库性能测试是业务选型和 PoC 测试中的重点关注指标，涉及多种评估标准和测试工具。OceanBase 数据库作为一款实时 HTAP 数据库，同时支持在线实时交易和实时分析两种场景。为了评估数据库 OLTP（联机事务处理）的性能，我们常用标准化测试工具（如 Sysbench 和 TPC-C 等）来衡量数据库在处理高并发事务、维持数据完整性及快速响应查询请求方面的能力。另一方面，为了评估数据库 OLAP（联机分析处理）的性能，我们通常会通过 TPC-H 这样的工具来模拟多维数据集的复杂查询和分析操作，检验系统对于大数据量的处理和响应能力。

除了上述提到的业界公认的测试标准模型，如果想评估数据库在实际业务场景下的综合性能，也可以使用 JMeter 来模拟用户实际业务场景与数据库的交互，使测试结果更全面，更贴近真实的应用场景。

并行导入测试

在 OLAP 场景下，大量数据的并行导入，即数据批处理能力也是 PoC 测试中不可或缺的一环。数据批量写入操作的速度和稳定性，对于需要处理大量数据迁移或同步任务的用户来说至关重要。OceanBase 数据库的并行执行框架能够将 DML 语句通过并发的方式进行执行（Parallel DML），对于多节点的数据库，能够实现多机并发写入，并且保证大事务的一致性。另外结合异步转储机制，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。

数据压缩测试

在完成业务数据导入后，数据压缩也是用户经常关注的测试点。随着存储成本的持续上升和对高

效数据访问的不断追求，低成本存储及高效处理海量数据信息已成为提升数据库产品竞争力的重要因素，数据库压缩技术也成为优化数据库性能和降低总成本的关键手段。进行数据压缩测试旨在量化数据压缩前后对存储空间的节省，评估压缩数据对查询和事务处理性能的具体影响，以及验证数据的完整性和恢复过程的可靠性。

高可用测试

数据库系统在应用架构中承担了数据存储和查询的重要角色，企业数据的高可用对保障业务连续性至关重要，高可用也是数据库测试中重点考察的因素。OceanBase 数据库基于 Paxos 协议实现了多副本容灾方案，对用户提供少数派故障时 RPO = 0 (Recovery Point Objective，数据恢复点目标) ，RTO < 8s (Recovery Time Objective，恢复时间目标) 的高可用能力。多副本容灾面向集群内少数派节点异常的场景，具备极好的故障恢复速度及数据零丢失能力。多副本容灾方案的实现，使事务日志持久化并在多个副本之间同步日志数据，基于 Paxos 协议保证日志数据在多数派副本持久化成功，同时通过成员变更提供容灾能力。

3.2 OceanBase 数据库性能的影响因素

数据库的性能往往受到多个方面因素的影响，从数据库软件层面来看，代码的优化、算法的选择，以及系统架构的设计，都直接影响着数据库的性能。OceanBase 数据库作为一款高性能的 HTAP 数据库，在版本的迭代过程中，一直持续致力于提升数据库性能并降低资源消耗。

除此之外，操作系统参数的调整同样不容忽视。操作系统是软件与硬件之间的桥梁，通过合理配置其参数，可以使系统更高效地管理资源并执行任务。这涉及对内存管理、进程调度、输入与输出操作等核心组件进行精细化控制。

在完成部署时软硬件资源环境的合理配置后，我们还可以通过运行时的数据资源分配、数据库参数调优、适当的运维操作，来更加充分利用已有的系统资源，在不影响系统稳定的前提下，发挥出数据库最大的性能。

操作系统参数

在操作系统参数上，我们可以从网络、内存、I/O 等角度进行配置，优化系统可用资源，提升性能。

类型	配置项	描述	建议
网络参数	net.core.somaxconn	Socket 监听队列的最大长度，频繁建立连接需要调大该值	默认为 128，建议配置为 2048
	net.core.netdev_max_backlog	协议栈处理的缓冲队列长度，设置过小有可能造成丢包	建议配置为 10000
	net.core.rmem_default	接收缓冲区队列的默认长度	建议配置为 16777216
	net.core.wmem_default	发送缓冲区队列的默认长度	建议配置为 16777216
	net.core.rmem_max	接收缓冲区队列的最大长度	建议配置为 16777216
	net.core.wmem_max	发送缓冲区队列的最大长度	建议配置为 16777216
	net.ipv4.ip_loc	本地 TCP/UDP 的端口范围，本地使用该	建议的端口范围为 [3500, 6553

	al_port_range	范围内的端口与远端发起连接	5]
	net.ipv4.tcp_rmem	Socket 接收缓冲区的大小，需配置三个值，从左到右分别为最小值、默认值、最大值	建议最小值、默认值、最大值分别配置为 4096、87380、16777216
	net.ipv4.tcp_wmem	Socket 发送缓冲区的大小，需配置三个值，从左到右分别为最小值、默认值、最大值	建议最小值、默认值、最大值分别配置为 4096、65536、16777216
	net.ipv4.tcp_max_syn_backlog	处于 SYN_RECVD 状态的连接数	建议配置为 16384
	net.ipv4.tcp_fin_timeout	Socket 主动断开之后，FIN-WAIT-2 状态的持续时间	建议配置为 15
	net.ipv4.tcp_tw_reuse	控制是否允许重用处于 TIME WAIT 状态的 Socket	建议配置为 1（允许重用）
	net.ipv4.tcp_slow_start_after_idle	控制是否禁止 TCP 连接从 Idle 状态恢复时执行慢启动，禁止后会降低某些情况的网络延迟	建议配置为 0（禁止）
虚拟内存配置	vm.swappiness	设置优先使用物理内存	建议配置为 0
	vm.max_map_count	进程可以拥有的虚拟内存区域数量	建议配置为 655360
AIO 配置	fs.aio-max-nr	异步 I/O 的请求数目	建议配置为 1048576

资源分配

磁盘划分

OBServer 节点运行时依赖系统日志、事务日志和数据文件，如果将这些日志文件设置到同一块盘上，由于磁盘硬件资源的竞争，可能存在如下的风险：

- 事务日志（CLOG）空间利用率超过 log_disk_utilization_threshold 值（默认 80%）时会开始回收，超过 log_disk_utilization_limit_threshold 值（默认 95%）时节点会停止写入。

- 在进行转储、合并等操作需要额外的 I/O 资源时，和业务读写的 I/O 可能会互相影响，造成业务抖动。
- 影响 obcdc 同步，导致同步数据慢。

应对策略：

- 在资源充足的情况下，推荐分别挂载三块磁盘并使用 SSD 存储，如果机器上没有三块磁盘，或者使用的是 RAID 磁盘阵列，需要对磁盘或者磁盘阵列的逻辑卷进行分区。
- 开启日志限流功能，控制系统日志所能占用的磁盘 I/O 带宽上限，示例如下。

```
alter system set syslog_io_bandwidth_limit='10M';
```

- 开启回收系统日志功能并设置最大文件数量，示例如下。

```
alter system set enable_syslog_recycle = true; alter system set max_syslog_file_co
unt = 1000;
```

Primary Zone

在 OceanBase 数据库中，Leader（主副本）承担了强一致性场景下事务中的读写请求，因此每个分区 Leader 的分布决定了流量在每个节点上的分布。

Leader 分布通过 Primary Zone 来控制，Primary Zone 描述了 Leader 副本的偏好位置，而 Leader 副本承载了业务的强一致读写流量，即 Primary Zone 决定了 OceanBase 数据库的流量分布。

对于多节点的 OceanBase 数据库集群，如果设置 Primary Zone 为 `RANDOM`，就可以将不同分区的 Leader 分散在不同 Zone 的节点上，使得整个集群机器利用率达到最大化。

分区表

在 OceanBase 数据库中，分区是指根据一定的规则，把一个表分解成多个更小的、更容易管理的部分。分区表使用水平拆分策略将一个大表切割为多个独立的分区。

使用分区的好处如下：

- 提高可用性

分区不可用并不意味着对象不可用，查询优化器自动从查询计划中删除未引用的分区。因此，当分区不可用时，查询不受影响。

- 更轻松地管理对象

  分区对象具有可以集体或单独管理的片段，DDL 语句可以操作分区而不是整个表或索引。因此，可以对重建索引或表等资源密集型任务进行分解。例如，可以一次只移动一个分区，如果出现问题，只需要重做分区移动，而不是表移动。此外，对分区进行 TRUNCATE 操作可以避免大量数据被 DELETE。

- 减少 OLTP 系统中共享资源的争用

  在 TP 场景中，分区可以减少共享资源的争用，例如，DML 分布在许多分区而不是一个表上。

- 增强数据仓库中的查询性能

  在 AP 场景中，分区可以加快即时查询的处理速度，分区键有天然的过滤功能。例如，查询一个季度的销售数据，当销售数据按照销售时间进行分区时，仅需查询一个分区或者几个分区，而不是整个表。

- 提供更好的负载均衡效果

  OceanBase 数据库的存储单位和负载均衡单位都是分区，不同的分区可以存储在不同的节点。因此，一个分区表可以将不同的分区分布在不同的节点，这样可以将一个表的数据比较均匀地分布在整个集群。

表组 Table Group

表组（Table Group）是一个逻辑概念，表示一组表的集合。默认情况下，不同表之间的数据是随机分布的，通过定义表组，可以控制一组表在物理存储上的邻近关系，减少分布式场景的开销，提升性能。

在 OceanBase 数据库 3.x 版本中，表组为定义了分区的表组，加入表组的表要求与表组的分区方式完全一致，限制了表加入表组，有较强的约束性；从 OceanBase 数据库 4.2.0 版本开始，表组没有了分区概念，只需要定义 `SHARDING` 属性，就可以很灵活地将不同分区方式的表加入表组。

对于 `SHARDING` 属性的表组，根据 `SHARDING` 属性值的不同，主要可以分为以下两大类。

- `SHARDING` 为 `NONE` 的表组：此类表组内的所有表的所有分区均聚集在同一台机器上，并且不限制表组内表的分区类型。
- `SHARDING` 不为 `NONE` 的表组：此类表组内每一张表的数据会打散分布在多台机器上。为了保证所有表的数据分布相同，还要求表组内所有表的分区定义一致，包括分区类型、分区个数、分区值等。系统会调度具有相同分区属性的分区聚集（对齐）在同一台机器上，从而实现 Partition Wise Join。

关于表组的详细介绍可参见官网《OceanBase 数据库》文档中 参考指南/数据库对象管理/MySQL 模式/创建和管理表组。

局部索引和全局索引

局部索引

分区表的局部索引和非分区表的索引类似，索引的数据结构还是和主表的数据结构保持一对一的关系，但由于主表已经做了分区，主表的每一个分区都有自己单独的索引数据结构。对每一个索引数据结构来说，里面的键（Key）只映射到自己分区中的主表数据，不会映射到其它分区中的主表，因此这种索引被称为局部索引。

全局索引

和分区表的局部索引相比，分区表的全局索引不再和主表的分区保持一对一的关系，而是将所有主表分区的数据合成一个整体来看，索引中的一个键可能会映射到多个主表分区中的数据（当索引键有重复值时）。更进一步，全局索引可以定义自己独立的数据分布模式，既可以选择非分区模式也可以选择分区模式；在分区模式中，分区的方式既可以和主表相同也可以和主表不同。由于全局索引的分区模式和主表的分区模式完全没有关系，看上去全局索引更像是另一张独立的表，因此也会将全局索引叫做索引表。

推荐使用全局索引的场景包括：

- 业务上除了主键外，还有其他列的组合需要满足全局唯一性的强需求，这个业务需求仅能通过全局性的唯一索引来实现。
- 业务的查询无法得到分区键的条件谓词，且业务表没有高并发的同时写入，为避免进行全分

区的扫描，可以根据查询条件构建全局索引，必要时可以将全局索引按照新的分区键来分区。

需要注意的是，全局索引虽然为全局唯一、数据重新分区带来了可能，解决了一些业务需要根据不同维度进行查询的强需求，但是为此付出的代价是每一笔数据的写入都有可能变成跨机的分布式事务，在高并发的写入场景下它将影响系统的写入性能。当业务的查询可以拥有分区键的条件谓词时，OceanBase 数据库依旧推荐构建局部索引，通过数据库优化器的分区裁剪功能，排除不符合条件的分区。这样的做法可以同时兼顾查询和写入的性能，让系统的总体性能表现更优。

数据库参数调优

为了提升用户体验和数据库的易用性，让每一个开发者在使用数据库时都能获得较好的性能，OceanBase 数据库在 4.x 版本做了大量的性能优化工作。让用户可以在基于基础参数调优的场景下，也能获得较好的数据库性能体验。这里我们仅对一些基本的参数进行调优建议，在此基础上，如果想进一步地提升数据库的性能，也可以针对运行环境以及业务场景，再进一步做个性化的参数调优。

OLTP 场景

OceanBase 数据库参数，执行 `obclient -h<host_ip> -P<host_port> -uroot@sys -A -p` 命令进入系统租户，执行如下命令：

```
# 关闭 SQL 审计功能
ALTER system SET enable_sql_audit=false;
# 关闭性能事件的信息收集功能
ALTER system SET enable_perf_event=false;
# 设置系统日志级别为 ERROR，减少日志
ALTER system SET syslog_level='ERROR';
# 关闭记录追踪日志
alter system set enable_record_trace_log=false;
```

OBProxy 参数，执行 `obclient -h<host_ip> -P<host_port> -uroot@sys -A -p` 命令进入系统租户，执行如下命令：

说明

需使用 OBProxy 节点 IP 和端口登录 OceanBase 数据库的系统租户，才可修改 OBProxy 配置项。

```
# 提高 OBProxy 运行时内存上限
ALTER proxyconfig SET proxy_mem_limited='4G';
# 关闭 OBProxy 的压缩协议功能
ALTER proxyconfig set enable_compression_protocol=false;
```

OLAP 场景

OceanBase 数据库参数，连接用户租户执行：

```
# 设置 SQL 工作区内存占整个租户内存百分比
SET GLOBAL ob_sql_work_area_percentage = 80;
# 设置 SQL 最大执行时间
SET GLOBAL ob_query_timeout = 36000000000;
# 设置事务超时时间
SET GLOBAL ob_trx_timeout = 36000000000;
# 设置最大网络包的大小
SET GLOBAL max_allowed_packet = 67108864;
# 租户在每个节点上可申请的并行执行线程数量
SET GLOBAL parallel_servers_target = 624;
```

合并与统计信息收集

合并

合并操作是将动静态数据做归并，把当前大版本的 SSTable 和 MemTable 与前一个大版本的全量静态数据进行合并，产生新的全量数据。合并期间 OceanBase 数据库会对数据进行两层压缩，第一层是数据库内部基于语义的编码压缩，第二层是基于用户指定压缩算法的通用压缩，使用 lz4 等压缩算法对编码后的数据再做一次“瘦身”。压缩不仅仅节省了存储空间，同时也会极大地提升查询性能。而更重要的是，对于 OceanBase 数据库这样 LSM-Tree 架构的存储系统，压缩对数据写入性能是几乎无影响的。

统计信息收集

在数据库中，优化器针对每一个输入的 SQL 查询都会尝试生成最优的执行计划，而生成最优的

执行计划往往需要实时有效的统计信息和准确的行数估计。统计信息实际上指的是优化器统计信息（optimizer statistics），它是一个描述数据库中表和列信息的数据集合，是代价模型选取最优执行计划非常关键的部分。优化器代价模型（optimizer cost model）依赖于查询中涉及到的表、列、谓词等对象的统计信息来选取计划、优化计划的选择。准确有效的统计信息能够帮助优化器选择到最优的执行计划。

在 OceanBase 数据库优化器中，统计信息以普通数据的形式存储在内部表中，并且会在本地维护一个统计信息缓存，以提高优化器对统计信息的访问速度。在 OceanBase 数据库 V4.0 之前，统计信息收集是在每日合并过程中完成，但是由于每日合并是增量合并，会影响统计信息的准确度，同时每日合并没法收集直方图信息，无法解决数据存在倾斜的场景。因此，OceanBase 数据库 V4.0 使用全新的统计信息，同时将统计信息收集和每日合并解耦，每日合并过程不再收集统计信息，执行计划也不再会受到每日合并的影响。

3.3 进行 Sysbench 测试

Sysbench 是一个基于 LuaJIT 构建的可编写脚本的多线程基准测试工具，可以执行 CPU、内存、线程、I/O 和数据库等方面的性能测试，常用于评估测试各种不同系统参数下的数据库负载情况，不需要修改源码，通过自定义 Lua 脚本就可以实现不同数据库业务类型的测试。

本文基于 x86 架构的 CentOS Linux 7.9 镜像环境，介绍两种对 OceanBase 数据库运行 Sysbench 测试的方式：

- 通过 obd test 命令一键进行 Sysbench 测试。
- 基于官方 Sysbench 工具手动 step by step 进行测试。

使用 obdiag 在测试前对集群进行巡检

OceanBase 数据库是原生分布式数据库系统，故障根因分析通常比较复杂，因为需综合考量众多因素，包括但不限于机器环境、配置参数、运行负载等。专家在排查问题的时候需要获取大量的信息来分析故障，如何高效地采集和分析故障场景下分散在各个节点的信息，便是 OceanBase 敏捷诊断工具（OceanBase Diagnostic Tool，简称 obdiag）需要解决的问题。在开始进行 Sysbench 测试前，我们可以使用 obdiag 工具给 OceanBase 数据库做个“体检”，详细步骤和说明可参见官网博客 试用 obdiag 在 sysbench 压测前进行巡检。

环境准备

- JDK：建议使用 1.8u131 及以上版本
- Sysbench：建议使用 1.0 及以上版本
- make：可执行 `sudo yum install make` 安装
- automake：可执行 `sudo yum install automake` 安装
- autoconf：可执行 `sudo yum install autoconf` 安装
- libtool：可执行 `sudo yum install libtool` 安装
- gcc：可执行 `sudo yum install gcc` 安装

- mariadb-devel：可执行 `sudo yum install mariadb-devel mariadb` 安装
- OBClient：详细介绍可参见 GitHub 仓库

注意

当使用的 OBClient 版本大于等于 2.2.0 时，会默认开启 OceanBase 2.0 协议以及全链路追踪的能力，在 Sysbench 测试中会影响性能，建议通过设置环境变量手动关闭（`export ENABLE_PROTOCOL_OB20=0`）。

测试方案

本次测试需要用到 5 台机器，Sysbench 和 obd 安装在一台机器上，OBProxy 单独部署在一台机器上，OceanBase 数据库使用 3 台机器组成一个 1:1:1 的集群。

注意

- 建议单独部署 OBProxy，以避免 OBProxy 和 OceanBase 数据库的资源竞争。
- 建议磁盘 iops 设置在 10000 以上，系统日志、事务日志、数据文件配置为三块盘。
- 使用 obd 部署集群时，建议不要使用 `obd cluster autodeploy` 命令，该命令为了保证稳定性，不会最大化资源利用率。建议根据实际环境，对 obd 配置文件进行定制化调整，最大化资源利用率。

测试环境（阿里云 ECS）

服务类型	ECS 类型	实例数	机器核心数	内存
OBServer	ecs.g7.8xlarge	3	32C	内存 128GB，每台机器系统盘 300GB，再挂载两块 400GB 云盘作为 Clog 盘和 Data 盘，性能级别为 PL1
Sysbench	ecs.c7.4xlarge	1	16C	32GB
OBProxy	ecs.c7.16xlarge	1	64C	128GB

软件版本

服务类型	软件版本
OceanBase 数据库	OceanBase_CE 4.2.1.0
OBProxy	OBProxy_CE 4.2.1.0
Sysbench	1.0.20

租户规格

部署成功后需创建进行 Sysbench 测试的租户及用户（sys 租户是管理集群的内置系统租户，请勿直接使用 sys 租户进行测试），设置租户的 `primary_zone` 为 `RANDOM`，`RANDOM` 表示新建表分区的 Leader 随机到任一 OBServer 节点。

```
CREATE RESOURCE UNIT sysbench_unit max_cpu 26, memory_size '100g';
CREATE RESOURCE POOL sysbench_pool unit = 'sysbench_unit', unit_num = 1, zone_list
=('zone1','zone2','zone3');
CREATE TENANT sysbench_tenant resource_pool_list=('sysbench_pool'),  zone_list('zo
ne1', 'zone2', 'zone3'), primary_zone=RANDOM, locality='F@zone1,F@zone2,F@zone3' s
et variables ob_compatibility_mode='mysql', ob_tcp_invited_nodes='%';
```

obd 一键测试

说明

使用 obd 进行一键测试时，测试集群需是被 obd 管理的集群。通过 obd 部署的集群默认被 obd 管理，通过其他方法部署的集群需执行 obd 接管操作，详细操作方法可参见官网《OceanBase 安装部署工具》文档中 使用指南/命令行/使用 obd 接管集群 一文。

1. 安装 ob-sysbench

   该工具封装了原生的 Sysbench，提升易用性，安装命令如下。

   ```
   sudo yum install -y yum-utils
   sudo yum-config-manager --add-repo https://mirrors.aliyun.com/oceanbase/OceanBase.
   repo
   sudo yum install ob-sysbench
   ```

2. 编写测试脚本

此处以脚本名为 `ob_sysbench.sh` 为例，您可自定义脚本名。脚本中 `deploy_name` 为部署集群的名称，`tenant_name` 为进行测试的租户名称（即 租户规格 中的 `sysbench_tenant`），您需根据实际部署集群名称和租户名称进行修改。

```
#!/bin/bash
export ENABLE_PROTOCOL_OB20=0

echo "run oltp_point_select test"
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_point_se
lect.lua --table-size=1000000 --threads=32 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_point_se
lect.lua --table-size=1000000 --threads=64 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_point_se
lect.lua --table-size=1000000 --threads=128 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_point_se
lect.lua --table-size=1000000 --threads=256 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_point_se
lect.lua --table-size=1000000 --threads=512 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_point_se
lect.lua --table-size=1000000 --threads=1024 --rand-type=uniform

echo "run oltp_read_only test"
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_onl
y.lua --table-size=1000000 --threads=32 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_onl
y.lua --table-size=1000000 --threads=64 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_onl
y.lua --table-size=1000000 --threads=128 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_onl
y.lua --table-size=1000000 --threads=256 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_onl
y.lua --table-size=1000000 --threads=512 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_onl
y.lua --table-size=1000000 --threads=1024 --rand-type=uniform

echo "run oltp_write_only test"
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_write_on
ly.lua --table-size=1000000 --threads=32 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_write_on
ly.lua --table-size=1000000 --threads=64 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_write_on
ly.lua --table-size=1000000 --threads=128 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_write_on
ly.lua --table-size=1000000 --threads=256 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_write_on
ly.lua --table-size=1000000 --threads=512 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_write_on
```

```
ly.lua --table-size=1000000 --threads=1024 --rand-type=uniform

echo "run oltp_read_write test"
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_wri
te.lua --table-size=1000000 --threads=32 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_wri
te.lua --table-size=1000000 --threads=64 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_wri
te.lua --table-size=1000000 --threads=128 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_wri
te.lua --table-size=1000000 --threads=256 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_wri
te.lua --table-size=1000000 --threads=512 --rand-type=uniform
obd test sysbench <deploy_name> --tenant=<tenant_name> --script-name=oltp_read_wri
te.lua --table-size=1000000 --threads=1024 --rand-type=uniform
```

3. 执行测试

```
./ob_sysbench.sh
```

运行脚本后，系统会详细列出运行步骤和输出，数据量越大耗时越久。`obd test sysbench` 命令会自动完成所有操作，包含测试数据的生成、性能参数调优、数据导入和测试。命令的详细介绍可参见官网《OceanBase 安装部署工具》文档中 obd 命令/测试命令组 中 **obd test sysbench**。

手动进行 Sysbench 测试

步骤一：安装 Sysbench

按照以下步骤安装 Sysbench。

1. 下载 Sysbench

   访问 Sysbench 的 GitHub 仓库 下载 Sysbench。

2. 解压 Sysbench

```
[admin@test ~]$ unzip sysbench-1.0.20.zip
```

3. 编译 Sysbench

进入 Sysbench 解压后的目录，运行如下命令编译 Sysbench：

```
[admin@test ~]$ cd sysbench-1.0.20
[admin@test sysbench-1.0.20]$ ./autogen.sh
[admin@test sysbench-1.0.20]$ ./configure --prefix=/usr/sysbench/ --with-mysql-inc
ludes=/usr/include/mysql/ --with-mysql-libs=/usr/lib64/mysql/ --with-mysql
[admin@test sysbench-1.0.20]$ make
[admin@test sysbench-1.0.20]$ sudo make install
```

参数说明：

参数名	说明
--prefix	指定 Sysbench 的安装目录
--with-mysql-includes	指定 mysql 的 includes 目录
--with-mysql-libs	指定 mysql 的 lib 目录
--with-mysql	Sysbench 默认支持 MySQL

4. 验证是否安装成功

运行以下命令，验证 Sysbench 是否安装成功：

```
[admin@test sysbench-1.0.20]$ /usr/sysbench/bin/sysbench --help
```

如果返回以下信息，则 Sysbench 安装成功。

```
Usage:
   sysbench [options]... [testname] [command]
Commands implemented by most tests: prepare run cleanup help
```

步骤二：环境调优

在执行 Sysbench 测试前，需要对 OceanBase 数据库的参数进行简单的设置。

OBProxy 调优，执行 `obclient -h<host_ip> -P<host_port> -uroot@sys -A -p` 命令进入系统租户。

说明

需使用 OBProxy 节点 IP 和端口登录 OceanBase 数据库的系统租户，才可修改 OBProxy 配置项。

```
# 提高 OBProxy 运行时内存上限
ALTER proxyconfig SET proxy_mem_limited='4G';
# 关闭 OBProxy 的压缩协议功能
ALTER proxyconfig set enable_compression_protocol=false;
```

OceanBase 数据库调优，执行 `obclient -h<host_ip> -P<host_port> -uroot@sys -A -p` 命令进入系统租户。

```
# 关闭 SQL 审计功能
ALTER system SET enable_sql_audit=false;
# 关闭性能事件的信息收集功能
ALTER system SET enable_perf_event=false;
# 设置系统日志级别为 ERROR，减少日志
ALTER system SET syslog_level='ERROR';
# 关闭记录追踪日志
alter system set enable_record_trace_log=false;
```

步骤三：进行 Sysbench 测试

说明

本节以使用测试用例 oltp_read_write.lua，启动 32 个进程为例。在 sysbench-1.0.20/src/lua 目录下，自带有针对不同场景的测试用例，比如 oltp_point_select.lua、oltp_read_only.lua 和 oltp_write_only.lua 等。

1. 清除数据

```
[admin@test lua]$ /usr/sysbench/bin/sysbench oltp_read_write.lua --mysql-host=x.x.x.x --mysql-port=xxxx --mysql-db=test --mysql-user=$user@$tenant --mysql-password=****** --table_size=1000000 --tables=30 --threads=32 --report-interval=10 --rand-type=uniform --time=60 cleanup
```

2. 初始化测试数据

```
[admin@test lua]$ /usr/sysbench/bin/sysbench oltp_read_write.lua --mysql-host=x.x.x.x --mysql-port=xxxx --mysql-db=test --mysql-user=$user@$tenant --mysql-password=****** --table_size=1000000 --tables=30 --threads=32 --report-interval=10 --rand-type=uniform --time=60 prepare
```

3. 执行测试

```
[admin@test lua]$ /usr/sysbench/bin/sysbench oltp_read_write.lua --mysql-host=x.x.x.x --mysql-port=xxxx --mysql-db=test --mysql-user=$user@$tenant --mysql-password=****** --table_size=1000000 --tables=30 --threads=32 --report-interval=10 --time=60 --rand-type=uniform --db-ps-mode=disable run
```

测试结果

Point Select 性能

Threads	V4.2.1 QPS	V4.2.1 95% Latency (ms)
32	138746.60	0.26
64	252231.37	0.29
128	447755.19	0.34
256	730315.66	0.48
512	1009966.93	0.90
1024	1012734.80	2.66

Read Only 性能

Threads	V4.2.1 QPS	V4.2.1 95% Latency (ms)
32	121733.00	4.65
64	221563.16	5.09
128	392138.56	5.67
256	577951.13	8.58
512	763726.51	17.01
1024	740835.95	38.94

Write Only 性能

Threads	V4.2.1 QPS	V4.2.1 95% Latency (ms)
32	43984.28	7.17
64	82554.92	6.55
128	114874.89	10.09
256	181982.10	12.52

512	253635.91	19.29
1024	292482.33	36.89

Read Write 性能

Threads	V4.2.1 QPS	V4.2.1 95% Latency (ms)
32	72554.47	11.87
64	139369.33	11.65
128	247061.25	12.30
256	313660.08	23.95
512	497734.89	25.74
1024	547816.87	54.83

Sysbench 高性能部署和问题分析

在 Sysbench 压测过程中，鉴于 OceanBase 数据库的安装配置、用户租户的创建以及 Sysbench 工具的使用细节等可能因环境差异而有所不同，可能出现实际测试结果不符合预期的情况。我们也提供一些性能问题的案例作为参考，来帮助大家解决可能遇到的性能问题。详细案例可参考 OceanBase 官网《OceanBase 数据库》文档中 参考指南/性能测试/OceanBase Sysbench 高性能部署和问题分析 一文。

3.4 进行 TPC-C 测试

本文基于 x86 架构的 CentOS Linux 7.9 镜像环境，介绍两种对 OceanBase 数据库运行 TPC-C 测试的方式：

- 通过 obd test 命令一键进行 TPC-C 测试。
- 手动 step by step 进行测试。

TPC-C 简介

TPC-C 是 TPC（Transaction Processing Performance Council，事务处理性能委员会）推出的 OLTP（联机事务处理）基准测试。自从 1992 年首次发布以来，TPC-C 已经成为评估数据库事务处理能力的行业标准之一。

数据库模型

TPC-C 模型中的包含了九个数据表，其中仓库的数量 W 可以根据系统的实际情况进行调整。假设 WAREHOUSE 表（仓库）的数量为 W，则数据库初始数据信息如下：

- 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）。
- NEW-ORDER 表中有 W×10×900（每个地区有 900 个新订单），每个订单随机生成 5~15 条 ORDER-LINE（订单明细记录）。
- ITEM 表中有 10 万条商品记录，与仓库数量无关。

事务类型

TPC-C 模型中包含以下五类事务：

- 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，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%。

评估标准

TPC-C 模型使用 tpmC（Transactions per Minute）来衡量系统最大有效吞吐量，其中 Transactions 以 NewOrder 为准，即最终衡量标准为每分钟处理的新订单数。

使用 obdiag 在测试前对集群进行巡检

OceanBase 数据库是原生分布式数据库系统，故障根因分析通常比较复杂，因为需综合考量众多因素，包括但不限于机器环境、配置参数、运行负载等。专家在排查问题的时候需要获取大量的信息来分析故障，如何高效地采集和分析故障场景下分散在各个节点的信息，便是 OceanBase 敏捷诊断工具（OceanBase Diagnostic Tool，简称 obdiag）需要解决的问题。在开始进行 TPC-C 测试前，我们可以使用 obdiag 工具给 OceanBase 数据库做个“体检”，详细步骤和说明可参见官网 OceanBase 敏捷诊断工具（obdiag）文档。

环境准备

- JDK：建议使用 1.8u131 及以上版本
- JDBC：建议使用 mysql-connector-java-5.1.47 版本，其他版本可能存在语法兼容性问题
- Ant：建议使用 apache-ant-1.10 及以上版本
- Benchmark SQL：建议使用 Benchmark SQL 5.0
- OBClient：详细介绍可参见 GitHub 仓库

测试方案

本次测试需要用到 5 台机器，Benchmark SQL 和 obd 安装在一台机器上，OBProxy 单独部署在一台机器上，OceanBase 数据库使用 3 台机器组成一个 1:1:1 的集群。

注意

- 建议单独部署 OBProxy，以避免 OBProxy 和 OceanBase 数据库的资源竞争。
- 建议磁盘 iops 设置在 10000 以上，系统日志、事务日志、数据文件配置为三块盘。
- 使用 obd 部署集群时，建议不要使用 `obd cluster autodeploy` 命令，该命令为了保证稳定性，不会最大化资源利用率。建议根据实际环境，对 obd 配置文件进行定制化调整，最大化资源利用率。

测试环境（阿里云 ECS）

服务类型	ECS 类型	实例数	机器核心数	内存
OBServer	ecs.g7.8xlarge	3	32C	内存 128GB，每台机器系统盘 300GB，再挂载两块 400GB 云盘作为 Clog 盘和 Data 盘，性能级别为 PL1
Benchmark SQL	ecs.c7.4xlarge	1	16C	32GB
OBProxy	ecs.c7.16xlarge	1	64C	128GB

软件版本

服务类型	软件版本
OceanBase 数据库	OceanBase_CE 4.2.1.0
OBProxy	OBProxy_CE 4.2.1.0
Benchmark SQL	Benchmark SQL V5.0

租户规格

部署成功后需创建进行 TPC-C 测试的租户及用户（sys 租户是管理集群的内置系统租户，请勿直接使用 sys 租户进行测试），设置租户的 `primary_zone` 为 `RANDOM`，`RANDOM` 表示新建表分区的 Leader 随机到任一 OBServer 节点。

```
CREATE RESOURCE UNIT tpcc_unit max_cpu 26, memory_size '100g';
CREATE RESOURCE POOL tpcc_pool unit = 'tpcc_unit', unit_num = 1, zone_list=('zone1
','zone2','zone3');
CREATE TENANT tpcc_tenant resource_pool_list=('tpcc_pool'),  zone_list('zone1', 'z
one2', 'zone3'), primary_zone=RANDOM, locality='F@zone1,F@zone2,F@zone3' set varia
bles ob_compatibility_mode='mysql', ob_tcp_invited_nodes='%';
```

测试规格

```
warehouses=1000
loadWorkers=40
terminals=600
runMins=5
newOrderWeight=45
paymentWeight=43
orderStatusWeight=4
deliveryWeight=4
stockLevelWeight=4
```

obd 一键测试

说明

使用 obd 进行一键测试时，测试集群需是被 obd 管理的集群。通过 obd 部署的集群默认被 obd 管理，通过其他方法部署的集群需执行 obd 接管操作，详细操作方法可参见官网《OceanBase 安装部署工具》文档中 使用指南/命令行/使用 obd 接管集群 一文。

1. 安装 obtpcc

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://mirrors.aliyun.com/oceanbase/OceanBase.
repo
sudo yum install obtpcc java
```

```

```

2. 执行测试

```
obd test tpcc <deploy_name> --tenant=<tenant_name> --warehouses=1000 --load-worker
s=40 --terminals=800 --run-mins=5
```

命令中的 deploy_name 表示部署集群名，tenant_name 表示进行测试的租户名（即 租户规格 中的 `tpcc_tenant`），您需根据实际部署集群名和租户名进行修改。`obd test tpcc` 命令会自动完成所有操作，包含测试数据的生成、性能参数调优、数据导入和测试。命令的详细介绍可参见官网《OceanBase 安装部署工具》文档中 obd 命令/测试命令组 中 **obd test tpcc**。

手动进行 TPC-C 测试

步骤一：安装 Ant

按照以下步骤安装 Ant。

1. 下载 Ant

```
[admin@test ~]$ wget "http://archive.apache.org/dist/ant/binaries/apache-ant-1.10.
6-bin.zip"
```

2. 安装 Ant

```
[admin@test ~]$ unzip apache-ant-1.10.6-bin.zip
```

3. 设置环境变量

```
[admin@test ~]$ sudo vim /etc/profile
```

配置内容如下：

```
#ant
export ANT_HOME=xx/apache-ant-1.10.6
export PATH=xx/apache-ant-1.10.6/bin:$PATH
```

执行如下命令使配置生效

```
[admin@test ~]$ source /etc/profile
```

4. 验证是否安装成功

   运行以下命令，验证 Ant 是否安装成功。

```
[admin@test ~]$ ant -version
```

   如果返回以下信息，则 Ant 安装成功。

```
Apache Ant(TM) version 1.10.6 compiled on May 2 2019
```

步骤二：安装 Benchmark SQL

按照以下步骤安装 Benchmark SQL。

1. 下载 Benchmark SQL

   您可访问 SourceForge 下载 Benchmark SQL。

2. 解压 Benchmark SQL

```
[admin@test ~]$ unzip benchmarksql-5.0.zip
```

3. 编译 Benchmark SQL

```
[admin@test ~]$ cd benchmarksql-5.0
[admin@test benchmarksql-5.0]$ ant
```

步骤三：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"))   //增加 OceanBase 数据库相关内容
            dbType = DB_OCEANBASE;
        else
        {
            log.error("unknown database type '" + iDB + "'");
            return;
        }
```

2. 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;
```

3. 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");    //增加 "AS L" 别名
                break;
```

4. 重新编译修改后的源码

```
[admin@test benchmarksql-5.0]# ant
```

5. 在 `benchmarksql-5.0/run` 目录下，创建文件 prop.oceanbase。

   文件内容如下：

```
db=oceanbase
driver=com.mysql.jdbc.Driver
conn=jdbc:mysql://$host_ip:$port/$db_name?rewriteBatchedStatements=true&allowMulti
Queries=true&useLocalSessionState=true&useUnicode=true&characterEncoding=utf-8&soc
ketTimeout=30000000
//请填写完整 user 信息
user=$user@$tenant
password=*****
warehouses=1000
loadWorkers=40
terminals=800
database=$db_name
//To run specified transactions per terminal- runMins must equal zero
runTxnsPerTerminal=0
//To run for specified minutes- runTxnsPerTerminal must equal zero
runMins=5
//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
//The default percentages of 45, 43, 4, 4 & 4 match the TPC-C spec
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
```

   prop.oceanbase 中的参数说明：

   - JDBC 连接串：`conn=jdbc:mysql://x.x.x.x(ip):xx(port)/xxxx(dbname)?rewriteBatchedStatements=true&allowMultiQueries=true&useLocalSessionState=true&useUnicode=true&characterEncoding=utf-8&socketTimeout=3000000`

- rewriteBatchedStatements：
  - 该参数非常重要，会严重影响导数据效率，不可以忽略。
  - 如果导数据较慢，可以用对应租户登录上去通过 show full processlist 检查是否开启。
  - new order 事务中也用到了 batch update，因此导数和 benchmark 阶段都需要开启。
- 并发数量（terminals）：800，MySQL 租户配置下并发需要结合具体配置动态调整。取值须在 (0, 10*`warehouses`] 范围内。
- useLocalSessionState：控制是否使用 autocommit，read_only 和 transaction isolation 的内部值（JDBC 端的本地值），建议设置为 `true`，如果设置为 `false`，则需要发语句到远端请求，增加发送请求频次，影响性能。
- warehouses/loadWorkers：这两项用于设置压测数据量，可以适当调整。

6. JDBC 连接串：`conn=jdbc:mysql://x.x.x.x(ip):xx(port)/xxxx(dbname)?rewriteBatchedStatements=true&allowMultiQueries=true&useLocalSessionState=true&useUnicode=true&characterEncoding=utf-8&socketTimeout=3000000`
7. rewriteBatchedStatements：
   - 该参数非常重要，会严重影响导数据效率，不可以忽略。
   - 如果导数据较慢，可以用对应租户登录上去通过 show full processlist 检查是否开启。
   - new order 事务中也用到了 batch update，因此导数和 benchmark 阶段都需要开启。

8. 该参数非常重要，会严重影响导数据效率，不可以忽略。
9. 如果导数据较慢，可以用对应租户登录上去通过 show full processlist 检查是否开启。
10. new order 事务中也用到了 batch update，因此导数和 benchmark 阶段都需要开启。
11. 并发数量（terminals）：800，MySQL 租户配置下并发需要结合具体配置动态调整。取值须在 (0, 10*`warehouses`] 范围内。
12. useLocalSessionState：控制是否使用 autocommit，read_only 和 transaction isolation 的内部值（JDBC 端的本地值），建议设置为 `true`，如果设置为 `false`，则需要发语句到远端请求，增加发送请求频次，影响性能。
13. warehouses/loadWorkers：这两项用于设置压测数据量，可以适当调整。
14. 修改 `benchmarksql-5.0/run/funcs.sh` 文件，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)      #添加 OceanBase 数据库类型
    cp="../lib/oceanbase/*:../lib/*"
    ;;
    esac
    myCP=".:${cp}:../dist/*"
    export myCP
}

...省略

case "$(getProp db)" in
    firebird|oracle|postgres|oceanbase)  #添加 OceanBase 数据库类型
    ;;
    "") 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
```

15. 添加 mysql java connector 驱动，推荐 mysql-connector-java-5.1.47.jar

```
[admin@test benchmarksql-5.0]# mkdir lib/oceanbase/
[admin@test benchmarksql-5.0]# cp xx/mysql-connector-java-5.1.47.jar lib/oceanbase
/
```

需将 `xx/mysql-connector-java-5.1.47.jar` 替换为实际的 mysql-connector-java-5.1.47.jar 存放路径。

16. 修改 `benchmarksql-5.0/run/runDatabaseBuild.sh` 文件

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

17. 改造 BenchMarkSQL5 中的 SQL

备份并重写 `benchmarksql-5.0/run/sql.common/tableCreates.sql`。

```
CREATE TABLE bmsql_config (
  cfg_name    varchar(30) PRIMARY KEY,
  cfg_value   varchar(50)
);

CREATE TABLEGROUP IF NOT EXISTS tpcc_group binding true partition by hash partitio
ns 96;

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 96;
```

```
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 96;

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 96;


CREATE TABLE bmsql_history (
  hist_id  integer AUTO_INCREMENT,
  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 96;

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 96;

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 96;

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 96;

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' partition by hash(s_w_id) partitions 96;
```

备份并重写 `benchmarksql-5.0/run/sql.common/tableDrops.sql`。

```
DROP TABLE bmsql_config;
DROP TABLE bmsql_new_order;
DROP TABLE bmsql_order_line;
DROP TABLE bmsql_oorder;
DROP TABLE bmsql_history;
DROP TABLE bmsql_customer;
DROP TABLE bmsql_stock;
DROP TABLE bmsql_item;
DROP TABLE bmsql_district;
DROP TABLE bmsql_warehouse;
DROP TABLEGROUP tpcc_group;
```

备份并重写 `benchmarksql-5.0/run/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;
```

备份并重写 `benchmarksql-5.0/run/sql.common/indexDrops.sql`。

```
ALTER TABLE bmsql_customer DROP INDEX bmsql_customer_idx1;
ALTER TABLE bmsql_oorder DROP INDEX bmsql_oorder_idx1;
```

步骤四：环境调优

在执行 TPC-C 测试前，需要对 OceanBase 数据库的参数进行简单的设置。

OBProxy 调优，执行 `obclient -h<host_ip> -P<host_port> -uroot@sys -A -p` 命令进入系统租户。

说明

需使用 OBProxy 节点 IP 和端口登录 OceanBase 数据库的系统租户，才可修改 OBProxy 配置项。

```
# 提高 OBProxy 运行时内存上限
ALTER proxyconfig SET proxy_mem_limited='4G';
# 关闭 OBProxy 的压缩协议功能
ALTER proxyconfig set enable_compression_protocol=false;
```

OceanBase 数据库调优，执行 `obclient -h<host_ip> -P<host_port> -uroot@sys -A -p` 命令进入系统租户。

```
# 关闭 SQL 审计功能
ALTER system SET enable_sql_audit=false;
# 关闭性能事件的信息收集功能
ALTER system SET enable_perf_event=false;
# 设置系统日志级别为 ERROR，减少日志
ALTER system SET syslog_level='ERROR';
# 关闭记录追踪日志
alter system set enable_record_trace_log=false;
```

步骤五：进行 TPC-C 测试

1. 初始化环境

```
[admin@test run]$ ./runDatabaseDestroy.sh prop.oceanbase
```

2. 创建表并导入数据

```
[admin@test run]$ ./runDatabaseBuild.sh prop.oceanbase
```

3. 执行合并

使用 root 用户登录到 OceanBase 集群的 sys 租户，执行如下命令将当前大版本的 SSTable

和 MemTable 与前一个大版本的全量静态数据进行合并，使存储层统计信息更准确，生成的执行计划更稳定。

`<your tenant name>` 为待进行测试的租户名称（即 租户规格 中的 `tpcc_tenant`），您需根据实际测试租户进行替换。

```
MySQL [oceanbase]> ALTER SYSTEM major freeze tenant=<your tenant name>;
Query OK, 0 rows affected
```

4. 查询合并是否完成

```
obclient [oceanbase]> SELECT * FROM oceanbase.CDB_OB_MAJOR_COMPACTION;
```

输出如下，当看到 `STATUS` 列为 `IDLE` 时，表示合并完成。

```
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+-------------------------
--+--------+----------+-----------------------+-------+
| TENANT_ID | FROZEN_SCN          | FROZEN_TIME                | GLOBAL_BROADCAST_
SCN | LAST_SCN            | LAST_FINISH_TIME           | START_TIM
E                   | STATUS | IS_ERROR | IS_SUSPENDED | INFO |
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+-------------------------
--+--------+----------+-----------------------+-------+
|         1 | 1709661601360541623 | 2024-03-06 02:00:01.360542 |  1709661601360541
623 | 1709661601360541623 | 2024-03-06 02:06:25.027267 | 2024-03-06 02:00:01.38279
4 | IDLE   | NO       | NO                    |       |
|      1001 | 1709661602742784187 | 2024-03-06 02:00:02.742784 |  1709661602742784
187 | 1709661602742784187 | 2024-03-06 02:05:36.148110 | 2024-03-06 02:00:02.78097
8 | IDLE   | NO       | NO                    |       |
|      1002 | 1709661600590790760 | 2024-03-06 02:00:00.590791 |  1709661600590790
760 | 1709661600590790760 | 2024-03-06 02:05:43.819029 | 2024-03-06 02:00:00.64104
4 | IDLE   | NO       | NO                    |       |
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+-------------------------
--+--------+----------+-----------------------+-------+
```

5. 手动收集统计信息

执行 `obclient -h<host_ip> -P<host_port> -u<user_name>@<tenant_name> -p -A -D<db_name>` 命令进入测试租户，执行如下命令。其中，`$db_name` 需替换为上文 `prop.oceanbase` 文件中配置的 `database` 值。

```
call dbms_stats.gather_schema_stats('$db_name',degree=>96);
```

```

```

6. 执行压力测试

```
[admin@test run]$ ./runBenchmark.sh prop.oceanbase
```

测试结果

性能测试数据受到多种因素的影响，包括硬件配置、数据库安装部署方式、用户租户资源分配等，实际性能数据可能因环境因素会有些差异，此处测试结果仅供参考。

```
TPC-C Result
Measured tpmC (NewOrders) = 289711.96
Measured tpmTOTAL = 644025.66
```

3.5 进行 TPC-H 测试

本文基于 x86 架构的 CentOS Linux 7.9 镜像环境，介绍两种对 OceanBase 数据库运行 TPC-H 测试的方式：

- 通过 obd test 命令一键进行 TPC-H 测试。
- 手动 step by step 进行测试。

TPC-H 简介

TPC-H（商业智能计算测试）是美国交易处理效能委员会（TPC，Transaction Processing Performance Council）组织制定的用来模拟决策支持类应用的一个测试集。目前，学术界和工业界普遍采用 TPC-H 来评价决策支持技术方面应用的性能。

数据库模型

TPC-H 模型是典型的雪花模型，共有以下 8 张表，其中 nation（国家）和 region（区域）两张表的数据量是固定的，其余 6 张表的数据量与比例因子 SF（Scale Factor）相关，可以指定为 1、100、1000 等，分别代表 1GB、100GB、1000GB，根据指定的 SF 确定每张表的数据量。

- part：表示零件信息，主键为 `p_partkey`，取值范围 1~SF*200000，与 `partsupp` 表关联。
- supplier：表示供应商信息，主键为 `s_suppkey`，取值范围1~SF*10000，与 `partsupp`、`customer`、`nation` 表关联。
- partsupp：表示供应商零件信息，主键为 `ps_partkey`、`ps_suppkey`，与 `part`、`supplier`、`lineitem` 表关联。
- customer：表示消费者信息，主键为 `c_custkey`，取值范围 1~SF*150000，与 `orders` 表关联。
- orders：表示订单信息，主键为 `o_orderkey`，取值范围 1~SF*1500000，与 `lineitem` 表关联。
- lineitem：表示在售商品信息，主键为 `l_orderkey`、`l_linenumber`，这是数据量最大的一张表。

- nation：表示国家信息，主键为 `n_nationkey`，固定有 25 个国家。
- region：表示地区信息，主键为 `r_regionkey`，固定有 5 个地区。

SQL 类型

TPC-H 标准共有 22 条 SQL，全都是查询操作，主要考验数据库的如下数据分析能力：

- Aggregation（聚合）
- Join（连接）
- Expression Calculation（表达式计算）
- Subqueries（子查询）
- Parallelism and Concurrency（并行处理与并发性）

评估标准

TPC-H 用 3NF 实现了一个数据仓库，共包含 8 个基本关系，其主要评价指标是各个查询的响应时间，即从提交查询到结果返回所需时间。TPC-H 基准测试的度量单位是每小时执行的查询数（QphH@size），其中 H 表示每小时系统执行复杂查询的平均次数，size 表示数据库规模的大小，它能够反映出系统在处理查询时的能力。

使用 obdiag 在测试前对集群进行巡检

OceanBase 数据库是原生分布式数据库系统，故障根因分析通常比较复杂，因为需综合考量众多因素，包括但不限于机器环境、配置参数、运行负载等。专家在排查问题的时候需要获取大量的信息来分析故障，如何高效地采集和分析故障场景下分散在各个节点的信息，便是 OceanBase 敏捷诊断工具（OceanBase Diagnostic Tool，简称 obdiag）需要解决的问题。在开始进行 TPC-H 测试前，我们可以使用 obdiag 工具给 OceanBase 数据库做个“体检”，详细步骤和说明可参见官网 OceanBase 敏捷诊断工具（obdiag）文档。

环境准备

- JDK：建议使用 1.8u131 及以上版本
- make：可执行 `sudo yum install make` 安装
- gcc：可执行 `sudo yum install gcc` 安装
- mysql-devel：可执行 `sudo yum install mysql-devel` 安装
- Python 连接数据库的驱动：可执行 `sudo yum install MySQL-python` 安装
- prettytable：可执行 `pip install prettytable` 安装
- OBClient：详细介绍可参见 GitHub 仓库

测试方案

本次测试需要用到 4 台机器，TPC-H Tool、OBProxy 和 obd 安装在一台机器上，OceanBase 数据库使用 3 台机器组成一个 1:1:1 的集群。

注意

- 建议磁盘 iops 设置在 10000 以上，系统日志、事务日志、数据文件配置为三块盘。
- 使用 obd 部署集群时，建议不要使用 `obd cluster autodeploy` 命令，该命令为了保证稳定性，不会最大化资源利用率。建议根据实际环境，对 obd 配置文件进行定制化调整，最大化资源利用率。

测试环境（阿里云 ECS）

服务类型	ECS 类型	实例数	机器核心数	内存
OBServer	ecs.g7.8xlarge	3	32C	内存 128GB，每台机器系统盘 300GB，再挂载两块 400GB 云盘作为 Clog 盘和 Data 盘，性能级别为 PL1
OBProxy、TPC-H Tool	ecs.c7.16xlarge	1	32C	128GB

软件版本

服务类型	软件版本
OceanBase 数据库	OceanBase_CE 4.2.1.0
OBProxy	OBProxy_CE 4.2.1.0
TPC-H	V3.0.0

租户规格

部署成功后需创建进行 TPC-H 测试的租户及用户（sys 租户是管理集群的内置系统租户，请勿直接使用 sys 租户进行测试），设置租户的 `primary_zone` 为 `RANDOM`，`RANDOM` 表示新建表分区的 Leader 随机到任一 OBServer 节点。

```
CREATE RESOURCE UNIT tpch_unit max_cpu 26, memory_size '100g';
CREATE RESOURCE POOL tpch_pool unit = 'tpch_unit', unit_num = 1, zone_list=('zone1
','zone2','zone3');
CREATE TENANT tpch_mysql resource_pool_list=('tpch_pool'),  zone_list('zone1', 'zo
ne2', 'zone3'), primary_zone=RANDOM, locality='F@zone1,F@zone2,F@zone3' set variab
les ob_compatibility_mode='mysql', ob_tcp_invited_nodes='%', secure_file_priv = '/
';
```

obd 一键测试

说明

使用 obd 进行一键测试时，测试集群需是被 obd 管理的集群。通过 obd 部署的集群默认被 obd 管理，通过其他方法部署的集群需执行 obd 接管操作，详细操作方法可参见官网《OceanBase 安装部署工具》文档中 使用指南/命令行/使用 obd 接管集群 一文。

1. 安装 obtpch

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://mirrors.aliyun.com/oceanbase/OceanBase.
repo
sudo yum install obtpch
sudo ln -s /usr/tpc-h-tools/tpc-h-tools/ /usr/local/
```

2. 执行测试

```
obd test tpch <deploy_name>  --tenant=<tenant_name> -s 100 --remote-tbl-dir=/tmp/t
pch100
```

- 命令中的 `deploy_name` 表示部署集群名，`tenant_name` 表示进行测试的租户名（即 租户规格 中的 `tpch_mysql`），您需根据实际部署集群名和租户名进行修改。

- `remote-tbl-dir` 远程目录具备足够的容量能存储 TPC-H 的数据，建议单独一块盘存储加载测试数据。

- `obd test tpch` 命令会自动完成所有操作，包含测试数据的生成、性能参数调优、数据导入和测试。命令的详细介绍可参见官网《OceanBase 安装部署工具》文档中 obd 命令/测试命令组 中 **obd test tpch**。

3. 命令中的 `deploy_name` 表示部署集群名，`tenant_name` 表示进行测试的租户名（即 租户规格 中的 `tpch_mysql`），您需根据实际部署集群名和租户名进行修改。
4. `remote-tbl-dir` 远程目录具备足够的容量能存储 TPC-H 的数据，建议单独一块盘存储加载测试数据。
5. `obd test tpch` 命令会自动完成所有操作，包含测试数据的生成、性能参数调优、数据导入和测试。命令的详细介绍可参见官网《OceanBase 安装部署工具》文档中 obd 命令/测试命令组 中 **obd test tpch**。

手动进行 TPC-H 测试

步骤一：环境调优

在执行 TPC-H 测试前，需要对 OceanBase 数据库的参数进行简单的设置。

执行 `obclient -h<host_ip> -P<host_port> -uroot@sys -A -p` 命令进入系统租户，执行如下命令。

```
ALTER SYSTEM flush plan cache GLOBAL;
ALTER system SET enable_sql_audit=false;
ALTER system SET enable_perf_event=false;
```

```
ALTER system SET syslog_level='PERF';
alter system set enable_record_trace_log=false;
```

执行 `obclient -h<host_ip> -P<host_port> -u<user_name>@<tenant_name> -A -p` 命令进入测试租户，执行如下命令。

```
# 设置 SQL 工作区内存占整个租户内存百分比
SET GLOBAL ob_sql_work_area_percentage = 80;
# 设置 SQL 最大执行时间
SET GLOBAL ob_query_timeout = 36000000000;
# 设置事务超时时间
SET GLOBAL ob_trx_timeout = 36000000000;
# 设置最大网络包的大小
SET GLOBAL max_allowed_packet = 67108864;
# 租户在每个节点上可申请的并行执行线程数量
SET GLOBAL parallel_servers_target = 624;
```

步骤二：安装 TPC-H Tool

按照以下步骤安装 TPC-H Tool。

1. 下载 TPC-H Tool

   您可访问 TPC-H Tool 下载页面，下载 TPC-H Tool。

2. 解压 TPC-H Tool

   ```
   [admin@test ~]$ unzip *-tpc-h-tool.zip
   ```

3. 修改 `Makefile` 文件

   在 `TPC-H_Tools_v3.0.0/dbgen` 目录下执行如下命令复制 `makefile.suite` 文件并重命名为 `Makefile`

   ```
   [admin@test dbgen]$ cp makefile.suite Makefile
   ```

   修改 `Makefile` 文件中的 `CC`、`DATABASE`、`MACHINE`、`WORKLOAD` 等参数定义。修改后内容如下：

   ```
   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
```

4. 修改 `tpcd.h` 文件

   在 `dbgen` 目录下修改 `tpcd.h` 文件，并添加新的宏定义。

```
[admin@test dbgen]$ vim tpcd.h
```

   添加如下内容：

```
#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
```

5. 编译文件

```
[admin@test dbgen]$ make
```

步骤三：生成数据

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，在 `dbgen` 目录下执行如下命令生成数据。

```
[admin@test dbgen]$ ./dbgen -s 100
[admin@test dbgen]$ mkdir tpch100
[admin@test dbgen]$ mv *.tbl tpch100
```

步骤五：生成查询 SQL

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，需将 SQL 语句中的 cpu_num 修改为实际并发数。

1. 复制 `qgen` 和 `dists.dss` 文件至 `queries` 目录

```
[admin@test dbgen]$ cp qgen queries
[admin@test dbgen]$ cp dists.dss queries
```

2. 在 `queries` 目录下创建 `gen.sh` 脚本生成查询 SQL

```
[admin@localhost queries]$ vim gen.sh
```

脚本内容如下：

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

3. 执行 `gen.sh` 脚本

```
[admin@localhost queries]$ chmod +x  gen.sh
[admin@localhost queries]$ ./gen.sh
```

4. 查询 SQL 进行调整

```
[admin@localhost queries]$ dos2unix *
```

调整后的查询 SQL 请参考 GitHub。您需要将 GitHub 给出的 SQL 语句中的 `cpu_num` 修改为实际并发数。建议并发数的数值与可用 CPU 总数相同，两者相等时性能最好，可以在 sys 租户下使用如下命令查看租户的可用 CPU 总数。

```
SELECT sum(cpu_capacity_max) FROM __all_virtual_server;
```

以 q1 为例，修改后的 SQL 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;
```

步骤六：新建表

在 `dbgen` 目录下创建 `load` 文件夹，并在 `load` 文件夹下创建表结构文件 `create_tpch_mysql_table_part.ddl`。

```
[admin@localhost dbgen]$ mkdir load
[admin@localhost dbgen]$ cd load
[admin@localhost load]$ vim create_tpch_mysql_table_part.ddl
```

文件内容如下：

```
DROP TABLE IF EXISTS lineitem;
DROP TABLE IF EXISTS orders;
DROP TABLE IF EXISTS partsupp;
DROP TABLE IF EXISTS part;
DROP TABLE IF EXISTS customer;
DROP TABLE IF EXISTS supplier;
DROP TABLE IF EXISTS nation;
DROP TABLE IF EXISTS region;
DROP TABLEGROUP IF EXISTS tpch_tg_lineitem_order_group;
DROP TABLEGROUP IF EXISTS tpch_tg_partsupp_part;

CREATE TABLEGROUP IF NOT EXISTS tpch_tg_lineitem_order_group binding true partitio
n by key 1 par
CREATE TABLEGROUP IF NOT EXISTS tpch_tg_partsupp_part binding true partition by ke
y 1 partitions

DROP TABLE IF EXISTS lineitem;
```

```
CREATE TABLE lineitem (
  l_orderkey BIGINT NOT NULL,
  l_partkey BIGINT NOT NULL,
  l_suppkey INTEGER NOT NULL,
  l_linenumber INTEGER 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 char(25) DEFAULT NULL,
  l_shipmode char(10) DEFAULT NULL,
  l_comment varchar(44) DEFAULT NULL,
  PRIMARY KEY(l_orderkey, l_linenumber))row_format = condensed
  tablegroup = tpch_tg_lineitem_order_group
  partition by key (l_orderkey) partitions cpu_num;

DROP TABLE IF EXISTS orders;
CREATE TABLE orders (
  o_orderkey bigint not null,
  o_custkey bigint not null,
  o_orderstatus char(1) default null,
  o_totalprice bigint default null,
  o_orderdate date not null,
  o_orderpriority char(15) default null,
  o_clerk char(15) default null,
  o_shippriority bigint default null,
  o_comment varchar(79) default null,
  PRIMARY KEY (o_orderkey))row_format = condensed
  tablegroup = tpch_tg_lineitem_order_group
  partition by key(o_orderkey) partitions cpu_num;

DROP TABLE IF EXISTS partsupp;
CREATE TABLE partsupp (
  ps_partkey bigint not null,
  ps_suppkey bigint not null,
  ps_availqty bigint default null,
  ps_supplycost bigint default null,
  ps_comment varchar(199) default null,
  PRIMARY KEY (ps_partkey, ps_suppkey))row_format = condensed
  tablegroup tpch_tg_partsupp_part
  partition by key(ps_partkey) partitions cpu_num;

DROP TABLE IF EXISTS part;
CREATE TABLE part (
  p_partkey bigint not null,
```

```
  p_name varchar(55) default null,
  p_mfgr char(25) default null,
  p_brand char(10) default null,
  p_type varchar(25) default null,
  p_size bigint default null,
  p_container char(10) default null,
  p_retailprice bigint default null,
  p_comment varchar(23) default null,
  PRIMARY KEY (p_partkey))row_format = condensed
  tablegroup tpch_tg_partsupp_part
  partition by key(p_partkey) partitions cpu_num;

DROP TABLE IF EXISTS customer;
CREATE TABLE customer (
  c_custkey bigint not null,
  c_name varchar(25) default null,
  c_address varchar(40) default null,
  c_nationkey bigint default null,
  c_phone char(15) default null,
  c_acctbal bigint 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 cpu_num;

DROP TABLE IF EXISTS supplier;
CREATE TABLE supplier (
  s_suppkey bigint not null,
  s_name char(25) default null,
  s_address varchar(40) default null,
  s_nationkey bigint default null,
  s_phone char(15) default null,
  s_acctbal bigint default null,
  s_comment varchar(101) default null,
  PRIMARY KEY (s_suppkey))row_format = condensed
  partition by key(s_suppkey) partitions cpu_num;

DROP TABLE IF EXISTS nation;
CREATE TABLE nation (
  n_nationkey bigint not null,
  n_name char(25) default null,
  n_regionkey bigint default null,
  n_comment varchar(152) default null,
  PRIMARY KEY (n_nationkey))row_format = condensed;

DROP TABLE IF EXISTS region;
CREATE TABLE region (
  r_regionkey bigint not null,
  r_name char(25) default null,
  r_comment varchar(152) default null,
```

```
PRIMARY KEY (r_regionkey))row_format = condensed;
```

步骤七：加载数据

您可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 `load.py` 脚本

   ```
   [admin@localhost load]$ vim load.py
   ```

   脚本内容如下：

   ```
   #!/usr/bin/env python
   #-*- encoding:utf-8 -*-
   import os
   import sys
   import time
   import commands
   hostname='$host_ip'  # 注意！！请填写某个 OBServer 节点，如 OBServer A 节点所在服务器的 I
   P 地址
   port='$host_port'              # OBServer A 节点的端口号
   tenant='$tenant_name'              # 租户名
   user='$user'              # 用户名
   password='$password'           # 密码
   data_path='$path'         # 注意！！请填写 OBServer A 节点所在服务器下 tbl 所在目录
   db_name='$db_name'             # 数据库名
   # 创建表
   cmd_str='obclient -h%s -P%s -u%s@%s -p%s -D%s < create_tpch_mysql_table_part.ddl'%
   (hostname,port,user,tenant,password,db_name)
   result = commands.getstatusoutput(cmd_str)
   print result
   cmd_str='obclient -h%s -P%s -u%s@%s -p%s  -D%s -e "show tables;" '%(hostname,port,
   user,tenant,password,db_name)
   result = commands.getstatusoutput(cmd_str)
   print result
   cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s -c  -D%s -e "load data /*+ parallel(80
   ) */ infile '%s/customer.tbl' into table customer fields terminated by '|';" """ %
   (hostname,port,user,tenant,password,db_name,data_path)
   result = commands.getstatusoutput(cmd_str)
   print result
   cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s -c  -D%s -e "load data /*+ parallel(80
   ) */ infile '%s/lineitem.tbl' into table lineitem fields terminated by '|';" """ %
   (hostname,port,user,tenant,password,db_name,data_path)
   result = commands.getstatusoutput(cmd_str)
   print result
   cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s -c -D%s -e "load data /*+ parallel(80
   ```

```
) */ infile '%s/nation.tbl' into table nation fields terminated by '|';" """ %(hos
tname,port,user,tenant,password,db_name,data_path)
result = commands.getstatusoutput(cmd_str)
print result
cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s -c  -D%s -e "load data /*+ parallel(80
) */ infile '%s/orders.tbl' into table orders fields terminated by '|';" """ %(hos
tname,port,user,tenant,password,db_name,data_path)
result = commands.getstatusoutput(cmd_str)
print result
cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s   -D%s -e "load data /*+ parallel(80)
*/ infile '%s/partsupp.tbl' into table partsupp fields terminated by '|';" """ %(h
ostname,port,user,tenant,password,db_name,data_path)
result = commands.getstatusoutput(cmd_str)
print result
cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s -c  -D%s -e "load data /*+ parallel(80
) */ infile '%s/part.tbl' into table part fields terminated by '|';" """ %(hostnam
e,port,user,tenant,password,db_name,data_path)
result = commands.getstatusoutput(cmd_str)
print result
cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s -c  -D%s -e "load data /*+ parallel(80
) */ infile '%s/region.tbl' into table region fields terminated by '|';" """ %(hos
tname,port,user,tenant,password,db_name,data_path)
result = commands.getstatusoutput(cmd_str)
print result
cmd_str=""" obclient -h%s -P%s -u%s@%s -p%s -c  -D%s -e "load data /*+ parallel(80
) */ infile '%s/supplier.tbl' into table supplier fields terminated by '|';" """ %
(hostname,port,user,tenant,password,db_name,data_path)
result = commands.getstatusoutput(cmd_str)
print result
```

2. 加载数据

加载数据需要安装 OBClient 客户端。

```
[admin@localhost load]$ python load.py
```

输出如下：

```
(0,'')
(0, 'obclient: [Warning] Using a password on the command line interface can be ins
ecure.   \nTABLE_NAME\nT1\nLINEITEM\nORDERS\nPARTSUPP\nPART\nCUSTOMER\nSUPPLIER\nN
ATION\nREGION')
(0, 'obclient: [Warning] Using a password on the command line interface can be ins
ecure.')
(0, 'obclient: [Warning] Using a password on the command line interface can be ins
ecure.')
(0, 'obclient: [Warning] Using a password on the command line interface can be ins
ecure.')
```

```
(0, 'obclient: [Warning] Using a password on the command line interface can be insecure.')
(0, 'obclient: [Warning] Using a password on the command line interface can be insecure.')
(0, 'obclient: [Warning] Using a password on the command line interface can be insecure.')
(0, 'obclient: [Warning] Using a password on the command line interface can be insecure.')
```

3. 执行合并

使用 root 用户登录到 OceanBase 集群的 sys 租户，执行如下命令将当前大版本的 SSTable 和 MemTable 与前一个大版本的全量静态数据进行合并，使存储层统计信息更准确，生成的执行计划更稳定。

`<your tenant name>` 为待进行测试的租户名称（即 租户规格 中的 `tpch_mysql`），您需根据实际测试租户进行替换。

```
obclient [oceanbase]> ALTER SYSTEM major freeze tenant=<your tenant name>;
```

4. 查询合并是否完成

```
obclient [oceanbase]> SELECT * FROM oceanbase.CDB_OB_MAJOR_COMPACTION;
```

输出如下，当看到 `STATUS` 列为 `IDLE` 时，表示合并完成。

```
+-----------+---------------------+----------------------------+----------------------+---------------------+----------------------------+----------------------------+--------+----------+--------------+------+
| TENANT_ID | FROZEN_SCN          | FROZEN_TIME                | GLOBAL_BROADCAST_SCN | LAST_SCN            | LAST_FINISH_TIME           | START_TIME                 | STATUS | IS_ERROR | IS_SUSPENDED | INFO |
+-----------+---------------------+----------------------------+----------------------+---------------------+----------------------------+----------------------------+--------+----------+--------------+------+
|         1 | 1709661601360541623 | 2024-03-06 02:00:01.360542 |  1709661601360541623 | 1709661601360541623 | 2024-03-06 02:06:25.027267 | 2024-03-06 02:00:01.382794 | IDLE   | NO       | NO           |      |
|      1001 | 1709661602742784187 | 2024-03-06 02:00:02.742784 |  1709661602742784187 | 1709661602742784187 | 2024-03-06 02:05:36.148110 | 2024-03-06 02:00:02.780978 | IDLE   | NO       | NO           |      |
|      1002 | 1709661600590790760 | 2024-03-06 02:00:00.590791 |  1709661600590790760 | 1709661600590790760 | 2024-03-06 02:05:43.819029 | 2024-03-06 02:00:00.641044 | IDLE   | NO       | NO           |      |
+-----------+---------------------+----------------------------+-------------------
```

```
----+--------------------+---------------------------+-------------------------
--+--------+---------+-------------+------+
```

5. 收集统计信息

   执行 `obclient -h<host_ip> -P<host_port> -u<user_name>@<tenant_name> -p -A -D<db_name>` 命令进入测试租户，执行如下命令。其中，`$db_name` 需替换为上文 `load.py` 文件中配置的 `db_name` 值。

```
obclient [oceanbase]> call dbms_stats.gather_schema_stats('$db_name',degree=>96);
```

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. 在 `queries` 目录下编写测试脚本 `tpch.sh`

```
[admin@localhost queries]$ vim tpch.sh
```

   脚本内容如下：

```
#!/bin/bash
TPCH_TEST="obclient -h ${host_ip} -P ${host_port} -uroot@tpch_mysql  -D ${db_name}  -
p${password} -c"
# warmup 预热
for i in {1..22}
do
    sql1="source db${i}.sql"
    echo $sql1| $TPCH_TEST >db${i}.log  || ret=1
done
# 正式执行
for i in {1..22}
do
    starttime=`date +%s%N`
    echo `date  '+[%Y-%m-%d %H:%M:%S]'` "BEGIN Q${i}"
    sql1="source db${i}.sql"
    echo $sql1| $TPCH_TEST >db${i}.log  || ret=1
    stoptime=`date +%s%N`
    costtime=`echo $stoptime $starttime | awk '{printf "%0.2f\n", ($1 - $2) / 1000
000000}'`
    echo `date  '+[%Y-%m-%d %H:%M:%S]'` "END,COST ${costtime}s"
done
```

2. 执行测试脚本

```
[admin@localhost queries]$ sh tpch.sh
```

测试结果

性能测试数据受到多种因素的影响，包括硬件配置、数据库安装部署方式、用户租户资源分配等，实际性能数据可能因环境因素会有些差异，此处测试结果仅供参考。

测试数据量：100GB

Query ID	三节点 OceanBase 集群 V4.2.1 查询响应时间（单位：秒）
Q1	2.24
Q2	0.48
Q3	1.49
Q4	0.66
Q5	0.95
Q6	0.14
Q7	1.35
Q8	1.09
Q9	4.46
Q10	0.95
Q11	0.19
Q12	1.34
Q13	1.86
Q14	0.41
Q15	0.88
Q16	0.67
Q17	1.57
Q18	0.91
Q19	0.64

Q20	1.12
Q21	2.52
Q22	1.11
Total	27.03

3.6 使用 JMeter 运行业务场景测试

JMeter 是由 Apache 软件基金会开发的一款开源的性能测试工具，基于 Java 平台构建，主要用于对软件应用进行负载和压力测试。JMeter 最初是为测试 Web 应用程序而设计，但后来也扩展了其他的测试功能。我们可以通过 JMeter 来模拟业务 SQL 并分析数据库服务器在不同负载情况下的性能。

环境准备

- 配置 Java 环境，JDK 版本要求在 1.8 以上。
- 安装 JMeter 工具，具体方法可参见 JMeter 文档。
- 下载 Java 驱动，使用 JMeter 测试 OceanBase 数据库性能前需先下载连接数据库的 Java 驱动，可从 MySQL官网 下载。推荐下载 `mysql-connector-java-5.1.47.jar`，下载后需将文件放到 JMeter 的 `lib` 文件夹中，并重启 JMeter。

测试方案

本次测试我们简单模拟一个交易业务的场景，仅使用最基本的 SQL 模型来演示 JMeter 的测试流程。在实际测试的过程中，可以按照实际的业务场景需求进行补充添加。

建表语句

请在 OceanBase 集群 MySQL 租户下的业务账户执行如下 SQL。

```
CREATE TABLE account(id bigint NOT NULL AUTO_INCREMENT PRIMARY KEY
 , name varchar(50) NOT NULL UNIQUE
 , value bigint NOT NULL
 , gmt_create timestamp DEFAULT current_timestamp NOT NULL
 , gmt_modified timestamp DEFAULT current_timestamp NOT NULL  );
```

测试 SQL

基本 SQL 模型如下，其中 `trans_amount` 为交易金额，`credit_id` 为交易支付方，`debit_id` 为交易收款方。

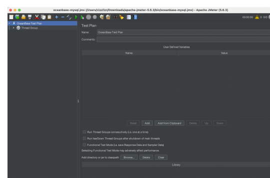
```
begin ;
update account set  value = value - ${trans_amount} , gmt_modified = current_timest
amp where id = ${credit_id} ;
update account set  value = value + ${trans_amount} , gmt_modified = current_timest
amp where id = ${debit_id} ;
-- commit or rollback;
commit;
```

测试步骤

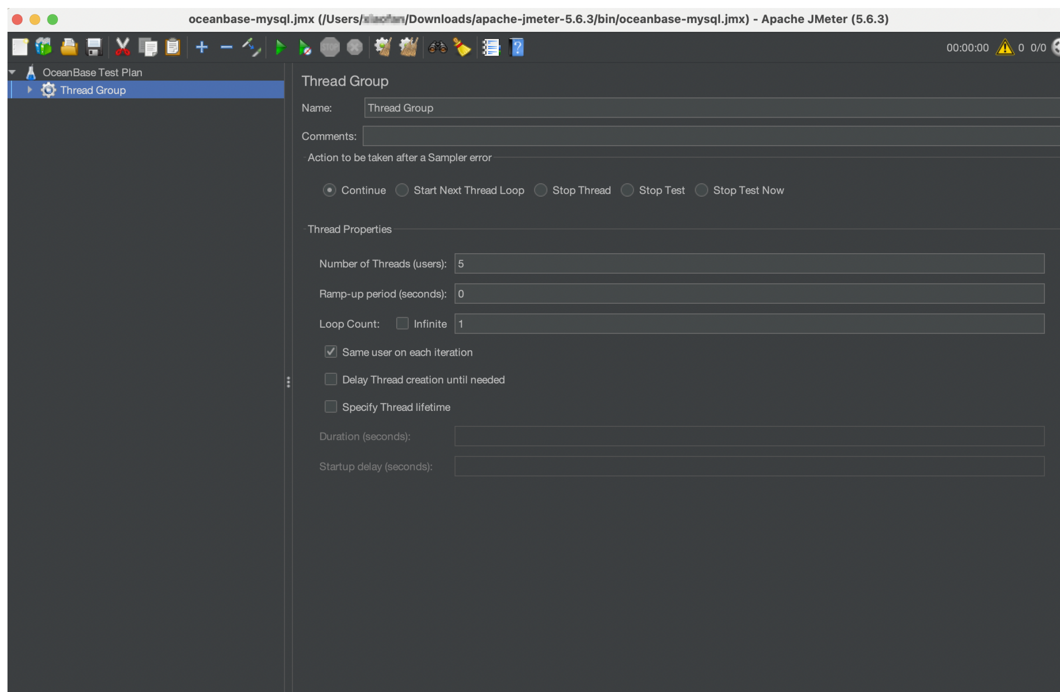
JMeter 能在命令行下运行，也可在图形界面运行。这里以图形界面为例介绍简单操作，详细的操作可参见 JMeter 的 用户手册。

新建测试计划和线程组

在 `apache-jmeter-xxx/bin/` 路径下，执行 `sh jmeter` 命令启动后，会进入 JMeter 的图形界面，下图示例中新建名为 `Oceanbase Test Plan` 的测试计划。

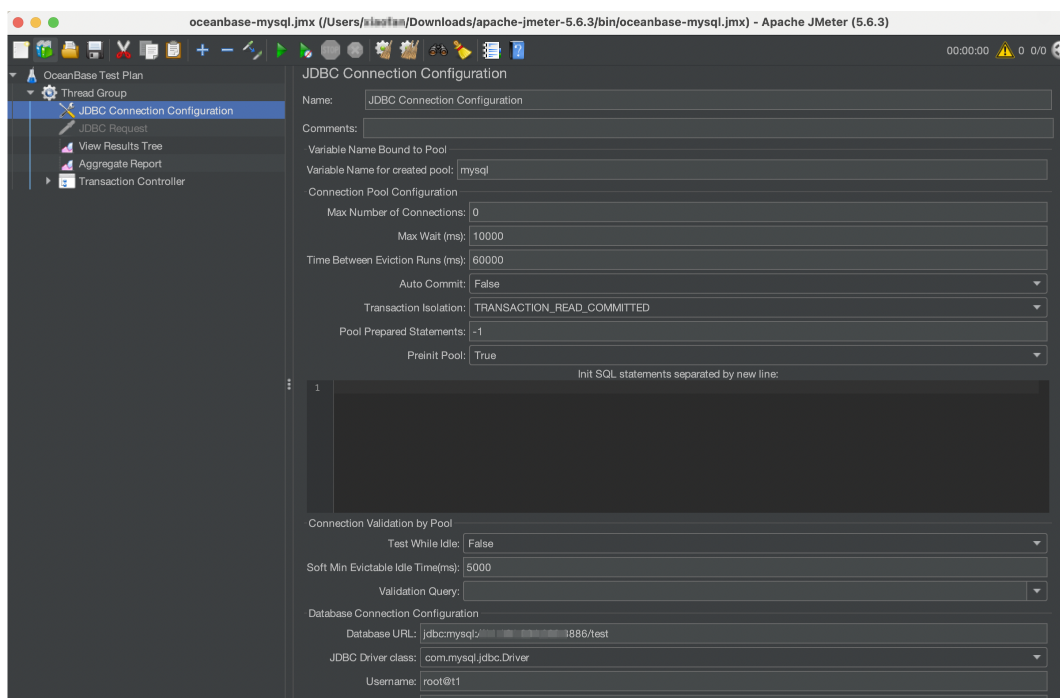


单击 **Oceanbase Test Plan**，选择 **Add** > **Threads(Users)** > **Thread-Group**，新建 Thread-Group。在整个 JMeter 的测试过程中，有很多可以配置的参数，具体参数说明可参见 JMeter 文档 Elements of a Test Plan。



添加 JDBC 连接

单击 **Thread-Group**，选择 **Add** > **Config Element** > **JDBC Connection Configuration**，新建 JDBC 配置文件。



上图中有很多属性，都是一个连接池常具备的参数，详情参考网上关于 Java 连接池配置的经验。

您需注意以下参数：

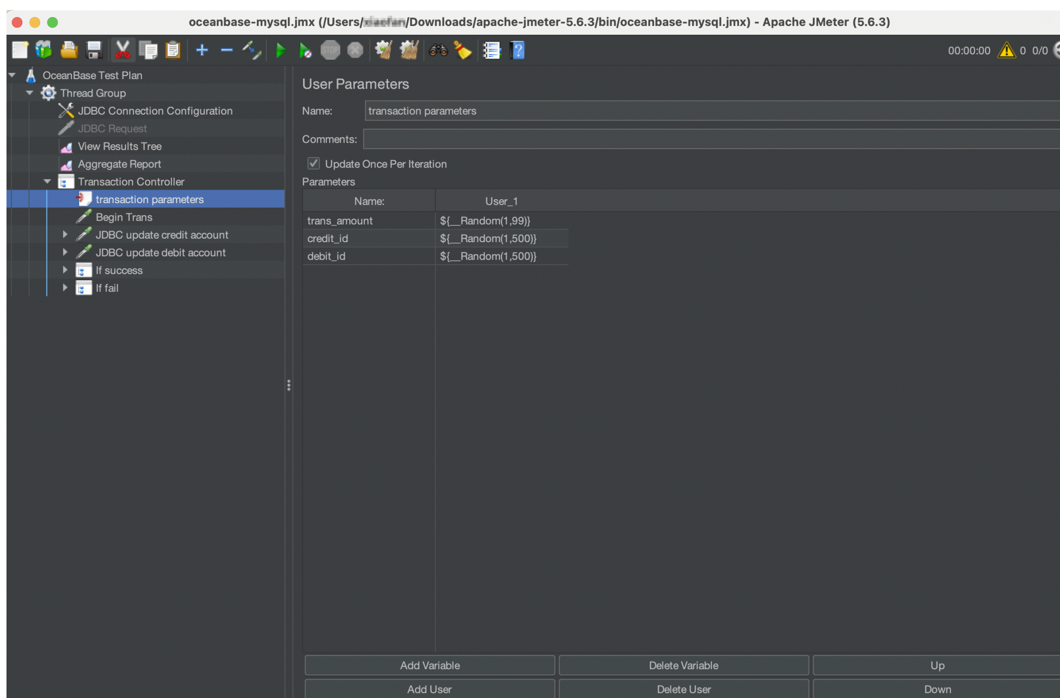
- Max Number of Connections：连接池里的最大连接数。如果压测线程数远高于该值，那么压测线程需要等待这个连接池创建或返还数据库连接（即到 OceanBase 数据库的连接）给它，如果等不到可能会报错。因此，在测试过程中，客户端压测线程获取不到连接，可能与 JDBC 配置相关，不一定与 OceanBase 数据库有直接关系，在 Java 应用中同理。
- Transaction Isolation：数据库连接使用的事务隔离级别，OceanBase 数据库默认的隔离级别为读已提交（Read Committed）。
- Test While Idle：连接探活（keepalive）设置，对应用很有必要。有时候应用会提示数据库在一个关闭的连接上执行 SQL，进而报错，这是因为连接池中的数据库连接因为其他原因断开了。因此，数据库连接池通常都需要探活机制。此处因为压测场景基本无闲置连接，所以可以设置为 `False`。
- Database URL：数据库的连接 URL，例如 `jdbc:oceanbase://10.10.10.1:2881/test`。
- JDBC Driver Class：数据库驱动中的 Main 类名。
- Username：执行测试的用户名，OceanBase 数据库的用户名格式比较特别，常用格式为 `用户名@租户名` 或 `集群名:租户名`，如 `root@t1`。

事务控制器

单击 **Thread-Group**，选择 **Add** > **Logic Controller** > **Transaction Controller**，新建事务控制器，用于将多个请求组合成一个事务，下面我们对事务控制器进行配置。

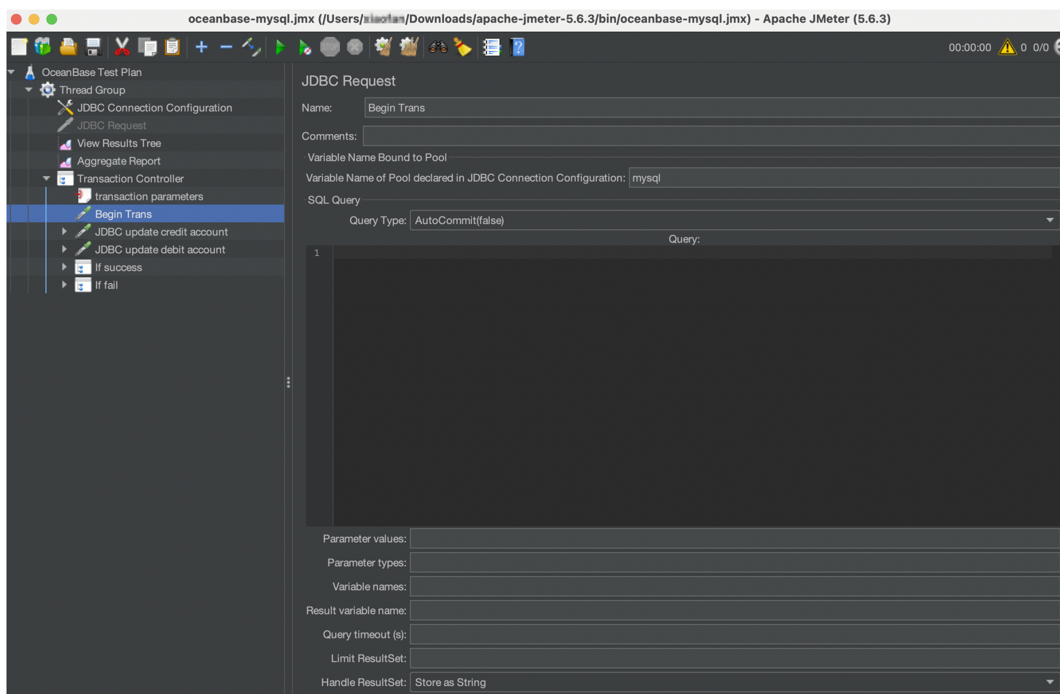
- 配置事务变量

  单击 **Transaction Controller**，选择 **Add** > **Pre Processors** > **User Parameters**，添加需要的变量。在本测试中，有三个变量：账户 A（`credit_id`），账户 B（`debit_id`）和转账金额（`trans_amount`）。账户参数和金额采取随机数，随机数的值不可超出测试数据的实际范围。



- 开启事务

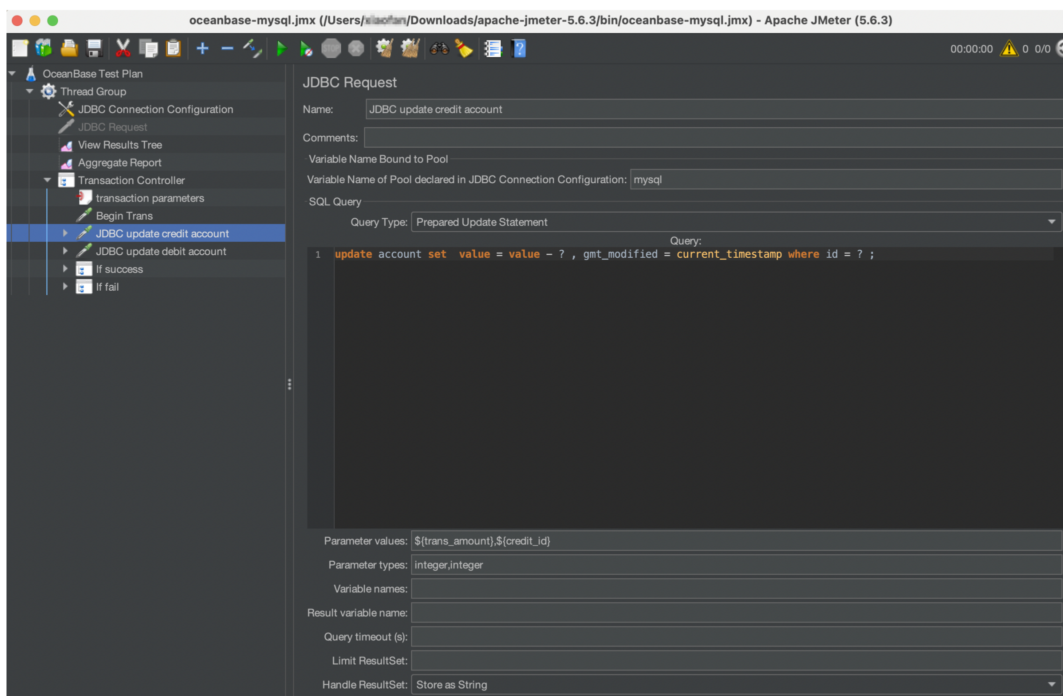
  单击 **Transaction Controller**，选择 **Add** > **Sampler** > **JDBC Request**，新增 JDBC 请求，设置 **Query Type** 为 `AutoCommit(false)`，开启显式事务。



- 新增扣减账户 A 余额的 JDBC 请求

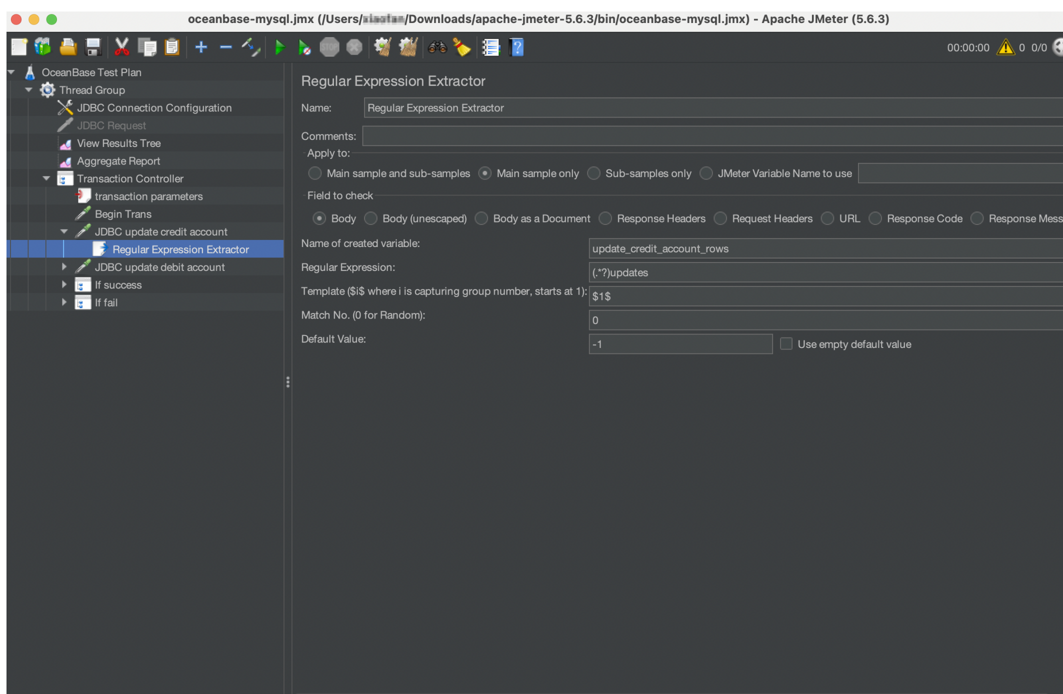
  单击 **Transaction Controller**，选择 **Add** > **Sampler** > **JDBC Request**，新增 JDBC 请求，将该 JDBC Request 命名为 `JDBC update credit account`，并添加扣减账户 A 余额的 SQL

请求。



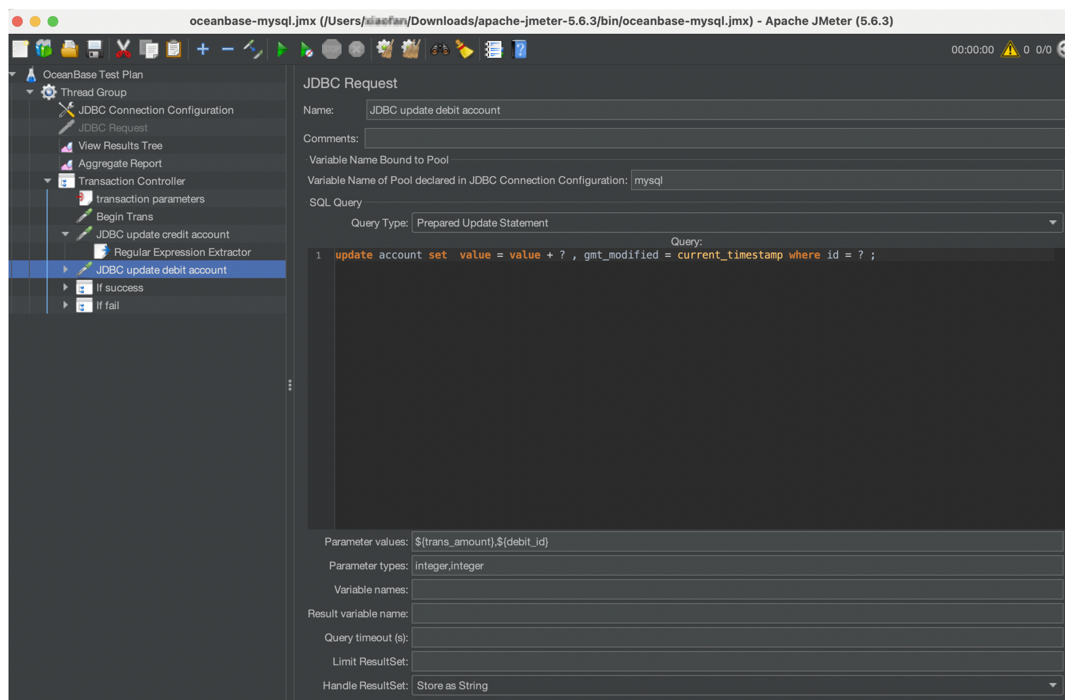
- 新增 Post 处理逻辑，获取扣减账户 A 余额的 JDBC 请求更新返回值

单击 **JDBC update credit account**，选择 **Add** > **Post Processors** > **Regular Expression Extractor**，新增一个 Post 处理逻辑，获取扣减账户 A 余额的 JDBC 请求更新返回值。



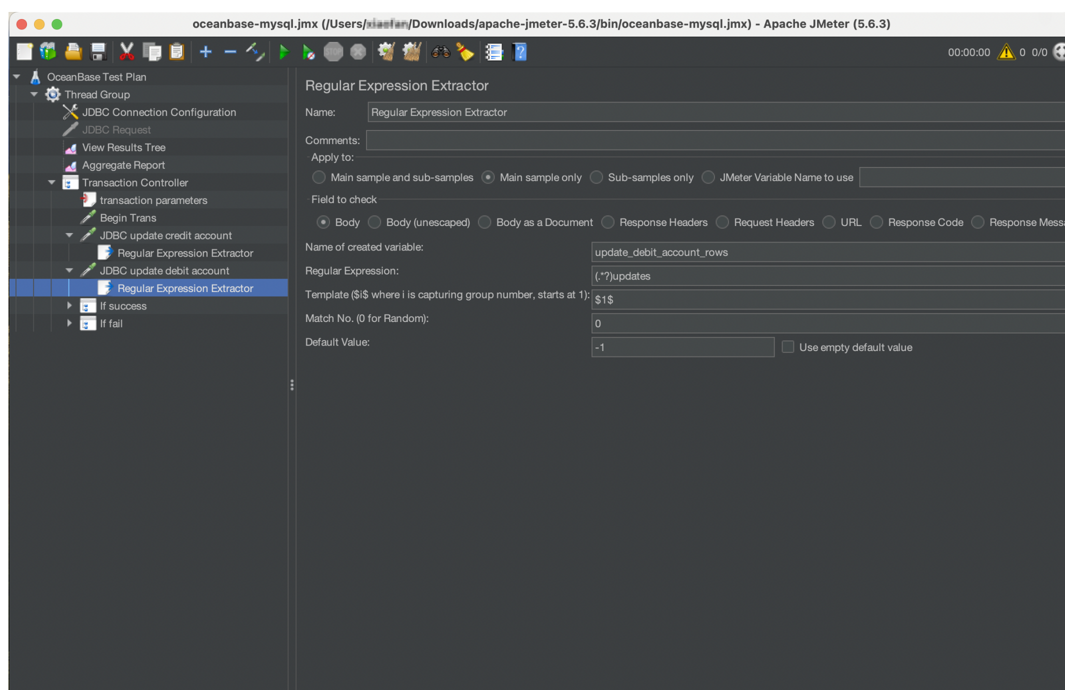
- 新增增加账户 B 余额的 JDBC 请求

单击 **Transaction Controller**，选择 **Add** > **Sampler** > **JDBC Request**，新增 JDBC 请求，将该 JDBC Request 命名为 `JDBC update debit account`，并添加增加账户 B 余额的 SQL 请求。



- 新增 Post 处理逻辑，获取增加账户 B 余额的 JDBC 请求更新返回值

单击 **JDBC update debit account**，选择 **Add** > **Post Processors** > **Regular Expression Extractor**，新增一个 Post 处理逻辑，获取增加账户 B 余额的 JDBC 请求更新返回值。

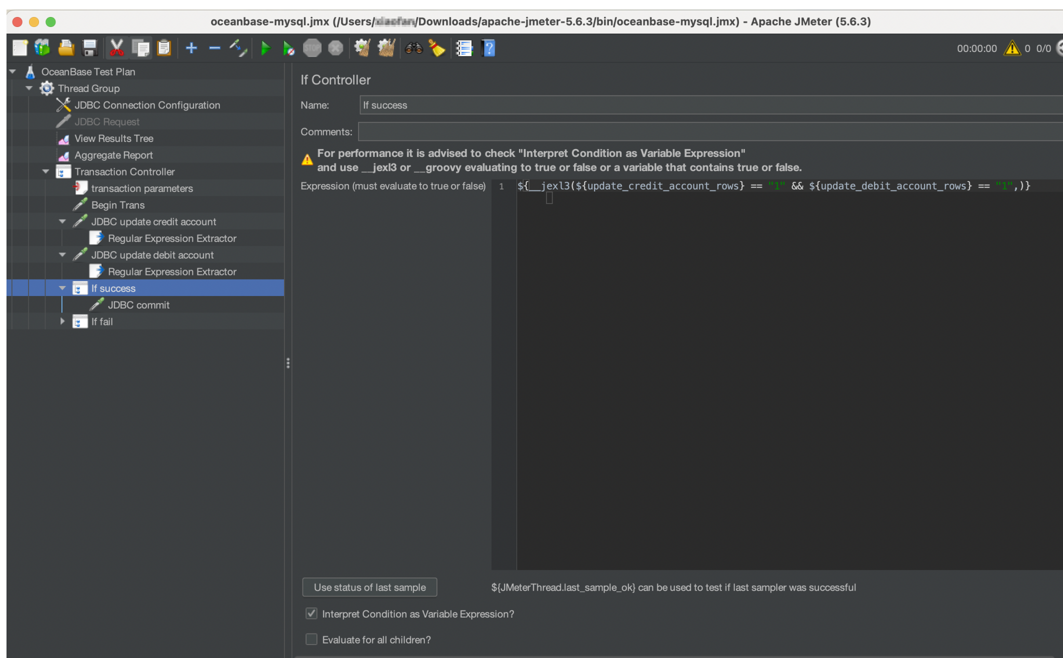


判断逻辑 - 成功流程

如果上面两笔账户更新成功，则提交事务。

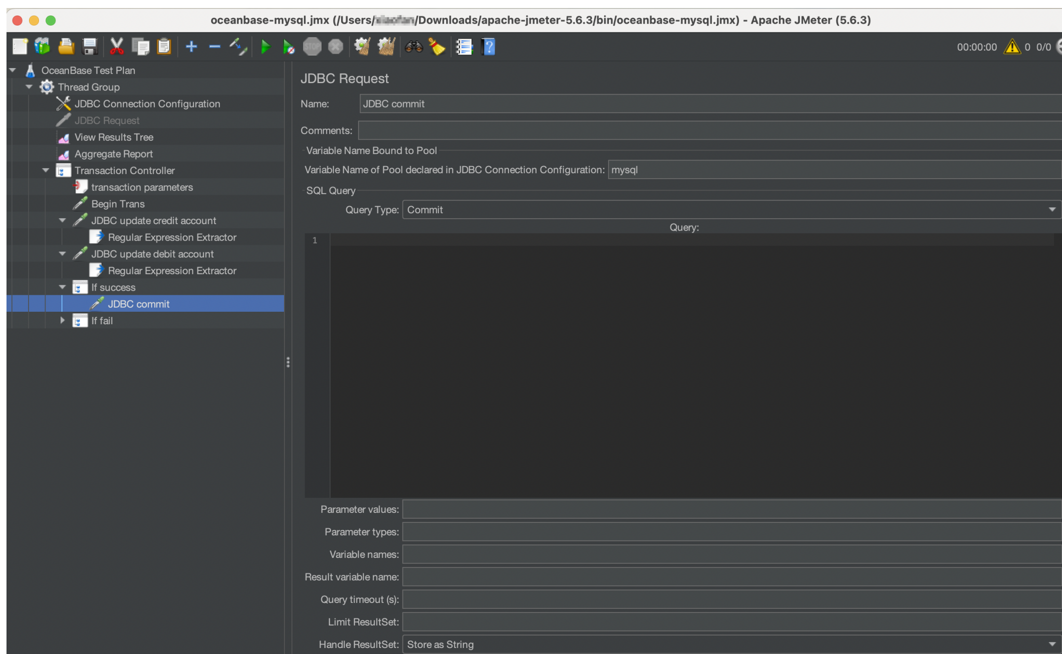
- 新增判断控制 IF

  单击 **Transaction Controller**，选择 **Add** > **Logic Controller** > **If Controller**，将该 If Controller 命名为 `If success`，并新增 If 判断。



- 新增更新成功后的提交动作

  单击 **If success**，选择 **Add** > **Sampler** > **JDBC Request**，新增 JDBC 请求，并设置 **Query Type** 为 `Commit`。

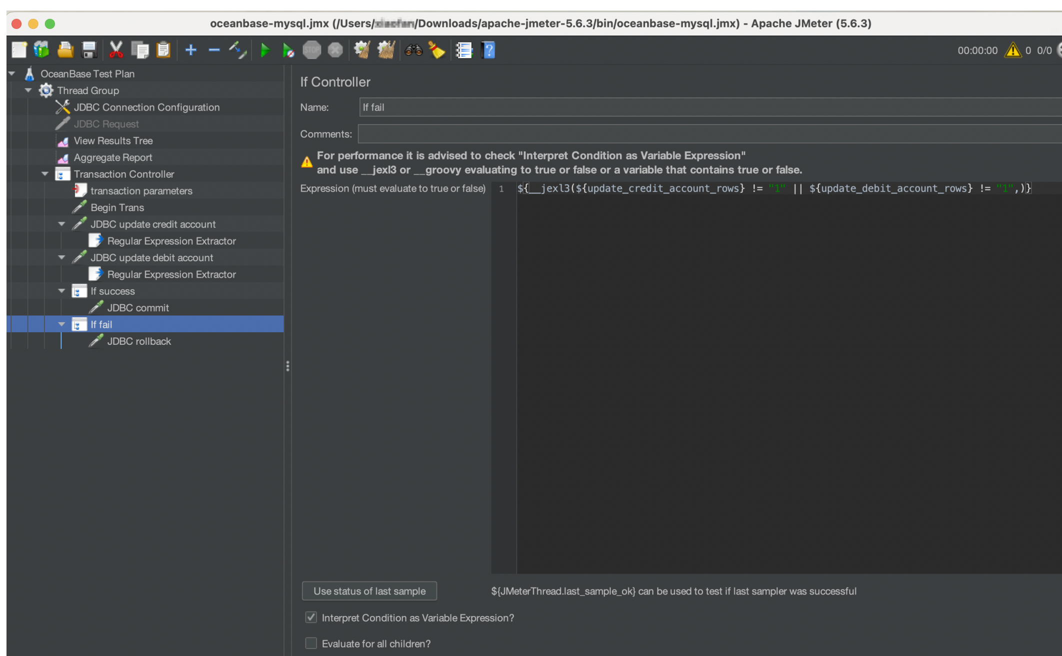


判断逻辑 - 失败流程

如果上面两笔账户的更新有一笔失败，则回滚事务。

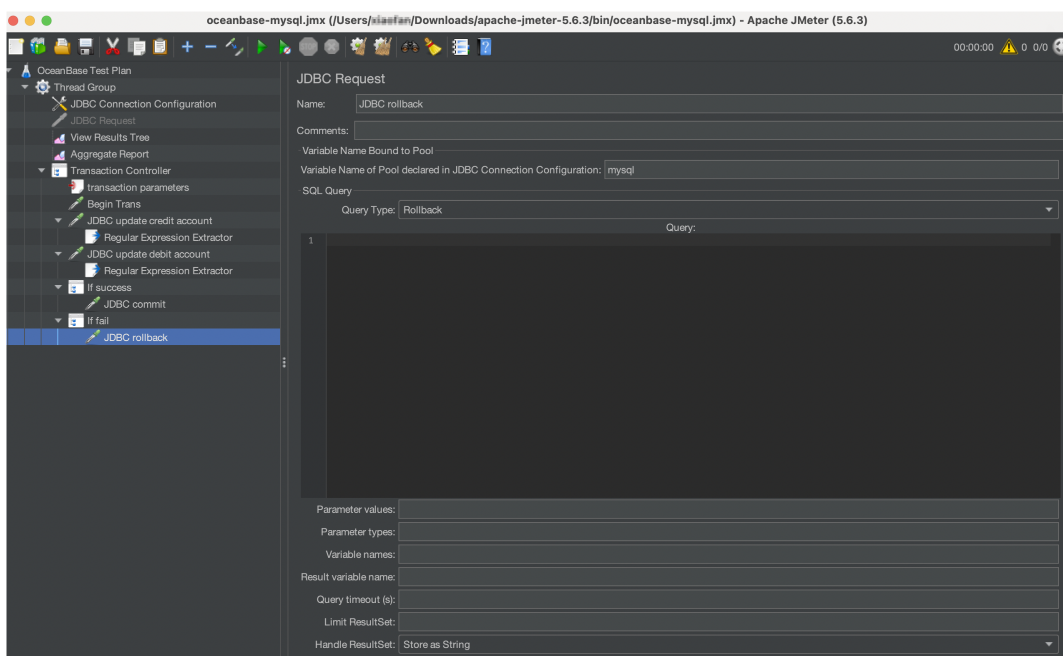
- 新增判断控制 IF

  单击 **Transaction Controller**，选择 **Add** > **Logic Controller** > **If Controller**，将该 If Controller 命名为 `If fail`，并新增 If 判断。



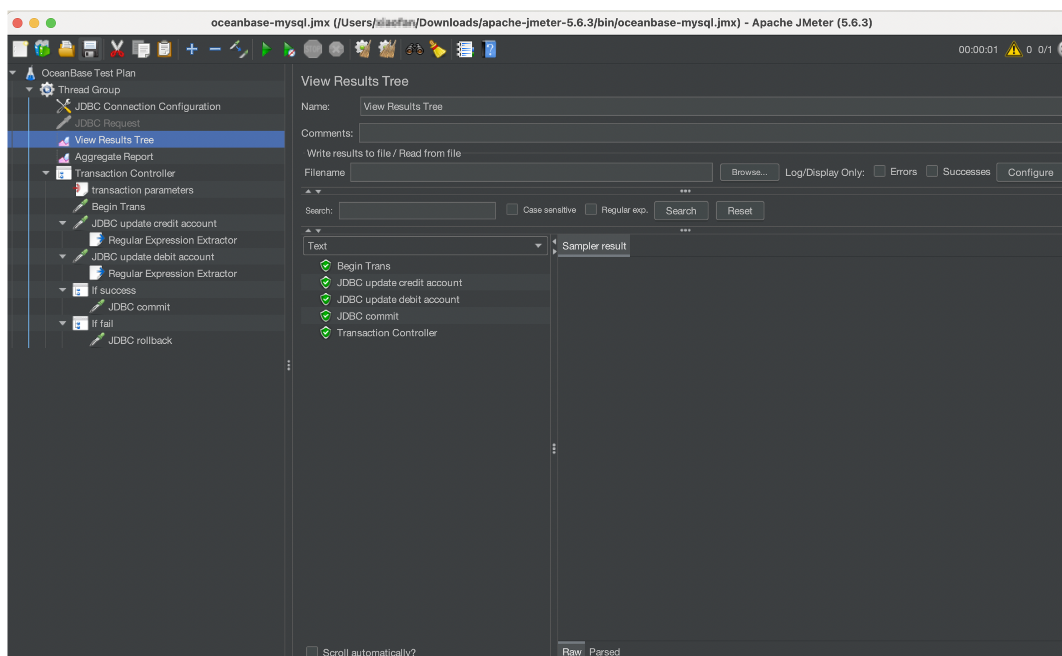
- 新增更新失败后的提交动作

单击 **If fail**，选择 **Add > Sampler > JDBC Request**，新增 JDBC 请求，并设置 **Query Type** 为 `Rollback`。



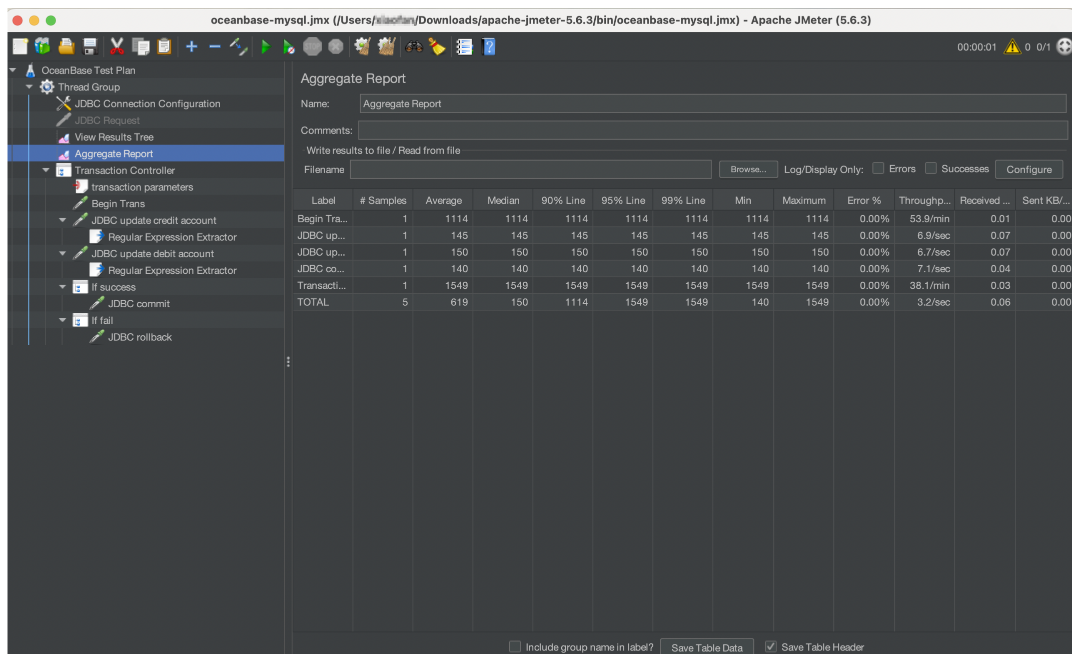
查看结果

单击 **Thread-Group**，选择 **Add > Listerner > View Results Tree**，新建结果树文件，可以查看成功或失败的结果。



单击 **Thread-Group**，选择 **Add > Listerner > Aggregate Report**，新建聚合报告，可以查看

汇总报告。



3.7 其他常见测试点

并行导入

我们可以以 TPC-H 测试中的 `customer` 表为基础，创建一张相同表结构的空表 `customer2`。然后使用 INSERT INTO ...SELECT 语句将 `customer` 表的全部 1500 万行数据插入新表 `customer2` 中。下面我们分别在关闭和开启 PDML 的情况下执行，观察其效果和区别。

首先，复制 `customer` 的表结构，创建空表 `customer2`。

```
CREATE TABLE customer2 LIKE customer;
```

- 不开启 PDML 执行

  创建好 `customer2` 表后，先以默认配置不开启并行的方式插入，因为这是一个 1500 万行的大事务，我们需要将 OceanBase 数据库 SQL 最大执行时间调整到更大的值（单位为 μs）：

  ```
  SET ob_query_timeout = 21600000000;
  ```

  执行如下命令插入数据：

  ```
  obclient [test]> INSERT INTO customer2 SELECT * FROM customer;
  ```

  输出如下，不开启并行的情况下，单个事务插入 1500 万行数据，OceanBase 数据库的耗时为 110 秒。

  ```
  Query OK, 15000000 rows affected (1 min 50.043 sec)
  Records: 15000000  Duplicates: 0  Warnings: 0
  ```

- 开启 PDML 执行

  通过添加一个 Hint，开启 PDML 的执行选项。注意再次插入前，需先执行如下命令将上次插入的数据清空。

  ```
  obclient [test]> TRUNCATE TABLE customer2;
  ```

一般来说，如果没有多个 SQL 并发执行，单条 SQL的并行度可以设置为租户 CPU 数。命令示例如下：

```
obclient [test]> INSERT /*+ parallel(22) enable_parallel_dml */ INTO customer2 SEL
ECT * FROM customer;
```

输出如下，可以看到开启 PDML 后，相同的表插入 1500 万行数据，OceanBase 数据库的耗时缩短为 17.3 秒左右。

```
Query OK, 15000000 rows affected (17.319 sec)
Records: 15000000  Duplicates: 0  Warnings: 0
```

对比两种情况的输出，PDML 特性带来的性能提升大约为 6.3 倍。这一特性可以在需要批量数据处理的场景中提供帮助。

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，这使得数据在磁盘中能够以更紧凑的方式存储。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，使数据存储在 OceanBase 数据库中可以获得非常好的压缩比，且这种数据压缩并未带来查询和写入性能的下降。

可通过向 OceanBase 数据中导入大量外部数据观察数据压缩比。数据准备方面，我们可以使用 3.5 进行TPC-H测试 中用到的 TPC-H 测试数据。完成数据导入后，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。

1. 主动触发合并

   使用 root 用户登录到 OceanBase 数据库的 sys 租户，执行如下命令。

   ```
   obclient [oceanbase]> ALTER SYSTEM major freeze tenant=mysql_tenant;
   ```

   `mysql_tenant` 为待进行数据压缩测试的租户，需根据实际租户名进行替换。

2. 查询合并是否完成

```
obclient [oceanbase]> SELECT * FROM oceanbase.CDB_OB_MAJOR_COMPACTION;
```

输出如下，当看到 `STATUS` 列为 `IDLE` 时，表示合并完成。

```
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+------------------------
--+--------+---------+------------+------+
| TENANT_ID | FROZEN_SCN          | FROZEN_TIME                | GLOBAL_BROADCAST_
SCN | LAST_SCN            | LAST_FINISH_TIME           | START_TIM
E                   | STATUS | IS_ERROR | IS_SUSPENDED | INFO |
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+------------------------
--+--------+---------+------------+------+
|         1 | 1709661601360541623 | 2024-03-06 02:00:01.360542 |  1709661601360541
623 | 1709661601360541623 | 2024-03-06 02:06:25.027267 | 2024-03-06 02:00:01.38279
4 | IDLE    | NO        | NO             |       |
|      1001 | 1709661602742784187 | 2024-03-06 02:00:02.742784 |  1709661602742784
187 | 1709661602742784187 | 2024-03-06 02:05:36.148110 | 2024-03-06 02:00:02.78097
8 | IDLE    | NO        | NO             |       |
|      1002 | 1709661600590790760 | 2024-03-06 02:00:00.590791 |  1709661600590790
760 | 1709661600590790760 | 2024-03-06 02:05:43.819029 | 2024-03-06 02:00:00.64104
4 | IDLE    | NO        | NO             |       |
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+------------------------
--+--------+---------+------------+------+
```

3. 查看数据存储占用情况

在 sys 租户下执行如下语句，查看导入至 OceanBase 数据库后的数据存储占用情况，比如这里我们查询 `lineitem` 表主副本的空间占用。

```
obclient [oceanbase]> select t1.table_name,
    round(sum(t2.data_size)/1024/1024/1024,2) as data_size_gb,
    round(sum(t2.required_size)/1024/1024/1024,2) as required_size_gb
    from dba_ob_tenants t,cdb_ob_table_locations t1,cdb_ob_tablet_replicas t2
    where t.tenant_id=t1.tenant_id
    and t1.svr_ip=t2.svr_ip
    and t1.tenant_id=t2.tenant_id
    and t1.ls_id=t2.ls_id
    and t1.tablet_id=t2.tablet_id
    and t1.role='leader'
    and t.tenant_name='perf'
    and t1.database_name='test'
    and t1.table_name='lineitem'
    group by t1.table_name
    order by 3 desc;
```

输出如下，`lineitem` 表数据导入前总数据量大小为 75G，压缩后的数据大小为 22.15G，实际需要的磁盘空间为 23.11G，压缩比约为 75/23.11=3.2。

说明

输出中 required_size_gb 值大于 data_size_gb，是因为数据存储以宏块（2MB）为最小单位。

```
+------------+--------------+------------------+
| table_name | data_size_gb | required_size_gb |
+------------+--------------+------------------+
| lineitem   |        22.15 |            23.11 |
+------------+--------------+------------------+
```

高可用

集群高可用容灾能力作为分布式架构数据库产品的亮点之一，RTO 指标是衡量该能力的重要依据。OceanBase 数据库 4.x 版本在 RTO 特性上已经取得了阶段性的成果，将 RTO 控制在了 8s 以内。

可通过 Sysbench 的 oltp_read_write.lua 测试模拟持续的业务请求，在压测过程中，通过 `kill -9` 等方式手动将一个 Leader（主副本）故障，观察 QPS 或 TPS 的持续跌零时间。

这里注意，我们需要使用 `--mysql-ignore-errors` 传参忽略 1062、2013、4265、5066、6002、6213、6224、6222、4746、4012、4009、4250、4009 和 4038 错误码，以防止客户端接收错误码后断连接，示例如下：

```
sysbench oltp_read_write.lua --mysql-host=xxx --mysql-port=xxx --mysql-user=root@mysql --mysql-db=sysbench --mysql-password=test --threads=400 --report-interval=1 --tables=10 --table_size=500000 --mysql-ignore-errors=1062,2013,4265,5066,6002,6213,6224,6222,4746,4012,4009,4250,4009,4038 --time=600 --db-ps-mode=disable run
```

输出结果如下，TPS 稳定值为 500 左右，21s～27s 发生了下跌，过程花了 7s 左右的时间，对应的就是 RTO < 8s 恢复了服务。

```
[ 13s ] thds: 400 tps: 190.00 qps: 3431.03 (r/w/o: 2700.03/380.00/351.00) lat (ms,99%): 1938.16 err/s: 0.00 reconn/s: 0.00
[ 14s ] thds: 400 tps: 452.00 qps: 8034.99 (r/w/o: 6234.99/885.00/915.00) lat (ms,99%): 2045.74 err/s: 0.00 reconn/s: 0.00
[ 15s ] thds: 400 tps: 464.00 qps: 8466.98 (r/w/o: 6591.98/945.00/930.00) lat (ms,
```

```
99%): 1304.21 err/s: 0.00 reconn/s: 0.00
[ 16s ] thds: 400 tps: 404.00 qps: 7723.98 (r/w/o: 6036.98/865.00/822.00) lat (ms,
99%): 1280.93 err/s: 0.00 reconn/s: 0.00
[ 17s ] thds: 400 tps: 515.99 qps: 8798.78 (r/w/o: 6818.83/971.98/1007.97) lat (ms
,99%): 1479.41 err/s: 0.00 reconn/s: 0.00
[ 18s ] thds: 400 tps: 546.01 qps: 9694.12 (r/w/o: 7528.10/1071.01/1095.01) lat (m
s,99%): 1149.76 err/s: 0.00 reconn/s: 0.00
[ 19s ] thds: 400 tps: 555.99 qps: 9973.84 (r/w/o: 7746.88/1109.98/1116.98) lat (m
s,99%): 861.95 err/s: 0.00 reconn/s: 0.00
[ 20s ] thds: 400 tps: 551.02 qps: 10089.34 (r/w/o: 7873.26/1121.04/1095.04) lat (
ms,99%): 846.57 err/s: 0.00 reconn/s: 0.00
[ 21s ] thds: 400 tps: 267.00 qps: 4857.99 (r/w/o: 3763.99/534.00/560.00) lat (ms,
99%): 831.46 err/s: 0.00 reconn/s: 0.00
[ 22s ] thds: 400 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,99%): 0.00 e
rr/s: 0.00 reconn/s: 0.00
[ 23s ] thds: 400 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,99%): 0.00 e
rr/s: 0.00 reconn/s: 0.00
[ 24s ] thds: 400 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,99%): 0.00 e
rr/s: 0.00 reconn/s: 0.00
[ 25s ] thds: 400 tps: 0.00 qps: 0.00 (r/w/o: 0.00/0.00/0.00) lat (ms,99%): 0.00 e
rr/s: 0.00 reconn/s: 0.00
[ 26s ] thds: 400 tps: 0.00 qps: 211.00 (r/w/o: 90.00/0.00/121.00) lat (ms,99%): 0
.00 err/s: 0.00 reconn/s: 178.00
[ 27s ] thds: 400 tps: 3.00 qps: 1765.86 (r/w/o: 1531.88/10.00/223.98) lat (ms,99%
): 6594.16 err/s: 0.00 reconn/s: 174.99
[ 28s ] thds: 400 tps: 449.03 qps: 10079.67 (r/w/o: 8246.55/935.06/898.06) lat (ms
,99%): 7895.16 err/s: 0.00 reconn/s: 2.00
[ 29s ] thds: 400 tps: 612.00 qps: 10188.00 (r/w/o: 7714.00/1268.00/1206.00) lat (
ms,99%): 816.63 err/s: 0.00 reconn/s: 0.00
[ 30s ] thds: 400 tps: 550.03 qps: 10166.58 (r/w/o: 7971.46/1084.06/1111.06) lat (
ms,99%): 831.46 err/s: 0.00 reconn/s: 0.00
[ 31s ] thds: 400 tps: 571.96 qps: 10246.31 (r/w/o: 7956.46/1139.92/1149.92) lat (
ms,99%): 831.46 err/s: 0.00 reconn/s: 0.00
```

第四章 OceanBase 数据库的迁移和同步

本章介绍 OceanBase 数据库的迁移和同步。

本章目录

4.1 OceanBase 数据库和 MySQL 兼容性介绍 188

4.2 迁移同步相关生态组件介绍 193

4.3 通过 OMS 进行数据迁移和同步 199

4.4 通过 oblogproxy 进行增量日志代理服务 210

4.5 使用导数工具进行数据迁移 213

4.6 使用 SQL 命令进行数据迁移 220

4.7 通过其他工具进行数据的迁移同步 241

4.1 OceanBase 数据库和 MySQL 兼容性介绍

大家都知道，想要很好地实施异构数据库之间的迁移和同步，首先就需要了解异构数据库之间的兼容能力，如数据类型、字符集和字符序、索引等。如果出现不兼容的情况，必然会导致数据的迁移同步失败。 因此，在迁移之前，就需要掌握源端数据库的功能特性，以及在目标端是否兼容。如果不兼容，是否有其他比较好的替代方案等。 这里以 OceanBase 数据库与 MySQL 数据库兼容程度（OceanBase 数据库 4.2.1 版本和 MySQL 数据库 8.x 版本）为例，做一个整体的兼容性比对，其他类型数据库，可参考 MySQL 数据库的功能特性对比。

OceanBase 数据库的 MySQL 模式兼容 MySQL 数据库 5.7/8.0 的绝大部分功能和语法。本节主要从以下几方面介绍 OceanBase 数据库的 MySQL 模式与原生 MySQL 数据库的不同：数据类型、字符串、过程性语言、系统视图、字符集、字符序、索引、SQL_MODE、分区支持、备份恢复。

说明

本文仅基于 OceanBase 数据库 V4.2.1 进行介绍，其他版本与 MySQL 数据库的兼容性对比可参见官网《OceanBase 数据库》文档对应版本 OceanBase 简介/与 MySQL 兼容性对比 一文。

数据类型

除 `SERIAL` 类型外，OceanBase 数据库支持 MySQL 数据库剩余所有数据类型。

字符串类型长度对比

类型	MySQL 数据库 V8.0	OceanBase 数据库 V4.2.1
CHAR	255 字符	256 字符
VARCHAR	65535 字符，实际 16383 字符左右	262144 字符
BINARY	255 字节	256 字节
VARBINARY	65535 字节，实际 65532 字符左右	1048576 字节
TINYBLOB	255 字节	255 字节

BLOB	65535 字节	65536 字节
MEDIUMBLOB	16777215 字节	16777216 字节
LONGBLOB	4294967295 字节（4GB）	536870911 字节
TINYTEXT	255 字节	255 字节
TEXT	65535 字节	65536 字节
MEDIUMTEXT	16777215 字节	6777216 字节
LONGTEXT	4294967295 字节（4GB）	536870911 字节

过程性语言

OceanBase 数据库社区版兼容了大部分 MySQL 数据库的 PL 功能。有关 PL 功能的详细信息，请参见官网《OceanBase 数据库》文档中 参考指南/SQL 参考/PL 参考。

OceanBase 数据库主要支持的 PL 功能如下：

- 数据类型
- 存储过程
- 自定义函数
- 触发器
- 异常处理

此外，OceanBase 数据库还支持特有的 MySQL PL 系统包，包括 `DBMS_RESOURCE_MANAGER`、`DBMS_STATS`、`DBMS_UDR`、`DBMS_XPLAN` 和 `DBMS_WORKLOAD_REPOSITORY` 等。

系统视图

OceanBase 数据库实现了 `information_schema` 和 `mysql` 这两个内部数据库中的大部分视图，但是由于架构不同，OceanBase 数据库并不保证所有视图均能实现，也不保证视图中所有的列含义与 MySQL 数据库相同。

更多系统视图的字段说明信息，请参见官网《OceanBase 数据库》文档中 参考指南/系统视图/系统试图总览。

字符集 & 字符序

本节只介绍 OceanBase 数据库社区版支持的字符集和字符序，表格中未提到的字符集和字符序暂不支持。

注意

随着版本迭代，字符集和字符序的支持情况会有一定的变化。以下支持情况以 OceanBase 数据库 4.2.1 版本为例，其他版本建议通过 `show charset` 和 `show collation` 命令查看支持情况。

OceanBase 数据库社区版支持的字符集和字符序如下：

字符集	字符序	说明
utf8mb4	utf8mb4_general_ci	使用通用排序规则。
utf8mb4	utf8mb4_bin	使用二进制排序规则。
binary	binary	使用二进制排序规则。
gbk	gbk_chinese_ci	使用中文语言排序规则。
gbk	gbk_bin	使用二进制排序规则。
utf16	utf16_general_ci	使用通用排序规则。
utf16	utf16_bin	使用二进制排序规则。
gb18030	gb18030_chinese_ci	使用中文语言排序规则。
gb18030	gb18030_bin	使用二进制排序规则。
latin1	latin1_swedish_ci	使用瑞典语/芬兰语排序规则。
latin1	latin1_bin	latin1 使用二进制排序规则。
gb18030_2022	gb18030_2022_bin	使用二进制排序规则。
gb18030_2022	gb18030_2022_chinese_ci	使用拼音排序规则。不区分大小写。MySQL 模式下该字符集的默认字符序。
gb18030_2022	gb18030_2022_chinese_cs	使用拼音排序规则。区分大小写。
gb18030_2022	gb18030_2022_radical_ci	使用部首笔画排序规则。不区分大小写。

gb18030_2022	gb18030_2022_radical_cs	使用部首笔画排序规则。区分大小写。
gb18030_2022	gb18030_2022_stroke_ci	使用笔画排序规则。不区分大小写。
gb18030_2022	gb18030_2022_stroke_cs	使用笔画排序规则。区分大小写。

更多字符集和字符序的详细介绍，可参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/基本元素/字符集和字符序 章节。

索引

OceanBase 数据库暂不支持以下索引类型，其他 MySQL 数据库支持的索引类型均支持。

索引类型	索引数据结构	MySQL 数据库	OceanBase 数据库
索引扩展	B-tree	支持	不支持
降序索引	B-tree	支持	不支持
全文索引	B-Tree	支持	不支持
HASH索引	B-Tree	支持	不支持
LOCK选项	/	支持	不支持
索引合并	B-Tree	支持	不支持

更多索引的详细介绍，可参见官网《OceanBase 数据库》文档 参考指南/系统原理/数据库对象/MySQL 模式/索引 章节。

SQL_MODE

目前在 OceanBase 数据库 4.2.1 版本中，所有 MySQL 数据库支持的 SQL_MODE 均已支持。详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Global 系统变量/sql_mode。

分区支持

OceanBase 数据库与 MySQL 数据库对分区的支持差异如下:

- OceanBase 数据库支持一级分区，模板化和非模板化二级分区；MySQL 数据库不支持非模板化二级分区。
- OceanBase 数据库的二级分区支持 Hash、Key、Range、Range Columns、List 和 List Columns 分区；MySQL 数据库的二级分区仅支持 Hash 分区和 Key 分区。

更多分区的详细介绍，可参见官网《OceanBase 数据库》文档 参考指南/数据库对象管理/MySQL 模式/创建和管理分区 章节。

备份恢复

OceanBase 数据库兼容了部分 MySQL 数据库的备份恢复特性，如:

- 支持全量备份和增量备份。
- 支持热备份。
- 支持表级恢复。

目前暂不支持如下特性:

- 不支持集群级别的备份恢复。
- 不支持冷备份。
- 不支持备份数据的有效性验证。
- 不支持租户内部部分数据库的备份恢复。

4.2 迁移同步相关生态组件介绍

在日常的数据库运维过程中，经常会遇到数据库替换、机房搬迁、业务测试、数据库升级等操作，需要对数据进行迁移和同步。虽然集群内部、表与表之间的数据归档、磁盘水位均衡、资源单元（UNIT）搬迁等操作在 OceanBase 数据库中可以通过简单命令快速发起，但是涉及异构数据源和集群间的数据同步等功能时就需要借助外部工具。

本文主要介绍几种常用的数据迁移方法及工具。

迁移方案具体支持情况如下：

迁移方案	结构迁移	全量数据迁移	增量数据迁移	数据校验	支持的数据源
OMS	支持	支持	支持	支持	支持如下数据源： • OceanBas • MySQL • MariaDB • PostgreSQL • GreenPlum • HBase • TiDB • Kafka • RocketMQ
oblogproxy	支持	不支持	支持	不支持	支持如下数据源： • OceanBase • MySQL Binlog 生态工具 • CDC 生态工具 • oblogclient
obloader & obdumper	支持	支持	不支持	不支持	OceanBase
SQL 命令迁移	支持	支持	不支持	不支持	支持如下数据源： • 主流数据库 • SQL 文本 • CSV 文件等
DataX	不支持	支持	不支持	不支持	支持的数据源较多，具体范围查看 官方

					介绍
Canal	支持	支持	支持	不支持	支持的数据源较多，具体范围查看 官方介绍
Flink cdc	支持	支持	支持	不支持	支持的数据源较多，具体范围查看 官方介绍
SeaTunnel	支持	支持	支持	不支持	支持的数据源较多，具体范围查看 官方介绍

OMS

OceanBase 迁移服务（OceanBase Migration Service，OMS）是 OceanBase 提供的一种支持同构或异构数据源与 OceanBase 数据库之间进行数据交互的服务，具备在线迁移存量数据和实时同步增量数据的能力。

OMS 社区版提供可视化的集中管控平台，用户只需要进行简单的配置即可实时迁移数据。OMS 社区版旨在帮助用户低风险、低成本、高效率地实现同构或异构数据库向 OceanBase 数据库进行实时数据迁移和数据同步。

OMS 的相关内容详见官网 OceanBase 迁移服务 文档。

oblogproxy

oblogproxy 是 OceanBase 的增量日志代理服务，它可以与 OceanBase 数据库建立连接并进行增量日志读取，为下游服务提供了变更数据捕获（CDC）的能力。

oblogproxy 有如下两种模式。

- Binlog 模式为 OceanBase 数据库兼容 MySQL binlog 而推出，支持现有的 MySQL binlog 增量解析工具实时同步 OceanBase 数据库，使 MySQL binlog 增量解析工具可以平滑切换到 OceanBase 数据库。
- CDC 模式用于解决数据同步，CDC 模式下 oblogproxy 可以订阅 OceanBase 数据库中的数据变更，并将这些数据变更实时同步至下游服务，实现数据的实时或准实时复制和同步。

oblogproxy 的相关内容详见官网 OceanBase 日志代理服务 文档。

obloader/obdumper

obloader 和 obdumper 是一款使用 Java 语言开发的客户端工具，目前该工具仅适用于 OceanBase 数据库。用户可以使用 obloader 将存储介质中的数据库对象的定义文件和表数据文件导入到 OceanBase 数据库中，相反也可以使用 obdumper 导出到存储介质中。

通常我们推荐 obloader 与 obdumper 搭配使用。如果用户希望借助于 obloader 完成数据迁移工作，它也兼容 mysqldump，Mydumper 等客户端工具导出的 CSV 格式的文件。用户希望借助于 obdumper 进行逻辑备份，可以直接将该工具集成到数据库运维系统中。obloader 专门优化了数据的导入性能，内置多种数据预处理函数，自动容错保证数据导入的稳定性，以及提供较为丰富的监控信息，以便于用户实时观测到数据文件导入的性能和进度。

obloader/obdumper 的相关内容详见官网 OceanBase 导数工具 文档。

SQL 命令迁移介绍

通过 SQL 命令，可以将数据写入到外部文件或者从外部文件写入到数据库中，是进行数据的迁移和导入导出时比较常见和简便的方式。同时也支持通过 SQL 命令将数据从一张表中迁移到另外一张表，并做相应的数据加工。

支持的命令包括 select into outfile、load data、insert into 等；同时也支持旁路导入，加速数据的导入。详细使用可参见 使用 SQL 命令进行数据迁移。

其他迁移工具

Canal 介绍

Canal 主要用途是基于 MySQL 数据库增量日志解析，提供增量数据订阅和消费。

早期阿里巴巴因为杭州和美国双机房部署，存在跨机房同步的业务需求，实现方式主要是基于业务 trigger 获取增量变更。从 2010 年开始，业务逐步尝试数据库日志解析获取增量变更进行同步，由此衍生出了大量的数据库增量订阅和消费业务。当前的 canal 支持源端 MySQL 版本包括 5.1.x、5.5.x、5.6.x、5.7.x 和 8.0.x。

Canal 工作原理如下：

1. Canal 模拟 MySQL slave 的交互协议，伪装自己为 MySQL slave，向 MySQL master 发送 dump 协议
2. MySQL master 收到 dump 请求，开始推送 binary log 给 slave（即 canal）
3. Canal 解析 binary log 对象（原始为 byte 流）

详细介绍请参见 Canal 介绍。

Flink CDC 介绍

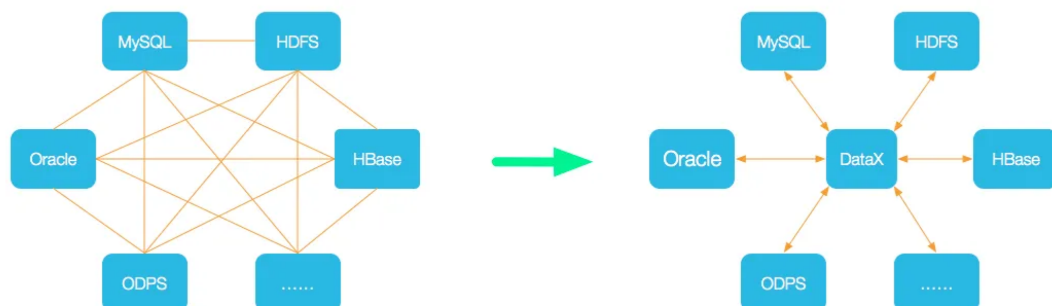
Flink CDC（Change Data Capture）是一种基于 Flink 的流式数据处理技术，用于捕获数据源的变化，并将变化发送到下游系统。Flink CDC 可以将数据源的变化转换为流式数据，并实时地将数据流发送到下游系统，以便下游系统及时处理这些变化。

Flink CDC 的工作原理是通过监听数据源的变化，将变化实时地抽取到 Flink 的数据流中，并将数据流发送到下游系统。Flink CDC 支持多种数据源，包括关系型数据库、NoSQL 数据库、消息队列等。

详细介绍请参见 Flink CDC 介绍。

DataX 介绍

DataX 是一个异构数据源离线同步工具，致力于实现包括关系型数据库（MySQL、Oracle、OceanBase 数据库等）、HDFS、Hive、ODPS、HBase、FTP 等各种异构数据源之间稳定高效的数据同步功能。



为了解决异构数据源同步问题，DataX 将复杂的网状同步链路变成了星型数据链路，DataX 作为中间传输载体负责连接各种数据源。当需要接入一个新的数据源时，只需要将此数据源对接到 DataX，便能跟已有的数据源做到无缝数据同步。

详细介绍请参见 DataX介绍。

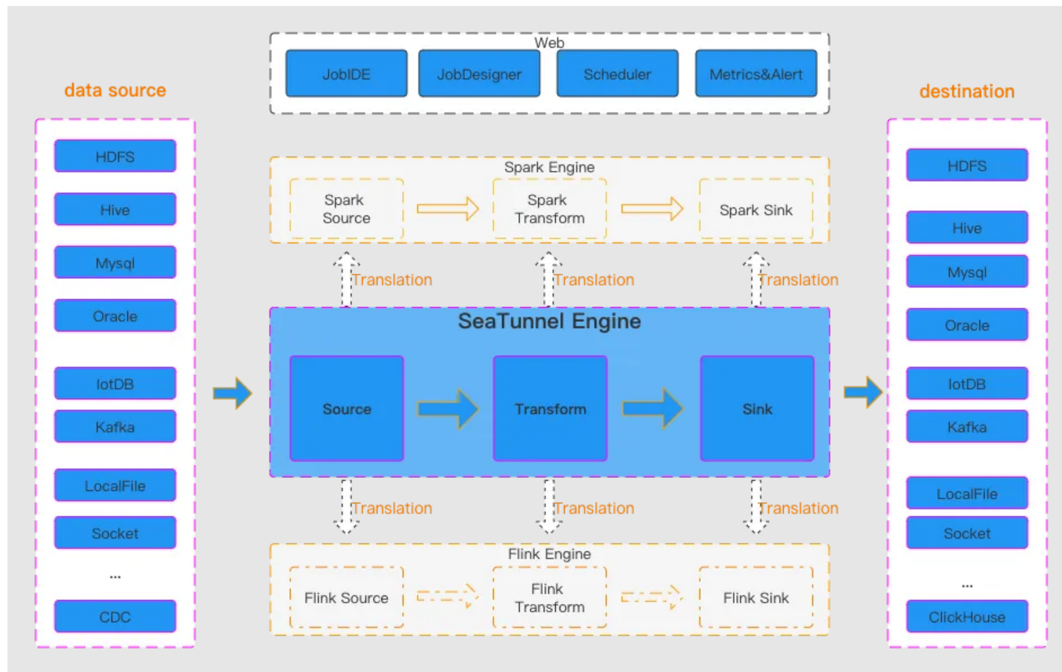
SeaTunnel 介绍

SeaTunnel 是一个非常易于使用的、超高性能的分布式数据集成平台，支持海量数据的实时同步。每天可稳定高效同步数百亿数据，已被近百家企业投入生产使用。

SeaTunnel 专注于数据集成和数据同步，旨在解决数据集成领域的常见问题：

- 数据源多样：常用数据源有数百种，版本不兼容。随着新技术的出现，更多的数据源不断出现。用户很难找到一个能够全面、快速支持这些数据源的工具。
- 同步场景复杂：数据同步需要支持离线全量同步、离线增量同步、CDC、实时同步、全库同步等多种同步场景。
- 资源需求高：现有的数据集成和数据同步工具往往需要大量的计算资源或 JDBC 连接资源来完成海量小表的实时同步，这增加了企业的负担。
- 缺乏质量和监控：数据集成和同步过程经常会出现数据丢失或重复的情况，同步过程缺乏监控，无法直观了解任务过程中数据的真实情况。
- 技术栈复杂：企业使用的技术组件不同，用户需要针对不同组件开发相应的同步程序来完成数据集成。
- 管理维护难度：受限于底层技术组件（Flink/Spark）不同，离线同步和实时同步往往需要分开开发和管理，增加了管理维护难度。

SeaTunnel 工作流程图如下所示：



详细介绍请参见 SeaTunnel 介绍。

4.3 通过 OMS 进行数据迁移和同步

本文将详细展开介绍如何部署和使用 OceanBase 迁移服务（OceanBase Migration Service，OMS）。

> **说明**
>
> 本教程中所贴官网文档链接版本均为编写教程时的最新版本，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

OMS 使用限制

本节为您介绍 OMS 社区版的使用限制。

- 数据迁移和数据同步功能中，OceanBase 数据库社区版和其它数据终端的适用版本如下。

分类	数据迁移	数据同步
OceanBase 数据库社区版	V3.1.0-CE、V3.1.1-CE、V3.1.2-CE、V3.1.3-CE、V3.1.4-CE、3.1.5-CE、V4.0.0-CE、V4.1.0-CE、V4.2.0-CE、V4.2.1-CE、V4.2.2-CE	V3.1.0-CE、V3.1.1-CE、V3.1.2-CE、V3.1.3-CE、V3.1.4-CE、3.1.5-CE、V4.0.0-CE、V4.1.0-CE、V4.2.0-CE、V4.2.1-CE、V4.2.2-CE
其它数据终端	• MySQL：V5.5、V5.6、V5.7、V8.0 • MariaDB：V10.2 • TiDB：V4.x、V5.x、V6.x、V7.x • PostgreSQL：V10.x、V12.x • GreenPlum：4 • HBase：1.2.0-cdh5.15.2	• Kafka：V0.9、V1.0、V2.x • RocketMQ：V4.7.1

- MySQL：V5.5、V5.6、V5.7、V8.0
- MariaDB：V10.2
- TiDB：V4.x、V5.x、V6.x、V7.x
- PostgreSQL：V10.x、V12.x
- GreenPlum：4
- HBase：1.2.0-cdh5.15.2
- Kafka：V0.9、V1.0、V2.x

- RocketMQ：V4.7.1
- OceanBase 云平台的适用版本为 V3.1.1-CE、V3.3.0-CE、V4.0.0-CE、V4.0.3-CE、V4.2.0-CE 和 V4.2.1-CE。
- 目前 OMS 社区版仅支持 X86 架构。

支持的迁移类型

迁移类型	描述
结构迁移	负责迁移源库中的数据对象定义（表、索引、约束、注释和视图等）至目标端数据库中，会自动过滤临时表。 当源端数据库非 OceanBase 数据库时，会依据目标 OceanBase 数据库租户类型的语法定义标准进行数据类型和 SQL 语法的自动转换和拼装，然后复制至目标库中。
全量迁移	迁移源库表的存量数据至目标端数据库对应的表中。您可以在 **全量迁移** 页面，查看 **表对象**、**表索引** 和 **全量迁移性能**。只有表对象和表索引均迁移完成，全量迁移的状态才会显示已完成。 在 **表索引** 页面，单击目标表对象后的 **查看创建语法**，即可查看索引创建语法。 全量迁移加上增量同步，可以确保目标端数据库与源端数据库的最终一致性。如果全量迁移过程中有失败的对象，会为您展示具体的失败原因。
增量同步	增量同步任务开始后，会同步源库发生变化的数据（新增、修改或删除）至目标端数据库对应的表中。 当源库不断有业务写入时，OMS 社区版会在全量数据迁移启动前，启动增量拉取模块，以拉取源实例中的增量更新数据，对其进行解析、封装，并存储至 OMS 社区版中。 当全量数据迁移完成后，OMS 社区版会启动增量数据回放模块，从增量数据拉取模块中获取增量数据。增量数据经过过滤、映射和转换后，再同步至目标实例中。
全量校验	在全量数据迁移完成，增量数据迁移至目标端并与源端基本追平后，OMS 社区版会自动发起一轮针对源库配置的数据表和目标表的全量数据校验任务。 增量数据同步过程中，您也可以发起自定义的数据校检。对于校验结果不一致的数据，您可以重新校验表中的全部数据或仅重新校验表中不一致的数据。
正向切换	正向切换（传统意义上的系统割接流程的抽象化、标准化）不会操作业务应用连接的切换，是 OMS 社区版的数据迁移项目配合应用切换需要执行的任务流。您需要保证在应用连接切换至目标端前完成正向切换的全部流程。

	正向切换是数据迁移必不可少的一个流程，通过正向切换，OMS 社区版可以确保完成了数据正向迁移的相关工作，并且您可以根据业务需求启动反向增量组件。正向切换主要涉及的工作如下: 1. 您需要自行确认已完成数据迁移，并等待正向同步延迟被追平。 2. OMS 社区版将会自动补充结构迁移阶段忽略的检查类约束、外键约束等对象。 3. OMS 社区版将会自动删除迁移依赖的附加隐藏列及唯一索引。 该操作仅在 OceanBase 数据库社区版之间的数据迁移项目存在，详情请参见官网《OceanBase 迁移服务》文档 数据迁移/功能介绍/数据迁移服务隐藏列机制说明。 4. 您需要自行补充迁移源端触发器、函数、存储过程等其它 OMS 社区版不支持的数据库对象至目标端。 5. 您需要自行禁用源端的触发器和外键约束（仅数据迁移项目存在反向增量时需要）。
反向增量	迁移完成后，针对于业务割接场景，可以引导用户在业务数据库完成切换前在 OMS 社区版上启动目标库至源库（即反向）的增量同步项目，实时回流业务切换后在目标端数据库产生的变更数据至源业务数据库。

部署 OMS

OMS 的部署可分为单节点部署和高可用部署，高可用部署又划分单地域多节点部署和多地域多节点部署。如想通过 OMS 实现容灾双活，则需要采用多地域多节点部署的模式。

单节点部署

OMS 社区版的单个节点可以提供全部 OMS 社区版能力。在部署 OMS 社区版时，如果您无需使用高可用环境，可以选择单节点部署。详细部署操作可参见官网《OceanBase 迁移服务》文档 部署 OMS 社区版/单节点部署。

单地域多节点部署

OMS 社区版支持通过多节点部署来实现高可用。在高可用架构中，OMS 社区版的每个节点均具备完整功能，后台任务框架基于数据库实现了分布式调度。当某个 OMS 节点不可用时，框架会自动将任务调度到可用的 OMS 社区版节点。

同时，OMS 社区版支持在单个地域部署多节点。详细部署操作可参见官网《OceanBase 迁移服务》文档 部署 OMS 社区版/单地域多节点部署。

说明

- 高可用特性不能自动恢复 Full-Import/Full-Verification 组件。
- 如果您使用的是生产环境，建议使用多节点部署。

多地域多节点部署

OMS 社区版支持多地域多节点部署，您可以在多个地域部署 OMS 社区版。详细部署操作可参见官网《OceanBase 迁移服务》文档 部署 OMS 社区版/多地域多节点部署。

检查部署状态

在部署机上安装 OMS 社区版之后，还需要进入 Docker 查看各个服务的运行状态，判断 OMS 社区版是否已在部署机上正常运行。详细检查操作可参见官网《OceanBase 迁移服务》文档 部署 OMS 社区版/检查部署状态

注意事项

- OMS 社区版 V4.1.1-CE 仅支持直接部署，不支持低版本升级。
- 如果您需要 OMS 社区版收集和展示历史监控数据，请部署 InfluxDB 时序数据库。详细操作可参见官网《OceanBase 迁移服务》文档 部署 OMS 社区版/（可选）部署时序数据库。
- OMS 社区版默认关闭高可用（HA）功能。如果您需要开启，可以通过在 OMS 管理控制台修改 `ha.config` 参数中 `enable` 的取值为 `true` 的方式启动 HA。详细操作可参见官网《OceanBase 迁移服务》文档 系统管理/系统参数/修改高可用功能的配置。

使用 OMS 进行数据迁移

数据迁移概述

OMS 社区版支持数据迁移功能，您可以通过该功能实现 OceanBase-CE、MySQL、TiDB 和 PostgreSQL 等多种类型的数据源与 OceanBase 数据库社区版进行实时数据传输，以及

OceanBase 数据库社区版 MySQL 租户间的数据迁移。

详细介绍可参见官网《OceanBase 数据库》文档 数据迁移/数据迁移概述。

数据迁移流程

OMS 社区版提供数据迁移功能，帮助您实现同构或异构数据源之间的数据迁移。适用于数据库升级、跨实例数据迁移、数据库拆分、扩容等业务场景。

OMS 社区版部署服务器需要同时保持源实例和目标实例的网络连通。您只需要配置好源库和目标库，并选择需要迁移的表，即可启动数据迁移项目。

您可以参考以下流程进行迁移前的准备工作、以及创建、管理数据迁移项目。

1. 完成准备工作

   使用 OMS 社区版迁移数据前，您需要对源或目标数据库进行创建迁移用户、为用户授权等准备工作。详细操作可参见官网《OceanBase 迁移服务》文档 新建及管理数据源/创建数据库用户。

2. 新建数据源

   在 OMS 社区版控制台，分别新建源端和目标端的数据源。详细操作可参见官网《OceanBase 迁移服务》文档 新建及管理数据源/新建数据源 章节。

3. 新建数据迁移项目

   根据业务需求，在数据迁移项目中选择源端、目标端、迁移类型和迁移对象。下文 创建数据迁移项目 中列举了常见的数据迁移项目搭建过程，可供参考。

4. 查看数据迁移项目的状态

   数据迁移项目启动后，会根据选择的迁移类型依次执行。详细操作可参见官网《OceanBase 迁移服务》文档 数据迁移/管理数据迁移项目/查看数据迁移项目的详情。

5. （可选）停止并释放数据迁移项目

   确认数据迁移项目成功，并不再需要同步源库和目标库的数据后，您可以清理当前的数据迁移项目。详细操作可参见官网《OceanBase 迁移服务》文档 数据迁移/管理数据迁移项目/释放和删除数据迁移项目。

创建数据迁移项目

上文 OMS 使用限制 中已经介绍了 OMS 适配的数据源以及使用限制，本节将列举几个迁移项目的创建，来指导大家如何使用 OMS 在线迁移数据功能。

- 迁移 MySQL 数据库的数据至 OceanBase 社区版
- 迁移 OceanBase 社区版的数据至 MySQL 数据库
- 迁移 HBase 数据库的数据至 OBKV
- OceanBase 社区版之间的数据迁移
- 容灾双活场景的数据迁移
- 迁移 TiDB 数据库的数据至 OceanBase 社区版
- 迁移 PostgreSQL 数据库的数据至 OceanBase 社区版

> **说明**
>
> 数据迁移项目创建完成之后，可以对已经创建的项目进行管理。详细操作可参见官网《OceanBase 迁移服务》文档 数据迁移/管理数据迁移项目 章节。

功能介绍

在创建数据迁移项目过程中，有很多功能选项可以进行配置，如 DML 过滤、DDL 同步、迁移对象匹配规则、通配符规则等，这些功能的详细配置操作可参见官网《OceanBase 迁移服务》文档 数据迁移/功能介绍 章节。本节重点介绍旁路导入和同步 DDL 功能。

旁路导入

OMS 社区版自 V4.2.2 起已经加入了旁路导入的功能。旁路导入是指 OceanBase 数据库支持向 data 文件中直接写入数据。旁路导入可以绕过 SQL 层的接口，在 data 文件中直接分配空间并插入数据，从而提高数据导入的效率。

可在创建数据迁移项目时，勾选 **迁移选项** 页面的 **Direct Load** 使用旁路导入功能。选择 **Direct Load** 后数据就会以旁路导入的方式写入，经测试旁路导入方式比普通导入方式效率提升 3-6 倍。

目前仅支持在下述类型的数据迁移项目创建时使用旁路导入的方式写入数据。

- 迁移 MySQL 数据库的数据至 OceanBase 社区版
- 迁移 HBase 数据库的数据至 OBKV
- OceanBase 社区版之间的数据迁移
- 容灾双活场景的数据迁移
- 迁移 TiDB 数据库的数据至 OceanBase 社区版

OMS 社区版中旁路导入功能的详细使用方法和使用限制可参见官网《OceanBase 迁移服务》文档 数据迁移/功能介绍/旁路导入介绍。

同步 DDL

在创建的迁移项目中，支持设置同步 DDL。只有设置同步 DDL 后，Create/Alter/Drop/Truncate Table 等 DDL 语句才会被同步至目标端。该功能常用于长期的数据迁移、数据同步项目，可以显著降低项目运维的成本。但是也有一定的使用限制，具体如下：

- 除支持的同步 DDL 类型外，其它类型均不支持。具体 DDL 范围请参见官网《OceanBase 迁移服务》文档 数据迁移/同步 DDL 的支持范围和使用限制 章节。
- 如果需要同步的表涉及其它类型的 DDL，数据迁移项目可能会中断并造成数据问题，且无法

恢复。

- 请勿在结构迁移和全量迁移阶段执行库或者表结构变更的 DDL 操作，否则可能导致数据迁移项目中断。

可在创建数据迁移项目时，勾选 **选择迁移类型** 页面的 **增量同步** > **同步 DDL**，使用同步 DDL 功能。在 **同步 DDL** 区域可选择 **源端执行** 或 **目标端执行**。

- **源端执行**：表示 DDL 操作仅在源端执行，并增量同步至目标端。
- **目标端执行**：表示 DDL 操作仅在目标端执行，并增量同步至源端。

同步 DDL 的详细介绍可参见官网《OceanBase 迁移服务》文档 数据迁移/功能介绍/同步 DDL。

使用 OMS 进行数据同步

数据同步概述

OMS 社区版支持 OceanBase 数据库社区版和 Kafka、RocketMQ 之间进行实时数据同步。OMS 社区版可以应用于实时数据仓库搭建、数据查询和报表分流等业务场景。

支持的项目类型

从源端至目标端的数据同步项目主要支持增量数据同步场景，即实时写入源端发生的 DML 变更至目标端。OMS 社区版支持同步的粒度最小为表级别，最大为租户级别。

OMS 社区版为您提供同步数据的功能，支持的项目类型及具体情况如下：

项目类型	结构同步	全量同步	增量同步	数据校验
OceanBase 数据库 > Kafka	支持	支持	支持	暂不支持
OceanBase 数据库 > RocketMQ	不涉及	支持	支持	暂不支持

数据同步流程

OMS 社区版提供数据同步功能，帮助您实现数据源之间的数据实时同步。适用于数据异地多

活、数据异地灾备、数据聚合和实时数据仓库等多种业务场景。

您可以参考以下流程进行同步前的准备工作，并创建和管理数据同步项目。

1. 完成准备工作

   使用 OMS 社区版同步数据前，请先在源端和目标端数据库中，为数据同步项目创建专用的数据库用户，并为该用户赋予相应的权限。详细操作可参见官网《OceanBase 迁移服务》文档 新建及管理数据源/创建数据库用户。

2. 新建数据源

   在 OMS 社区版控制台，分别新建源端和目标端的数据源。详细操作可参见官网《OceanBase 迁移服务》文档 新建及管理数据源/新建数据源 章节。

3. 新建数据同步项目

   根据业务需求，在数据同步项目中选择源端、目标端、同步类型和同步对象。下文 创建数据同步项目 中列举了常见的数据同步项目搭建过程，可供参考。

4. 查看数据同步项目的状态

   数据同步项目启动后，会根据选择的同步类型依次执行。详细操作可参见官网《OceanBase 迁移服务》文档 数据同步/管理数据同步项目/查看数据同步项目的详情。

5. （可选）停止并释放数据同步项目

确认数据同步项目成功，并不再需要同步源库和目标库的数据后，您可以清理当前的数据同步项目。详细操作可参见官网《OceanBase 迁移服务》文档 数据同步/管理数据同步项目/释放和删除数据同步项目。

创建数据同步项目

目前 OMS 社区版支持 OceanBase 数据库社区版和 Kafka、RocketMQ 之间进行实时数据同步。

- Kafka 是目前广泛应用的高性能分布式流计算平台，OMS 社区版支持 OceanBase 数据库社区版与自建 Kafka 数据源之间的数据实时同步，扩展消息处理能力，广泛应用于实时数据仓库搭建、数据查询和报表分流等业务场景。同步项目创建的详细操作可参见官网《OceanBase 迁移服务》文档 数据同步/创建 OceanBase 社区版至 Kafka 的数据同步项目。
- 消息队列 RocketMQ 是阿里云基于 Apache RocketMQ 构建的低延迟、高并发、高可靠的分布式消息中间件。OMS 社区版的数据同步功能可以帮助您实现 OceanBase 数据库社区版的物理表和 RocketMQ 数据源之间的数据实时同步，扩展消息处理能力。同步项目创建的详细操作可参见官网《OceanBase 迁移服务》文档 数据同步/创建 OceanBase 社区版至 RocketMQ 的数据同步项目。

管理数据同步项目

数据同步项目创建完成之后，可以对已经创建的项目进行管理。详细操作可参见官网《OceanBase 迁移服务》文档 数据同步/管理数据同步项目 章节。

功能介绍

在创建数据同步项目过程中，有很多功能选项可以进行配置，如 DML 的过滤、DDL 同步等。这些功能的配置操作可详见官网《OceanBase 迁移服务》文档 数据同步/功能介绍 章节。

数据格式说明

OMS 社区版同步源端数据至 Kafka 和 RocketMQ 时，支持序列化方式控制数据同步至目标端的

消息格式。序列化方式包括 Default、Canal、Dataworks（支持 V2.0）、SharePlex、DefaultExtendColumnType、Debezium、DebeziumFlatten、DebeziumSmt 和 Maxwell（仅目标端为 Kafka 时支持）。

详细介绍可参见官网《OceanBase 迁移服务》文档 数据同步/功能介绍/数据格式说明。

说明

OMS 社区版除支持数据迁移同步外，还支持运维监控、系统管理等功能，详细介绍请参见官网 OceanBase 迁移服务 对应版本文档。

4.4 通过 oblogproxy 进行增量日志代理服务

oblogproxy 是 OceanBase 的增量日志代理服务，它可以与 OceanBase 数据库建立连接并进行增量日志读取，为下游服务提供了变更数据捕获（CDC）的能力。

> **说明**
>
> 本教程中所贴官网文档链接版本均为编写教程时的最新版本，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

oblogproxy 介绍

oblogproxy 有 2 种模式，分别是 Binlog 模式和 CDC 模式。

- Binlog 模式为 OceanBase 数据库兼容 MySQL binlog 而推出，支持现有的 MySQL binlog 增量解析工具实时同步 OceanBase 数据库，使 MySQL binlog 增量解析工具可以平滑切换到 OceanBase 数据库。详细介绍可参见官网《OceanBase 日志代理服务》文档 Binlog 模式 章节。
- CDC 模式用于解决数据同步，CDC 模式下 oblogproxy 可以订阅 OceanBase 数据库中的数据变更，并将这些数据变更实时同步至下游服务，实现数据的实时或准实时复制和同步。详细介绍可参见官网《OceanBase 日志代理服务》文档 CDC 模式 章节。

oblogproxy 安装

您可以通过下载安装包和使用源码两种方式安装 oblogproxy。本文以下载安装包安装为例介绍如何安装 oblogproxy，详细的安装方法介绍可参见官网《OceanBase 日志代理服务》文档 oblogproxy 安装。

> **注意**
>
> oblogproxy 会单独占用资源，建议和 OceanBase 数据库分开部署，避免影响数据库性能。

1. 从官网 OceanBase 软件下载中心 或 GitHub 仓库 下载 oblogproxy 安装包。

2. 执行如下命令安装 oblogproxy。

```
rpm -i oblogproxy-{version}.{arch}.rpm
```

默认安装目录为 `/usr/local/oblogproxy`。

oblogproxy 的配置文件默认放在 `conf/conf.json` 中。在不完全了解参数用途情况下，我们不建议您进行修改。 详细的配置文件内容介绍可参见官网《OceanBase 日志代理服务》文档 配置文件。

通过 Binlog 模式同步

Binlog 模式为兼容 MySQL binlog 而诞生，支持现有的 MySQL binlog 生态工具同步 OceanBase 数据库，使现有的 MySQL binlog 生态工具可以平滑切换至 OceanBase 数据库。

Binlog 模式与 MySQL 5.7 版本保持一致，具备相同的功能：

- 支持 Position 模式订阅。在 Position 模式下，订阅方基于指定的 binlog 文件名和偏移量来定位和读取变更日志，也称为 binlog position，在这种模式下，订阅是基于物理位置。
- 支持 GTID 模式订阅，GTID（Global Transaction ID，全局事务 ID）是一种用于唯一标识和追踪分布式环境中事务的机制。在 GTID 模式下，每个事务都有一个唯一的全局事务 ID，不受 binlog 文件的变化和偏移量的影响。在这种模式下，订阅是基于逻辑位置。

在使用 Binlog 模式之前建议先查看官网《OceanBase 日志代理服务》文档中如下内容，了解 Binlog 模式的基本原理、与 MySQL binlog 的兼容性对比。

- Binlog 模式/Binlog 模式介绍
- Binlog 模式/Binlog 模式与 MySQL binlog 兼容性对比

使用限制

- Binlog 模式不支持 OceanBase 数据库针对 enum 和 set 实现的拓展语义。例如，set 定义数支持超过 64 个、支持重复、enum 支持插入未定义数据（例如 `''`）等。
- Binlog 模式不支持 varchar(65536) 定义。
- Binlog 模式不支持 gis 类型。

使用 Binlog 模式的详细操作步骤可参见官网《OceanBase 日志迁移服务》文档 Binlog 模式/使用 Binlog 模式。

使用 CDC 模式同步

CDC 模式用于实时数据同步，oblogproxy 在 CDC 模式下可以订阅 OceanBase 数据库中的数据变更（需配合 oblogclient 来完成数据订阅），并将这些数据变更实时同步至下游服务，实现数据的实时或准实时复制和同步。CDC 模式使用时主要分为以下几个步骤：

1. oblogclient 启动后发送消息到 oblogproxy，并指定要订阅的 OceanBase 数据库的账户名密码、库表、增量链路位点等信息（oblogproxy 本身是无状态的，订阅的 OceanBase 数据库、库表等信息需要由 oblogclient 来传入）。
2. oblogproxy 收到来自客户端的请求后，首先对要已订阅的 OceanBase 数据库进行鉴权等操作，若不通过则返回给客户端失败原因；若通过，则启动 oblogreader 子进程，并在启动时将连接等相关信息传递至 oblogreader 子进程。
3. oblogreader 启动后会拉取并解析 clog，按照一定的数据格式发送至下游来完成数据订阅。

详细操作步骤可参见官网《OceanBase 日志代理服务》文档 CDC 模式/使用 CDC 模式。

4.5 使用导数工具进行数据迁移

说明

本文以导数工具 V4.2.8 为例进行介绍，所贴官网文档链接版本亦为 V4.2.8，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

导数工具介绍

OceanBase 导数工具包含导入工具 obloader 和导出工具 obdumper。

obloader

obloader 是一款使用 Java 语言开发的客户端工具，目前该工具仅适用于 OceanBase 数据库。用户可以使用该工具将存储介质中的数据库对象的定义文件和表数据文件导入到 OceanBase 数据库中。通常我们推荐 obloader 与 obdumper 搭配使用。如果用户希望借助于 obloader 完成数据迁移工作，它也兼容 mysqldump、Mydumper 等客户端工具导出的 CSV 格式的文件。

obloader 专门优化了数据的导入性能，内置多种数据预处理函数，自动容错保证数据导入的稳定性，并且提供较为丰富的监控信息，以便于用户实时观测到数据文件导入的性能和进度。

obdumper

obdumper 是一款使用 Java 语言开发的客户端工具，目前该工具仅适用于 OceanBase 数据库。用户可以使用该工具将 OceanBase 数据库中定义的对象和表数据以指定的文件格式导出到存储介质中。如果用户希望借助于 obdumper 进行逻辑备份，可以直接将该工具集成到数据库运维系统中（注：不支持增量备份）。

与 mysqldump 等客户端导出工具相比，obdumper 具备以下显著的优势：

- 快速的数据导出能力，设计了多种数据查询策略，大幅提升导出的性能。
- 丰富的数据交换能力，支持将表中数据以多种格式导出到多种存储介质。
- 强大的数据处理能力，导出前对数据进行压缩，加密，脱敏，预处理等。

关于导数工具更详细的介绍可参见官网《OceanBase 导数工具》文档 产品介绍。

环境准备和安装

运行导数工具前，请确认已拥有合适的运行环境和运行权限。

运行环境

环境	要求
系统版本	支持 Linux/macOS/Windows 7 及之后版本。
Java 环境	请安装 Oracle JDK 1.8.0_3xx，配置 `JAVA_HOME` 环境变量。
字符集	推荐使用 UTF-8 文件编码。
JVM 参数	请编辑 `bin/obloader` 和 `bin/obdumper` 脚本修改 JVM 内存参数，避免出现 JVM 内存不足。

说明

OpenJDK 1.8 部分小版本存在严重的 GC Bug，运行导入导出工具会发生 OOM 或者夯住，请用户安装 OpenJDK 1.8 最新的小版本。

运行权限

obloader 运行权限

使用 obloader 连接 OceanBase 数据库导入数据时，连接数据库的账号需要拥有 `CREATE`、`SELECT`、`INSERT`、`UPDATE` 等命令的执行权限。可通过 `SHOW GRANTS` 语句查看用户被授予的权限。示例语句：

```
obclient> SHOW GRANTS FOR user1;
```

输出如下：

```
+------------------------------------------+
| Grants for user1@%                       |
+------------------------------------------+
```

```
| GRANT CREATE ON *.* TO 'user1'               |
| GRANT SELECT ON `db1`.* TO 'user1'           |
| GRANT INSERT ON `db1`.* TO 'user1'           |
| GRANT UPDATE ON `db1`.* TO 'user1'           |
| GRANT SELECT ON `oceanbase`.* TO 'user1'     |
+-------------------------------------------+
```

说明

- 导入数据前，用户需要拥有 oceanbase 数据库的查询权限。
- 使用 obloader 连接 OceanBase 数据库 Oracle 模式租户导入数据库对象定义和数据时，建议使用数据库管理员（DBA）角色的账号连接数据库。

obdumper 运行权限

使用 obdumper 连接 OceanBase 数据库导出数据时，连接数据库的账号需要拥有 `CREATE`、`SELECT` 等命令的执行权限。可通过 `SHOW GRANTS` 语句查看用户被授予的权限。示例语句：

```
obclient> SHOW GRANTS FOR user1;
```

输出如下：

```
+-------------------------------------------+
| Grants for user1@%                        |
+-------------------------------------------+
| GRANT CREATE ON *.* TO 'user1'            |
| GRANT SELECT ON `db1`.* TO 'user1'        |
| GRANT SELECT ON `oceanbase`.* TO 'user1'  |
+-------------------------------------------+
```

说明

- 导出数据前，用户需要拥有 oceanbase 数据库的查询权限。
- 使用 obdumper 连接 OceanBase 数据库 Oracle 模式租户导出数据库对象定义时，建议使用数据库管理员（DBA）角色的账号连接数据库；使用 obdumper 连接 OceanBase 数据库 Oracle 模式租户导出数据时，对连接数据库的账号无角色要求。

其它说明

- 当 OceanBase 数据库为 V4.0.0 之前版本时，导数工具命令行中需要指定 `--sys-user` 和 `--sys-password` 选项。
- 命令行选项 `--sys-user` 和 `--sys-password` 必须指定为 sys 租户下拥有查询系统表和查询视图权限的用户。

安装

导数工具是一个软件压缩包，用户只需要在宿主机上解压即可运行该工具。

具体的步骤如下：

1. 在官网 OceanBase 软件中心 下载导数工具软件包。
2. 解压软件包后，进入运行脚本所在的目录。

```
# Windows
cd {ob-loader-dumper}/bin/windows

# Linux or macOS
cd {ob-loader-dumper}/bin
```

3. 执行下述语句查看命令行选项的帮助。

   - 运行 obloader

   ```
   # Windows
   call obloader.bat --help

   # Linux or macOS
   ./obloader --help
   ```

   - 运行 obdumper

   ```
   # Windows
   call obdumper.bat --help

   # Linux or macOS
   ./obdumper --help
   ```

对应命令行选项的详细介绍可参见官网《OceanBase 导数工具》文档 导入数据/obloader 命令行选项 和 导出数据/obdumper 命令行选项。

4. 运行 obloader

```
# Windows
call obloader.bat --help

# Linux or macOS
./obloader --help
```

5. 运行 obdumper

```
# Windows
call obdumper.bat --help

# Linux or macOS
./obdumper --help
```

导出数据

obdumper 导出，分为导出 DDL 表结构和表数据。表数据的导出支持多种格式，包括 CSV、SQL、CUT 等，同时也支持将数据文件导出到 Amazon S3 以及 Aliyun OSS 上。对于不想导出的列，可以进行过滤，也可通过控制文件对导出的数据进行预处理等。

- 导出示例，详情请参见官网《OceanBase 导数工具》文档 导出数据/快速入门 中 **步骤 5：导出数据**。
- obdumper 通过命令行选项指定导出所需要的信息，选项的详细介绍请参见官网《OceanBase 导数工具》文档 导出数据/命令行选项。
- 通过预定义的控制文件，可以对导出的数据进行预处理。在定义控制文件时，用户可以为每一个列配置对应的预处理函数，也可使用条件表达式进行简单的逻辑运算和算术运算以实现更复杂的数据处理能力。具体定义和配置请参见官网《OceanBase 导数工具》文档 导出数据/数据处理 章节。
- obdumper 的性能可从命令行选项、虚拟机内存和数据库内核等三个方面进行调优，详情请参见官网《OceanBase 导数工具》文档 导出数据/性能调优。

- obdumper 导出过程中遇到的常见问题汇总请参见官网《OceanBase 导数工具》文档 导出数据/常见问题。

导入数据

obloader 导入，即将表结构 DDL 文件或者数据文件，按照相应的格式导入到数据库中。支持导入的数据文件格式包括 CSV、SQL、POS、CUT，同时也支持从 Amazon S3、Aliyun OSS、Apache Hadoop 等导入数据到 OceanBase数据库。对于不想导入的列，可以进行过滤，也可通过控制文件对导出的数据进行预处理等。

- 导入示例，详情请参见官网《OceanBase 导数工具》文档 导入数据/快速入门 中 **步骤 6：导入数据**。
- obloader 通过命令行选项指定导入所需要的信息，选项的详细介绍请参见官网《OceanBase 导数工具》文档 导入数据/命令行选项。
- 通过预定义的控制文件，可以对导入的数据进行预处理。在定义控制文件时，用户可以为每一个列配置对应的预处理函数，也可使用条件表达式进行简单的逻辑运算和算术运算以实现更复杂的数据处理能力。具体定义和配置请参见官网《OceanBase 导数工具》文档 导入数据/数据处理 章节。
- obloader 的性能可从命令行选项、虚拟机内存和数据库内核等三个方面进行调优，详情请参见官网《OceanBase 导数工具》文档 导入数据/性能调优。
- 如果导入的数据中存在 Bad Record 或者 Discard Record 类型错误时，用户可以使用 obloader V4.2.4 及之后的版本控制此类脏数据对进程结束状态。用户可以根据生成的报错文件，手动修复此类错误，详情请参见官网《OceanBase 导数工具》文档 导入数据/错误处理。
- obloader 导入过程中遇到的常见问题汇总请参见官网《OceanBase 导数工具》文档 导入数据/常见问题。

旁路导入

OceanBase 数据库支持通过旁路导入的方式向数据库插入数据，即 OceanBase 数据库支持向

data 文件中直接写入数据。旁路导入可以绕过 SQL 层的接口，在 data 文件中直接分配空间并插入数据，从而提高数据导入的效率。

旁路导入功能可以在下述场景中使用：

- 数据迁移与同步。对于数据迁移和同步，通常需要将大量各种格式的数据从不同的数据源向 OceanBase 数据库进行迁移，传统的 SQL 接口性能可能在时效性上无法得到满足。
- 传统 ETL。当数据在源端进行了抽取和转化之后，在装载到目标端时，通常需要在短时间内加载大量的数据。使用旁路导入技术，会提升数据导入的性能。
- 从文本文件或者其他数据源向 OceanBase 数据库加载数据，利用旁路导入技术，也能够提升加载数据效率。

导数工具旁路导入功能的详细介绍请参见官网《OceanBase 导数工具》文档 导入数据/旁路导入。

安全功能

导数工具默认在命令行中显式指定密码等敏感信息即可运行。为了加强信息的安全性，导数工具 V4.2.0 及之后版本支持对命令行中的敏感信息提供加密和解密的方法。详情请参见官网《OceanBase 导数工具》文档 安全功能。

问题自查

在使用 obdumper/obloader 过程中，有时会遇到一些问题，导数工具有整理相关文档帮助用户自助查看导数工具的运行状态，并进行问题分析。详情请参见官网《OceanBase 导数工具》文档 问题自查。

4.6 使用 SQL 命令进行数据迁移

使用 SQL 命令进行数据迁移，是比较常见和简便的方式。可使用 SELECT INTO OUTFILE 语句将数据导出到外部文件；也可使用 LOAD DATA 语法或者 source 的方式将数据从外部文件导入；表与表之间的数据迁移，也可以使用 INSERT INTO 或者 MERGE INTO 语法。本文就各种方式展开介绍。

> **说明**
>
> 本教程中所贴官网文档链接版本均为编写教程时的最新版本，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

SELECT INTO OUTFILE 导出

使用 SELECT INTO OUTFILE 语句导出数据是常用的一种数据导出方式。SELECT INTO OUTFILE 语句能够对需要导出的字段做出限制，这很好地满足了某些不需要导出主键字段的场景。配合 LOAD DATA INFILE 语句导入数据，是一种很便利的数据导入导出方式。

语法

```
SELECT column_list_option
INTO OUTFILE file_route_option
    [format_of_field_option]
    [start_and_end_option]
FROM table_name_list
[WHERE where_conditions]
[GROUP BY group_by_list [HAVING having_search_conditions]]
[ORDER BY order_expression_list]

column_list_option:
    column_name[,column_name]...

file_route_option:
    '/path/file'
    | 'oss://$PATH/$FILENAME/?host=$HOST&access_id=$ACCESS_ID&access_key=$ACCESSKE
Y'

format_of_field_option:
```

```
    {FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']

start_and_end_option:
    LINES
        [STARTING BY 'string']
        [TERMINATED BY 'string']
```

参数	是否必填	描述	示例
column_list_option	是	导出的列选项。如果要选中全部数据可以用 `*` 表示。 `column_name`：列名称。	`SELECT col1,col2,col3 ...`
file_route_option	是	选择导出的文件路径，支持导出到阿里云 OSS 中。 **说明** 由于阿里云 OSS 有文件大小的限制，对于超过 5GB 的文件，导出到 OSS 时会被拆分成多个文件，每个文件小于 5GB。	`... INTO OUTFILE '/home/admin/student.sql' ...`
format_of_field_option	否	导出字段格式选项。指定输出文件中各个字段的格式，通过 `FIELDS` 或 `COLUMNS` 子句来指定。 • `TERMINATED BY`：用来指定字段值之间的符号。例如，`TERMINATED BY ','` 指定了逗号作为两个字段值之间的标志。 • `ENCLOSED BY`：用来指定包裹字段值的符号。例如，`ENCLOSED BY '"'` 表示字符值放在双引号之间。如果使用了 `OPTIONALLY` 关键字，则仅对字符串类型的值使用指定字符包裹。 • `ESCAPED BY`：用来指定转义字符。例如，	`... TERMINATED BY ',' ENCLOSED BY '"' ...`

		`ESCAPED BY '*'` 表示将星号（*）指定为转义字符来取代默认的转义字符（`\`）。	
start_and_end_option	否	导出数据行的开始和结束符选项。指定输出文件中每一行的开始和结束字符，通过 `LINES` 子句设置。 • `STARTING BY`：指定每一行开始的字符。 • `TERMINATED BY`：指定每一行的结束字符。	`... LINES TERMINATED BY '\n' ...` 表示一行将以换行符作为结束标志。
FROM table_name_list	是	指定选择数据的对象。	`... FROM tbl1,tbl2 ...`
WHERE where_conditions	否	指定筛选条件，查询结果中仅包含满足条件的数据。	`... WHERE col1 > 100 ...`
GROUP BY group_by_list	否	指定分组的字段，通常与聚合函数配合使用。 **说明** SELECT 子句后面的所有列中，没有使用聚合函数的列，必须出现在 GROUP BY 子句后面。	`... GROUP BY col1,col2 ...`
HAVING having_search_conditions	否	筛选分组后的各组数据。`HAVING` 子句与 `WHERE` 子句类似，但是 `HAVING` 子句可以使用累计函数（如 `SUM`、`AVG` 等）。	`... HAVING SUM(col1) < 160 ...`
ORDER BY order_expression_list	否	指定结果集按照一个列或者多个列用来 ASC 或 DESC 显示查询结果。不指定 ASC 或者 DESC 时，默认为 ASC。 • `ASC`：表示升序。 • `DESC`：表示降序。	`... ORDER BY col1,col2 DESC ...`

示例

本文以数据导出到设备本地为例，提供数据的导出示例。

1. 在租户 `mysql001` 的 `test` 库中创建表 `tbl1` 并插入数据。

```
obclient [test]> CREATE TABLE tbl1(col1 INT PRIMARY KEY,col2 varchar(128),col3 INT
);
Query OK, 0 rows affected

obclient [test]> INSERT INTO tbl1 VALUES(1,'one',80),(2,'two',90),(3,'three',100);
Query OK, 3 rows affected
Records: 3  Duplicates: 0  Warnings: 0

obclient [test]> SELECT * FROM tbl1;
+------+-------+------+
| col1 | col2  | col3 |
+------+-------+------+
|    1 | one   |   80 |
|    2 | two   |   90 |
|    3 | three |  100 |
+------+-------+------+
3 rows in set
```

2. 设置导出的文件路径。

   设置系统变量 `secure_file_priv`，配置导入或导出文件时可以访问的路径。

   注意

   由于安全原因，设置系统变量 secure_file_priv 时，只能通过本地 Socket 连接数据库执行修改该全局变量的 SQL 语句。系统变量的详细介绍和设置后的连接示例可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Global 系统变量/secure_file_priv。

   1. 登录到要连接的 OBServer 节点。

   2. 执行以下命令，通过本地 Unix Socket 连接方式连接租户 `mysql001`。

   ```
   [admin@test ~]$ obclient -S /home/admin/oceanbase/run/sql.sock -uroot@mys
   ql001 -p******
   ```

   3. 设置导出路径为 `/home/admin`。

   ```
   obclient [(none)]> SET GLOBAL secure_file_priv = "/home/admin";
   ```

3. 登录到要连接的 OBServer 节点。

4. 执行以下命令，通过本地 Unix Socket 连接方式连接租户 `mysql001`。

```
[admin@test ~]$ obclient -S /home/admin/oceanbase/run/sql.sock -uroot@mysql001 -p*
*****
```

5. 设置导出路径为 `/home/admin`。

```
obclient [(none)]> SET GLOBAL secure_file_priv = "/home/admin";
```

6. 重新连接数据库后，使用 `SELECT INTO OUTFILE` 语句导出表 `tbl1` 中的数据。导出文件名为 `tbl1.sql`；指定逗号作为两个字段值之间的标志；对字符串类型的值使用 `"` 字符包裹；使用换行符作为结束标志。

```
obclient [test]> SELECT * INTO OUTFILE '/home/admin/tbl1.sql'
    FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
    LINES TERMINATED BY '\n'
    FROM tbl1;
```

7. 登录机器，在设备本地的 `/home/admin` 目录下查看导出的文件信息。

```
[admin@test ~]$ cat tbl1.sql
1,"one",80
2,"two",90
3,"three",100
```

LOAD DATA 导入

可使用 LOAD DATA 语法从外部导入数据到数据库中。目前支持从服务器端、客户端和 OSS 文件中导入。

注意

- 带有触发器（Trigger）的表禁止使用 `LOAD DATA` 语句。
- 要从外部文件导入数据，您需要具有 `FILE` 权限及以下设置：
    - 加载服务器端文件时，需要提前设置系统变量 secure_file_priv，配置导入或导出文件时可以访问的路径。

    - 加载客户端本地文件时，需要在启动 MySQL/OBClient 客户端时添加 `--local-infile[=1]` 选项来启用从本地文件系统加载数据的功能。

- 加载服务器端文件时，需要提前设置系统变量 secure_file_priv，配置导入或导出文件时可以访问的路径。
- 加载客户端本地文件时，需要在启动 MySQL/OBClient 客户端时添加 `--local-infile[=1]` 选项来启用从本地文件系统加载数据的功能。

OceanBase 数据库 LOAD DATA 语句支持加载以下输入文件：

- 服务器端（OBServer 节点）文件：文件位于 OceanBase 数据库的 OBServer 节点上。可以使用 `LOAD DATA INFILE` 语句，将服务器端文件中加载数据到数据库表中。
- 客户端（本地）文件：文件位于客户端本地文件系统中。可以使用 `LOAD DATA LOCAL INFILE` 语句，将客户端本地文件中的数据加载到数据库表中。

    说明

    OceanBase 数据库 MySQL 模式从 V4.2.2 版本起，开始支持使用 `LOAD DATA LOCAL INFILE` 语法加载本地数据文件。并且在执行 `LOAD DATA LOCAL INFILE` 命令时，系统会自动添加 `IGNORE` 选项。

- OSS 文件：文件位于 OSS 文件系统。可以使用 `LOAD DATA REMOTE_OSS INFILE` 语句，将 OSS 文件中的数据加载到数据库表中。

LOAD DATA 语句目前可以对 CSV 格式的文本文件进行导入，整个导入的过程可以分为以下的流程：

1. 解析文件：OceanBase 数据库会根据用户输入的文件名读取文件中的数据，并且根据指定的并行度来决定并行或者串行解析输入文件中的数据。
2. 分发数据：由于 OceanBase 数据库是分布式数据库，各个分区的数据可能分布在各个不同的 OBServer 节点，LOAD DATA 语句会对解析出来的数据进行计算，决定数据需要被发送到哪个 OBServer 节点。
3. 插入数据：当目标 OBServer 节点收到数据后，在本地执行 INSERT 操作将数据插入到对应的分区当中。

LOAD DATA 具体语法介绍及示例请参见官网《OceanBase 数据库》参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/LOAD DATA。

表与表之间的数据迁移

表与表之间的数据迁移，可以通过 SQL 语句快速实现，如使用 INSERT INTO 或者 MERGE INTO 语句，主要实现同租户中，不同表之间数据的迁移。

INSERT INTO

语法

```
INSERT INTO target_table_name[(target_col_name[, target_col_name] ...)]
SELECT [(source_col_name[, source_col_name] ...)]
FROM source_table_name
[WHERE expr];
```

参数解释：

参数	描述
target_table_name	数据迁移的目标表。
target_col_name	目标表的列名称。如果要更新目标表的全部列数据，可以不填写列名称。
source_col_name	源表的列名称。选择需要迁移的列，如果要选中全部数据可以用 * 表示。 **注意** 选择的列数量需要与目的表中的列数量保持一致。
source_table_name	数据迁移的源表。
WHERE expr	迁移数据的筛选条件，不填表示迁移 `SELECT` 选中的所有行记录。

示例

此处以将表 `tbl1` 中符合条件（`age` > 10）的数据插入表 `tbl2` 中为例。

1. 查看表 `tbl1` 中数据

   ```
   obclient [test]> SELECT * FROM tbl1;
   ```

   输出如下：

```
+------+------+------+
| id   | name | age  |
+------+------+------+
|    1 | ab   |    8 |
|    2 | bc   |   18 |
|    3 | cd   |   14 |
|    4 | de   |   19 |
|    5 | ef   |    6 |
|    6 | fg   |   15 |
+------+------+------+
```

2. 查看表 `tbl2` 的表结构

```
obclient [test]> DESC tbl2;
```

输出如下：

```
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| col1  | int(11) | YES  |     | NULL    |       |
| col2  | int(11) | YES  |     | NULL    |       |
+-------+---------+------+-----+---------+-------+
```

3. 查看表 `tbl2` 中数据

```
obclient [test]> SELECT * FROM tbl2;
```

输出如下，表示表 `tbl2` 中无数据。

```
Empty set
```

4. 将表 `tbl1` 中符合条件（`age` > 10）的数据插入表 `tbl2` 中

```
obclient [test]> INSERT INTO tbl2 SELECT id,age FROM tbl1 WHERE age > 10;
```

5. 再次查看表 `tbl2` 中数据

```
obclient [test]> SELECT * FROM tbl2;
```

输出如下，表 `tbl2` 中已存在表 `tbl1` 中 `age`>10 的数据。

```
+------+------+
| col1 | col2 |
+------+------+
|    2 |   18 |
|    3 |   14 |
|    4 |   19 |
|    6 |   15 |
+------+------+
```

旁路导入

OceanBase 数据库支持通过旁路导入的方式向数据库插入数据，即 OceanBase 数据库支持向 data 文件中直接写入数据。旁路导入可以绕过 SQL 层的接口，直接在 data 文件中直接分配空间并插入数据，从而提高数据导入的效率。

使用场景

旁路导入功能可以在下面的场景中使用：

- 数据迁移与同步。对于数据迁移和同步，通常需要将大量的各种格式的数据从不同的数据源向 OceanBase 数据库进行迁移，传统的 SQL 接口性能可能在时效性上无法得到满足。
- 传统 ETL。当数据在源端进行了抽取和转化之后，在装载到目标端时，通常需要在短时间内加载大量的数据，使用旁路导入技术，会提升数据导入的性能。而对于 ELT 技术，在装载数据的过程中，也可以通过旁路导入技术提高效率。
- 从文本文件或者其他数据源向 OceanBase 数据库加载数据，利用旁路导入技术，也能够提升加载数据效率。

功能列表

目前 OceanBase 数据库支持以下语句进行旁路导入：

- `LOAD DATA /*+ direct */`
- `INSERT /*+ append */ INTO SELECT`

注意

旁路导入会把所有的已有的数据都写一遍。如果原表的数据比较大，导入的数据比较少，不建议使用旁路导入功能。

使用 LOAD DATA 语句旁路导入数据

`LOAD DATA` 语句通过 Hint 使用 `direct` 来表示是否走旁路导入。

使用限制

- 不支持两个语句同时写一个表，因为导入的过程中会先加表锁。
- 不支持在触发器（Trigger）使用。
- 支持 `lob` 类型，但是性能比较差，`lob` 会走原来事务写入数据的路径。
- 不支持在多行事务中运行。

注意事项

为提高数据导入速率，OceanBase 数据库在 `LOAD DATA` 操作中采用了并行设计。在该过程中，需要导入的数据被划分为多个子任务以并行方式执行，每个子任务都作为一个独立的事务进行处理，并且执行顺序是随机的。因此，使用过程中需要注意以下事项:

- 无法保证整体数据导入的原子性。
- 对于无主键表来说，数据写入的顺序可能与文件中的数据顺序不一致。

语法

```
LOAD DATA /*+ direct(need_sort,max_error) parallel(N) */ INFILE 'file_name' ...
```

参数解释:

参数	描述
direct	表示走旁路导入。
need_sort	表示是否需要 OceanBase 数据库对数据进行排序。

	值为 `bool` 类型： • `true`：表示需要排序。 • `false`：表示不需要排序。
max_error	表示最大的容忍的错误的行数。值为 `INT` 类型，超过这个数值`LOAD DATA` 会报失败。
parallel(N)	加载数据的并行度，`N` 默认为 `4`。

示例

此处以从服务端文件导入数据为例，OceanBase 数据库 LOAD DATA 语句旁路导入数据还支持加载本地文件（LOCAL INFILE），更多 LOAD DATA INFILE 的示例信息可参见官网《OceanBase 数据库》文档 数据迁移/从 CSV 文件迁移数据到 OceanBase 数据库/使用 LOAD DATA 语句导入数据。

1. 登录到要连接 OBServer 节点所在的机器，在家目录下创建测试数据。

   说明

   OceanBase 数据库中的 `LOAD DATA` 语句仅支持加载 OBServer 节点本地的输入文件。因此，需要在导入之前将文件拷贝到某台 OBServer 节点上。

   ```
   [admin@test ~]$ vi tbl1.csv
   ```

   内容如下

   ```
   1,11
   2,22
   3,33
   ```

2. 设置导入的文件路径。

   设置系统变量 `secure_file_priv`，配置导入或导出文件时可以访问的路径。

   注意

> 由于安全原因，设置系统变量 secure_file_priv 时，只能通过本地 Socket 连接数据库执行修改该全局变量的 SQL 语句。系统变量的详细介绍和设置后的连接示例可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Global 系统变量/secure_file_priv。

1. 登录到要连接 OBServer 节点所在的机器。

2. 执行以下命令，通过本地 Unix Socket 连接方式连接租户 `mysql001`。

```
[admin@test ~]$ obclient -S /home/admin/oceanbase/run/sql.sock -uroot@mys
ql001 -p******
```

3. 设置导入路径为 `/home/admin`。

```
obclient [(none)]> SET GLOBAL secure_file_priv = "/home/admin";
```

3. 登录到要连接 OBServer 节点所在的机器。
4. 执行以下命令，通过本地 Unix Socket 连接方式连接租户 `mysql001`。

```
[admin@test ~]$ obclient -S /home/admin/oceanbase/run/sql.sock -uroot@mysql001 -p*
*****
```

5. 设置导入路径为 `/home/admin`。

```
obclient [(none)]> SET GLOBAL secure_file_priv = "/home/admin";
```

6. 重新连接数据库后，使用 `LOAD /*+ APPEND */ DATA` 语句导入数据。

```
obclient [test]> CREATE TABLE tbl1(col1 INT PRIMARY KEY,col2 INT);
Query OK, 0 rows affected

obclient [test]> SELECT * FROM tbl1;
Empty set

obclient [test]> LOAD DATA /*+ direct(true,1024) parallel(16) */INFILE '/home/admi
n/tbl1.csv' INTO TABLE tbl1 FIELDS TERMINATED BY ',';
Query OK, 3 rows affected
Records: 3  Deleted: 0  Skipped: 0  Warnings: 0

obclient [test]> SELECT * FROM tbl1;
+------+------+
| col1 | col2 |
+------+------+
```

```
|     1 |    11 |
|     2 |    22 |
|     3 |    33 |
+------+-------+
3 rows in set
```

使用INERT INTO 语句旁路导入数据

`INSERT INTO SELECT` 语句通过 Hint 使用 `append` 加上 `enable_parallel_dml` 来走旁路导入。

使用限制

- 只支持 PDML（Parallel Data Manipulation Language，并行数据操纵语言），非 PDML 不能用旁路导入。
- 不支持能两个语句同时写一个表，因为导入的过程中会先加表锁。
- 不支持在触发器（Trigger）使用。
- 支持 `lob` 类型，但是性能比较差，`lob` 会走原来事务写入数据的路径。
- 不支持在多行事务（包含多个操作的事务）中运行。

语法

```
INSERT /*+ append enable_parallel_dml parallel(N) */ INTO  table_name select_sente
nce
```

参数解释：

参数	描述
append	表示走旁路导入。

enable_parallel_dml parallel(N)	加载数据的并行度，
	`N`
	默认为
	`4`
	。
	说明

	一般情况下，`enable_parallel_dml` Hint 和 `parallel` Hint 必须配合使用才能开启并行 DML。不过，当目标表的 Schema 上指定了表级别的并行度时，仅需指定 `enable_parallel_dml` Hint。

示例

此处以使用旁路导入将表 `tbl2` 中的部分数据导入到 `tbl1` 中为例。

1. 查看表 `tbl1` 中数据

```
obclient [test]> SELECT * FROM tbl1;
```

输出如下，表 `tbl1` 中无数据。

```
Empty set
```

2. 查看表 `tbl2` 中数据

```
obclient [test]> SELECT * FROM tbl2;
```

输出如下：

```
+------+------+-------+
| col1 | col2 | col3  |
+------+------+-------+
|    1 | a1   |    11 |
|    2 | a2   |    22 |
|    3 | a3   |    33 |
+------+------+-------+
```

3. 使用旁路导入将表 `tbl2` 中的部分数据导入到 `tbl1` 中

```
obclient [test]> INSERT /*+ append enable_parallel_dml parallel(16) */ INTO tbl1 S
ELECT t2.col1,t2.col3 FROM tbl2 t2;
```

4. 再次查看表 `tbl1` 中数据

```
obclient [test]> SELECT * FROM tbl1;
```

输出如下，表 `tbl1` 中已包含表 `tbl2` 中数据。

```
+------+------+
| col1 | col2 |
+------+------+
|    1 |   11 |
|    2 |   22 |
|    3 |   33 |
+------+------+
```

在 `EXPLAIN EXTENDED` 语句的返回结果的 `Note` 中，查看是否通过旁路导入功能写入数据。命令如下：

```
obclient [test]> EXPLAIN EXTENDED INSERT /*+ append enable_parallel_dml parallel(1
6) */ INTO tbl1 SELECT t2.col1,t2.col3 FROM tbl2 t2;
```

输出如下：

```
+---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
------------------------------------------------------------------+
| Query Pla
n

                                                       |
+---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
------------------------------------------------------------------+
| ===========================================================================
=
                                                                        |
| |ID|OPERATOR                    |NAME           |EST.ROWS|EST.TIME(us)
|
                                                                        |
| ---------------------------------------------------------------------------
-
                                                                        |
| |0 |OPTIMIZER STATS MERGE       |               |3       |27
|
                                                                        |
| |1 | PX COORDINATOR             |               |3       |27
|
                                                                        |
| |2 |  EXCHANGE OUT DISTR        |:EX10001       |3       |27
|
                                                                        |
| |3 |   INSERT                   |               |3       |26
|
```

```
                                                                        |
| |4 |    EXCHANGE IN DISTR          |               |3       |1
|
                                                                        |
| |5 |     EXCHANGE OUT DISTR (RANDOM)|:EX10000       |3       |1
|
                                                                        |
| |6 |      OPTIMIZER STATS GATHER    |               |3       |1
|
                                                                        |
| |7 |       SUBPLAN SCAN             |ANONYMOUS_VIEW1|3       |1
|
                                                                        |
| |8 |        PX BLOCK ITERATOR       |               |3       |1
|
                                                                        |
| |9 |         TABLE SCAN             |t2             |3       |1
|
                                                                        |
| ===========================================================================
=
                                                                        |
| Outputs & filters
:

                                                   |
| ------------------------------------
-

                                     |
|   0 - output(nil), filter(nil), rowset=25
6

                                |
|   1 - output([column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col1(0x7f0ba6a51800
))(0x7f0ba6a522c0)], [column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col3(0x7f0ba6
a51ac0))(0x7f0ba6a59630)]),                                      |
| filter(nil), rowset=25
6

                                          |
|   2 - output([column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col1(0x7f0ba6a51800
))(0x7f0ba6a522c0)], [column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col3(0x7f0ba6
a51ac0))(0x7f0ba6a59630)]),                                      |
| filter(nil), rowset=25
6

                                          |
|       dop=1
6
```

```
                                                    |
|   3 - output([column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col1(0x7f0ba6a51800
))(0x7f0ba6a522c0)], [column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col3(0x7f0ba6
a51ac0))(0x7f0ba6a59630)]),                                                  |
| filter(nil
)

                                                     |
|       columns([{tbl1: ({tbl1: (tbl1.__pk_increment(0x7f0ba6a51d80), tbl1.col1(0x7
f0ba6a30a90), tbl1.col2(0x7f0ba6a30d50))})}]), partitions(p0)
,
   |
|       column_values([T_HIDDEN_PK(0x7f0ba6a52040)], [column_conv(INT,PS:(11,0),NU
LL,ANONYMOUS_VIEW1.col1(0x7f0ba6a51800))(0x7f0ba6a522c0)], [column_conv(INT,PS:(11
,0),NULL,ANONYMOUS_VIEW1.col3(0x7f0ba6a51ac0))(0x7f0ba6a59630)]) |
|   4 - output([column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col1(0x7f0ba6a51800
))(0x7f0ba6a522c0)], [column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col3(0x7f0ba6
a51ac0))(0x7f0ba6a59630)],                                                   |
| [T_HIDDEN_PK(0x7f0ba6a52040)]), filter(nil), rowset=25
6

         |
|   5 - output([column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col1(0x7f0ba6a51800
))(0x7f0ba6a522c0)], [column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col3(0x7f0ba6
a51ac0))(0x7f0ba6a59630)],                                                   |
| [T_HIDDEN_PK(0x7f0ba6a52040)]), filter(nil), rowset=25
6

         |
|       dop=1
6

                                                    |
|   6 - output([column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col1(0x7f0ba6a51800
))(0x7f0ba6a522c0)], [column_conv(INT,PS:(11,0),NULL,ANONYMOUS_VIEW1.col3(0x7f0ba6
a51ac0))(0x7f0ba6a59630)]),                                                  |
| filter(nil), rowset=25
6

                                         |
|   7 - output([ANONYMOUS_VIEW1.col1(0x7f0ba6a51800)], [ANONYMOUS_VIEW1.col3(0x7f0
ba6a51ac0)]), filter(nil), rowset=25
6
                              |
|       access([ANONYMOUS_VIEW1.col1(0x7f0ba6a51800)], [ANONYMOUS_VIEW1.col3(0x7f0
ba6a51ac0)]
)
                                                      |
|   8 - output([t2.col1(0x7f0ba6a50d40)], [t2.col3(0x7f0ba6a512f0)]), filter(nil)
, rowset=25
6
```

```
                                                                  |
|   9 - output([t2.col1(0x7f0ba6a50d40)], [t2.col3(0x7f0ba6a512f0)]), filter(nil)
, rowset=25
6
                                                                  |
|       access([t2.col1(0x7f0ba6a50d40)], [t2.col3(0x7f0ba6a512f0)]), partitions(p
0
)
                                                                           |
|       is_index_back=false, is_global_index=false
,
               |
|       range_key([t2.__pk_increment(0x7f0ba6a6ccf0)]), range(MIN ; MAX)always tru
e
                                                                  |
| Used Hint
:
                                                              |
| ----------------------------------
-
                           |
| /*
+
                                                                     |
|
                                                                  |
|       USE_PLAN_CACHE( NONE
)
                                        |
|       PARALLEL(16
)
                                                    |
|       ENABLE_PARALLEL_DM
L
                                           |
|       APPEN
D
                                                           |
|       APPEN
D
                                                           |
```

```
| *
/
                                                                  |
| Qb name trace
:
                                                      |
| ------------------------------------
-
                              |
|   stmt_id:0, stmt_type:T_EXPLAI
N
                                   |
|   stmt_id:1, INS$
1
                                                 |
|   stmt_id:2, SEL$
1
                                                 |
| Outline Data
:
                                                       |
| ------------------------------------
-
                              |
| /*
+
                                                                 |
|       BEGIN_OUTLINE_DAT
A
                                           |
|       FULL(@"SEL$1" "test"."t2"@"SEL$1"
)
                         |
|       USE_PLAN_CACHE( NONE
)
                                     |
|       PARALLEL(16
)
                                                 |
```

```
|       ENABLE_PARALLEL_DM
L
                                       |
|       OPTIMIZER_FEATURES_ENABLE('4.0.0.0'
)
                      |
|       APPEN
D
                                                            |
|       APPEN
D
                                                            |
|       END_OUTLINE_DAT
A
                                          |
| *
/
                                                                      |
| Optimization Info
:
                                                      |
| ------------------------------------
-
                           |
| t2
:
                                                                     |
|       table_rows:
3
                                                      |
|       physical_range_rows:
3
                                             |
|       logical_range_rows:
3
                                              |
|       index_back_rows:
0
                                                 |
```

```
|       output_rows:
3
                                              |
|       est_method:local_storag
e
                                   |
|       optimization_method:cost_base
d
                             |
|       avaiable_index_name:[tbl2
]
                                 |
|       table_id:500004:estimation info:(table_type:12, version:-1--1--1, logical_
rc:3, physical_rc:3)
]
                                              |
|       stats version:
0
                                            |
| Plan Type
:
                                                       |
|       DISTRIBUTE
D
                                                |
| Note
:
                                                            |
|       Degree of Parallelism is 16 because of hin
t
               |
|       Direct-mode is enabled in insert into selec
t
              |
+-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
-------------------------------------------------------------------+
86 rows in set
```

4.7 通过其他工具进行数据的迁移同步

除前几节介绍工具和命令可以实现数据的导入导出外，目前 OceanBase 也支持很多第三方工具。如开源的 Flink CDC、DataX、Canal 等工具，这些工具的功能都略有不同，因此适用场景也不太一样，本节将详细介绍如何使用这些工具，实现各类数据库与OceanBase之间的数据迁移和同步。

说明

本教程中所贴官网文档链接版本均为编写教程时的最新版本，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

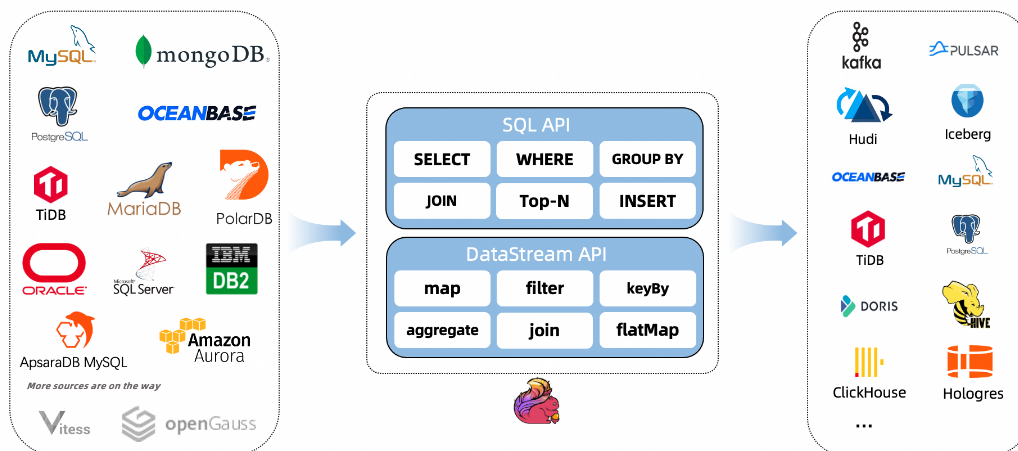
Flink CDC 实现数据迁移同步

Flink CDC介绍

CDC （Change Data Capture，即变更数据捕获）能够帮助您监测并捕获数据库的变动。CDC 提供的数据可以做很多事情，比如：做历史库、做近实时缓存、提供给消息队列（MQ），用户消费 MQ 做分析和审计等。

Flink CDC（CDC Connectors for Apache Flink）是 Apache Flink 的一组 Source 连接器，它支持从大多数据库中实时地读取存量历史数据和增量变更数据，能够将数据库的全量和增量数据同步到消息队列和数据仓库中。Flink CDC 也可以用于实时数据集成，您可以使用它将数据库数据实时导入数据湖或者数据仓库。

同时，Flink CDC 还支持数据加工，您可以通过它的 SQL Client 对数据库数据做实时关联、打宽、聚合，并将结果写入到各种存储中。



支持的连接器

连接器	数据库	驱动
mongodb-cdc	MongoDB: 3.6、4.x、5.0	MongoDB Driver: 4.3.4
mysql-cdc	• MySQL：5.6、5.7、8.0.x • RDS MySQL：5.6、5.7、8.0.x • PolarDB MySQL：5.6、5.7、8.0.x • Aurora MySQL：5.6、5.7、8.0.x • MariaDB：10.x • PolarDB X：2.0.1	JDBC Driver: 8.0.28
oceanbase-cdc	• OceanBase CE：3.1.x、4.x • OceanBase EE：2.x、3.x、4.x	OceanBase Driver: 2.4.x
oracle-cdc	Oracle: 11、12、19、21	Oracle Driver: 19.3.0.0
postgres-cdc	PostgreSQL: 9.6、10、11、12、13、14	DBC Driver: 42.5.1
sqlserver-cdc	Sqlserver: 2012、2014、2016、2017、2019	JDBC Driver: 9.4.1.jre8
tidb-cdc	TiDB：5.1.x、5.2.x、5.3.x、5.4.x、6.0.0	JDBC Driver: 8.0.27
db2-cdc	Db2：11.5	Db2 Driver: 11.5.0.0
vitess-cdc	Vitess: 8.0.x、9.0.x	MySql JDBC Driver: 8.0.26

配置 Flinck CDC 数据迁移同步

详细操作请参见官网《OceanBase 数据库》文档：

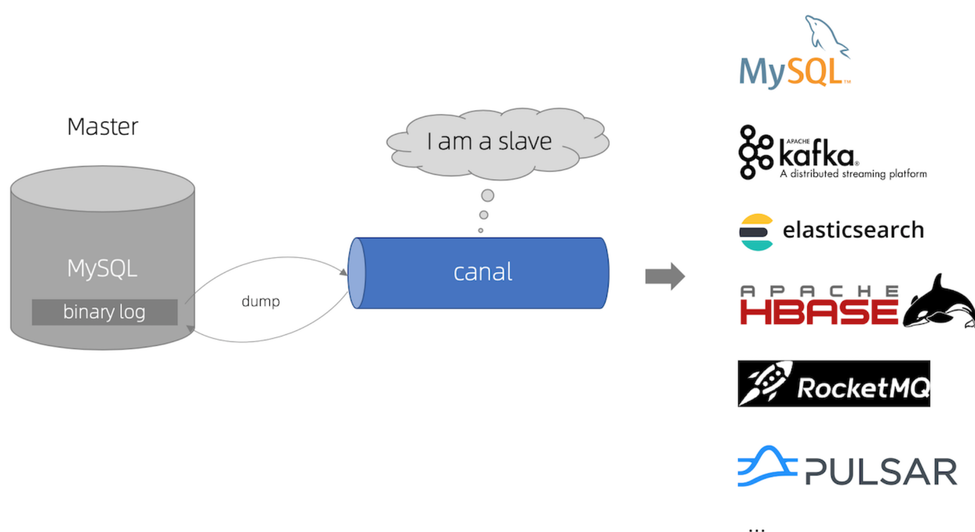
- 数据迁移/从 MySQL 数据库迁移数据到 OceanBase 数据库/使用 Flink CDC 从 MySQL 数据库同步数据到 OceanBase 数据库
- 数据迁移/从 OceanBase 数据库迁移数据到 MySQL 数据库/使用 Flink CDC 从 OceanBase 数据库迁移数据到 MySQL 数据库

Canal 实现数据迁移同步

Canal介绍

Canal 是 Alibaba 开源的一个产品，主要用途是基于 MySQL 数据库增量日志解析，提供增量数据订阅和消费。

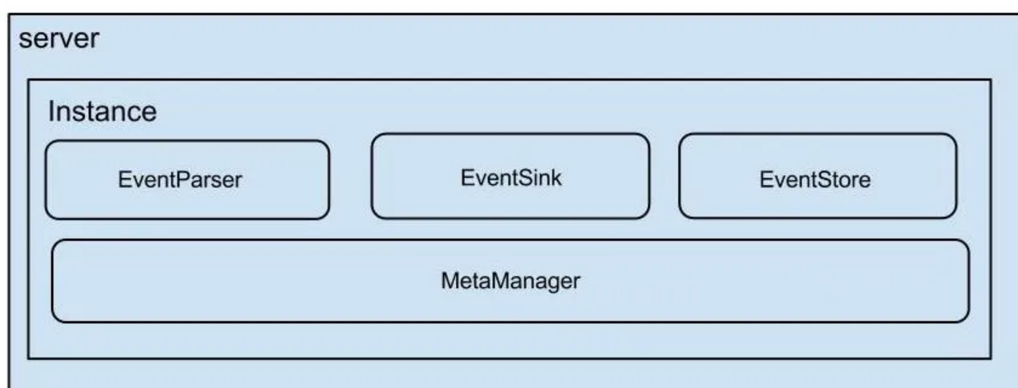
Canal 工作原理如下图所示



- Canal 模拟 MySQL slave 的交互协议，伪装自己为 MySQL slave，向 MySQL master 发送 dump 协议
- MySQL master 收到 dump 请求，开始推送 binary log 给 slave（即 canal）
- Canal 解析 binary log 对象（原始为 byte 流）

架构和组件

Canal 的架构和组件如下图：



说明：

- server：代表一个 canal 运行实例，对应于一个 jvm。
- Instance：对应一个数据队列，一个 server 对应 1~n 个 Instance。Instance 模块中包含如下内容：
    - eventParser：数据源接入，模拟 slave 协议和 master 进行交互，协议解析。
    - eventSink：Parser 和 Store 链接器，进行数据过滤、加工和分发的工作。
    - eventStore：数据存储。
    - metaManager：增量订阅 & 消费信息管理器。

- eventParser：数据源接入，模拟 slave 协议和 master 进行交互，协议解析。
- eventSink：Parser 和 Store 链接器，进行数据过滤、加工和分发的工作。
- eventStore：数据存储。
- metaManager：增量订阅 & 消费信息管理器。

配置 Canal 数据迁移同步

详细操作请参见官网《OceanBase 数据库》文档：

- 数据迁移/从 MySQL 数据库迁移数据到 OceanBase 数据库/使用 Canal 从 MySQL 数据库

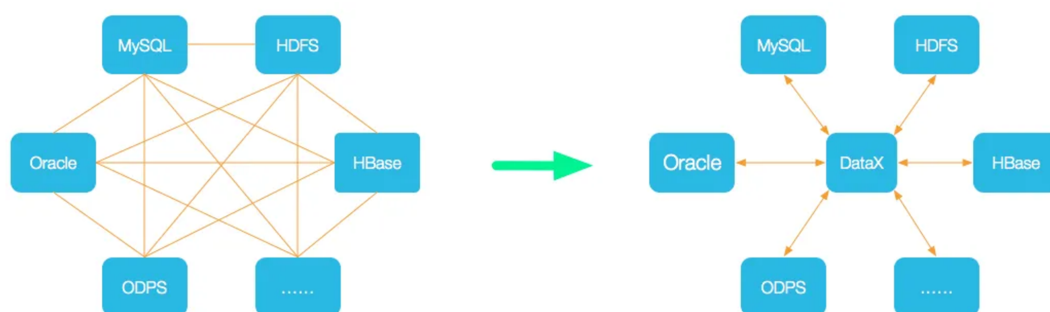
同步数据到 OceanBase 数据库

- 数据迁移/从 OceanBase 数据库迁移数据到 MySQL 数据库/使用 Canal 从 OceanBase 数据库同步数据到 MySQL 数据库

DataX 实现数据的迁移

DataX 介绍

DataX 是阿里云 DataWorks 数据集成的开源版本，是阿里巴巴集团内被广泛使用的离线数据同步工具/平台。DataX 实现了包括 MySQL、Oracle、SQLserver、Postgre、HDFS、Hive、ADS、HBase、TableStore(OTS)、MaxCompute(ODPS)、Hologres、DRDS 、OceanBase 等各种异构数据源之间高效的数据同步功能。

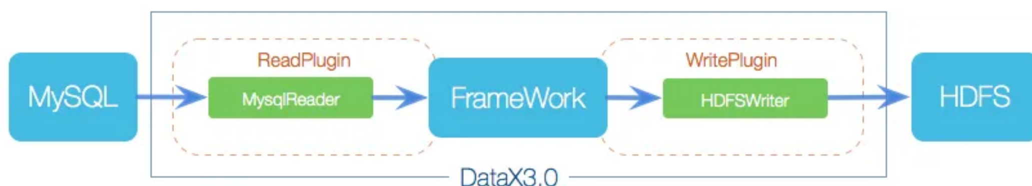


为了解决异构数据源同步问题，DataX 将复杂的网状同步链路变成了星型数据链路，DataX 作为中间传输载体负责连接各种数据源。当需要接入一个新的数据源的时候，只需要将此数据源对接到 DataX，便能跟已有的数据源做到无缝数据同步。

DataX 在阿里巴巴集团内被广泛使用，承担了所有大数据的离线同步业务，并已持续稳定运行了 6 年之久。目前每天完成同步 8w 多道作业，每日传输数据量超过 300TB。

DataX 架构设计

DataX 本身作为离线数据同步框架，采用 Framework + plugin 架构构建。将数据源读取和写入抽象成为 Reader/Writer 插件，纳入到整个同步框架中。



- Reader：Reader 为数据采集模块，负责采集数据源的数据，将数据发送给 Framework。
- Writer：Writer 为数据写入模块，负责不断向 Framework 取数据，并将数据写入到目的端。
- Framework：Framework 用于连接 Reader 和 Writer，作为两者的数据传输通道，并处理缓冲、流控、并发、数据转换等核心技术问题。

DataX 3.0 插件体系

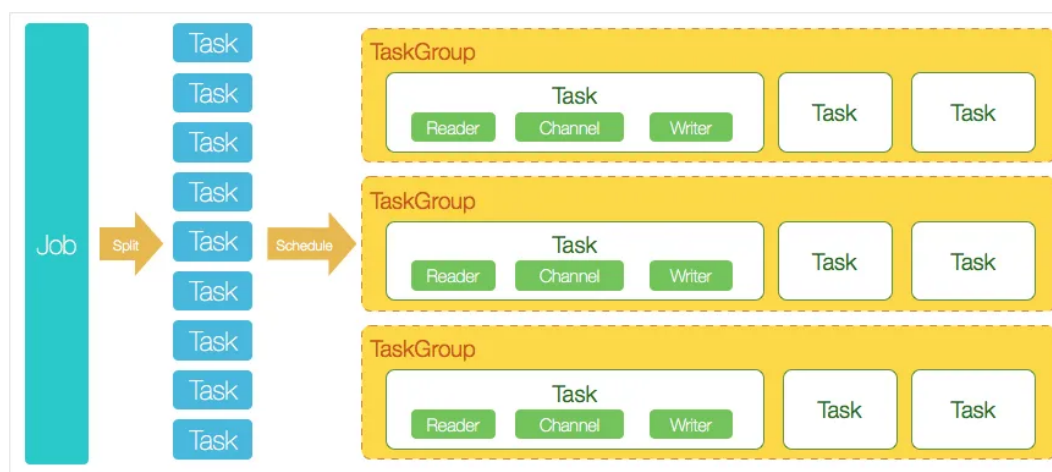
经过几年积累，DataX 目前已经有了比较全面的插件体系，主流的 RDBMS 数据库、NOSQL、大数据计算系统都已经接入。DataX 目前支持数据如下：

类目	数据源	Reader(读)	Writer(写)	文档
RDBMS 关系型数据库	MySQL	▯	▯	读、写
	Oracle	▯	▯	读、写
	OceanBase	▯	▯	读、写
	SQLServer	▯	▯	读、写
	PostgreSQL	▯	▯	读、写
	DRDS	▯	▯	读、写
	达梦	▯	▯	读、写
	通用 RDBMS（支持所有关系型数据库）	▯	▯	读、写
阿里云数仓数据存储	ODPS	▯	▯	读、写
	ADS		▯	写
	OSS	▯	▯	读、写
	OCS	▯	▯	读、写
NoSQL 数据存储	OTS	▯	▯	读、写
	Hbase0.94	▯	▯	读、写

	Hbase1.1	▯	▯	读、写
	MongoDB	▯	▯	读、写
	Hive	▯	▯	读、写
无结构化数据存储	TxtFile	▯	▯	读、写
	FTP	▯	▯	读、写
	HDFS	▯	▯	读、写
	Elasticsearch		▯	写

DataX 核心架构

DataX 3.0 开源版本支持单机多线程模式完成同步作业运行，本节按一个 DataX 作业生命周期的时序图，从整体架构设计非常简要说明 DataX 各个模块相互关系。



核心模块介绍

1. DataX 完成单个数据同步的作业，我们称之为 Job。DataX 接收到一个 Job 之后，将启动一个进程来完成整个作业同步过程。DataX Job 模块是单个作业的中枢管理节点，承担了数据清理、子任务切分（将单一作业计算转化为多个子 Task）、TaskGroup 管理等功能。
2. DataX Job 启动后，会根据不同的源端切分策略，将 Job 切分成多个小的 Task（子任务），以便于并发执行。Task 便是 DataX 作业的最小单元，每一个 Task 都会负责一部分数据的同步工作。
3. 切分多个 Task 之后，DataX Job 会调用 Scheduler 模块，根据配置的并发数据量，将拆分成的 Task 重新组合，组装成 TaskGroup（任务组）。每一个 TaskGroup 负责以一定的并发

级别运行完毕分配好的所有 Task，默认单个任务组的并发数量为 5。

4. 每一个 Task 都由 TaskGroup 负责启动，Task 启动后，会固定启动 Reader—>Channel—>Writer 的线程来完成任务同步工作。
5. DataX 作业运行起来之后，Job 监控并等待多个 TaskGroup 模块任务完成，等待所有 TaskGroup 任务完成后 Job 成功退出。否则，异常退出，进程退出值非 0。

DataX 调度流程

举例来说，用户提交了一个 DataX 作业，并且配置了 20 个并发，目的是将一个 100 张分表的 MySQL 数据同步到 ODPS 里。 DataX 的调度决策思路是：

1. DataX Job 根据分库分表切分成 100 个 Task。
2. 根据 20 个并发，DataX 计算共需要分配 4 个 TaskGroup。
3. 4 个 TaskGroup 平分切分好的 100 个 Task，每一个 TaskGroup 负责以 5 个任务的并发度执行其分配的 25 个 Task。

配置 DataX 数据迁移

详细操作请参见官网《OceanBase 数据库》文档：

- 数据迁移/从 MySQL 数据库迁移数据到 OceanBase 数据库/使用 DataX 迁移 MySQL 表数据到 OceanBase 数据库
- 数据迁移/从 OceanBase 数据库迁移数据到 MySQL 数据库/使用 Datax 迁移 OceanBase 表数据到 MySQL 数据库
- 数据迁移/从 Oracle 数据库迁移数据到 OceanBase 数据库/使用 DataX 迁移 Oracle 表数据到 OceanBase 数据库
- 数据迁移/从 OceanBase 数据库迁移数据到 Oracle 数据库/使用 Datax 迁移 OceanBase 表数据到 Oracle 数据库
- 数据迁移/从 CSV 文件迁移数据到 OceanBase 数据库/使用 DataX 迁移 CSV 文件到 OceanBase 数据库

SeaTunnel 实现数据的迁移同步

在 迁移同步相关生态组件介绍 中已经对 SeaTunnel 做了简单介绍，本节介绍如何部署并使用 SeaTunnel。

SeaTunnel 部署

环境配置

在开始安装 SeaTunnel 之前，需要配置 Java 环境（Java 8 或 11，高于 Java 8 的其他版本理论上也可以工作）。

下载软件

如果机器可以联网，可以直接在终端下载，命令如下。

```
export version="2.3.3"
wget "https://archive.apache.org/dist/seatunnel/${version}/apache-seatunnel-${version}-bin.tar.gz"
tar -xzvf "apache-seatunnel-${version}-bin.tar.gz"
```

如果无法联网，可以从 SeaTunnel 官网 下载安装包，并传到机器上进行解压安装。

安装连接器

根据机器是否可以联网，分为如下两种处理方法：

- 在线安装

  自 2.2.0-beta 版本起，SeaTunnel 的二进制包默认不提供连接器依赖，所以第一次使用时，需要执行以下命令安装连接器，`2.3.3` 为需要的连接器版本。

  ```
  sh bin/install-plugin.sh 2.3.3
  ```

  一般情况下不需要安装所有的连接器插件，可以通过配置来指定需要的插件，配置文件地址 `config/plugin_config`。例如，只需要 connector-console 插件，那么可以修改 `plugin.properties` 为如下内容：

  ```
  --seatunnel-connectors--
  connector-console
  ```

```
--end--
```

- 离线安装

  机器不能联网的情况下，需要手动下载连接器，并上传到 `connectors/seatunnel` 目录下。

  如果手动下载连接器来安装连接器插件，则需要特别注意以下事项：

  - `connectors` 目录包含以下子目录，如果不存在，需要手动创建。

    ```
    flink
    flink-sql
    seatunnel
    spark
    ```

  - 可以只下载需要的 V2 连接器插件并将其放在 `seatunnel` 目录中

- `connectors` 目录包含以下子目录，如果不存在，需要手动创建。

  ```
  flink
  flink-sql
  seatunnel
  spark
  ```

- 可以只下载需要的 V2 连接器插件并将其放在 `seatunnel` 目录中

配置同步任务

成功安装 SeaTunnel 后，可参考本节内容编写配置文件，来启动数据同步任务。这里以将 MySQL 数据同步到 OceanBase 数据库为例。

	源端	目标端
数据库	MySQL	OceanBase MySQL模式
表	mysql2ob	mysql2ob
IP地址	10.10.10.1	10.10.10.2

1. 在源端 MySQL 的 `test` 库中，创建表 `mysql2ob`，并写入三条数据。

   ```
   MySQL [test]> create table mysql2ob(id int primary key, name varchar(20));
   Query OK, 0 rows affected (0.01 sec)
   ```

```
MySQL [test]> insert into mysql2ob values(1,'oceanbase');
Query OK, 1 row affected (0.00 sec)

MySQL [test]> insert into mysql2ob values(2,'oracle');
Query OK, 1 row affected (0.00 sec)

MySQL [test]> insert into mysql2ob values(3,'mysql');
Query OK, 1 row affected (0.00 sec)
```

2. 在目标端 OceanBase 数据库 MySQL 模式租户的 `test` 库下，同样创建 `mysql2ob` 表。

```
obclient [test]> create table mysql2ob(id int primary key, name varchar(20));
```

3. 新建配置文件(mysql_to_oceanbase.conf)，配置文件在 `config` 目录下。

```
[admin@test ~]$ vim config/mysql_to_oceanbase.conf
```

配置文件内容如下：

```
env {
        job.mode = "STREAMING"
        execution.parallelism = 1
        checkpoint.interval = 10000
}
source {
    MySQL-CDC {
        result_table_name = "test"
        parallelism = 1
        server-id = 5656
        username = "root"
        password = "******"
        table-names = ["test.mysql2ob"]
        base-url = "jdbc:mysql://10.10.10.1:3306/test"
    }
}

sink {
  jdbc {
        url = "jdbc:mysql://10.10.10.2:2883/test"
        driver = "com.mysql.jdbc.Driver"
        user = "root@obtest#obcluster"
        password = "******"
        generate_sink_sql = true
        database = "test"
        table = "mysql2ob"
  }
}
```

4. 编写完成之后，启动同步任务

```
[admin@test ~]$ bash ./bin/seatunnel.sh --config ./config/mysql_to_oceanbase.conf
-e local
```

打印日志如下：

```
2024-02-29 18:56:25,664 WARN  org.apache.seatunnel.core.starter.seatunnel.args.Cli
entCommandArgs$MasterTypeValidator -
*********************************************************************************
********
-e and --deploy-mode deprecated in 2.3.1, please use -m and --master instead of it
*********************************************************************************
********
Feb 29, 2024 6:56:25 PM com.hazelcast.internal.config.AbstractConfigLocator
INFO: Loading configuration '/opt/seatunnel/apache-seatunnel-2.3.3/config/seatunne
l.yaml' from System property 'seatunnel.config'
Feb 29, 2024 6:56:25 PM com.hazelcast.internal.config.AbstractConfigLocator
INFO: Using configuration file at /opt/seatunnel/apache-seatunnel-2.3.3/config/sea
tunnel.yaml
Feb 29, 2024 6:56:25 PM org.apache.seatunnel.engine.common.config.SeaTunnelConfig
INFO: seatunnel.home is /opt/seatunnel/apache-seatunnel-2.3.3
Feb 29, 2024 6:56:25 PM com.hazelcast.internal.config.AbstractConfigLocator
INFO: Loading configuration '/opt/seatunnel/apache-seatunnel-2.3.3/config/hazelcas
t.yaml' from System property 'hazelcast.config'
Feb 29, 2024 6:56:25 PM com.hazelcast.internal.config.AbstractConfigLocator
INFO: Using configuration file at /opt/seatunnel/apache-seatunnel-2.3.3/config/haz
elcast.yaml
2024-02-29 18:56:26,178 WARN  com.hazelcast.instance.AddressPicker - [LOCAL] [seat
unnel-997250] [5.1] You configured your member address as host name. Please be awa
re of that your dns can be spoofed. Make sure that your dns configurations are cor
rect.
2024-02-29 18:56:26,178 INFO  com.hazelcast.instance.AddressPicker - [LOCAL] [seat
unnel-997250] [5.1] Resolving domain name 'localhost' to address(es): [127.0.0.1]
2024-02-29 18:56:26,179 INFO  com.hazelcast.instance.AddressPicker - [LOCAL] [seat
unnel-997250] [5.1] Interfaces is disabled, trying to pick one address from TCP-I
P config addresses: [localhost/127.0.0.1]
2024-02-29 18:56:26,202 INFO  org.apache.seatunnel.engine.server.SeaTunnelServer
- SeaTunnel server start...
2024-02-29 18:56:26,204 INFO  com.hazelcast.system - [localhost]:5801 [seatunnel-9
97250] [5.1] Based on Hazelcast IMDG version: 5.1.0 (20220228 - 21f20e7)
2024-02-29 18:56:26,204 INFO  com.hazelcast.system - [localhost]:5801 [seatunnel-9
97250] [5.1] Cluster name: seatunnel-997250
2024-02-29 18:56:26,204 INFO  com.hazelcast.system - [localhost]:5801 [seatunnel-9
97250] [5.1]


 -----               -----                           -
```

```
/  ___|             |_   _|                           | |
\ `--.   ___   __ _   | |   _   _  _ __   _ __    ___ | |
 `--. \ / _ \ / _` |  | |  | | | || '_ \ | '_ \  / _ \| |
/\__/ /|  __/| (_| |  | |  | |_| || | | || | | ||  __/| |
\____/  \___| \__,_|  \_/   \__,_||_| |_||_| |_| \___||_|

......
```

在源端写入数据之后，可以看到会自动同步到目标端。以上就是一个简单的同步任务搭建，大家有兴趣可以尝试更多的同步场景。

第五章 运维 OceanBase 数据库

本章介绍如何对 OceanBase 数据库进行运维。

本章目录

5.1 使用 OCP 进行运维 255

5.2 使用 obd 进行运维 280

5.3 使用 ob-operator 进行运维 299

5.4 使用命令行进行运维 313

5.1 使用 OCP 进行运维

社区版 OCP 是一个专为 OceanBase 数据库设计的企业级管理平台，它帮助用户轻松管理数据库集群的全生命周期，包括安装、运维、性能监控、配置、升级以及主机的添加和删除等。OCP 还提供了租户管理功能，支持租户创建、结构拓扑显示、性能监控、会话管理和参数管理。此外，其监控告警功能允许用户设定集群、租户和主机的监控规则，并通过 HTTP 或脚本通道发送告警通知。系统管理方面，OCP 允许查看和管理任务，以及自定义参数配置。在安全层面，OCP 支持用户和角色管理，确保数据库的安全访问。

更多 OCP 基础管理和运维功能可以参见官网《OceanBase 云平台》文档 集群管理 章节和 运维最佳实践 章节。

本文着重介绍以下 OCP 常用的运维功能。

> **说明**
>
> 本教程中所贴官网文档链接版本均为编写教程时的最新版本或长期支持版本（LTS），若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

OCP 接管集群

OCP 接管集群指的是将一个外部集群（非当前 OCP 部署）的管理权转移到当前 OCP 平台上，实现和使用 OCP 部署的集群同样的管理、监控、运维等功能。

> **说明**
>
> - 目前已发布的 OCP 新版本均支持接管 OceanBase 集群和 OBProxy 集群功能。
> - OCP 目前仅支持接管通过 OCP 部署的 OBProxy 集群（自 V4.2.1 支持），接管 OceanBase 集群时无此限制。

OCP 接管 OceanBase 数据库

当 obd 或 OCP 部署的 OceanBase 集群需要使用新的 OCP 进行管理时，可通过接管功能将

OceanBase 数据库接管到当前 OCP 中，并使用当前 OCP 对新接管的集群继续运维和管理。目前支持接管手动创建、obd 创建或其它 OCP 创建的 OceanBase 数据库。

使用 OCP 接管 OceanBase 数据库前，需了解如下信息：

- 支持接管纯手工安装的 OceanBase 集群，但需要满足接管检查，一般推荐接管 obd 或者 OCP 部署的集群。
- OCP 默认会接管自己的 MetaDB，但不支持一些功能性的运维操作，例如：重启集群、删除租户等（这也是不推荐 MetaDB 和业务集群混用的主要原因之一）。

  使用 root 用户登录 OceanBase 数据库的 ocp_meta 租户，在 `meta_database` 库下执行如下命令更新元数据。

  ```
  update meta_database.config_properties set value='' where key='ocp.ob.cluster.ops.
  blacklist';
  ```

- 当 OCP 需要接管另一个 OCP 部署的 OceanBase 数据库时，一定要先进行集群迁出操作，否则两个 OCP 管理一个集群时 ocp-agent 的配置会互相覆盖，影响使用。迁出的详细操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群/迁出集群。
- 使用 OCP 接管 obd 部署的 OceanBase 数据库后，不建议后续再通过 obd 对这套 OceanBase 数据库进行运维管理。为防止 OCP 和 obd 共同管理一套集群出现配置混乱，可通过如下步骤在 obd 中删除管理信息。

  1. 查看部署集群名称输出如下：

     ```
     +-------------------------------------------------------------+
     |                         Cluster List                        |
     +------+------------------------------------+-----------------+
     | Name | Configuration Path                 | Status (Cached) |
     +------+------------------------------------+-----------------+
     | demo | /home/admin/.obd/cluster/demo      | running         |
     | test | /home/admin/.obd/cluster/test      | running         |
     +------+------------------------------------+-----------------+
     ```

  2. 删去对应集群的管理信息此处以部署集群名为 `test` 为例，您需根据实际部署集群名进行替换。

     ```
     rm -rf /home/admin/.obd/cluster/test
     ```

3. 验证

   再次执行 `obd cluster list` 命令查看 obd 管理的集群，输出中已无被接管集群，表示已成功删除对应管理信息。

- 查看部署集群名称

```
obd cluster list
```

输出如下：

```
+------------------------------------------------------------------+
|                          Cluster List                            |
+------+--------------------------------------+-----------------+
| Name | Configuration Path                   | Status (Cached) |
+------+--------------------------------------+-----------------+
| demo | /home/admin/.obd/cluster/demo        | running         |
| test | /home/admin/.obd/cluster/test        | running         |
+------+--------------------------------------+-----------------+
```

- 删去对应集群的管理信息

此处以部署集群名为 `test` 为例，您需根据实际部署集群名进行替换。

```
rm -rf /home/admin/.obd/cluster/test
```

- 验证

再次执行 `obd cluster list` 命令查看 obd 管理的集群，输出中已无被接管集群，表示已成功删除对应管理信息。

- 接管时如果忘记 `proxyro@sys` 密码，可参见官网《OceanBase 云平台》文档 OBProxy 管理/管理 OBProxy 集群/proxyro 账号管理 一文进行修改。

- 接管前如果已经添加过要接管集群的主机，需要确保主机中填写的机房（IDC）和地区（Region）信息和待接管集群（`DBA_OB_ZONES` 视图查看）的对应信息保持一致。

若不一致可参考如下步骤修改待接管集群的 IDC 和 Region。

1. 使用 root 用户登录待接管集群的 sys 租户

```
obclient -h10.10.10.1 -uroot@sys -P2883 -p
```

```

```

使用 OBClient 客户端连接 OceanBase 租户的详细介绍可参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 OBClient 客户端连接 OceanBase 租户。

2. 修改待接管集群的 IDC 和 Region 信息

```
ALTER SYSTEM ALTER ZONE zone1 SET REGION 'xxx',IDC 'xxx';
```

您需对所有 Zone 执行上述命令。

3. 查看待接管集群的 IDC 和 Region 信息

```
SELECT * FROM oceanbase.DBA_OB_ZONES;
```

DBA_OB_ZONES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_ZONES。

- 使用 root 用户登录待接管集群的 sys 租户

```
obclient -h10.10.10.1 -uroot@sys -P2883 -p
```

使用 OBClient 客户端连接 OceanBase 租户的详细介绍可参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 OBClient 客户端连接 OceanBase 租户。

- 修改待接管集群的 IDC 和 Region 信息

```
ALTER SYSTEM ALTER ZONE zone1 SET REGION 'xxx',IDC 'xxx';
```

您需对所有 Zone 执行上述命令。

- 查看待接管集群的 IDC 和 Region 信息

```
SELECT * FROM oceanbase.DBA_OB_ZONES;
```

DBA_OB_ZONES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_ZONES。

- 接管的 MetaDB 默认关闭 SQL 采集，即 SQL 诊断中无数据是正常现象。可通过在系统参数管理中开启 `ocp.perf.collect.metadb.enabled` 开启 SQL 采集功能，具体操作可参见官网《OceanBase 云平台》文档 系统管理/管理系统参数/修改系统参数。

使用 OCP 接管 OceanBase 集群的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群/接管集群。

OCP 接管 OBProxy

可通过接管操作将其他 OCP 部署的 OBProxy 服务添加到当前 OCP 中已有的 OBProxy 集群中，并通过当前 OCP 对新组成的 OBProxy 集群继续运维和管理。

使用 OCP 接管 OBProxy 集群前，需了解如下信息：

- 使用接管功能之前，需要先使用 OCP 部署一个 OBProxy 集群（可以是单节点），即当前 OCP 中存在 OBProxy 集群才可接管其它 OBProxy 集群。
- 接管前，建议先在原 OCP 上迁出待接管的 OBProxy 集群。详细操作可参见官网《OceanBase 云平台》文档 OBProxy 管理/管理 OBProxy 集群/迁出 OBProxy 集群 或 运维最佳实践/迁出 OBProxy 集群。

> **注意**
>
> 接管 OBProxy 集群需要 `root@proxysys` 用户密码，若忘记该密码，需在迁出前重置密码。重置密码的具体操作可参见官网《OceanBase 云平台》文档 OBProxy 管理/管理 OBProxy 集群/修改 proxysys 密码。

- 仅支持接管 OCP 部署的 OBProxy 服务，obd 部署的 OBProxy 暂不支持接管。
- 接管过程会检查主机中 OBProxy 进程，因此需要待接管 OBProxy 的服务状态是正常的，并且保证主机上有且仅有一个 OBProxy 进程。
- 当前 OCP 中已有的 OBProxy 集群需要先关联待接管 OBProxy 服务所要连接的 OceanBase 集群，即 OBProxy 集群可连接的 OceanBase 集群需要包含待接管的 OBProxy 可连接的 OceanBase 集群。
- 待接管的 OBProxy 服务版本不能和当前 OBProxy 集群版本差太大，建议保证大版本统一，

例如 V4.1.x，第三位的 x 版本号可以不一致。

使用 OCP 接管 OBProxy 的具体操作可参见官网《OceanBase 云平台》文档 OBProxy 管理/管理 OBProxy Server/接管 OBProxy 或 运维最佳实践/接管 OBProxy。

OCP 迁出集群

迁出集群指的是将当前 OCP 管理的 OceanBase 集群或 OBProxy 集群从平台的管理范围内移除，移除后的集群可被其他 OCP 服务接管并管理。

OCP 迁出 OceanBase 集群

当需要使用新的 OCP 管理原 OCP 管理的 OceanBase 集群时，可在原 OCP 中将对应的 OceanBase 集群迁出。将 OCP 管理 OceanBase 集群迁出管理，不会影响集群的业务和业务数据，也可重新被 OCP 接管并管理。

使用 OCP 迁出 OceanBase 数据库前，需了解如下信息：

- 社区版 OCP 自 4.2.2 版本起支持该功能。
- 运维或升级状态下的 OceanBase 集群不支持迁出，即集群仅处于 `STOPPED`、`RUNNING` 或 `UNAVAILABLE` 状态时支持迁出。具体查看方法可参见官网《OceanBase 云平台》文档 参考指南/API 参考/集群信息/查询集群列表。
- 迁出集群后，如果被迁出的 OceanBase 集群主机没有其他服务，建议删除对应主机。具体操作可参见官网《OceanBase 云平台》文档 主机管理/删除主机。

使用 OCP 迁出 OceanBase 集群的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群/迁出集群。

OCP 迁出 OBProxy 集群

通过 OCP 迁出 OceanBase 集群后，如需搭配迁出 OBProxy 集群，OCP 也支持将 OCP 管理的 OBProxy 集群迁出管理，并且不会影响 OceanBase 集群的业务。OBProxy 集群迁出后也可重新被 OCP 接管并管理。

使用 OCP 迁出 OBProxy 集群前，需了解如下信息：

- 社区版 OCP 自 4.2.2 版本起支持该功能。
- 仅当前 OBProxy 集群处于在线状态时支持迁出。
- 迁出集群后，如果被迁出的 OBProxy 集群主机没有其他服务，建议删除对应主机。

使用 OCP 迁出 OBProxy 集群的具体操作可参见官网《OceanBase 云平台》文档 OBProxy 管理/管理 OBProxy 集群/迁出 OBProxy 集群 或 运维最佳实践/迁出 OBProxy 集群。

OCP 数据备份和恢复

备份恢复是 OceanBase 数据库常用的容灾方案之一，通过 NFS、OSS、COS 备份介质实现租户级别或集群级别策略的远程物理备份，以保障数据的安全。同时也可以通过恢复功能从备份数据中恢复数据。

备份介质

数据备份前需要先部署备份介质，目前 OceanBase 集群支持 NFS、OSS、COS 三种备份介质，建议使用 OSS 作为备份的目的端，OSS 作为无状态的对象存储，相较有状态的 NFS4 具有更高的稳定性。如需要使用 NFS 作为备份目的端，建议使用专用的 NFS 硬件设备。

部署备份介质前需了解如下信息：

- NFS 介质需要使用 NFS4.1 或之后版本，NFS3 版本稳定性较差。
- 多节点集群环境必须使用备份介质，否则无法进行数据备份；单机测试环境可以本地备份，但存在安全风险。
- 使用 OSS 备份介质需保证 OceanBase 数据库为 4.1.0 或之后版本，使用 COS 备份介质需保证 OceanBase 数据库为 V4.2.1 或之后版本。
- 如果使用 NFS 备份介质，OceanBase 集群每个节点均需要部署 NFS 客户端。
- 尤其注意 NFS 客户端和服务端目录访问权限问题，OceanBase 数据库进程用户需要有读写权限。

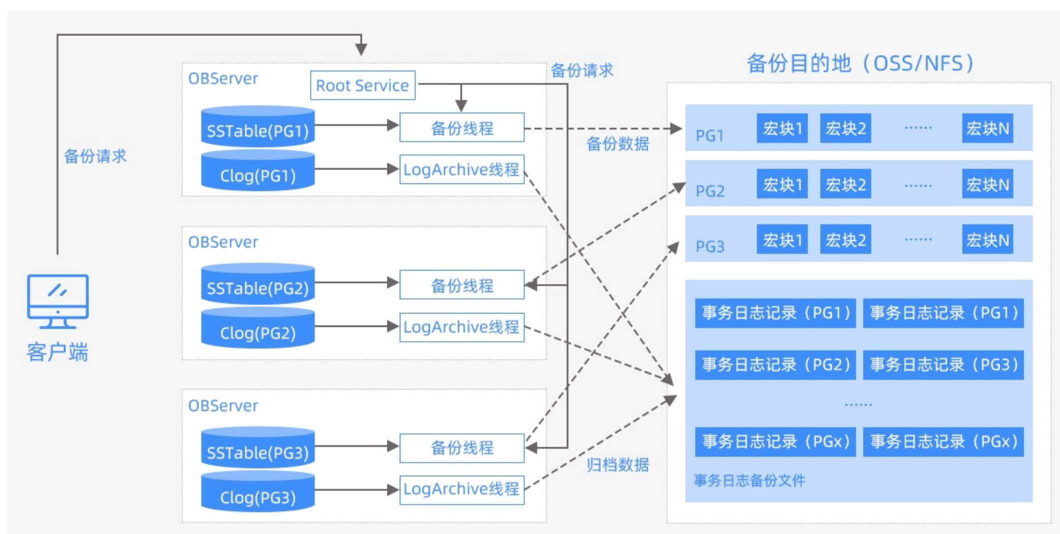
此操作为物理备份前必备过程，具体操作可参见官网《OceanBase 数据库》文档 管理数据库/备份恢复/部署 NFS。

数据备份

部署备份介质后，即可在 OCP 界面侧边栏选择 **备份恢复** -> **备份**，新建租户级别或者集群级别的备份策略，其中集群级别的备份策略也是租户粒度的所有用户租户的数据备份。新建备份策略后，不会立即触发备份，默认每天凌晨四点触发，也可以手动触发 **立即备份** 功能。

> **说明**
>
> 若手动触发 **立即备份**，且备份策略中未勾选 **自动合并**，建议在触发备份前手动进行一次合并。合并的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群合并/执行合并 或 租户管理/管理租户合并/执行合并。

OceanBase 数据库物理备份的架构如下图所示。



使用 OCP 进行数据备份前，需了解如下信息：

- sys 租户和 Meta 租户不支持数据备份功能，目前仅支持对用户租户进行备份。

  Meta 租户的详细介绍可参见官网《OceanBase 数据库》文档 租户介绍 中 **Meta 租户** 章节。

- 数据备份分为归档日志备份和数据备份，数据备份的前提是先开启归档日志备份，即不支持仅做数据备份。

- 不支持对状态处于 **升级中** 的集群进行数据备份，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。

- OceanBase 数据库 4.x 版本二次备份功能默认关闭，需在 **系统参数** 中修改

`ocp.backup.advanced-secondary-backup.enable` 的参数值为 `true`。

> **说明**
>
> 目前二次备份功能仅在使用 OSS 备份介质时支持。

修改系统参数的具体操作可参见官网《OceanBase 云平台》文档 系统管理/管理系统参数/修改系统参数。

- 创建备份策略后，若在自动备份前手动触发 **立即备份**，需要选择数据备份方式为 **全量备份**。
- 数据备份的清理遵循 `全量+增量=完整数据` 原则，如果仅有一份全量备份，即使达到过期备份清理保留周期也不会触发该全量备份的清理。
- 禁止直接以手动执行 rm 命令的方式删除正在执行备份的数据文件，此操作可能会导致归档日志备份失败，数据备份无法完成等问题。
- OCP 上删除备份策略不会删除已经备份的数据文件，如果确认备份数据可以丢弃，可以手动执行 rm 命令删除备份目录。
- **备份恢复** 界面中如果 **可恢复时间** 显示为空或者无法选择 **源租户** 信息，可能是因为 obd 部署的 OceanBase 数据库安装目录的 `bin` 目录下缺少 `ob_admin` 文件，可在官网 OceanBase 软件下载中心 下载 **工具集成包（OceanBase Utils）**，并执行如下命令解压获取，解压后将 `usr/bin/ob_admin` 文件复制到 OceanBase 数据库安装目录的 `bin` 目录下。

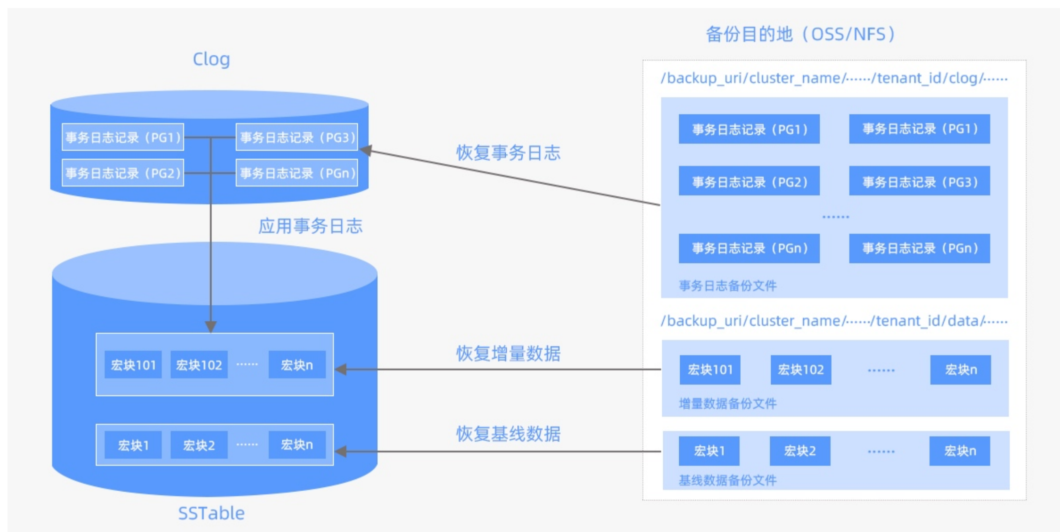
```
rpm2cpio oceanbase-ce-utils-<version> | cpio -idmv ./usr/bin/ob_admin
```

使用 OCP 进行数据备份的具体操作可参见官网《OceanBase 云平台》文档 备份恢复 章节。

数据恢复

OceanBase 数据库支持租户级别的恢复，恢复是基于已有数据的备份重建新租户的过程。恢复过程包括租户系统表和用户表的 Restore 和 Recover 过程。Restore 是将恢复需要的基线数据恢复到目标租户的 OBServer 节点，Recover 是将基线数据备份过程中产生的增量数据日志恢复到 OBServer 节点中。

OceanBase 数据库的物理恢复架构如下图所示。



使用 OCP 进行数据恢复前，需了解如下信息：

- 建议数据恢复的目标集群和源集群数据库版本保持一致。
- OceanBase 数据库当前仅支持将低版本的备份数据恢复到同版本或高版本中，但不支持 OceanBase 数据库 3.x 的备份数据恢复到 OceanBase 数据库 4.x 版本。
- 对于特殊版本如 OceanBase 数据库 4.0.x 版本的备份数据不支持恢复到 OceanBase 数据库 4.1.x 及之后版本。当前 4.2.x 版本备份的数据也无法恢复到 4.3.x 版本。
- 恢复界面中如果 **可恢复时间** 显示为空或者无法选择 **源租户** 信息，可能是因为 obd 部署的 OceanBase 数据库安装目录的 `bin` 目录下缺少 `ob_admin` 文件，可在官网 OceanBase 软件下载中心 下载 **工具集成包（OceanBase Utils）**，并执行如下命令解压获取，解压后将 `usr/bin/ob_admin` 文件复制到 OceanBase 数据库安装目录的 `bin` 目录下。

```
rpm2cpio oceanbase-ce-utils-<version> | cpio -idmv ./usr/bin/ob_admin
```

- OCP 上使用数据恢复有两种场景：
  - 如果通过新建备租户的方式进行备份恢复，目标租户默认会和源租户形成主备租户关系，同时会进行数据同步。若需要解除同步状态，可以进行主备租户解耦操作。新建备租户和主备租户解耦的具体操作可参见官网《OceanBase 云平台》文档 租户管理/管理租户/新建备租户 和 容灾管理/主备租户切换/主备租户解耦。
  - 如果选择直接使用备份文件的方式进行数据恢复，需要指定目标租户恢复时间点，并不会保持数据同步。

- 如果通过新建备租户的方式进行备份恢复，目标租户默认会和源租户形成主备租户关系，同时会进行数据同步。若需要解除同步状态，可以进行主备租户解耦操作。新建备租户和主备租户解耦的具体操作可参见官网《OceanBase 云平台》文档 租户管理/管理租户/新建备租户 和 容灾管理/主备租户切换/主备租户解耦。
- 如果选择直接使用备份文件的方式进行数据恢复，需要指定目标租户恢复时间点，并不会保持数据同步。

使用 OCP 进行数据恢复的具体操作可参见官网《OceanBase 云平台》文档 备份恢复 章节。

OCP 升级集群

使用 OCP 管理的集群主要有 OceanBase 集群和 OBProxy 集群，升级所管理的集群时可按以下三个流程进行。

升级 OCP

使用 OCP 升级集群之前需要先确认是否需要升级 OCP。可以通过版本号进行确认，OCP 服务在 V4.2.0 开始和 OceanBase 数据库版本进行对齐，需要确保 OCP 服务版本和要升级的集群目标版本保持一致，建议使用最新版 OCP。同时也可以通过 OCP 发布版本的 Release Notes 确认版本升级依赖关系。

目前 OCP 主要有两种部署方式，通过 obd 图形化界面部署 OCP 和通过容器部署 OCP，不同方式部署的 OCP 的升级操作也不一样。

- 使用 obd 升级 OCP

  升级前，需了解如下信息：

  - 仅支持 OCP V4.0.3 及之后版本通过该方式进行升级。
  - OCP 节点需要安装 JAVA 环境，目前仅支持 JDK1.8 版本，并要求构建版本号至少需为 161。
  - obd 可升级容器部署的 OCP，升级过程中会关闭该容器，并重新使用 RPM 包部署新的

OCP 服务，启用新的 OCP 服务后，建议不再使用容器方式管理该 OCP。

- 获取到需要升级的 OCP 软件包后，推荐使用 `obd web upgrade` 命令图形化界面升级方式。

- 仅支持 OCP V4.0.3 及之后版本通过该方式进行升级。
- OCP 节点需要安装 JAVA 环境，目前仅支持 JDK1.8 版本，并要求构建版本号至少需为 161。
- obd 可升级容器部署的 OCP，升级过程中会关闭该容器，并重新使用 RPM 包部署新的 OCP 服务，启用新的 OCP 服务后，建议不再使用容器方式管理该 OCP。
- 获取到需要升级的 OCP 软件包后，推荐使用 `obd web upgrade` 命令图形化界面升级方式。
- 使用容器升级 OCP

  该方法适用于升级通过容器部署的 OCP，也可以将 RPM 部署的 OCP，重新升级成容器版本，升级前需了解如下信息：

  - 升级前需要先停止容器。

  - 新启动的 OCP 容器名不能和升级前的容器名相同，否则会启动失败。

  - 如果需要将 RPM 部署的 OCP 升级成容器版本，需要先杀掉 OCP-SERVER 服务进程，通过拉取 OCP 镜像方式，使用原有的 MetaDB 租户信息进行连接配置，即可部署成容器版本。

- 升级前需要先停止容器。
- 新启动的 OCP 容器名不能和升级前的容器名相同，否则会启动失败。
- 如果需要将 RPM 部署的 OCP 升级成容器版本，需要先杀掉 OCP-SERVER 服务进程，通过拉取 OCP 镜像方式，使用原有的 MetaDB 租户信息进行连接配置，即可部署成容器版本。

使用 obd 升级 OCP 和使用容器升级 OCP 的具体操作可分别参见官网《OceanBase 云平台》文档 升级 OceanBase 云平台/升级社区版 OceanBase 云平台/使用图形化界面升级 OCP 和 升级 OceanBase 云平台/升级社区版 OceanBase 云平台/使用容器升级 OCP。

升级 OceanBase 集群

适用场景：版本缺陷修复、版本特性支持等。

使用 OCP 升级 OceanBase 集群前，需了解如下信息：

- 升级集群前，需要上传 3 个 OceanBase 软件包：数据库包（oceanbase-ce-*.rpm）、数据库依赖包（oceanbase-ce-libs-*.rpm）、数据库工具集成包（oceanbase-ce-utils-*.rpm）。
- 升级前需确认当前版本是否支持升级到目标版本，具体可以参看目标版本的发布记录中 **升级说明**。OceanBase 数据库的版本发布记录可访问 链接 查看。
- 升级集群前，集群状态需要保证健康正常的，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。
- 升级集群时，如果集群中的 Zone 个数大于等于 3，推荐选择 **滚动升级**，可在线升级。滚动升级前需要确保所有租户至少为 3 副本，否则会因为升级过程中租户无法保证多数派导致升级预检查无法通过。
- 不支持滚动升级单 Zone 集群环境，因为在升级过程中会停止数据库服务，影响业务使用，需要提前申请升级时间窗口。
- 升级集群时，重点关注升级任务完成 **Pre check for upgrade** 阶段后无法再对集群升级任务进行回滚，如果遇到任务失败，严禁手动 **跳过** 或 **回滚** 任务或任务 **设置为成功**，可能会造成集群不可逆损坏，危及业务生产，建议到官网 问答区 发帖，会有专业人员协助解决。

使用 OCP 升级 OceanBase 集群的具体操作可参见官网《OceanBase 云平台》文档 集群管理/升级版本。

升级 OBProxy 集群

适用场景：版本缺陷修复、版本特性支持、版本配套要求等。

使用 OCP 升级 OBProxy 集群前，需了解如下信息：

- 升级 OBProxy 集群时，会同时升级集群中所有 OBProxy，因此该 OBProxy 集群将停止对外服务。如当前 OBProxy 集群已承载业务，建议根据当前配置的负载均衡，分批升级集群中的 OBProxy 服务。

- 对存在多个 CPU 架构的 OBProxy 集群进行升级时，若某个 CPU 架构中缺少所选择的安装包版本，将无法进行升级。

使用 OCP 升级 OBProxy 集群的具体操作可参见官网《OceanBase 云平台》文档 OBProxy 管理/管理 OBProxy 集群/升级 OBProxy 集群。

OCP 扩缩容 OceanBase 集群及租户

OceanBase 有两种添加/缩减节点的场景：横向添加/缩减节点会增加/减少 Zone 的个数，以实现控制租户副本个数的目的；纵向添加/缩减节点会增加/减少 OBServer 节点的个数，以实现控制租户资源上限的目的。因此可以对租户副本数和资源使用限制进行调整，以满足不同业务场景下的容灾、性能、运维等要求。

OCP 扩容 OceanBase 集群及租户

通过 OCP 的扩容功能可拓展机房数量，提供更可靠的高可用能力或实现业务迁移机房。通过扩容可以在几乎不影响业务访问的情况下，在其他机房增加 OBServer 节点。

横向扩容

横向扩容通过新增 Zone 实现，添加 Zone 时会在该 Zone 下新增 OBServer 服务，从而增加集群的副本数量，但不能增加集群的资源（CPU、内存、磁盘容量）水平使用上限，例如：数据磁盘存储数据上限不会提升、单 Zone 下的整体资源不会提升。

适用场景：单机扩容成分布式集群或者少 Zone 集群扩容成多 Zone 集群以提高容灾能力。

1. 新增 Zone

   使用 OCP 新增 Zone 前，需了解如下信息：

   - 扩容 Zone 前需确认 **软件包管理** 中是否包含 OceanBase 数据库的三个安装包：数据库包（oceanbase-ce-*.rpm）、数据库依赖包（oceanbase-ce-libs-*.rpm）、数据库工具集成包（oceanbase-ce-utils-*.rpm）。

   - 扩容 Zone 前需先添加主机，主机节点建议先进行服务器初始化，具体操作可参见本教

程第二章 部署前准备。

- 扩容 Zone 前，集群需处于 **运行中** 状态，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。

- 扩容任务失败时支持回滚，若回滚失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

- 扩容 Zone 完成后，如果 Primary Zone 是顺序优先级策略，不会增加集群计算性能。如果 Primary Zone 是随机策略，一般会对分区表的性能有一些影响，具体需要看业务使用场景。例如：对分区表而言扩容后分区会打散到更多节点，每个节点分区数减少，计算速度提升，但跨了更多个节点数可能增加网络 I/O 等。

使用 OCP 新增 Zone 的具体操作可参见官网《OceanBases 云平台》文档 集群管理/管理 Zone/新增 Zone。

2. 扩容 Zone 前需确认 **软件包管理** 中是否包含 OceanBase 数据库的三个安装包：数据库包（oceanbase-ce-*.rpm）、数据库依赖包（oceanbase-ce-libs-*.rpm）、数据库工具集成包（oceanbase-ce-utils-*.rpm）。
3. 扩容 Zone 前需先添加主机，主机节点建议先进行服务器初始化，具体操作可参见本教程第二章 部署前准备。
4. 扩容 Zone 前，集群需处于 **运行中** 状态，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。
5. 扩容任务失败时支持回滚，若回滚失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。
6. 扩容 Zone 完成后，如果 Primary Zone 是顺序优先级策略，不会增加集群计算性能。如果 Primary Zone 是随机策略，一般会对分区表的性能有一些影响，具体需要看业务使用场景。例如：对分区表而言扩容后分区会打散到更多节点，每个节点分区数减少，计算速度提升，但跨了更多个节点数可能增加网络 I/O 等。
7. 新增租户副本

扩容 Zone 完成后并不会自动为租户添加上副本，需要进入 **租户管理** 页面点选所需的租户，进行新增副本操作，具体操作步骤可参见官网《OceanBase 云平台》文档 租户管理/管理副本/新增副本。

使用 OCP 新增租户副本前，需了解如下信息：

1. OceanBase 数据库为 V4.0.0 及以上、V4.2.0 以下版本时，仅支持新增全能型副本，即 F 副本。

2. OceanBase 数据库为 V4.2.0 及以上版本时，仅支持新增全能型副本和只读副本，即 F 副本和 R 副本。

3. 增加 Unit 规格时，建议保持租户的日志盘大小为租户内存大小的 3~4 倍。

8. OceanBase 数据库为 V4.0.0 及以上、V4.2.0 以下版本时，仅支持新增全能型副本，即 F 副本。
9. OceanBase 数据库为 V4.2.0 及以上版本时，仅支持新增全能型副本和只读副本，即 F 副本和 R 副本。
10. 增加 Unit 规格时，建议保持租户的日志盘大小为租户内存大小的 3~4 倍。

使用 OCP 横向扩容的整体操作可参见官网《OceanBase 云平台》文档 运维最佳实践/扩展 OceanBase 集群及租户的高可用。

纵向扩容

纵向扩容通过在已有的 Zone 中添加 OBServer 节点实现，可增加 Zone 的资源（CPU、内存、磁盘容量）使用上限，并提升集群计算能力，例如：1-1-1 架构集群纵向扩容成 2-2-2 架构，单 Zone 下的整体资源会提升，计算能力会提升。

适用场景：集群性能达到瓶颈、数据存储达到阀值。

1. 新增 OBServer 节点

   使用 OCP 新增 OBServer 节点前，需了解如下信息：

   - 扩容 OBServer 节点前，扩容节点的主机需要和集群硬件架构的主机一致，例如：CPU

架构。

- 扩容 OBServer 节点前，需要先添加主机，主机节点建议先进行服务器初始化，并确保待扩容节点和源集群服务器资源配置保持一致，具体操作可参见本教程第二章 部署前准备。

> **说明**
>
> OCP 添加 OBServer 节点时会对服务器节点做一些初始化动作，此处建议手动按源集群服务器初始化过程进行配置，是为了确保万无一失。

- 扩容 OBServer 节点前需确认 **软件包管理** 中是否包含 OceanBase 数据库的三个安装包：数据库包（oceanbase-ce-*.rpm）、数据库依赖包（oceanbase-ce-libs-*.rpm）、数据库工具集成包（oceanbase-ce-utils-*.rpm）。

- 扩容 OBServer 节点前，集群需处于 **运行中** 状态，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。

- 扩容 OBServer 节点时，扩容任务失败支持回滚，如果回滚失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

使用 OCP 新增 OBServer 节点的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理 OBServer/添加 OBServer 节点。

2. 扩容 OBServer 节点前，扩容节点的主机需要和集群硬件架构的主机一致，例如：CPU 架构。
3. 扩容 OBServer 节点前，需要先添加主机，主机节点建议先进行服务器初始化，并确保待扩容节点和源集群服务器资源配置保持一致，具体操作可参见本教程第二章 部署前准备。

> **说明**
>
> OCP 添加 OBServer 节点时会对服务器节点做一些初始化动作，此处建议手动按源集群服务器初始化过程进行配置，是为了确保万无一失。

4. 扩容 OBServer 节点前需确认 **软件包管理** 中是否包含 OceanBase 数据库的三个安装包：数据库包（oceanbase-ce-*.rpm）、数据库依赖包（oceanbase-ce-libs-*.rpm）、数据库工具集成包（oceanbase-ce-utils-*.rpm）。
5. 扩容 OBServer 节点前，集群需处于 **运行中** 状态，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。
6. 扩容 OBServer 节点时，扩容任务失败支持回滚，如果回滚失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。
7. 扩容租户 Unit 数量

   扩容 OBServer 节点后，租户的数据并不会自动均衡到新节点，需要进入 **租户管理** 页面点选所需的租户，进行修改 Unit 操作，具体操作可参见官网《OceanBase 云平台》文档 租户管理/管理副本/编辑 Zone 中的租户副本。修改 Unit 操作主要是通过增加租户可使用同 Zone 下的 OBServer 节点数，在新扩容节点上生成同样配置的 Unit 资源池，将同一 Zone 下的租户数据分区均衡到新节点。

   使用 OCP 扩容租户 Unit 数量前，需了解如下信息：

   - OceanBase 数据库 V4.0 及之后版本，要求租户中每个 Zone 下的 Unit 数量保持一致。例如：集群架构是 2-2-1 或 2-1-1 时，不允许租户的 Unit 设置为 2，只有 2-2-2 架构下，租户的 Unit 才能设置为 2。

   - 当普通租户所属集群为 V4.2.0 及以上版本，且租户参数 `enable_rebalance` 为 `false` 的情况下，无法修改 Zone 的 Unit 数量。

   - 扩容租户的 Unit 数量不能大于所在 Zone 内 OBServer 节点的数量。

   - 扩容租户 Unit 前，集群与租户需为 **运行中** 状态。可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态；单击左侧导航栏的 **租户** 进入 **租户** 页面，在 **租户列表** 中查看对于租户状态。

   - 租户扩容 Unit 时，如果扩容任务失败，不推荐进行任务回滚，回滚任务时租户将不会自动缩容回原先规格，也不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区

发帖，会有专业人员协助解决。

8. OceanBase 数据库 V4.0 及之后版本，要求租户中每个 Zone 下的 Unit 数量保持一致。例如：集群架构是 2-2-1 或 2-1-1 时，不允许租户的 Unit 设置为 2，只有 2-2-2 架构下，租户的 Unit 才能设置为 2。
9. 当普通租户所属集群为 V4.2.0 及以上版本，且租户参数 `enable_rebalance` 为 `false` 的情况下，无法修改 Zone 的 Unit 数量。
10. 扩容租户的 Unit 数量不能大于所在 Zone 内 OBServer 节点的数量。
11. 扩容租户 Unit 前，集群与租户需为 **运行中** 状态。可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态；单击左侧导航栏的 **租户** 进入 **租户** 页面，在 **租户列表** 中查看对于租户状态。
12. 租户扩容 Unit 时，如果扩容任务失败，不推荐进行任务回滚，回滚任务时租户将不会自动缩容回原先规格，也不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

使用 OCP 纵向扩容的整体操作可参见官网《OceanBase 云平台》文档 最佳运维实践/扩容 OceanBase 集群及租户。

OCP 缩容 OceanBase 集群及租户

横向缩容

横向缩容主要指减少租户的副本数量并删除对应 Zone 及 Zone 下的 OBServer 节点，适用于需要腾出硬件资源，或集群维护需要进行 Zone 缩容的场景。

1. 删除副本

   使用 OCP 删除副本前，需了解如下信息：

   - 删除副本前，OceanBase 集群、租户必须为 **运行中** 状态，其他状态不允许进行操作。可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态；单击左侧导航栏的 **租户** 进入 **租户** 页面，在 **租户列表** 中查看对于租户状态。

- 删除副本前需确保删除副本后租户的副本能满足多数派。即 3 副本租户最多只能删除 1 个副本。

- 删除副本前，建议主动将主副本切走，即修改租户的 Primary Zone 优先级，确保副本删除时不影响业务，具体操作可参见官网《OceanBase 云平台》文档 租户管理/修改 Zone 优先级。

使用 OCP 删除副本的具体操作可参见官网《OceanBase 云平台》文档 租户管理/管理副本/删除 Zone 中的租户副本。

2. 删除副本前，OceanBase 集群、租户必须为 **运行中** 状态，其他状态不允许进行操作。可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态；单击左侧导航栏的 **租户** 进入 **租户** 页面，在 **租户列表** 中查看对于租户状态。
3. 删除副本前需确保删除副本后租户的副本能满足多数派。即 3 副本租户最多只能删除 1 个副本。
4. 删除副本前，建议主动将主副本切走，即修改租户的 Primary Zone 优先级，确保副本删除时不影响业务，具体操作可参见官网《OceanBase 云平台》文档 租户管理/修改 Zone 优先级。
5. 删除 Zone

使用 OCP 删除 Zone 前，需了解如下信息：

- 缩容 Zone 前，集群状态需要处于 **运行中**，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。

- 缩容 Zone 前，需要保证 Zone 上已无租户副本，即需要先进行上一步删除副本的操作。

- 缩容 Zone 时，扩容任务失败支持回滚，如果回滚失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

使用 OCP 删除 Zone 的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理

Zone/删除 Zone。

6. 缩容 Zone 前，集群状态需要处于 **运行中**，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。
7. 缩容 Zone 前，需要保证 Zone 上已无租户副本，即需要先进行上一步删除副本的操作。
8. 缩容 Zone 时，扩容任务失败支持回滚，如果回滚失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

使用 OCP 横向缩容的整体操作可参见官网《OceanBase 云平台》文档 运维最佳实践/缩减 OceanBase 集群及租户的高可用。

纵向缩容

纵向缩容通过减少已有 Zone 中的 OBServer 节点实现，您需先执行缩减租户 Unit 数量后再删除对应的 OBServer 节点。

1. 缩减租户 Unit 数量

   当集群需要缩容 OBServer 节点，或腾出集群资源供其他租户使用时，可通过减少租户 Unit 数量来减少租户同 Zone 下可使用的 OBServer 节点数。该过程会随机选取同 Zone 下的 OBServer 节点，并将该节点的数据迁移到同 Zone 其他正在使用的 OBServer 节点上，删除缩容节点上的 Unit 资源池，释放资源（CPU、内存等）。因为数据盘和事务日志盘是预占用空间，不会影响该节点磁盘预占用空间大小。

   使用 OCP 缩减租户 Unit 数量前，需了解如下信息：

   - 缩容租户 Unit 前，集群与租户需为 **运行中** 状态。可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态；单击左侧导航栏的 **租户** 进入 **租户** 页面，在 **租户列表** 中查看对于租户状态。

   - 缩容租户 Unit 前，建议对租户进行合并，释放租户正在使用的内存，提高缩容效率。执行合并的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群合并/执行合并。

   - 缩容租户 Unit 前，需要确保剩下的 OBServer 节点磁盘剩余资源足够支撑缩容租户的

数据。

- 如果需要对指定的 OBServer 节点进行缩容，需要手动迁移 Unit，具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群资源/查看 Unit 分布。

- 当普通租户所属集群为 V4.2.0 及以上版本，且租户参数 `enable_rebalance` 为 `false` 的情况下，无法修改 Zone 的 Unit 数量。

- 在多 Zone 集群场景下，OceanBase 数据库为 V4.0.0 或以上版本时，不支持减少单个 Zone 的 Unit 数量，且副本类型仅支持全能型副本类型。例如：2-2-2 架构集群或租户，租户只能缩容成 1-1-1 架构，不能缩容成 2-1-1 或 2-2-1 架构。

- 缩容租户时，如果缩容任务失败，不推荐进行任务回滚，也不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

通过 OCP 缩减租户 Unit 数量的具体操作可参见官网《OceanBase 云平台》文档 租户管理/管理副本/编辑 Zone 中的租户副本。

2. 缩容租户 Unit 前，集群与租户需为 **运行中** 状态。可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态；单击左侧导航栏的 **租户** 进入 **租户** 页面，在 **租户列表** 中查看对于租户状态。
3. 缩容租户 Unit 前，建议对租户进行合并，释放租户正在使用的内存，提高缩容效率。执行合并的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群合并/执行合并。
4. 缩容租户 Unit 前，需要确保剩下的 OBServer 节点磁盘剩余资源足够支撑缩容租户的数据。
5. 如果需要对指定的 OBServer 节点进行缩容，需要手动迁移 Unit，具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理集群资源/查看 Unit 分布。
6. 当普通租户所属集群为 V4.2.0 及以上版本，且租户参数 `enable_rebalance` 为 `false` 的情况下，无法修改 Zone 的 Unit 数量。

7. 在多 Zone 集群场景下，OceanBase 数据库为 V4.0.0 或以上版本时，不支持减少单个 Zone 的 Unit 数量，且副本类型仅支持全能型副本类型。例如：2-2-2 架构集群或租户，租户只能缩容成 1-1-1 架构，不能缩容成 2-1-1 或 2-2-1 架构。
8. 缩容租户时，如果缩容任务失败，不推荐进行任务回滚，也不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。
9. 删除 OBServer 节点

   在已有的 Zone 中删除 OBServer 节点，释放出服务器资源，例如：2-2-2 架构集群纵向缩容成 1-1-1 或 2-1-1 或 2-2-1 架构，建议缩容时尽量在 Zone 中删除相同数量的 OBServer 节点，以保证 Zone 之间的资源均等，即推荐缩容成 N-N-N 集群架构(N 值保持一致)。

   使用 OCP 删除 OBServer 节点前，需了解如下信息：

   - 缩容 OBServer 节点前，集群需为 **运行中** 状态，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。

   - 缩容 OBServer 节点前，需要确认待缩容的 OBServer 节点的租户已经完成 Unit 迁移，可在任务中查看对应任务确认。

   - 缩容 OBServer 节点时，如果缩容任务失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

   使用 OCP 删除 OBServer 节点的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理 OBServer/删除 OBServer 节点。
10. 缩容 OBServer 节点前，集群需为 **运行中** 状态，可单击左侧导航栏的 **集群** 进入 **集群** 页面，在 **集群列表** 中查看对应集群状态。
11. 缩容 OBServer 节点前，需要确认待缩容的 OBServer 节点的租户已经完成 Unit 迁移，可在任务中查看对应任务确认。
12. 缩容 OBServer 节点时，如果缩容任务失败，不建议对任务进行 **跳过** 或 **设置为成功**，建议到官网 问答区 发帖，会有专业人员协助解决。

使用 OCP 纵向缩容的整体操作可参见官网《OceanBase 云平台》文档 运维最佳实践/缩容 OceanBase 集群及租户。

OCP 替换 OBServer 节点

OCP 替换功能适用于大于等于 3 副本的集群中出现故障（服务器无法启动、硬件故障等）OBServer 节点的场景，或者部署大于等于 3 副本的集群时参数设置不合理但又不能整体重装解决时的场景，例如：数据磁盘预占用太大，可通过替换操作快速解决节点故障的潜在危险，实现集群的高可用能力、业务的稳定性和运维的高效性。

有替换服务器

适用场景：有空闲的服务器储备，集群节点因外部因素故障短期无法修复等。

有替换服务器的情况下，使用 OCP 替换 OBServer 节点前，需了解如下信息：

- 需要集群当前是可用状态，即通过集群能正常访问租户。
- 空闲服务器需要和当前集群硬件架构匹配，且该主机需与原来的主机在同一机房内。
- 空闲服务器软硬件资源（CPU、内存、磁盘等）需要大于或等于故障节点，特别是磁盘容量。
- 替换服务器前，需要先添加主机，主机节点建议先进行服务器初始化，具体操作可参见本教程第二章 部署前准备。
- 替换服务器前，**软件包管理** 中需要有 OceanBase 数据库、OCP-agent 的软件包，且版本需要和源集群保持一致。
- 替换服务器时，如果故障节点 OBServer 节点已经 **离线** 状态，替换过程中需要按提示选择跳过主机运维操作。
- 替换服务器时，如果替换任务如果失败，不建议对任务进行 **跳过** 或 **设置为成功** 操作，建议到官网 问答区 发帖，会有专业人员协助解决。
- 因为替换功能仅针对 OBServer 服务，如果原节点部署过 OBProxy 服务，替换完成后，还需要单独在替换节点上重新部署 OBProxy 服务。添加 OBProxy 的具体操作可参见官网《OceanBase 云平台》文档 运维最佳实践/扩容 OBProxy 集群。

使用 OCP 替换 OBServer 节点的具体操作可参见官网《OceanBase 云平台》文档 运维最佳实践/OceanBase 集群主机故障处理。

无替换服务器

当没有替换的服务器时，服务器的故障问题可能存在隐藏的风险，例如服务故障后集群可能处于 **运维中** 状态，此时 OCP 对集群做某些运维操作会因为集群状态不满足而失败。该场景下如果必须使用这些运维操作时，例如升级集群，不同的应急恢复后会可能带来一些使用风险，比如：应急操作踢出故障节点后将不再保证高可用，或者集群性能会降低等。

适用场景：故障节点 observer 服务进程在线，能接受业务性能或集群高可用风险。

无替换服务器的情况下，可考虑如下方法处理故障 OBServer 节点：

- 如果短时间可以维修好服务器故障，可以先调整永久下线时间参数 `server_permanent_offline_time`，再停止 observer 进程，永久下线时间需要按服务器维修好重新上线时间评估。
- 如果短时间无法维修好服务器故障，且集群是多 Zone 多 OBServer 节点集群，例如：2-2-2架构集群，可以考虑手动迁移 Unit 到同 Zone 下的其他节点（前提是剩余的 OBServer 节点有足够的资源容纳迁移的 Unit），再删除故障 OBServer 节点。
- 如果短时间无法维修好服务器故障，集群是多 Zone 单 OBServer 节点集群，可以考虑缩容集群方案。

使用 OCP 替换 OBServer 节点的具体操作可参见官网《OceanBase 云平台》文档 运维最佳实践/OceanBase 集群主机故障处理。

5.2 使用 obd 进行运维

obd 作为开源社区 OceanBase 的生态组件安装部署工具，同时也支持部分部署服务的管理能力，例如部署组件的重启、升级、扩容、诊断等，接下来将从常用的运维动作中介绍如何使用 obd 进行运维。

> **说明**
>
> 本教程中所贴官网文档链接版本均为编写教程时的最新版本或长期支持版本（LTS），若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

可执行如下命令查看 obd 一级目录的基础信息，这些信息可以帮助我们了解 obd 在管理和运维时的运作方式和注意事项。

```
[admin@test001 ~]$ tree -L 1 ~/.obd
```

输出如下：

```
/home/admin/.obd
├── cluster         # 存放部署集群名称对应配置文件，不建议直接编辑该配置文件，如需修改集群配置可使用 obd cluster edit-config 命令操作。该目录下存在 .data 隐藏文件，记录部署名称、组件相关信息、部署状态信息。不建议直接编辑该配置文件。
├── config_parser
├── lock
├── log             # 存放 obd 命令操作的日志。
├── mirror          # 存放本地仓库的安装包和远程仓库的 repo 文件。
├── optimize
├── plugins         # 存放组件相关插件。
├── repository
└── version         # 记录 obd 版本。
```

使用 obd 调整 OceanBase 集群参数

obd 中可通过 `obd cluster edit-config` 命令修改部署配置文件，以实现对 OceanBase 集群系统参数的调整。可执行如下命令查看 `parameter.yaml` 文件，获取 obd 支持修改的 OceanBase 集群参数。

```
find ~/.obd/plugin/${component} "parameter.yaml"
```

这里的 `${component}` 需要替换为待查看的组件名，本节中可替换为 `oceanbase-ce` 查看 OceanBase 数据库的参数。

`parameter.yaml` 文件中存在如下三种修改生效方式，具体生效的方式也可以查看修改配置文件后黑屏打印的执行生效提示。

- `need_redeploy` 为 `true` 表示对应参数需要执行 `obd cluster redeploy` 命令生效。该命令会卸载服务，清理数据并重新部署新的集群，请谨慎操作。
- `need_restart` 为 `true` 表示对应参数需要执行 `obd cluster restart` 命令重启生效。该命令会重启部署服务，生产环境需要注意业务中断影响。
- `need_reload` 为 `true` 表示对应参数需要执行 `obd cluster reload` 命令生效，可放心修改。

接下来将通过修改 OceanBase 集群集群资源参数并扩容用户租户资源的示例来介绍通用的 obd 运维操作步骤。

1. 查看服务列表信息

   ```
   [admin@test001 ~]$ obd cluster list
   ```

   输出如下：

   ```
   +----------------------------------------------------------+
   |                       Cluster List                       |
   +------+--------------------------------+------------------+
   | Name | Configuration Path             | Status (Cached)  |
   +------+--------------------------------+------------------+
   | test | /home/admin/.obd/cluster/test  | running          |
   +------+--------------------------------+------------------+
   ```

2. 修改配置文件

   部署配置文件中包含的参数，优先执行 `obd cluster edit-config` 命令修改；部署配置文件中没有包含的参数，可以登录数据库执行 SQL 命令（`alter system set参数='参数值';`）修改。

   ```
   [admin@test001 ~]$ obd cluster edit-config test
   ```

执行命令后会打开配置文件，在配置文件中找到 oceanbase-ce 模块，内容示例如下：

```
oceanbase-ce:
  servers:
    - 10.10.10.1
  global:
    home_path: /home/admin/oceanbase
    data_dir: /obdata/data
    redo_dir: /obdata/log
    devname: eth0
    mysql_port: 2881
    rpc_port: 2882
    zone: zone1
    cluster_id: 1
    memory_limit: 32G
    system_memory: 5G
    datafile_size: 100G
    datafile_next: 0G
    datafile_maxsize: 0G
    log_disk_size: 100G
    cpu_count: 16
    production_mode: false
    enable_syslog_wf: false
    enable_syslog_recycle: true
    max_syslog_file_count: 10
    root_password: ******
```

修改配置文件中 `memory_limit` 参数为 `50G`、`system_memory` 参数为 `10G`、`log_disk_size` 参数为 `150G`，修改后执行 `:x` 保存配置文件。

调整集群资源相关参数时，需注意调大参数数值时，不能超出服务器剩余资源范围；调小参数数值时，不能低于租户已分配占用的数值范围。资源参数的调整之间有关联关系，具体可参见 部署前准备 中 **OceanBase 数据库常用资源参数简介及计算方式** 一节。

> **注意**
>
> `datafile_size` 参数不支持调小。

3. 重载配置

配置文件修改并保存后，obd 会输出对应的重载命令，输出内容示例如下：

```
Search param plugin and load ok
Search param plugin and load ok
Parameter check ok
```

```
Save deploy "test" configuration
Use `obd cluster reload test` to make changes take effect.
Trace ID: 12ece7da-f32d-11ee-ae7a-00163e046d79
If you want to view detailed obd logs, please run: obd display-trace 12ece7da-f32d
-11ee-ae7a-00163e046d79
```

执行上述输出内容中的 `obd cluster reload test` 命令即可使修改生效。

```
[admin@test001 ~]$ obd cluster reload test
```

执行后输出如下：

```
Get local repositories and plugins ok
Load cluster param plugin ok
Open ssh connection ok
Cluster status check ok
Connect to observer 10.10.10.1:2881 ok
test reload
Trace ID: 70529604-f32d-11ee-a216-00163e046d79
If you want to view detailed obd logs, please run: obd display-trace 70529604-f32d
-11ee-a216-00163e046d79
```

4. 验证修改

使用 root 用户登录 OceanBase 数据库的 `sys` 租户，执行如下命令查看对应参数是否有变动，此处以查看 `memory_limit` 为例。

```
obclient [oceanbase]> show parameters like 'memory_limit';
```

输出如下：

```
+--------+----------+----------------+----------+--------------+-----------+-------
+--------------------------------------------------------------------------------
--------------------------------------------+----------+---------+----------+--
-----------------+
| zone   | svr_type | svr_ip         | svr_port | name         | data_type | value
| inf
o
                                            | section  | scope   | source   | edit_l
evel         |
+--------+----------+----------------+----------+--------------+-----------+-------
+--------------------------------------------------------------------------------
--------------------------------------------+----------+---------+----------+--
-----------------+
| zone1  | observer | 10.10.10.1     |     2882 | memory_limit | NULL      | 50G
```

```
| the size of the memory reserved for internal use(for testing purpose), 0 means f
ollow memory_limit_percentage. Range: 0, [1G,). | OBSERVER | CLUSTER | DEFAULT | D
YNAMIC_EFFECTIVE |
+-------+----------+---------------+----------+-------------+----------+-------
+-------------------------------------------------------------------------------
-----------------------------------------------+----------+---------+---------+--
-----------------+
```

5. 确认集群可用资源变化

```
obclient [oceanbase]> select zone,svr_ip,svr_port,cpu_capacity,cpu_assigned_max,cp
u_capacity-cpu_assigned_max as cpu_free,round(memory_limit/1024/1024/1024,2) as me
mory_total_gb,round((memory_limit-mem_capacity)/1024/1024/1024,2) as system_memory
_gb,round(mem_assigned/1024/1024/1024,2) as mem_assigned_gb,round((mem_capacity-me
m_assigned)/1024/1024/1024,2) as memory_free_gb,round(log_disk_capacity/1024/1024/
1024,2) as log_disk_capacity_gb,round(log_disk_assigned/1024/1024/1024,2) as log_d
isk_assigned_gb,round((log_disk_capacity-log_disk_assigned)/1024/1024/1024,2) as l
og_disk_free_gb,round((data_disk_capacity/1024/1024/1024),2) as data_disk_gb,round
((data_disk_in_use/1024/1024/1024),2) as data_disk_used_gb,round((data_disk_capaci
ty-data_disk_in_use)/1024/1024/1024,2) as data_disk_free_gb from gv$ob_servers;
```

输出如下：

```
+-------+----------------+----------+--------------+------------------+----------+
-----------------+------------------+-----------------+----------------+----------
------------+----------------------+------------------+--------------+------------
-------+-------------------+
| zone  | svr_ip          | svr_port | cpu_capacity | cpu_assigned_max | cpu_free
| memory_total_gb | system_memory_gb | mem_assigned_gb | memory_free_gb | log_disk
_capacity_gb | log_disk_assigned_gb | log_disk_free_gb | data_disk_gb | data_disk_
used_gb | data_disk_free_gb |
+-------+----------------+----------+--------------+------------------+----------+
-----------------+------------------+-----------------+----------------+----------
------------+----------------------+------------------+--------------+------------
-------+-------------------+
| zone1 | 10.10.10.1      |     2882 |           16 |               16 |        0
|           50.00 |            10.00 |           27.00 |          13.00
|               150.00 |               100.00 |            50.00 |       100.00
|              0.04 |             99.96 |
+-------+----------------+----------+--------------+------------------+----------+
-----------------+------------------+-----------------+----------------+----------
------------+----------------------+------------------+--------------+------------
-------+-------------------+
```

6. 查看集群租户资源已分配情况

```
select t4.tenant_id,t4.tenant_name,t1.name resource_pool_name, t1.unit_count, t2.`
name` unit_config_name, t2.max_cpu, t2.min_cpu, round(t2.memory_size/1024/1024/102
```

```
4,2) mem_size_gb, round(t2.log_disk_size/1024/1024/1024,2) log_disk_size_gb, t2.ma
x_iops, t2.min_iops, t3.unit_id, t3.zone, concat(t3.svr_ip,':',t3.`svr_port`) obse
rver from dba_ob_resource_pools t1 join dba_ob_unit_configs t2 on (t1.unit_config_
id=t2.unit_config_id) join dba_ob_units t3 on (t1.`resource_pool_id` = t3.`resourc
e_pool_id`) left join dba_ob_tenants t4 on (t1.tenant_id=t4.tenant_id) order by t4
.tenant_name,t3.zone;
```

输出如下：

```
+-----------+-------------+------------------+-----------+-----------------+--
-------+---------+------------+------------------+---------------------+---------
------------+---------+-------+---------------------+
| tenant_id | tenant_name | resource_pool_name | unit_count | unit_config_name | m
ax_cpu | min_cpu | mem_size_gb | log_disk_size_gb | max_iops            | min_iop
s            | unit_id | zone   | observer            |
+-----------+-------------+------------------+-----------+-----------------+--
-------+---------+------------+------------------+---------------------+---------
------------+---------+-------+---------------------+
|         1 | sys         | sys_pool           |          1 | sys_unit_config
|       3 |       3 |        2.00 |             4.00 | 9223372036854775807 | 92233
72036854775807 |       1 | zone1 | 10.10.10.1:2882     |
|      1002 | test_tenant | test_tenant_pool   |          1 | test_tenant_unit
|      13 |      13 |       25.00 |            96.00 | 9223372036854775807 | 92233
72036854775807 |    1001 | zone1 | 10.10.10.1:2882     |
+-----------+-------------+------------------+-----------+-----------------+--
-------+---------+------------+------------------+---------------------+---------
------------+---------+-------+---------------------+
```

7. 按需调大用户租户 test_tenant 资源规格

说明

如果集群资源扩容较大，建议 sys 租户（非 system_memory 参数）也进行资源扩容。因为 sys 租户承载了 Root Service 和其他关键系统进程，防止运行缓慢影响整个集群的响应时间和处理能力。

此处以将租户内存由 `25G` 调整为 `35G`，租户事务日志空间由 `96G` 调整为 `105G` 为例。

```
obclient [oceanbase]> alter resource unit test_tenant_unit MEMORY_SIZE '35G',LOG_D
ISK_SIZE '105G';
```

调整租户资源参数时，需要遵守原则。

- 调大参数数值时，不能超出集群剩余资源范围。

- 调小参数数值时，租户内存最小不能低于 `__min_full_resource_pool_memory` 限制，可使用 root 用户登录 OceanBase 集群的 sys 租户，执行如下命令查看 `__min_full_resource_pool_memory` 参数。

```
select * from oceanbase.GV$OB_PARAMETERS where name ='__min_full_resourc
e_pool_memory';
```

- 调小参数数值时，租户 CPU 最小不能低于 1 核，`MAX_CPU` 取值不能小于 `MIN_CPU`。
- 调小参数数值时，租户 `LOG_DISK_SIZE` 参数建议是租户内存大小的 3~4 倍。

8. 调大参数数值时，不能超出集群剩余资源范围。
9. 调小参数数值时，租户内存最小不能低于 `__min_full_resource_pool_memory` 限制，可使用 root 用户登录 OceanBase 集群的 sys 租户，执行如下命令查看 `__min_full_resource_pool_memory` 参数。

```
select * from oceanbase.GV$OB_PARAMETERS where name ='__min_full_resource_pool_mem
ory';
```

10. 调小参数数值时，租户 CPU 最小不能低于 1 核，`MAX_CPU` 取值不能小于 `MIN_CPU`。
11. 调小参数数值时，租户 `LOG_DISK_SIZE` 参数建议是租户内存大小的 3~4 倍。
12. 重复查看集群可用资源变化情况

```
obclient [oceanbase]> select zone,svr_ip,svr_port,cpu_capacity,cpu_assigned_max,cp
u_capacity-cpu_assigned_max as cpu_free,round(memory_limit/1024/1024/1024,2) as me
mory_total_gb,round((memory_limit-mem_capacity)/1024/1024/1024,2) as system_memory
_gb,round(mem_assigned/1024/1024/1024,2) as mem_assigned_gb,round((mem_capacity-me
m_assigned)/1024/1024/1024,2) as memory_free_gb,round(log_disk_capacity/1024/1024/
1024,2) as log_disk_capacity_gb,round(log_disk_assigned/1024/1024/1024,2) as log_d
isk_assigned_gb,round((log_disk_capacity-log_disk_assigned)/1024/1024/1024,2) as l
og_disk_free_gb,round((data_disk_capacity/1024/1024/1024),2) as data_disk_gb,round
((data_disk_in_use/1024/1024/1024),2) as data_disk_used_gb,round((data_disk_capaci
ty-data_disk_in_use)/1024/1024/1024,2) as data_disk_free_gb from gv$ob_servers;
```

输出如下:

```
+--------+----------------+----------+--------------+------------------+----------
-----------------+------------------+-----------------+----------------+----------
------------+---------------------+-----------------+--------------+------------
```

```
-------+-------------------+
| zone  | svr_ip          | svr_port | cpu_capacity | cpu_assigned_max | cpu_free
| memory_total_gb | system_memory_gb | mem_assigned_gb | memory_free_gb | log_disk
_capacity_gb | log_disk_assigned_gb | log_disk_free_gb | data_disk_gb | data_disk_
used_gb | data_disk_free_gb |
+-------+----------------+----------+--------------+------------------+----------+
----------------+------------------+-----------------+----------------+----------
------------+----------------------+------------------+--------------+------------
-------+-------------------+
| zone1 | 10.10.10.1      |     2882 |           16 |               16 |        0
|           50.00 |            10.00 |           37.00 |           3.00
|              150.00 |                109.00 |             41.00 |       100.00
|             0.10 |            99.90 |
+-------+----------------+----------+--------------+------------------+----------+
----------------+------------------+-----------------+----------------+----------
------------+----------------------+------------------+--------------+------------
-------+-------------------+
```

使用 obd 重启管理的服务

obd 可使用 `obd cluster stop` 和 `obd cluster start` 命令，或者使用 `obd cluster restart` 命令来重启部署的服务组件，同时也可以指定特定组件、特定节点的组件进行重启。详细的命令介绍可参见官网《OceanBase 安装部署工具》文档 obd 命令/集群命令组 中对应命令。

本节以重启 3 节点混合部署的 OBProxy 集群和 OceanBase 集群环境服务为例，介绍如何使用 obd 进行重启。

重启集群

```
[admin@@test001 /home/admin]$ obd cluster restart test2
```

执行命令前需了解如下内容：

- 重启组件的前提是所有组件涉及的节点 SSH 已连通，否则即使指定 IP 节点也无法进行重启操作。
- 所有组件均重启完成后，执行 `obd cluster list` 命令查看对应部署名称的状态才会是 `running`。如果存在组件重启失败，后续组件不会再进行重启动作。

> **注意**
>
> `obd cluster list` 中部署名称的状态非实时获取，当服务器发生重启时，可能会出现组件的服务不正常，而部署状态却为 `running` 的情况，因此建议服务器重启后执行 `obd cluster display` 命令确认所有组件是否正常。

- 部署名称状态如果非 `running`，会影响后续管理操作，例如：升级、扩容等。

重启命令的输出如下，展示集群中所有组件的状态。

```
Get local repositories and plugins ok
Load cluster param plugin ok
Open ssh connection ok
Cluster status check ok
Check before restart observer ok
Connect to observer ok
Server check ok
Observer restart ok
Wait for observer init ok
+---------------------------------------------------+
|                      observer                     |
+----------------+---------+------+-------+--------+
| ip             | version | port | zone  | status |
+----------------+---------+------+-------+--------+
| 10.10.10.1     | 4.2.1.2 | 2881 | zone1 | ACTIVE |
| 10.10.10.2     | 4.2.1.2 | 2881 | zone3 | ACTIVE |
| 10.10.10.3     | 4.2.1.2 | 2881 | zone2 | ACTIVE |
+----------------+---------+------+-------+--------+
obclient -h10.10.10.1 -P2881 -uroot -p'******' -Doceanbase -A

Check before start obproxy ok
Stop obproxy ok
Start obproxy ok
obproxy program health check ok
Connect to obproxy ok
+---------------------------------------------------+
|                      obproxy                      |
+----------------+------+-----------------+--------+
| ip             | port | prometheus_port | status |
+----------------+------+-----------------+--------+
| 10.10.10.1     | 2883 | 2884            | active |
| 10.10.10.2     | 2883 | 2884            | active |
| 10.10.10.3     | 2883 | 2884            | active |
+----------------+------+-----------------+--------+
obclient -h10.10.10.1 -P2883 -uroot@proxysys -p'******' -Doceanbase -A

test2 restart
Trace ID: 471fd8d0-f33e-11ee-88aa-00163e046d79
```

```
If you want to view detailed obd logs, please run: obd display-trace 471fd8d0-f33e
-11ee-88aa-00163e046d79
```

重启部署配置文件中指定的服务组件

```
[admin@test001 /home/admin]$ obd cluster restart test2 -c obproxy-ce
```

执行命令前需了解如下内容：

- `-c` 指定的组件名称为部署配置文件中的组件模块名称，非执行 `obd cluster display` 命令输出中展示的名称。
- 指定多个组件名称时，多个组件需使用英文逗号（`,`）间隔。
- 单独重启某个服务组件时，并不会更新部署名称的状态。例如整个集群服务都停止的情况下，集群部署状态为 `stopped`，此时单独重启某个服务组件，或依次单独启动集群中的服务组件，最终集群部署状态仍然是 `stopped`。

重启命令的输出如下，仅展示指定的服务组件状态。

```
Get local repositories and plugins ok
Load cluster param plugin ok
Open ssh connection ok
Cluster status check ok
Check before start obproxy ok
Stop obproxy ok
Start obproxy ok
obproxy program health check ok
Connect to obproxy ok
+----------------------------------------------------+
|                      obproxy                       |
+-----------------+------+-----------------+--------+
| ip              | port | prometheus_port | status |
+-----------------+------+-----------------+--------+
| 10.10.10.1      | 2883 | 2884            | active |
| 10.10.10.2      | 2883 | 2884            | active |
| 10.10.10.3      | 2883 | 2884            | active |
+-----------------+------+-----------------+--------+
obclient -h10.10.10.1 -P2883 -uroot -p'******' -Doceanbase -A
succeed
Trace ID: 669dc098-f3dc-11ee-bafb-00163e046d79
If you want to view detailed obd logs, please run: obd display-trace 669dc098-f3dc
-11ee-bafb-00163e046d79
```

重启部署配置文件中指定节点的指定服务组件

```
[admin@test001 /home/admin]$ obd cluster start test2 -c oceanbase-ce  -s xx.xx.xx.
2
```

执行命令前需了解如下内容：

- 指定多个组件多个节点 IP 时，多个组件或多个节点 IP 均需使用英文逗号（,）间隔。
- 如果指定多个 IP 时存在错误的 IP，obd 会跳过错误的 IP 信息，仅对正确的 IP 进行重启操作。
- 如果指定多个组件服务时存在错误的组件名称，obd 会终止整个重启操作。

重启命令的输出如下：

```
Get local repositories ok
Search plugins ok
Open ssh connection ok
Load cluster param plugin ok
Cluster status check ok
Check before start observer ok
Start observer ok
observer program health check ok
Connect to observer 10.10.10.2:2881 ok
succeed
Trace ID: bb2082a0-f3db-11ee-8a93-00163e046d79
If you want to view detailed obd logs, please run: obd display-trace bb2082a0-f3db
-11ee-8a93-00163e046d79
```

使用 obd 升级管理的服务

这里主要介绍如何升级 obd 部署的集群环境下各个组件，包括 OceanBase 数据库、OBProxy、OCP Express 等组件。obd 不支持同时升级多个组件，因此本节将按升级 obd、升级 OceanBase 数据库、升级 OBProxy 集群、升级 OCP Express 的升级顺序进行介绍。

说明

- 使用 obd 升级集群组件时，待升级的集群需是被 obd 管理的集群，否则 obd 无法获取集群的信息。obd

部署的集群默认由 obd 管理，非 obd 部署的集群可执行接管操作进行接管，具体操作可参见官网《OceanBase 安装部署工具》文档 使用指南/命令行/使用 obd 接管集群。

- 本节内容不包含升级 OCP 的介绍，使用 obd 升级 OCP 的具体操作可参见官网《OceanBase 云平台》文档 升级 OceanBase 云平台/升级社区版 OceanBase 云平台/使用图形化界面升级 OCP。

升级 obd

根据当前环境是否可以连接外部网络，有如下两种升级 obd 的方法，建议根据实际环境选择合适的升级方法。

在线升级

1. 打开远程仓库

```
[admin@test001 /home/admin]$ obd mirror enable remote
```

2. 执行升级命令

```
[admin@test001 /home/admin]$ obd update
```

离线升级

离线环境下，您需先获取 obd 升级目标版本的安装包，并将安装包上传到当前环境的任一目录下。

1. 关闭远程仓库

```
[admin@test001 /home/admin]$ obd mirror disable remote
```

2. 克隆安装包到 obd 本地仓库

```
[admin@test001 /home/admin]$ obd mirror clone ob-deploy-*.rpm
```

3. 执行升级命令

```
[admin@test001 /home/admin]$ obd update
```

升级 OceanBase 数据库

执行升级 OceanBase 数据库操作前，您需了解如下内容：

- 升级前需检查 obd 版本是否需要更新，建议使用 obd 最新版本。
- 如果集群存在备租户，您需先升级备租户所在的集群，或者执行 `obd cluster tenant switchover` 命令进行主备切换。建议主备集群均升级至相同版本。`obd cluster tenant switchover` 命令的详细介绍可参见官网《OceanBase 安装部署工具》文档 obd 命令/集群命令组 中 **obd cluster tenant switchover** 小节。
- 升级前需确认当前版本是否支持升级到目标版本，具体可以参看目标版本的发布记录中 **升级说明**。OceanBase 数据库的版本发布记录可访问 链接 查看。
- 升级前需检查部署名称状态是 `running`，且 OceanBase 集群状态无异常。
- 如果升级失败，支持重复执行 obd 升级命令，但禁止在不明升级失败原因的情况下修改集群系统参数，例如：`enable_ddl`、`enable_upgrade_mode`。升级失败时，可执行黑屏打印的 `obd display-trace xxxx` 命令查看日志，或者直接查看升级日志 `upgrade_post.log` 进行检查。
- 升级默认使用的滚动升级方式且无法修改，期间会停止升级节点数据库服务。如果集群 Zone 个数小于三，或者为单机集群，需要注意升级对业务带来的影响。
- 升级完成后，如果升级目标版本为 V4.0.x、V4.1.x 和 V4.2.0 BETA 版本，需要还原用户租户的 Primary Zone 优先级，否则可能影响业务性能。
- 升级完成后建议升级其他组件服务至配套版本。

升级 OceanBase 数据库的详细操作可参见官网《OceanBase 安装部署工具》文档 使用指南/命令行/升级 OceanBase 数据库。

升级 OBProxy 集群

根据当前环境是否可以连接外部网络，有如下两种升级 OBProxy 集群的方法，建议根据实际环境选择合适的升级方法。

在线升级

1. 打开远程仓库

```
[admin@test001 /home/admin]$ obd mirror enable remote
```

2. 查看远程仓库中 OBProxy 版本

```
[admin@test001 /home/admin]$ obd mirror list oceanbase.community.stable | grep obp
roxy-ce
```

输出如下，最后一列字符串即为 obproxy-ce 对应版本的 hash 值。

```
[admin@test001 /home/admin]$ obd mirror list oceanbase.community.stable | grep obp
roxy-ce
| obproxy-ce                     | 4.1.0.0 | 7.el8                 | x86_64 | 9f64a
13645980e5c767fdbfc6aa0ff3fca7c2fe468d40aeb6c45ea2cccd863ba |
| obproxy-ce                     | 4.2.0.0 | 7.el8                 | x86_64 | bac14
3e7cdedd98fd4458a66291b403b42d1f8efffb896452105020331b7053 |
| obproxy-ce                     | 4.2.1.0 | 11.el8                | x86_64 | eff4c
01cef815c1323b70ae5105cc09d5711852baf3638b6cd12c5f9d09ffe7c |
| obproxy-ce                     | 4.2.3.0 | 3.el8                 | x86_64 | 14958
440fa70d669cf08671b3e5a2c3bdbec235fbe72960a9bc51e61b4ed6f8d |
```

3. 执行升级命令

此处以将 OBProxy 从低版本升级到 V4.2.3 为例，您需根据实际需要替换对应版本号及 hash 值。

```
[admin@test001 /home/admin]$ obd cluster upgrade test2 -c obproxy-ce -V 4.2.3.0 --
usable=14958440fa70d669cf08671b3e5a2c3bdbec235fbe72960a9bc51e61b4ed6f8d
```

离线升级

离线环境下，您需先获取 OBProxy 升级目标版本的安装包，并将安装包上传到当前环境的任一目录下。

1. 关闭远程仓库

```
[admin@test001 /home/admin]$ obd mirror disable remote
```

2. 克隆安装包到 obd 本地仓库

```
[admin@test001 /home/admin]$ obd mirror clone obproxy-ce-*.rpm
```

3. 查看本地仓库安装包信息

```
[admin@test001 /home/admin]$ obd mirror list local |grep obproxy-ce
| obproxy-ce                  | 4.2.3.0    | 3.el7                        | x86_64    | 0490e
bc04220def8d25cb9cac9ac61a4efa6d639 |
| obproxy-ce                  | 4.2.1.0    | 11.el7                       | x86_64    | 0aed4
b782120e4248b749f67be3d2cc82cdcb70d |
```

4. 执行升级命令

```
[admin@test001 /home/admin]$ obd cluster upgrade test2 -c obproxy-ce -V 4.2.3.0 --
usable=0490ebc04220def8d25cb9cac9ac61a4efa6d639
```

5. 验证 OBProxy 版本

升级操作完成后，您可使用 root 用户登录 OceanBase 数据库的 sys 租户，执行如下命令验证 OBProxy 版本。

```
show proxyinfo binary\G
```

升级 OCP Express

说明

目前仅支持通过 obd 黑屏命令行方式升级 OCP Express。

执行升级 OCP Express 操作前，您需了解如下内容：

- 升级前需检查 obd 版本是否需要更新，建议使用 obd 最新版本。
- 离线环境下除了克隆 OCP Express 安装包到本地仓库外，还需要克隆配套版本的 OBAgent 安装包。
- 升级前需要先确认当前 OCP Express 版本和升级目标版本信息，因为同版本和跨版本升级方式不一样。例如：同版本升级使用 `obd cluster reinstall` 命令，跨版本升级使用 `obd cluster upgrade` 命令。

- 如果升级失败不会影响业务数据库的使用，可以通过执行 `obd display-trace xxxx` 命令进行问题排查。
- 升级完成后，OCP Express 登录密码不会重置，可以登录后查看 **帮助** -> **关于 ocp express**，查看版本号与发布日期验证是否已升级到目标版本。

升级 OCP Express 的详细操作可参见官网《OceanBase 安装部署工具》文档 使用指南/命令行/升级 OCP Express。

使用 obd 扩容管理的服务

obd 在 V2.5.0 起已经支持除 `ocp-server` 和 `oblogproxy` 组件以外的全部组件的扩容能力，具体可以参看使用 obd 扩容与组件变更。接下来将主要介绍使用 obd 扩容 OceanBase 集群时需要注意的事项。

注意事项

执行扩容前，您需了解如下内容：

- 扩容前，需检查 obd 版本是否需要更新，建议使用最新版本。
- 扩容前，扩容的节点建议先进行服务器初始化，其中部署用户和 SSH 连接信息需要确保一致。详细介绍可参见本教程第二章 部署前准备。
- 扩容前，需要确保 OceanBase 集群状态正常。
- 扩容前，需要确保已经安装有 OBClient 客户端。
- 扩容节点的配置文件不允许修改原集群配置中的 `depends`、`global` 或其他 `server` 部分的配置，配置新节点的基础信息即可，因此需要保证扩容节点服务器资源配置、目录挂载等和原集群服务器保持一致。
- 扩容时，`obd cluster scale_out` 命令会将新节点扩容配置信息更新到源集群部署配置文件中，并对扩容节点进行环境预检查，通过后会在扩容节点安装 OceanBase 数据库并启动服务，并添加到源集群中。
- 扩容后，可执行 `obd cluster display` 命令检查扩容集群状态是否正常。
- 扩容后，需要对租户进行增加副本操作，否则仍然是单副本租户。具体操作可参见官网

《OceanBase 数据库》文档 管理数据库/副本管理/副本分布/Locality 常见操作/增加副本。

- 建议扩容后按需调整租户的 Primary Zone 优先级。
- 目前 obd 不支持对集群进行缩容操作。
- 扩容后支持纵向扩容 OBServer 节点，扩容配置文件中指定源集群 Zone 名称即可，扩容完成后，可以对进行租户资源水平扩展操作。具体操作可参见官网《OceanBase 数据库》文档 参考指南/系统原理/分布式数据库对象/动态扩容和缩容/租户内资源的扩容和缩容/租户资源水平扩缩容。

操作步骤

此处以已部署一套单机环境，将该单机环境扩容成 3 Zone 的集群环境为例介绍如何使用 obd 进行扩容。

1. 查看单机环境信息

```
[admin@test001 /home/admin]$obd cluster display Expansion
```

此处以部署集群名为 `Expansion` 为例，您需根据实际部署集群名进行替换。输出如下：

```
Get local repositories and plugins ok
Open ssh connection ok
Cluster status check ok
Connect to observer xx.xx.xx.1:2881 ok
Wait for observer init ok
+----------------------------------------------------+
|                     observer                       |
+----------------+---------+------+-------+--------+
| ip             | version | port | zone  | status |
+----------------+---------+------+-------+--------+
| 10.10.10.1     | 4.2.1.2 | 2881 | zone1 | ACTIVE |
+----------------+---------+------+-------+--------+
obclient -hxx.xx.xx.1 -P2881 -uroot -p'******' -Doceanbase -A
```

2. 编写扩容节点配置文件

此处以扩容配置文件为 Expansion1_3.yaml 为例，您可自定义文件名。

```
[admin@test001 /home/admin]$ vim Expansion1_3.yaml
```

```

```

文件内容如下：

```
oceanbase-ce:
  servers:
    - name: server2
      ip: 10.10.10.2
    - name: server3
      ip: 10.10.10.3
  server2:
    mysql_port: 2881
    rpc_port: 2882
    home_path: /home/admin/oceanbase
    zone: zone2
  server3:
    mysql_port: 2881
    rpc_port: 2882
    home_path: /home/admin/oceanbase
    zone: zone3
```

3. 执行扩容命令

```
[admin@test001 /home/admin]$obd cluster scale_out Expansion  -c Expansion1_3.yaml
```

输出如下内容即表示扩容成功：

```
succeed
Connect to observer 10.10.10.1:2881 ok
scaling out ok
Waiting for observers ready ok
Execute ` obd cluster display Expansion ` to view the cluster status
```

4. 验证扩容结果

```
[admin@test001 /home/admin]$ obd cluster display Expansion
```

输出如下

```
Get local repositories and plugins ok
Open ssh connection ok
Cluster status check ok
Connect to observer 10.10.10.1:2881 ok
Wait for observer init ok
+----------------------------------------------------+
```

```
|                      observer                            |
+-----------------+---------+------+-------+--------+
| ip              | version | port | zone  | status |
+-----------------+---------+------+-------+--------+
| 10.10.10.1      | 4.2.1.2 | 2881 | zone1 | ACTIVE |
| 10.10.10.2      | 4.2.1.2 | 2881 | zone2 | ACTIVE |
| 10.10.10.3      | 4.2.1.2 | 2881 | zone3 | ACTIVE |
+-----------------+---------+------+-------+--------+
obclient -h10.10.10.1 -P2881 -uroot -p'******' -Doceanbase -A
```

5.3 使用 ob-operator 进行运维

ob-operator 是一款基于 Kubernetes Operator 框架构建的工具，用于在 Kubernetes 中管理 OceanBase 集群。它提供了一种简单可靠的方式来实现 OceanBase 集群的容器化部署，可以简化 OceanBase 集群的运维。ob-operator 定义了 OceanBase 数据库相关的各种资源并且实现相应的调协逻辑，因此用户可以像管理 Kubernetes 的原生资源一样，通过声明式的方式管理 OceanBase 数据库。当前最新版本已支持通过 Dashboard 提供给用户管理 K8s 中的 OceanBase 集群的界面，调用 K8s 的 API 来实现 OceanBase 集群的资源管理，包括基础概览、创建集群、创建租户、监控指标、备份恢复、集群升级、集群扩缩容等，目前仅支持社区版 OceanBase 数据库镜像。完整运维文档可以参见官网 OceanBase K8s 运维工具 文档，此章节将介绍通过配置文件实现部分核心运维操作时的注意事项。

ob-operator 组成成分

CRD (Custom Resource Definition)：自定义资源，提供了 K8s 中资源的定义。

```
$kubectl get crds | grep oceanbase.com
```

输出如下：

```
obtenantbackups.oceanbase.oceanbase.com          2024-05-07T11:29:58Z
obtenantrestores.oceanbase.oceanbase.com         2024-05-07T11:29:58Z
obtenantoperations.oceanbase.oceanbase.com       2024-05-07T11:29:58Z
obtenants.oceanbase.oceanbase.com                2024-05-07T11:29:58Z
obresourcerescues.oceanbase.oceanbase.com        2024-05-07T11:29:58Z
obclusteroperations.oceanbase.oceanbase.com      2024-05-30T04:01:44Z
obclusters.oceanbase.oceanbase.com               2024-05-07T11:29:58Z
obparameters.oceanbase.oceanbase.com             2024-05-07T11:29:58Z
observers.oceanbase.oceanbase.com                2024-05-07T11:29:58Z
obtenantbackuppolicies.oceanbase.oceanbase.com   2024-05-07T11:29:58Z
obzones.oceanbase.oceanbase.com                  2024-05-07T11:29:58Z
```

Controller-manager：自定义的程序，处理 OceanBase 数据库相关的资源变更。

```
$kubectl get deployment -n oceanbase-system
```

输出如下：

```
NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
oceanbase-controller-manager     1/1     1            1           113s
```

Dashboard：web 程序，白屏的方式来管理 K8s 中的 OceanBase 数据库资源。

```
$kubectl get deployment -n dashboard-test
```

输出如下：

```
NAME                                                     READY   UP-TO-DATE   AVAILABL
E   AGE
oceanbase-dashboard-oceanbase-dashboard-1723107702   1/1     1
1            17h
```

不同部署模式的高可用

详细介绍可参见官网《OceanBase K8s 运维工具》文档 使用指南/高可用 章节。

集群模式	说明	限制
常规模式	直接使用 pod ip 来 关联 observer，一般能够容忍少数派的故障，如果使用 calico 作为网络插件，可以从多数派故障中进行节点故障恢复，详细介绍可参见官网《OceanBase K8s 运维工具》文档使用指南/高可用/故障恢复。	建议使用 calico 做网络插件，OceanBase 集群部署至少三个 Zone，租户至少三副本。
单体模式	使用 127.0.0.1作为地址，适用于小规格的部署场景，最好和分布式存储系统一起使用，靠存储的冗余来达到高可用，或者进行租户备份做数据灾备。租户备份的详细介绍可参见官网《OceanBase K8s 运维工具》文档使用指南/高可用/租户备份。	需要 OceanBase 数据库版本不低于 V4.2.0.0，只能启动单节点的数据库，没有拓展性。
service 模式	为每个 pod 创建一个对应的 service (IP)，使用 service 的地址来关联 observer，以此来达到 IP 固定的目的。	需要 OceanBase 数据库版本不低于 V4.2.1.4 ，但暂不支持 4.2.2.x 版本。

说明

除了单机模式外，常规模式和 service 模式还可以通过利用 OceanBase 数据库的主备租户能力，可以建立两个租户的主备关系，在故障发生时可以很快切换，能保证业务受到的影响更小。主备租户的详细介绍可参见官网《OceanBase K8s 运维工具》文档 使用指南/高可用/物理备库。

OceanBase 集群的运维

通过修改 K8s 中的资源来实现，目前实现逻辑是通过集群的资源传导到对应的目标资源上去。

创建租户

具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/租户管理/创建租户。

注意事项：

1. 创建租户前，确保 ob-operator V2.1.0 及以上，建议使用最新版本。
2. 指定资源名（kind）、命名空间（namespace）、集群（spec.obcluster）、租户名（spec.tenantName）、Unit 数量（spec.unitNum）、字符集（spec.charset）、连接白名单（spec.connectWhiteList）、资源大小（pools.*）等。
3. 创建租户前，OceanBase 集群可正常运行使用。
4. 创建租户前，业务租户事务日志盘资源建议为租户内存的 3~4 倍。
5. 创建租户前，配置文件中业务租户内存时不能小于 `__min_full_resource_pool_memory` 参数，查看方式：`select * from oceanbase.GV$OB_PARAMETERS where name ='__min_full_resource_pool_memory';`。

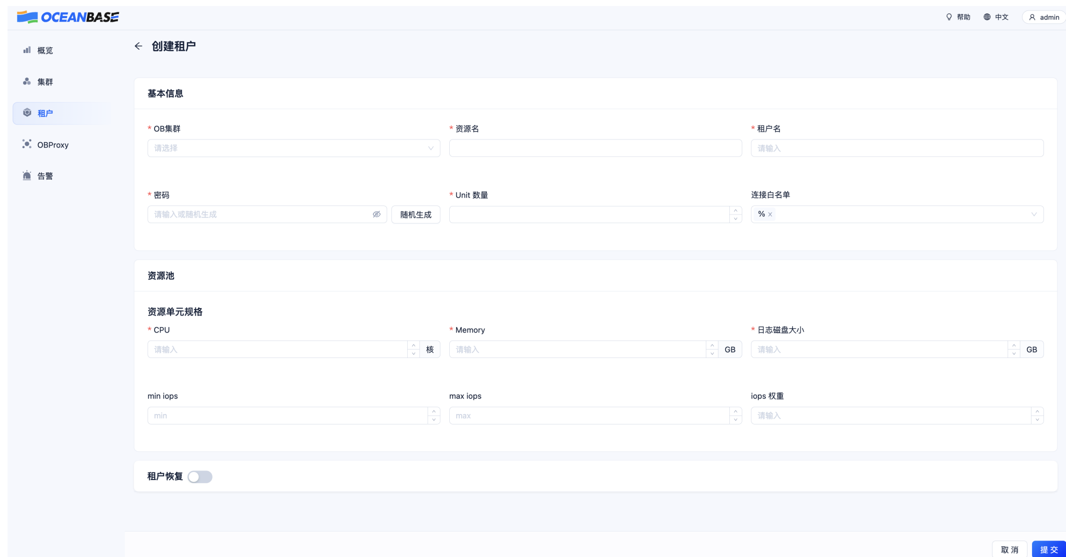
6. 创建租户后，创建业务租户下的普通用户，需要登录业务租户使用 SQL 语句进行创建。
7. 创建租户后，若配置文件未指定 `spec.credentials.root` 字段中的 `secret` 内容，则新建完成的业务租户 `root` 用户密码默认为空，如果需要修改密码，推荐 Dashboard 白屏修改。

例如下面的 YAML 文件应用到 K8s 集群（在集群已处在运行状态的前提下）后，ob-operator 将会自动创建指定规格的资源规格、资源单元和资源池，并且创建租户利用该资源池。

```
kubectl apply -f demo-tenant.yaml
```

```
apiVersion: oceanbase.oceanbase.com/v1alpha1
kind: OBTenant
metadata:
  name: demo-tenant
  namespace: default
spec:
  obcluster: test
  tenantName: demo_tenant
  unitNum: 1
  charset: utf8mb4
  connectWhiteList: "%"
  forceDelete: true
  pools:
    - zone: zone1
      type:
        name: Full
        replica: 1
        isActive: true
      resource:
        maxCPU: 1
        memorySize: 2Gi
        minCPU: 1
        maxIops: 1024
        minIops: 1024
        iopsWeight: 2
        logDiskSize: 4Gi
```

Dashboard 白屏创建租户界面：



修改租户资源

具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/租户管理/修改租户/资源管理。

直接在 YAML 上修改资源配置后重新应用即可。比如管理租户副本和其他租户配置项操作也是类似的操作。具体介绍可参见官网《OceanBase K8s 运维工具》中 使用指南/租户管理/修改租户/副本管理 和 使用指南/租户管理/修改租户/其他配置项修改。

例如将下面的配置文件中的 `memorySize`（2Gi -> 5Gi）进行修改，(建议同时调整 `logDiskSize` 是 `memorySize` 3~4 倍大小)，再次应用到集群中，ob-operator 会将资源池进行扩容。

```
kubectl apply -f demo-tenant.yaml
```

```
apiVersion: oceanbase.oceanbase.com/v1alpha1
kind: OBTenant
metadata:
  name: demo-tenant
  namespace: default
spec:
  obcluster: test
  tenantName: demo_tenant
  unitNum: 1
  charset: utf8mb4
  connectWhiteList: "%"
  forceDelete: true
  pools:
    - zone: zone1
```

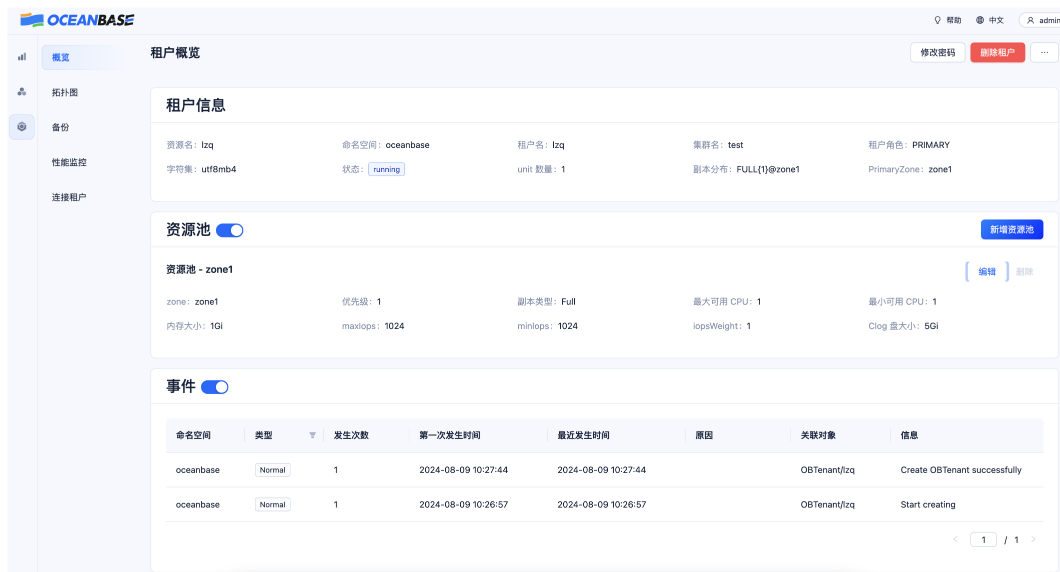
```
      type:
        name: Full
        replica: 1
        isActive: true
      resource:
        maxCPU: 1
        memorySize: 5Gi
        minCPU: 1
        maxIops: 1024
        minIops: 1024
        iopsWeight: 2
        logDiskSize: 15Gi
```

注意

某些租户运维操作会涉及多个租户，不适合定义在租户资源当中，所以专门定义了租户运维资源，通过此类资源可完成修改用户密码、日志回放、Switchover、Failover 等操作。租户运维的详细介绍可参见官网《OceanBase K8s 运维工具》文档 使用指南/租户管理/租户运维操作。

除了能够作用于多个同类型资源，运维操作资源也可以作为“操作日志”作为归档，后续可用于查看该租户都经历过哪些操作以及操作的配置各是如何的。

Dashboard 白屏修改租户资源界面：



数据备份

具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/高可用/租户备份。

ob-operator 中的数据备份采用备份策略的方式，在其中指定好所在集群、需要备份的租户、数据恢复窗口、备份地址、备份周期等。备份策略创建到集群中后，ob-operator 会先执行一次全量备份。之后会按照指定的定时周期来执行增量和全量备份。当前 ob-operator 数据备份只有租户级别备份。

注意事项：

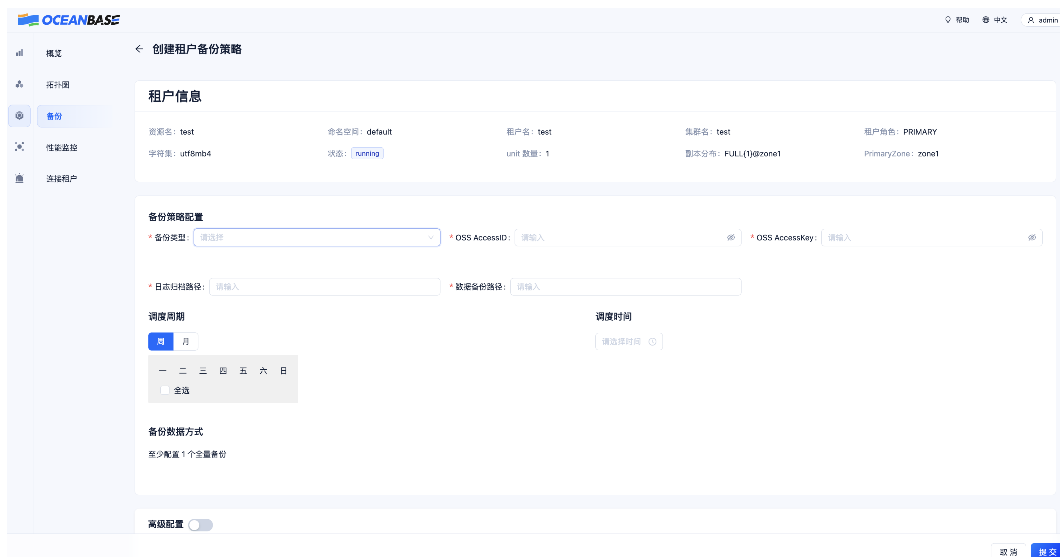
1. 备份前，OceanBase 集群所有节点均需处于正常状态，处于 `升级中` 的集群状态不支持进行数据备份。
2. 备份前，当前仅支持 NFS 和 OSS 两种备份介质，如果使用 NFS 作为备份介质，需要部署 NFS，且部署 OceanBase 集群设置了备份的 volume，数据库可以正常访问并有读写权限。部署 NFS 的具体操作可参见官网《OceanBase 数据库》文档 管理数据库/备份恢复/部署 NFS。
3. 备份前，建议先进行一次手动合并，确保数据的一致性和提高备份的效率。手动合并的具体操作可参见官网《OceanBase 数据库》文档 参考指南/系统管理/存储管理/合并管理/手动触发合并。
4. 备份后，禁止直接手动 `rm` 方式删除正在执行备份的数据备份的文件，此操作可能会导致归档日志备份失败，数据备份无法完成等。

5. 备份后，关于数据备份的清理需要遵循 `全量+增量=完整数据` 原则，如果仅有一份全量备份时，即使达到过期备份清理保留周期也不会触发该全量备份的清理。

   例如下面的备份策略表示在每周六的 00:30 执行全量备份，在每天的 01:30 执行增量备份。备份数据的目的地类型是 NFS（需要在创建集群时先指定挂载）。

```
apiVersion: oceanbase.oceanbase.com/v1alpha1
kind: OBTenantBackupPolicy
metadata:
  name: obtenantbackuppolicy-sample
  namespace: oceanbase
spec:
  obClusterName: "test"
  tenantCRName: "demo-tenant"
  jobKeepWindow: "1d"
  dataClean:
    recoveryWindow: "8d"
  logArchive:
    destination:
      type: "NFS"
      path: "t1/log_archive_custom_1019"
    switchPieceInterval: "1d"
  dataBackup:
    destination:
      type: "NFS"
      path: "t1/data_backup_custom_enc"
    fullCrontab: "30 0 * * 6"
    incrementalCrontab: "30 1 * * *"
```

Dashboard 白屏创建备份策略界面：



数据恢复

具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/高可用/数据恢复。

ob-operator 支持的 OceanBase 数据库 4.x 版本仅支持租户级别的物理恢复，所以进行物理恢复的配置仍在租户的文件中。数据恢复依赖关键资源为 OBTenant 和 OBTenantRestore，分别表示租户和租户恢复任务。因数据恢复行为的最终结果是产生一个新的租户，所以使用在 `OBTenant Spec` 中配置的 `Source` 字段来指定租户的恢复源。在租户资源中指定了数据恢复相关的字段之后，Operator 会创建出 OBTenantRestore 资源来执行具体的恢复任务。

注意事项：

1. 建议确保数据恢复的目标集群和源集群数据库版本保持一致。
2. OceanBase 数据库当前仅支持将低版本的备份数据恢复到同版本或高版本中，但不支持 OceanBase 数据库 3.x 版本的备份数据恢复到 OceanBase 数据库 4.x 版本。
3. 特殊版本：OceanBase 数据库 4.0.x 版本的备份数据也不支持恢复到 OceanBase 数据库 4.1.x 及以上版本。
4. 恢复的租户可以通过配置文件 `tenantRole: STANDBY/PRIMARY` 参数配置是否恢复成备租户，不设置默认恢复成主租户，即独立租户，恢复完成后不会和源租户再进行数据同步。
5. 确保目标集群状态正常，可正常使用。
6. 需要确认目标集群剩余资源足够恢复租户，特别是数据磁盘空间。
7. 如果恢复源为 NFS 类型，需要确定挂载到 OceanBase 集群的备份 Volume 可用，且目标集群可以正常访问并有读写权限。

以下示例恢复一个主租户，也可以通过调整 `tenantRole` 为 `STANDBY` 和备租户相关的配置恢复备租户。具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/高可用/物理备库。

```
apiVersion: oceanbase.oceanbase.com/v1alpha1
kind: OBTenant
metadata:
  name: t1s
  namespace: oceanbase
spec:
  obcluster: test
  tenantName: t1s
  unitNum: 1
  charset: utf8mb4
  connectWhiteList: '%'
```

```
  forceDelete: true
  tenantRole: PRIMARY # 指定恢复出主租户
  source: # 租户数据来源
    restore:
      bakDataSource:
        type: "OSS" # 备份介质中备份数据访问信息
        path: "oss://operator-backup-data/backup-t1?host=oss-cn-xxxx.aliyuncs.com"
        ossAccessSecret: "oss-access"
      archiveSource:
        type: "OSS" # 备份介质中归档数据访问信息
        path: "oss://operator-backup-data/archive-t1?host=oss-cn-xxxx.aliyuncs.com
"
        ossAccessSecret: "oss-access"
      until:
        unlimited: false  # true 表示恢复到最新位点
        timestamp: "2024-05-05 10:00:00"  # 需要恢复到的时间戳
    tenant: t1  # 指定主租户，仅在恢复成为备租户时生效
  pools:
    - zone: zone1
      type:
        name: Full
        replica: 1
        isActive: true
      resource:
        maxCPU: 1000m
        memorySize: 2Gi
        minCPU: 1
        maxIops: 1024
        minIops: 1024
        logDiskSize: 5Gi
```

扩容集群

扩容集群可分为横向扩容和纵向扩容，本节将分别为您介绍。

横向扩容集群

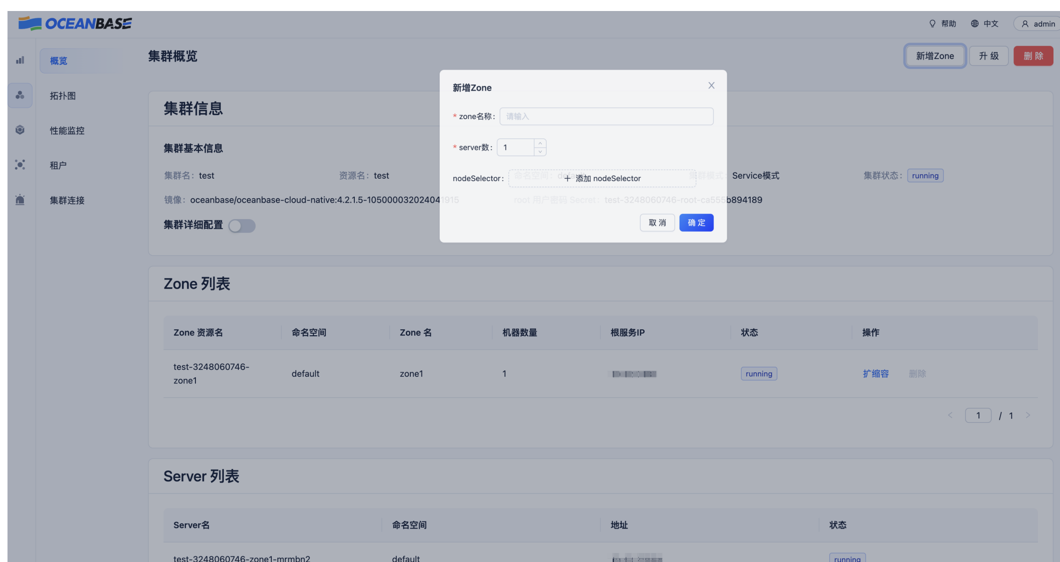
具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/集群管理/Zone 管理/在集群中增加 Zone。

通过修改原部署时的配置文件，在添加新的 Zone 节点，实现集群横向扩容，从而增加租户副本数量。

注意事项：

1. 扩容前，OceanBase 集群需要处于 `running` 状态。
2. 扩容前，服务器要有足够的资源，能够支持新增 Zone 所需资源。
3. 扩容前，ob-operator 会根据 OceanBase 数据库部署配置文件中的 nodeSelector 来决定 OBServer 节点的分布，如果需要指定节点可以配置 Kubernetes 的 node 节点配置 label，否则由 K8s 自动选节点。具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/集群管理/集群创建。
4. 扩容完成后，并不会自动为租户添加上副本，需要对租户进行副本管理，增加副本，具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/租户管理/修改租户/副本管理。
5. 扩容完成后，如果 Primary Zone 是顺序优先级策略，不能增加集群计算性能。如果 Primary Zone 是随机策略，一般对分区表的性能是有一些影响的，具体需要看业务使用场景。例如：对分区表而言扩容后分区会打散到更多节点，每个节点分区数减少，计算速度提升，但跨了更多个节点数可能增加网络 I/O 等。

Dashboard 白屏新增 Zone 界面：



纵向扩容集群

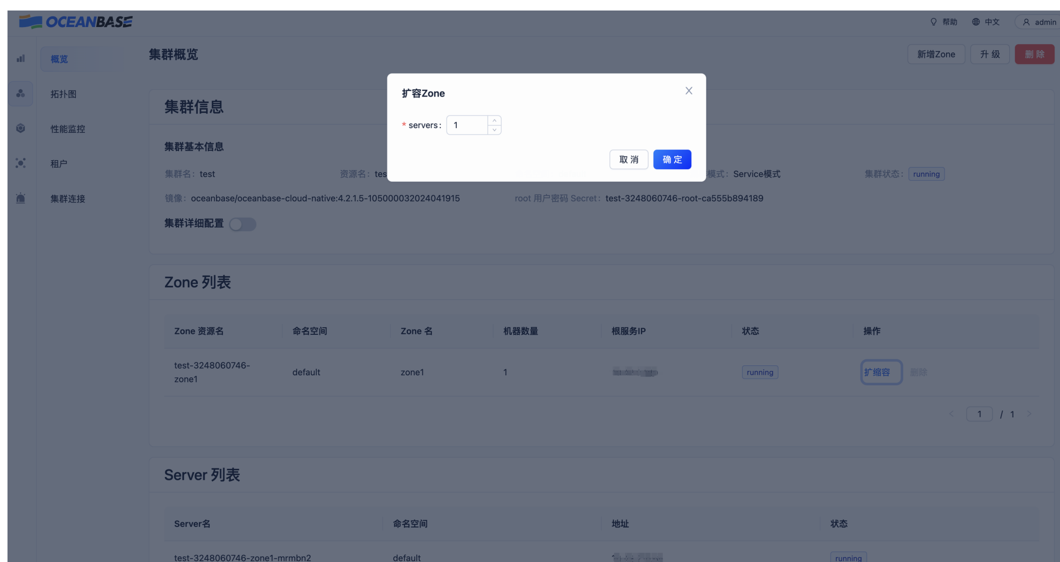
具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/集群管理/节点管理/向 Zone 内添加 OBServer 节点。

通过修改原部署时的配置文件，在原 Zone 中添加新的 observer 节点，实现集群纵向扩容，从而可以增加租户资源占用，提升租户业务性能和集群存储上限。

注意事项：

1. 扩容前，OceanBase 集群需要处于 `running` 状态。
2. 扩容前，服务器要有足够的资源，能够支持新增 observer 所需资源。
3. 扩容前，ob-operator 会根据 OceanBase 数据库部署配置文件中的 nodeSelector 来决定 OBServer 节点的分布，如果需要指定节点可以配置 Kubernetes 的 node 节点配置 label，否则由 K8s 自动选节点。具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/集群管理/集群创建。
4. 扩容前，部署配置文件调整 `topology` 下各个 Zone 的 replica 个数即可，建议保持一致。
5. 扩容后，租户的数据并不会自动均衡到新节点，需要对租户修改资源池 unit num，才会在新扩容节点生成同样的 Unit 资源池，将同一 Zone 下的租户数据分区均衡到新节点。修改资源池 unit num 的具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/租户管理/修改租户/资源管理。

Dashboard 白屏扩容 Zone 中的 OBServer 节点界面：



缩容集群

缩容集群可分为横向缩容和纵向缩容，本节将分别为您介绍。

横向缩容集群

具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/集群管理/节点管理/从 Zone 中减少 OBServer 节点。

通过修改原部署时的配置文件，在减少原有的 Zone 节点，实现集群横向缩容，从而可以减少租户副本数量。

注意事项：

1. 缩容前，OceanBase 集群需要处于 `running` 状态。
2. 缩容前，需要确认减少 Zone 后仍需要满足多数派。即 N 个 Zone 的集群缩容后 Zone 个数需要大于等于 N/2 个 Zone，但不支持缩容成单 Zone。
3. 缩容前，需要保证 Zone 上已无租户副本，即需要先进行缩容租户副本，具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/租户管理/修改租户/副本管理。

纵向缩容集群

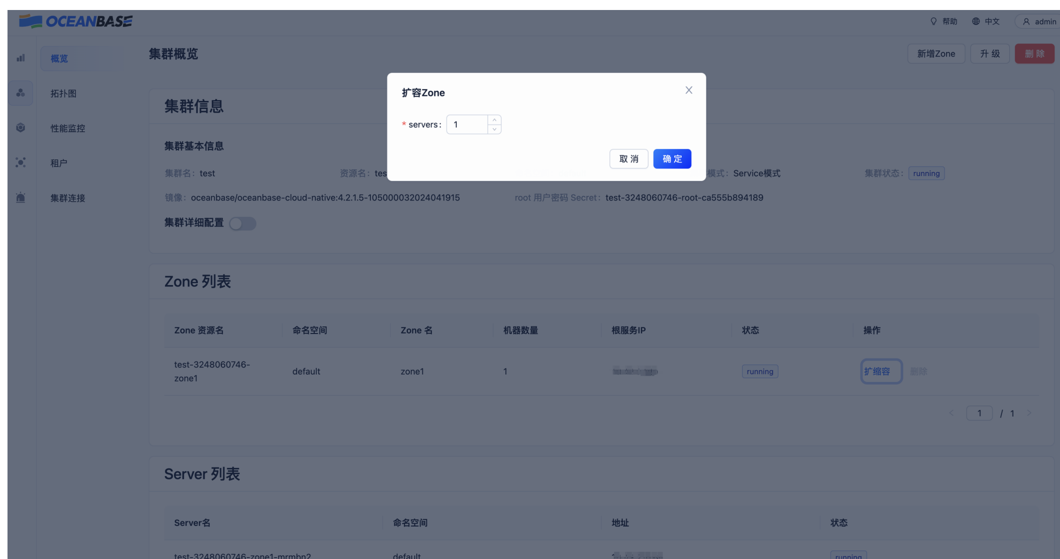
具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/集群管理/节点管理/从 Zone 中减少 OBServer 节点。

通过修改原部署时的配置文件，在原 Zone 中减少 OBServer 节点，实现集群纵向缩容，从而可以降低服务器资源占用，同时也会增加集群磁盘使用率。

注意事项：

1. 缩容前，OceanBase 集群需要处于 `running` 状态。
2. 缩容前，会把缩容的 OBServer 节点迁移到同 Zone 下的其他 OBServer 节点上，因此需要确保剩下 OBServer 节点有足够的资源支撑，特别是磁盘空间。

Dashboard 白屏缩容 Zone 中的 OBServer 节点界面：



升级集群

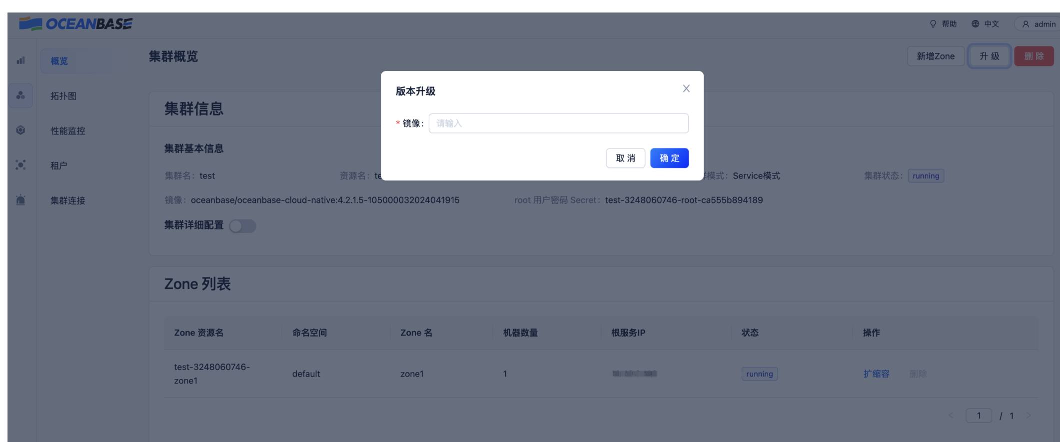
具体操作可参见官网《OceanBase K8s 运维工具》文档 使用指南/集群管理/集群升级。

通过修改原部署时的配置文件，将 `spec` 下的 `image` 更新为目标镜像即可进行 OceanBase 集群的升级操作。

注意事项：

1. 升级前，需要确保待升级的集群是 `running` 状态。
2. 升级前，需要检查是否需要升级 ob-operator，建议使用适配的最新版的数据库镜像文件。升级 ob-operator 的具体操作可参见官网《OceanBase K8s 运维工具》文档 ob-operator 升级。

Dashboard 白屏升级集群界面：



5.4 使用命令行进行运维

本文将着重介绍常用的 SQL 语句和数据库不同场景异常时收集日志自查思路等。

说明

本教程中所贴官网文档链接版本均为编写教程时的最新版本或长期支持版本（LTS），若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

使用 SQL 命令对数据库运维是数据库最基本的运维方式，目前官方文档对集群进行基本管理均是通过 SQL 命令实现，且介绍比较全面，具体可以查看官网《OceanBase 数据库》文档 管理数据库 章节。

常用 SQL

集群基础 SQL

查看集群名称

用户在使用 OBProxy 代理登录集群时，有时会遇到报错 `ERROR 4669 (HY000): cluster not exist`，原因是配置的集群名称不正确，例如：使用 obd 部署集群时，`obd cluster list` 展示的是部署名称，而非集群名称。您可登录 OceanBase 数据库执行如下命令最直接获取 OceanBase 数据库名称。

```
show parameters like 'cluster';
```

输出如下，`value` 对应的值即为 OceanBase 数据库的名称，此示例中为 `metadb`。

```
+-------+----------+----------------+----------+---------+-----------+--------+---
-----------------+----------+---------+---------+-------------------+------------
---+------------+
| zone  | svr_type | svr_ip         | svr_port | name    | data_type | value  | in
fo                 | section  | scope   | source  | edit_level        |
+-------+----------+----------------+----------+---------+-----------+--------+---
-----------------+----------+---------+---------+-------------------+
| zone1 | observer | 10.10.10.1     |     2882 | cluster | STRING    | metadb | Na
```

```
me of the cluster | OBSERVER | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
+-------+----------+---------------+---------+---------+-----------+--------+---
-----------------+----------+---------+---------+-------------------+
```

查看数据库版本

获取数据库版本方式有多种，除了执行 `./bin/observer --version` 命令通过二进制文件获取具体版本，在数据库中也能执行以下命令获取具体版本号信息。

- 查看参数

```
SHOW VARIABLES like 'version_comment';
```

输入如下：

```
+----------------+---------------------------------------------------------------
--------------------------------------------------+
| Variable_name   | Valu
e
                          |
+----------------+---------------------------------------------------------------
--------------------------------------------------+
| version_comment | OceanBase_CE 4.2.1.2 (r102000042023120514-ccdde7d34de421336c53
62483d64bf2b73348bd4) (Built Dec  5 2023 14:34:01) |
+----------------+---------------------------------------------------------------
--------------------------------------------------+
```

- 查看 DBA_OB_SERVERS 视图

```
SELECT BUILD_VERSION FROM oceanbase.DBA_OB_SERVERS;
```

输出如下，DBA_OB_SERVERS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_SERVERS。

```
+--------------------------------------------------------------------------------
----------+
| BUILD_VERSIO
N                                                                            |
+--------------------------------------------------------------------------------
----------+
| 4.2.1.2_102000042023120514-ccdde7d34de421336c5362483d64bf2b73348bd4(Dec  5 2023
14:34:01) |
+--------------------------------------------------------------------------------
```

```
----------+
```

查看 Root Service 主节点

Root Service 是集群的控制中心，负责资源管理、容灾、负载均衡和 Schema 管理等。在多 Zone 集群中，Root Service 会有多个副本，但只有一个为主节点，即 Leader，这个主节点负责处理集群的管理和协调任务。如果主节点发生故障，需要查看 Root Service 主节点日志信息，可以通过以下 SQL 获取主节点 IP。

```
SELECT svr_ip as RootService FROM oceanbase.DBA_OB_SERVERS WHERE with_rootserver='
yes';
```

输出如下，DBA_OB_SERVERS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_SERVERS。

```
+-----------------+
| RootService     |
+-----------------+
| 10.10.10.1      |
+-----------------+
```

查看各个 Zone 的状态、IDC、Region、TYPE 等信息

在 OCP 接管场景有时会遇到 IDC/Region 信息不匹配问题，原因是主机设置 IDC/Region 和部署的集群设置 IDC/Region 不一致，可以通过以下 SQL 查看各个 Zone 的 IDC/Region 信息，通过修改 Zone 的信息可以匹配主机配置保持一致。

```
SELECT * FROM oceanbase.DBA_OB_ZONES;
```

输出如下，DBA_OB_ZONES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_ZONES。

```
+-------+----------------------------+----------------------------+--------+------
-------+------------+------------+
| ZONE  | CREATE_TIME                | MODIFY_TIME                | STATUS | ID
C          | REGION     | TYPE       |
+-------+----------------------------+----------------------------+--------+------
-------+------------+------------+
```

```
| zone1 | 2024-06-04 11:21:23.398969 | 2024-06-04 11:21:45.087430 | ACTIVE | defau
lt_idc | sys_region | ReadWrite |
+--------+----------------------------+----------------------------+--------+------
-------+------------+------------+
```

查看隐藏参数

OceanBase 数据库中存在一些以下滑线（`__`）开头的参数，我们称之为隐藏参数，是默认情况下不建议直接调整的系统参数。隐藏参数无法通过 `show parameters`命令查看，但可以和正常参数一样通过 `alter system set` 命令修改。如需查看某个隐藏参数的信息，可执行如下命令。

此处以查看创建租户最小内存资源限制隐藏参数为例：

```
SELECT * FROM oceanbase.GV$OB_PARAMETERS WHERE name ='__min_full_resource_pool_mem
ory';
```

输出如下，GV$OB_PARAMETERS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_PARAMETERS。

```
+----------------+----------+-------+---------+-----------+----------------------
----------+-----------+------------+----------------------------------------------
--------------------+--------------+-------------------+---------------+---------
--+
| SVR_IP         | SVR_PORT | ZONE  | SCOPE   | TENANT_ID | NAM
E                          | DATA_TYPE | VALUE      | INF
O                                                                   | SECTION       | ED
IT_LEVEL        |
+----------------+----------+-------+---------+-----------+----------------------
----------+-----------+------------+----------------------------------------------
--------------------+--------------+-------------------+
| 10.10.10.1     |     2882 | zone1 | CLUSTER |      NULL | __min_full_resource_po
ol_memory | INT        | 2147483648 | the min memory value which is specified for
a full resource pool. | LOAD_BALANCE | DYNAMIC_EFFECTIVE |
+----------------+----------+-------+---------+-----------+----------------------
----------+-----------+------------+----------------------------------------------
--------------------+--------------+-------------------+
```

查看租户下所有类型的表

您可执行如下命令通过 DBA_OBJECTS 视图查看或统计用户租户下不同对象类型的信息或数量，

DBA_OBJECTS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OBJECTS。

1. 查看对象类型分类

```
SELECT distinct(OBJECT_TYPE) FROM oceanbase.DBA_OBJECTS;
```

输出如下：

```
+----------------+
| OBJECT_TYPE    |
+----------------+
| TABLE          |
| VIRTUAL TABLE  |
| INDEX          |
| VIEW           |
| PACKAGE        |
| PACKAGE BODY   |
| DATABASE       |
| TABLEGROUP     |
+----------------+
```

2. 根据对象类型查看表

此处以查看 `TABLE` 类型的表为例，执行如下 SQL 命令后将会输出租户下所有 `TABLE` 类型的表。

```
SELECT OBJECT_NAME FROM oceanbase.DBA_OBJECTS WHERE OBJECT_TYPE='TABLE';
```

查看历史参数修改记录

当数据库参数被修改时，如需了解修改前参数配置，可以通过以下 SQL 获取修改记录，便于进行参数还原等。

```
SELECT * FROM oceanbase.DBA_OB_ROOTSERVICE_EVENT_HISTORY WHERE event='admin_set_co
nfig' and value2 like '%xxxxx%';
```

以修改 enable_rereplication 参数为例，输出如下，DBA_OB_ROOTSERVICE_EVENT_HISTORY 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/

oceanbase.DBA_OB_ROOTSERVICE_EVENT_HISTORY。

```
+----------------------------+--------------+------------------+-------+--------+-
------+---------------------------------------------------------------------------
----------------------------------------------------------------------------------
------------+----------+--------+-------+--------+-------+--------+-------+-------
-+------------+---------------+-------------+
| TIMESTAMP                  | MODULE       | EVENT            | NAME1 | VALUE1 |
NAME2 | VALUE
2

| NAME3    | VALUE3 | NAME4 | VALUE4 | NAME5 | VALUE5 | NAME6 | VALUE6 | EXTRA_INF
O | RS_SVR_IP      | RS_SVR_PORT |
+----------------------------+--------------+------------------+-------+--------+-
------+---------------------------------------------------------------------------
----------------------------------------------------------------------------------
------------+----------+--------+-------+--------+-------+--------+-------+-------
-+------------+---------------+-------------+
| 2024-07-29 11:05:04.577456 | root_service | admin_set_config | ret   | 0      |
arg   | {name:"enable_rereplication", value:"false", comment:"", zone:"", server:"
0.0.0.0:0", tenant_name:"", exec_tenant_id:1, tenant_ids:[], want_to_set_tenant_co
nfig:false} | is_inner | 0      |       |        |       |        |
|        |            | 10.10.10.1     |        2882 |
+----------------------------+--------------+------------------+-------+--------+-
------+---------------------------------------------------------------------------
----------------------------------------------------------------------------------
------------+----------+--------+-------+--------+-------+--------+-------+-------
-+------------+---------------+-------------+
```

集群近期 1 小时内发生的事件

可以通过以下 SQL 查看近期集群不同模块发生的调度事件，帮助我们了解集群运行稳定性。例如节点故障时，RootService 服务会负责监控和调度节点，节点拉黑、永久下线节点、迁移 Unit 等，查看具体事件信息帮助了解事发自动恢复过程。

```
SELECT `TIMESTAMP`,module,EVENT,name1,value1,name2,value2,rs_svr_ip
  FROM oceanbase.DBA_OB_ROOTSERVICE_EVENT_HISTORY
  WHERE module IN ( 'server', 'root_service', 'balancer' ) AND `TIMESTAMP` > SUBDA
TE( now(), INTERVAL 1  HOUR )
  ORDER BY `TIMESTAMP` DESC LIMIT 50;
```

输出如下：

```

+----------------------------+--------------+-------------------+-----------+-----
```

```
---+-------+------------------------------------------------------------------------
---------+----------------+
| TIMESTAMP                  | module       | EVENT              | name1      | valu
e1 | name2 | value
2                                                                          | rs_svr_
ip       |
+----------------------------+--------------+--------------------+-----------+-----
---+-------+------------------------------------------------------------------------
---------+----------------+
| 2024-07-18 02:00:02.709958 | root_service | root_minor_freeze | ret        |
0       | arg    | {tenant_ids:[1], server_list:[], zone:"", tablet_id:{id:0}, ls_id:{
id:-1}}     | 10.10.10.1       |
| 2024-07-18 02:00:02.685112 | root_service | root_major_freeze | tenant_id |
1       | ret    |
0                                                                               | 10
.10.10.1      |
| 2024-07-18 02:00:02.348725 | root_service | root_minor_freeze | ret        |
0       | arg    | {tenant_ids:[1002], server_list:[], zone:"", tablet_id:{id:0}, ls_
id:{id:-1}} | 10.10.10.1       |
| 2024-07-18 02:00:02.324432 | root_service | root_major_freeze | tenant_id | 100
2    | ret    |
0                                                                               | 10
.10.10.1     |
| 2024-07-18 02:00:02.291545 | root_service | root_minor_freeze | ret        |
0       | arg    | {tenant_ids:[1010], server_list:[], zone:"", tablet_id:{id:0}, ls_
id:{id:-1}} | 10.10.10.1       |
| 2024-07-18 02:00:02.262967 | root_service | root_major_freeze | tenant_id | 101
0    | ret    |
0                                                                               | 10
.10.10.1     |
| 2024-07-18 02:00:01.672662 | root_service | root_minor_freeze | ret        |
0       | arg    | {tenant_ids:[1009], server_list:[], zone:"", tablet_id:{id:0}, ls_
id:{id:-1}} | 10.10.10.1       |
| 2024-07-18 02:00:01.658700 | root_service | root_major_freeze | tenant_id | 100
9    | ret    |
0                                                                               | 10
.10.10.1     |
| 2024-07-18 02:00:01.216653 | root_service | root_minor_freeze | ret        |
0       | arg    | {tenant_ids:[1001], server_list:[], zone:"", tablet_id:{id:0}, ls_
id:{id:-1}} | 10.10.10.1       |
| 2024-07-18 02:00:01.202710 | root_service | root_major_freeze | tenant_id | 100
1    | ret    |
0                                                                               | 10
.10.10.1     |
+----------------------------+--------------+--------------------+-----------+-----
---+-------+------------------------------------------------------------------------
---------+----------------+
```

DBA_OB_ROOTSERVICE_EVENT_HISTORY 视图的详细介绍可参见官网《OceanBase 数据库》

文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_ROOTSERVICE_EVENT_HISTORY。

查看集群资源相关 SQL

- 查看集群 CPU、内存、磁盘参数配置信息

```
show parameters where name in ('memory_limit','memory_limit_percentage','system_me
mory','log_disk_size','log_disk_percentage','datafile_size','datafile_disk_percent
age');
```

输出如下：

```
+-------+----------+----------------+----------+-------------------------+-------
----+-------+---------------------------------------------------------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
----------+-----------+---------+---------+------------------+
| zone  | svr_type | svr_ip         | svr_port | name                    | data_t
ype | value | inf
o


                                                                          | secti
on    | scope    | source   | edit_level        |
+-------+----------+----------------+----------+-------------------------+-------
----+-------+---------------------------------------------------------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
----------+-----------+---------+---------+------------------+
| zone1 | observer | 10.10.10.1     |     2882 | log_disk_percentage     | NUL
L      | 0     | the percentage of disk space used by the log files. Range: [0,99
] in integer;only effective when parameter log_disk_size is 0;when log_disk_percen
tage is 0: a) if the data and the log are on the same disk, means log_disk_percent
age = 30 b) if the data and the log are on the different disks, means log_disk_per
ecentage = 90 | LOGSERVICE | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
| zone1 | observer | 10.10.10.1     |     2882 | log_disk_size           | NUL
L      | 100G  | the size of disk space used by the log files. Range: [0, +∞
)


                   | LOGSERVICE | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
| zone1 | observer | 10.10.10.1     |     2882 | memory_limit_percentage  | NUL
L      | 80    | the size of the memory reserved for internal use(for testing purp
ose). Range: [10, 95
```

```
]
                                                                              | OBSER
VER    | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
| zone1 | observer | 10.10.10.1      |      2882 | system_memory            | NUL
L       | 3G     | the memory reserved for internal use which cannot be allocated t
o any outer-tenant, and should be determined to guarantee every server functions n
ormally. Range: [0M,
)
                                                                               | OBSE
RVER    | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
| zone1 | observer | 10.10.10.1      |      2882 | memory_limit             | NUL
L       | 30G    | the size of the memory reserved for internal use(for testing purp
ose), 0 means follow memory_limit_percentage. Range: 0, [1G,)
.
                                  | OBSERVER    | CLUSTER | DEFAULT | DYNAMIC_EFFEC
TIVE |
| zone1 | observer | 10.10.10.1      |      2882 | datafile_disk_percentage | NUL
L       | 0      | the percentage of disk space used by the data files. Range: [0,99
] in intege
r


  | SSTABLE     | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
| zone1 | observer | 10.10.10.1      |      2882 | datafile_size            | NUL
L       | 100G   | size of the data file. Range: [0, +∞
)


                                          | SSTABLE     | CLUSTER | DEFAULT | DYNAM
IC_EFFECTIVE |
+--------+----------+----------------+----------+--------------------------+-------
----+-------+---------------------------------------------------------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
----------+------------+---------+---------+-------------------+
```

- 查看集群 server 级资源分配情况

```
SELECT zone,concat(SVR_IP,':',SVR_PORT) observer,
  cpu_capacity_max cpu_total,cpu_assigned_max cpu_assigned,
  cpu_capacity-cpu_assigned_max as cpu_free,
  round(memory_limit/1024/1024/1024,2) as memory_total,
  round((memory_limit-mem_capacity)/1024/1024/1024,2) as system_memory,
  round(mem_assigned/1024/1024/1024,2) as mem_assigned,
  round((mem_capacity-mem_assigned)/1024/1024/1024,2) as memory_free,
  round(log_disk_capacity/1024/1024/1024,2) as log_disk_capacity,
  round(log_disk_assigned/1024/1024/1024,2) as log_disk_assigned,
```

```
  round((log_disk_capacity-log_disk_assigned)/1024/1024/1024,2) as log_disk_free,
  round((data_disk_capacity/1024/1024/1024),2) as data_disk,
  round((data_disk_in_use/1024/1024/1024),2) as data_disk_used,
  round((data_disk_capacity-data_disk_in_use)/1024/1024/1024,2) as data_disk_free
  FROM oceanbase.GV$OB_SERVERS;
```

输出如下，GV$OB_SERVERS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_SERVERS。

```
+-------+---------------------+----------+--------------+----------+-------------
-+----------------+--------------+-------------+-------------------+-------------
----+---------------+-----------+----------------+----------------+
| zone  | observer            | cpu_total | cpu_assigned | cpu_free | memory_tota
l | system_memory | mem_assigned | memory_free | log_disk_capacity | log_disk_assi
gned | log_disk_free | data_disk | data_disk_used | data_disk_free |
+-------+---------------------+----------+--------------+----------+-------------
-+----------------+--------------+-------------+-------------------+-------------
----+---------------+-----------+----------------+----------------+
| zone1 | 10.10.10.1:2882     |        16 |            7 |        9 |        30.0
0 |          3.00 |        10.00 |       17.00 |            100.00 |            2
7.00 |         73.00 |    100.00 |           1.52 |          98.48 |
+-------+---------------------+----------+--------------+----------+-------------
-+----------------+--------------+-------------+-------------------+-------------
----+---------------+-----------+----------------+----------------+
```

- 查看集群租户级资源分配和磁盘使用情况

```
SELECT a.zone,a.svr_ip,b.tenant_name,b.tenant_type, a.max_cpu, a.min_cpu,
  round(a.memory_size/1024/1024/1024,2) memory_size_gb,
  round(a.log_disk_size/1024/1024/1024,2) log_disk_size,
  round(a.log_disk_in_use/1024/1024/1024,2) log_disk_in_use,
  round(a.data_disk_in_use/1024/1024/1024,2) data_disk_in_use
  FROM oceanbase.GV$OB_UNITS a join oceanbase.dba_ob_tenants b on a.tenant_id=b.te
nant_id order by b.tenant_name;
```

输出如下，GV$OB_UNITS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_UNITS。

```
+-------+----------------+-------------+-------------+---------+---------+--------
--------+---------------+-----------------+------------------+
| zone  | svr_ip         | tenant_name | tenant_type | max_cpu | min_cpu | memory_
size_gb | log_disk_size | log_disk_in_use | data_disk_in_use |
+-------+----------------+-------------+-------------+---------+---------+--------
--------+---------------+-----------------+------------------+
| zone1 | 10.10.10.1     | META$1002   | META        |    NULL |    NULL
|          1.00 |          1.20 |            0.95 |             0.10 |
```

```
| zone1 | 10.10.10.1      | META$1004    | META          |     NULL |     NULL
|          1.00 |          1.20 |             0.90 |             0.08 |
| zone1 | 10.10.10.1      | ocp_meta     | USER          |        2 |        2
|          3.00 |         10.80 |             4.31 |             0.92 |
| zone1 | 10.10.10.1      | ocp_monitor | USER          |        2 |        2
|          3.00 |         10.80 |             2.91 |             0.22 |
| zone1 | 10.10.10.1      | sys          | SYS           |        3 |        3
|          2.00 |          3.00 |             1.33 |             0.06 |
+--------+-----------------+--------------+--------------+----------+----------+--------
--------+---------------+----------------+-------------------+
```

- 查看某租户下所有表占用磁盘大小

```
SELECT /*+ query_timeout(30000000) */ a.TENANT_ID, a.DATABASE_NAME, a.TABLE_NAME,
a.TABLE_ID,
  sum(case when b.nested_offset = 0 then IFNULL(b.data_block_count+b.index_block_c
ount+b.linked_block_count, 0) * 2 * 1024 * 1024 else IFNULL(b.size, 0) end) /1024.
0/1024/1024 as data_size_in_GB
  FROM oceanbase.CDB_OB_TABLE_LOCATIONS a inner join oceanbase.__all_virtual_table
_mgr b on a.svr_ip = b.svr_ip and a.svr_port=b.svr_port and a.tenant_id = b.tenant
_id and a.LS_ID = b.LS_ID and a.TABLET_ID = b.TABLET_ID and a.role ='LEADER' and a
.tenant_id = ${租户ID}
  and b.table_type >= 10 and b.size > 0 group by a.TABLE_ID;
```

您需将 `${租户ID}` 替换为待查看租户的租户 ID（可通过 `select * from oceanbase.DBA_OB_TENANTS;` 命令获取）。DBA_OB_TENANTS 视图和 CDB_OB_TABLE_LOCATIONS 视图的详细介绍可分别参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_TENANTS 和 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_TABLE_LOCATIONS。

- 查看某表单副本占用磁盘大小

```
SELECT sum(size)/1024/1024/1024 FROM (SELECT DATABASE_NAME,TABLE_NAME,TABLE_ID,PAR
TITION_NAME,TABLET_ID,`ROLE`
  FROM oceanbase.DBA_OB_TABLE_LOCATIONS ) AA full join
  (SELECT distinct(TABLET_ID) ,size
  FROM  oceanbase.GV$OB_SSTABLES ) BB on AA.TABLET_ID=BB.TABLET_ID
  WHERE  AA.role='leader' and AA.table_name='${table_name}';
```

您需将 `${table_name}` 替换为待查看的表名，输出如下：

```
+---------------------------+
| sum(size)/1024/1024/1024 |
+---------------------------+
```

```
|          0.000017967074 |
+---------------------------+
```

DBA_OB_TABLE_LOCATIONS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_TABLE_LOCATIONS，GV$OB_SSTABLES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_SSTABLES。

- 统计租户的大小

```
SELECT t.tenant_name,
  round(sum(t2.data_size)/1024/1024/1024,2) as data_size_gb,
  round(sum(t2.required_size)/1024/1024/1024,2) as required_size_gb
  FROM oceanbase.DBA_OB_TENANTS t,oceanbase.CDB_OB_TABLE_LOCATIONS t1,oceanbase.CD
B_OB_TABLET_REPLICAS t2
  WHERE t.tenant_id=t1.tenant_id and t1.svr_ip=t2.svr_ip and t1.tenant_id=t2.tenan
t_id and t1.ls_id=t2.ls_id and t1.tablet_id=t2.tablet_id
  group by t.tenant_name
  order by 3 desc;
```

输出如下：

```
+-----------------------+--------------+------------------+
| tenant_name           | data_size_gb | required_size_gb |
+-----------------------+--------------+------------------+
| sys                   |         0.04 |             0.04 |
| ob_archivedata_tenant |         0.02 |             0.02 |
| META$1002             |         0.01 |             0.01 |
| META$1010             |         0.01 |             0.01 |
+-----------------------+--------------+------------------+
```

DBA_OB_TENANTS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_TENANTS，CDB_OB_TABLE_LOCATIONS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_TABLE_LOCATIONS，CDB_OB_TABLET_REPLICAS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_TABLET_REPLICAS。

表分区相关 SQL

本节所用 SQL 均是通过查看 DBA_OB_TABLE_LOCATIONS 视图获取信息，DBA_OB_TABLE_LOCATIONS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/字典视图/oceanbase.DBA_OB_TABLE_LOCATIONS。

- 查看 `test` 库下 `t1` 表 Leader 分布

```
SELECT * FROM oceanbase.DBA_OB_TABLE_LOCATIONS WHERE DATABASE_NAME='test' and TABL
E_NAME='t1' and ROLE='LEADER' and TABLE_TYPE='USER TABLE';
```

输出如下：

```
+---------------+------------+----------+------------+----------------+----------
--------+------------+---------------+-----------+-------+-------+----------------+
----------+--------+---------------+-----------------+-----------+----------------
+---------------+----------+
| DATABASE_NAME | TABLE_NAME | TABLE_ID | TABLE_TYPE | PARTITION_NAME | SUBPARTITI
ON_NAME | INDEX_NAME | DATA_TABLE_ID | TABLET_ID | LS_ID | ZONE  | SVR_IP
| SVR_PORT | ROLE   | REPLICA_TYPE | DUPLICATE_SCOPE | OBJECT_ID | TABLEGROUP_NAM
E | TABLEGROUP_ID | SHARDING |
+---------------+------------+----------+------------+----------------+----------
--------+------------+---------------+-----------+-------+-------+----------------+
----------+--------+---------------+-----------------+-----------+----------------
+---------------+----------+
| test          | t1         |   525671 | USER TABLE | NULL           | NUL
L                | NULL       |          NULL |    225152 |  1002 | zone1 | 10.10.10
.1   |     2882 | LEADER | FULL          | NONE            |    525671 | NUL
L              |          NULL | NULL     |
+---------------+------------+----------+------------+----------------+----------
--------+------------+---------------+-----------+-------+-------+----------------+
----------+--------+---------------+-----------------+-----------+----------------
+---------------+----------+
```

- 计算 `test` 库下分区个数（包含索引分区）

```
SELECT count(*) FROM oceanbase.DBA_OB_TABLE_LOCATIONS WHERE DATABASE_NAME='test' a
nd ROLE='LEADER';
```

输出如下：

```
+----------+
| count(*) |
+----------+
|      242 |
+----------+
```

- 计算 `test` 库在某个 OBServer 节点上分区个数

```
SELECT count(*) FROM oceanbase.DBA_OB_TABLE_LOCATIONS
  WHERE SVR_IP='<server_ip>'
  and database_name='test'
  and ROLE='LEADER';
```

您需将 `<server_ip>` 替换为想要查询的 OBServer 节点 IP，输出如下。

```
+----------+
| count(*) |
+----------+
|      117 |
+----------+
```

表分布

本节所用 SQL 均是通过查看 CDB_OB_TABLE_LOCATIONS 视图获取信息，CDB_OB_TABLE_LOCATIONS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_TABLE_LOCATIONS。

- Partition 分布

```
SELECT svr_ip, count(1) FROM oceanbase.CDB_OB_TABLE_LOCATIONS
  WHERE tenant_id = 1002 group by svr_ip order by svr_ip;
```

输出如下：

```
+----------------+----------+
| svr_ip         | count(1) |
+----------------+----------+
| 10.10.10.1     |      1156|
+----------------+----------+
```

- Partition Leader 分布

```
SELECT svr_ip, count(1) FROM oceanbase.CDB_OB_TABLE_LOCATIONS
  WHERE tenant_id = 1002 and role = 'leader' group by svr_ip order by svr_ip;
```

输出如下：

```
+----------------+----------+
| svr_ip         | count(1) |
+----------------+----------+
|  10.10.10.1    |     1156 |
+----------------+----------+
```

- 业务库下 Leader 分布

```
SELECT svr_ip, count(1) FROM oceanbase.CDB_OB_TABLE_LOCATIONS
  WHERE tenant_id = 1002 and role = 'LEADER' and DATABASE_NAME not in ('oceanbase'
,'mysql') group by svr_ip order by svr_ip;
```

输出如下:

```
+----------------+----------+
| svr_ip         | count(1) |
+----------------+----------+
|  10.10.10.1    |      566 |
+----------------+----------+
```

转储合并相关 SQL

- 手动发起所有租户转储

```
ALTER SYSTEM MINOR FREEZE tenant= all_user;
```

- 手动发起单个租户转储，以业务名为 `test` 为例

```
ALTER SYSTEM MINOR FREEZE tenant= 'test';
```

- 查看已经转储次数和内存阈值

```
SELECT * FROM oceanbase.gv$ob_memstore;
```

输出如下，GV$OB_MEMSTORE 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_MEMSTORE。

```
+----------------+----------+-----------+-------------+----------------+----------
--+----------------+-----------------+
| SVR_IP         | SVR_PORT | TENANT_ID | ACTIVE_SPAN | FREEZE_TRIGGER | FREEZE_CN
T | MEMSTORE_USED | MEMSTORE_LIMIT |
```

```
+----------------+----------+-----------+-------------+---------------+----------
--+---------------+----------------+
| 10.10.10.1     |     2882 |         1 |   138412032 |     290393948 |          
0 |     138412032 |     1288490160 |
| 10.10.10.1     |     2882 |      1001 |    75497472 |      90013604 |         9
2 |      73400320 |      429496720 |
| 10.10.10.1     |     2882 |      1009 |    48234496 |      90642744 |         9
9 |      44040192 |      429496720 |
| 10.10.10.1     |     2882 |      1010 |    37748736 |     452510424 |          
0 |      37748736 |     2147483640 |
+----------------+----------+-----------+-------------+---------------+----------
--+---------------+----------------+
```

• 查看转储配置

```
SHOW PARAMETERS WHERE name in ('memstore_limit_percentage','freeze_trigger_percent
age','writing_throttling_trigger_percentage','memory_limit');
```

输出如下：

```
+-------+----------+----------------+----------+---------------------------------
-----+----------+-------+--------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
-------------------------------------------------------------------+----------+--
-------+---------+-------------------+---------------+-----------+
| zone  | svr_type | svr_ip         | svr_port | nam
e                                   | data_type | value | inf
o





                                       | section   | scope     | source    | edit_leve
l         | default_value | isdefault |
+-------+----------+----------------+----------+---------------------------------
-----+----------+-------+--------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
-------------------------------------------------------------------+----------+--
-------+---------+-------------------+---------------+-----------+
| zone1 | observer | 10.10.10.1     |     2882 | memstore_limit_percentag
```

```
e              | INT        | 0      | used in calculating the value of MEMSTORE_LIMI
T parameter: memstore_limit_percentage = memstore_limit / memory_size, where MEMOR
Y_SIZE is determined when the tenant is created. Range: [0, 100). 1. the system wi
ll use memstore_limit_percentage if only memstore_limit_percentage is set.2. the s
ystem will use _memstore_limit_percentage if both memstore_limit_percentage and _m
emstore_limit_percentage is set.3. the system will adjust automatically if both me
mstore_limit_percentage and _memstore_limit_percentage set to 0(by default). | TEN
ANT    | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE | 0                 |          1 |
| zone1 | observer | 10.10.10.1      |      2882 | memory_limi
t                          | CAPACITY   | 30G     | the size of the memory reserved f
or internal use(for testing purpose), 0 means follow memory_limit_percentage. Rang
e: 0, [1G,)
.



                                                                                     | O
BSERVER | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE | 0M              |          0 |
| zone1 | observer | 10.10.10.1      |      2882 | writing_throttling_trigger_percen
tage | INT        | 60      | the threshold of the size of the mem store when writing
_limit will be triggered. Rang:(0,100]. setting 100 means turn off writing limi
t



                                                                                | TRANS
| TENANT   | DEFAULT | DYNAMIC_EFFECTIVE | 60              |          1 |
| zone1 | observer | 10.10.10.1      |      2882 | freeze_trigger_percentag
e              | INT        | 20      | the threshold of the size of the mem store whe
n freeze will be triggered. Rang:(0,100
)




                                      | TENANT    | TENANT    | DEFAULT | DYNAMIC_EFF
ECTIVE | 20              |          1 |
+--------+----------+---------------+----------+------------------------------------
-----+-----------+-------+-----------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
---------------------------------------------------------------------+----------+--
-------+---------+-------------------+---------------+-----------+
```

- 查看全局合并是否开启

```
SHOW PARAMETERS LIKE 'enable_major_freeze';
```

```

```

输出如下：

```
+-------+----------+---------------+----------+---------------------+-----------+
-------+-------------------------------------------------------------------------
----------------------+-------------+--------+--------+------------------+
| zone  | svr_type | svr_ip        | svr_port | name                | data_type
| value | inf
o
           | section      | scope    | source   | edit_level        |
+-------+----------+---------------+----------+---------------------+-----------+
-------+-------------------------------------------------------------------------
----------------------+-------------+--------+--------+------------------+
| zone1 | observer | 10.10.10.1    |     2882 | enable_major_freeze | BOOL
| True  | specifies whether major_freeze function is turned on. Value:  True:turne
d on;  False: turned off | ROOT_SERVICE | CLUSTER | DEFAULT | DYNAMIC_EFFECTIVE |
+-------+----------+---------------+----------+---------------------+-----------+
-------+-------------------------------------------------------------------------
----------------------+-------------+--------+--------+------------------+
```

- 手动发起所有租户合并

```
ALTER SYSTEM MAJOR FREEZE TENANT= all_user;
```

- 手动发起单个租户合并，以用户租户名为 `test` 为例

```
ALTER SYSTEM MAJOR FREEZE TENANT='test';
```

- 查看租户各个 Zone 的合并过程

```
SELECT * FROM oceanbase.CDB_OB_ZONE_MAJOR_COMPACTION;
```

输出如下，CDB_OB_ZONE_MAJOR_COMPACTION 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_ZONE_MAJOR_COMPACTION。

```
+-----------+-------+---------------------+---------------------+----------------
-----------+----------------------------+--------+
| TENANT_ID | ZONE  | BROADCAST_SCN       | LAST_SCN            | LAST_FINISH_TIM
E         | START_TIME                 | STATUS |
+-----------+-------+---------------------+---------------------+----------------
-----------+----------------------------+--------+
|         1 | zone1 | 1721152800548834000 | 1721152800548834000 | 2024-07-17 02:02
```

```
:45.384450 | 2024-07-17 02:00:00.592597 | IDLE   |
|       1001 | zone1 | 1721152803954045000 | 1721152803954045000 | 2024-07-17 02:04
:38.201694 | 2024-07-17 02:00:04.011316 | IDLE   |
|       1009 | zone1 | 1721152804388328000 | 1721152804388328000 | 2024-07-17 02:05
:28.617318 | 2024-07-17 02:00:04.438908 | IDLE   |
|       1010 | zone1 | 1721152800394534000 | 1721152800394534000 | 2024-07-17 02:02
:33.460294 | 2024-07-17 02:00:00.448116 | IDLE   |
+-----------+-------+---------------------+---------------------+-----------------
-----------+----------------------------+--------+
```

- 查看所有租户的合并信息

```
SELECT * FROM oceanbase.CDB_OB_MAJOR_COMPACTION;
```

输出如下，CDB_OB_MAJOR_COMPACTION 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_MAJOR_COMPACTION。

```
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+-------------------------
--+--------+----------+---------------+------+
| TENANT_ID | FROZEN_SCN          | FROZEN_TIME                | GLOBAL_BROADCAST_
SCN | LAST_SCN            | LAST_FINISH_TIME           | START_TIM
E                  | STATUS | IS_ERROR | IS_SUSPENDED | INFO |
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+-------------------------
--+--------+----------+---------------+------+
|         1 | 1721152800548834000 | 2024-07-17 02:00:00.548834 |  1721152800548834
000 | 1721152800548834000 | 2024-07-17 02:02:45.381779 | 2024-07-17 02:00:00.57535
8 | IDLE   | NO       | NO            |      |
|      1001 | 1721152803954045000 | 2024-07-17 02:00:03.954045 |  1721152803954045
000 | 1721152803954045000 | 2024-07-17 02:04:38.198968 | 2024-07-17 02:00:03.99654
5 | IDLE   | NO       | NO            |      |
|      1009 | 1721152804388328000 | 2024-07-17 02:00:04.388328 |  1721152804388328
000 | 1721152804388328000 | 2024-07-17 02:05:28.614793 | 2024-07-17 02:00:04.42330
2 | IDLE   | NO       | NO            |      |
|      1010 | 1721152800394534000 | 2024-07-17 02:00:00.394534 |  1721152800394534
000 | 1721152800394534000 | 2024-07-17 02:02:33.456848 | 2024-07-17 02:00:00.42945
1 | IDLE   | NO       | NO            |      |
+-----------+---------------------+----------------------------+------------------
----+---------------------+----------------------------+-------------------------
--+--------+----------+---------------+------+
```

- 查看合并剩余表数量

```
SELECT * FROM oceanbase.GV$OB_COMPACTION_PROGRESS
  WHERE UNFINISHED_TABLET_COUNT != 0;
```

GV$OB_COMPACTION_PROGRESS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/性能视图/GV$OB_COMPACTION_PROGRESS。

- 查看某租户合并延迟信息

```
SELECT * FROM oceanbase.__all_rootservice_event_history WHERE name1='tenant_id' an
d value1='${租户ID}' and module = 'daily_merge' order by gmt_create desc  limit 1;
```

您需将 `${租户ID}` 替换为待查看租户的租户 ID，可通过 `select * from oceanbase.DBA_OB_TENANTS;` 命令获取，DBA_OB_TENANTS 视图的详细介绍可分别参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.DBA_OB_TENANTS。输出如下：

```
+----------------------------+--------------+----------------+-----------+---------+-
---------------------+----------------------+--------+--------+--------+--------+----
---+--------+-------+--------+------------+----------------+--------------+
| gmt_create                 | module       | event          | name1     | value1 |
name2                | value2               | name3 | value3 | name4 | value4 | nam
e5 | value5 | name6 | value6 | extra_info | rs_svr_ip      | rs_svr_port |
+----------------------------+--------------+----------------+-----------+---------+-
---------------------+----------------------+--------+--------+--------+--------+----
---+--------+-------+--------+------------+----------------+--------------+
| 2024-07-18 02:04:45.332640 | daily_merge | global_merged | tenant_id | 1001    |
global_broadcast_scn | 1721239201197860000 |        |        |        |
|       |        |        |        |             | 10.10.10.1      |         2882 |
+----------------------------+--------------+----------------+-----------+---------+-
---------------------+----------------------+--------+--------+--------+--------+----
---+--------+-------+--------+------------+----------------+--------------+
```

- 查看当前正在执行的转储/合并任务

```
SELECT * FROM oceanbase.GV$OB_TABLET_COMPACTION_PROGRESS WHERE TYPE='MINI_MERGE';
```

`TYPE` 中设置为 `MINI_MERGE` 表示查看转储任务，冻结 MemTable 通过转储变成 Mini SSTable；`TYPE` 中设置为 `MAJOR_MERGE` 表示查看合并任务。GV$OB_TABLET_COMPACTION_PROGRESS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/性能视图/

GV$OB_TABLET_COMPACTION_PROGRESS。

- 查看转储/合并历史信息

```
SELECT * FROM oceanbase.GV$OB_TABLET_COMPACTION_HISTORY
  WHERE TYPE='MINI_MERGE' order by START_TIME desc limit 3;
```

TYPE 中设置为 MINI_MERGE 表示查看转储历史信息，冻结 MemTable 通过转储变成 Mini SSTable；TYPE 中设置为 MAJOR_MERGE 表示查看合并历史信息。输出如下：

```
+---------------+----------+-----------+-------+-----------+------------+---------
------------+----------------------------+----------------------------+-----------
------------------------+-------------+-------------------+-----------------------
--------+------------------------------+--------------------------------------+---
--------------+----------------------+-------------------+---------------------+-
-----------------------------+----------------------------+-----------------+-----
----------+-----------------------------------------------------------------------
-+---------------+----------------------------------------------------------------
----+-------------+-----------+---------------------------------------------------
--------------+-------------------+
| SVR_IP          | SVR_PORT | TENANT_ID | LS_ID | TABLET_ID | TYPE         | COMPACTI
ON_SCN      | START_TIME                 | FINISH_TIME                | TASK_I
D                           | OCCUPY_SIZE | MACRO_BLOCK_COUNT | MULTIPLEXED_MACRO_
BLOCK_COUNT | NEW_MICRO_COUNT_IN_NEW_MACRO | MULTIPLEXED_MICRO_COUNT_IN_NEW_MACRO
| TOTAL_ROW_COUNT | INCREMENTAL_ROW_COUNT | COMPRESSION_RATIO | NEW_FLUSH_DATA_RAT
E | PROGRESSIVE_COMPACTION_ROUND | PROGRESSIVE_COMPACTION_NUM | PARALLEL_DEGREE |
PARALLEL_INFO | PARTICIPANT_TABL
E                                                      | MACRO_ID_LIST | COMMENT
S                                                           |
+---------------+----------+-----------+-------+-----------+------------+---------
------------+----------------------------+----------------------------+-----------
------------------------+-------------+-------------------+-----------------------
--------+------------------------------+--------------------------------------+---
--------------+----------------------+-------------------+---------------------+-
-----------------------------+----------------------------+-----------------+-----
----------+-----------------------------------------------------------------------
-+---------------+----------------------------------------------------------------
----+
| 10.10.10.1     |     2882 |      1002 |     1 |       373 | MINI_MERGE | 1721152
976299008000 | 2024-07-17 02:03:08.509109 | 2024-07-17 02:03:08.524133 | YB420BA1C
C62-00061A0898BDDE08-0-0 |       14045 |                 1
|                             0 |                            1
|                                    0 |             101 |                   101
|                 1 |                4896 |                            0
|                          0 |               1 | -             | table_cnt=1,start
_scn=1721112282660948001,end_scn=1721152976299008000; | 18831         | comment="c
ost_mb=2;time=add_time:1721152988508926|total=15.02ms;"; |
| 10.10.10.1     |     2882 |      1002 |     1 |       121 | MINI_MERGE | 1721152
```

```
980999256000 | 2024-07-17 02:03:03.506837 | 2024-07-17 02:03:03.576037 | YB420BA1C
C62-00061A0898BDDC45-0-0 |           0 |                 0
|                              0 |                            0
|                                     0 |              0 |                       0
|                 1 |                    0 |                          0
|                           0 |                 1 | -             | table_cnt=1,start
_scn=1721152803751899001,end_scn=1721152980999256000; |              | comment="t
ime=add_time:1721152983506636|total=69.20ms;";          |
| 10.10.10.1      |      2882 |      1002 |      1 |           1 | MINI_MERGE | 1721152
973925564001 | 2024-07-17 02:02:58.504770 | 2024-07-17 02:02:58.569947 | YB420BA1C
C62-00061A0898BDDA52-0-0 |        2204 |                 1
|                              0 |                            1
|                                     0 |             18 |                      18
|                 1 |                   818 |                          0
|                           0 |                 1 | -             | table_cnt=1,start
_scn=1721152803751899001,end_scn=1721152973925564001; | 18736         | comment="c
ost_mb=2;time=add_time:1721152978504549|total=65.18ms;"; |
+----------------+----------+-----------+-------+-----------+------------+---------
------------+-----------------------------+-----------------------------+----------
------------------------+-------------+--------------------+-----------------------
--------+------------------------------+---------------------------------------+---
--------------+------------------------+--------------------+---------------------+-
------------------------------+----------------------------+------------------+-----
----------+-----------------------------------------------------------------------
-+----------------+-------------------------------------------------------------------
----+
```

GV$OB_TABLET_COMPACTION_HISTORY 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/性能视图/GV$OB_TABLET_COMPACTION_HISTORY。

- 查看某张表的合并记录

```
SELECT * FROM oceanbase.GV$OB_TABLET_COMPACTION_HISTORY
  WHERE TABLET_ID IN
  (SELECT TABLET_ID FROM oceanbase.CDB_OB_TABLE_LOCATIONS WHERE TABLE_NAME = '${ta
ble_name}') ORDER BY START_TIME DESC limit 1;
```

您需将 `${table_name}` 替换为待查看的表名。GV$OB_TABLET_COMPACTION_HISTORY 视图和 CDB_OB_TABLE_LOCATIONS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/性能视图/GV$OB_TABLET_COMPACTION_HISTORY 和 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_TABLE_LOCATIONS。输出如下：

```
+----------------+----------+-----------+-------+-----------+---------------+-----
```

```
----------------+----------------------------+----------------------------+-------
----------------------------+-------------+--------------------+------------------
------------+-------------------------------+-------------------------------------
+-----------------+-----------------------+--------------------+------------------
--+------------------------------+----------------------------+-----------------+-
--------------+-------------------+---------------+-------------------------------
--------------------------+-------------+-----------+---------------+-----------
------+
| SVR_IP         | SVR_PORT | TENANT_ID | LS_ID | TABLET_ID | TYPE             | COMP
ACTION_SCN       | START_TIME                 | FINISH_TIME                | TASK_I
D                            | OCCUPY_SIZE | MACRO_BLOCK_COUNT | MULTIPLEXED_MACRO_
BLOCK_COUNT | NEW_MICRO_COUNT_IN_NEW_MACRO | MULTIPLEXED_MICRO_COUNT_IN_NEW_MACRO
| TOTAL_ROW_COUNT | INCREMENTAL_ROW_COUNT | COMPRESSION_RATIO | NEW_FLUSH_DATA_RAT
E | PROGRESSIVE_COMPACTION_ROUND | PROGRESSIVE_COMPACTION_NUM | PARALLEL_DEGREE |
PARALLEL_INFO | PARTICIPANT_TABLE | MACRO_ID_LIST | COMMENT
S                                                 |
+----------------+----------+-----------+-------+-----------+-----------------+-----
----------------+----------------------------+----------------------------+-------
----------------------------+-------------+--------------------+------------------
------------+-------------------------------+-------------------------------------
+-----------------+-----------------------+--------------------+------------------
--+------------------------------+----------------------------+-----------------+-
--------------+-------------------+---------------+-------------------------------
--------------------------+
| 10.10.10.1     |     2882 |         1 |     1 |    200030 | MDS_MINI_MERGE | 172
1273439647032000 | 2024-07-18 11:30:40.065549 | 2024-07-18 11:30:40.072639 | YB420
BA1CC62-00061A08028BAA9E-0-0 |           0 |                  0
|                            0 |                              0
|                      0 |                 0 |                        0
|                 1 |                          0 |                            0
|                        0 |                 0 | -
|                   |                | comment="time=add_time:1721273439825215|tota
l=7.09ms;"; |
+----------------+----------+-----------+-------+-----------+-----------------+-----
----------------+----------------------------+----------------------------+-------
----------------------------+-------------+--------------------+------------------
------------+-------------------------------+-------------------------------------
+-----------------+-----------------------+--------------------+------------------
--+------------------------------+----------------------------+-----------------+-
--------------+-------------------+---------------+-------------------------------
--------------------------+
```

执行计划缓存相关

- 查看执行计划信息

```
SELECT * FROM oceanbase.GV$OB_PLAN_CACHE_PLAN_stat where query_sql like '%关键字%';
```

您需将 `%关键字%` 替换为 SQL 语句的关键字。

- 清空某租户的所有计划缓存

```
ALTER SYSTEM FLUSH PLAN CACHE TENANT = '${tenant_name}' global;
```

您需将 `${tenant_name}` 替换为待清空租户的租户名。

- 清空某租户下面某个库的所有计划缓存

```
ALTER SYSTEM FLUSH PLAN CACHE databases='${database}' tenant='${tenant_name}' GLOBAL
;
```

您需将 `${database}` 替换为对应的库名，并将 `${tenant_name}` 替换为 `${database}` 所在租户的租户名。

收集统计信息相关

- 手动收集统计信息
  - call 方式收集 `test` 库下的 `test1` 表

    ```
    call dbms_stats.gather_table_stats('test','test1');
    ```

  - call 方式收集 `test` 下的所有表，并且设置并行度为 `128`

    ```
    call dbms_stats.gather_schema_stats('test', degree=>128);
    ```

  - ANALYZE 方式收集 `test` 库下的 `test1` 表

    ```
    ALTER SYSTEM SET enable_sql_extension = true;
    ANALYZE TABLE test.test1 COMPUTE STATISTICS;
    ```

- call 方式收集 `test` 库下的 `test1` 表

```
call dbms_stats.gather_table_stats('test','test1');
```

- call 方式收集 `test` 下的所有表，并且设置并行度为 `128`

```
call dbms_stats.gather_schema_stats('test', degree=>128);
```

- ANALYZE 方式收集 `test` 库下的 `test1` 表

```
ALTER SYSTEM SET enable_sql_extension = true;
ANALYZE TABLE test.test1 COMPUTE STATISTICS;
```

- 查看收集后的 `test1` 表统计信息

```
SELECT * FROM oceanbase.DBA_TAB_STATISTICS WHERE TABLE_NAME='test1';
```

DBA_TAB_STATISTICS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/字典视图/oceanbase.DBA_TAB_STATISTICS。

- 查看租户收集表统计信息时耗时排序

```
SELECT table_id,tenant_id,task_id,ret_code,start_time,end_time,TIMEDIFF(end_time,
start_time) as times FROM oceanbase.__all_virtual_table_opt_stat_gather_history or
der by times desc limit 5;
```

输出如下:

```
+----------+-----------+--------------------------------------+----------+--------
--------------------+----------------------------+-----------------+
| table_id | tenant_id | task_id                              | ret_code | start_t
ime                        | end_time                   | times           |
+----------+-----------+--------------------------------------+----------+--------
--------------------+----------------------------+-----------------+
|   525657 |      1002 | 36507c6d-32fb-11ef-a31f-00163e048137 |        0 | 2024-06
-25 22:00:00.173423 | 2024-06-25 22:01:51.433552 | 00:01:51.260129 |
|      458 |         1 | 5f2ad8bf-387b-11ef-8765-00163e035a65 |        0 | 2024-07
-02 22:00:00.136656 | 2024-07-02 22:00:01.743898 | 00:00:01.607242 |
|      458 |         1 | 3646d5bc-32fb-11ef-a31f-00163e06beb9 |        0 | 2024-06
-25 22:00:00.085670 | 2024-06-25 22:00:01.663990 | 00:00:01.578320 |
|      458 |      1001 | 60c1e763-33c4-11ef-a31f-00163e06beb9 |        0 | 2024-06
-26 22:00:00.252122 | 2024-06-26 22:00:01.822895 | 00:00:01.570773 |
|      458 |         1 | 3fddfbd3-4163-11ef-885e-00163e048137 |        0 | 2024-07
-14 06:00:00.235859 | 2024-07-14 06:00:01.800427 | 00:00:01.564568 |
+----------+-----------+--------------------------------------+----------+--------
--------------------+----------------------------+-----------------+
```

其他

- 查看 Root Service 切换信息

```
SELECT * FROM __all_rootservice_event_history WHERE module='root_service' and even
t in ('start_rootservice','stop_rootservice') order by gmt_create desc;
```

- 通过 table_id 查表相关信息

  table_id 在租户中是唯一的，但在集群中不同租户可以使用重复的 ID。table_id 对应的不一定是表，也可能是索引、视图等。可通过 CDB_OB_TABLE_LOCATIONS 视图查出 table 对象对应的表名、库名、租户名、Leader、表类型等等，示例如下。

```
SELECT * FROM oceanbase.CDB_OB_TABLE_LOCATIONS where table_id = 500008;
```

  输出如下，CDB_OB_TABLE_LOCATIONS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_OB_TABLE_LOCATIONS。

```
+-----------+---------------+------------------+----------+------------+----------
------+------------------+-----------+---------------+-----------+-------+------
-+---------------+----------+----------+--------------+------------------+--------
---+-----------------+-------------------+----------+
| TENANT_ID | DATABASE_NAME | TABLE_NAME       | TABLE_ID | TABLE_TYPE | PARTITION
_NAME | SUBPARTITION_NAME | INDEX_NAME | DATA_TABLE_ID | TABLET_ID | LS_ID | ZON
E   | SVR_IP          | SVR_PORT | ROLE     | REPLICA_TYPE | DUPLICATE_SCOPE | OBJEC
T_ID | TABLEGROUP_NAME | TABLEGROUP_ID | SHARDING |
+-----------+---------------+------------------+----------+------------+----------
------+------------------+-----------+---------------+-----------+-------+------
-+---------------+----------+----------+--------------+------------------+--------
---+-----------------+-------------------+----------+
|         1 | test          | cluster_slowlogs |   500008 | USER TABLE | NUL
L           | NULL              | NULL       |           NULL |     200002 |      1
| zone1 | 10.10.10.1     |      2882 | FOLLOWER | FULL           | NONE
|    500008 | NULL              |            NULL | NULL      |
+-----------+---------------+------------------+----------+------------+----------
------+------------------+-----------+---------------+-----------+-------+------
-+---------------+----------+----------+--------------+------------------+--------
---+-----------------+-------------------+----------+
```

- 查看索引是否生效

  - sys 租户下查索引是否生效，状态为 `VALID` 表示正常

```
SELECT * FROM oceanbase.CDB_INDEXES where status<>'VALID' and INDEX_NAME=
'%<索引名>%';
```

需要将 `<索引名>` 替换为实际要查询的索引名，输出为空表示索引全部正常。

CDB_INDEXES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_INDEXES。

  - 用 table_id 查询通过索引名查询 table_id 等信息，需要将 `<索引名>` 替换为实际要查询的索引名。通过输出的 `table_id` 查询索引创建失败信息，输出为空表示索引全部正常。

  ```
  SELECT * FROM  oceanbase.__all_virtual_ddl_error_message WHERE object_id
  = '${table_id}';
  ```

- sys 租户下查索引是否生效，状态为 `VALID` 表示正常

```
SELECT * FROM oceanbase.CDB_INDEXES where status<>'VALID' and INDEX_NAME='%<索引名>
%';
```

需要将 `<索引名>` 替换为实际要查询的索引名，输出为空表示索引全部正常。CDB_INDEXES 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/SYS 租户系统视图/字典视图/oceanbase.CDB_INDEXES。

- 用 table_id 查询

通过索引名查询 table_id 等信息，需要将 `<索引名>` 替换为实际要查询的索引名。

```
SELECT table_id, table_name FROM oceanbase.__all_virtual_table_history WHERE table
_name like '%<索引名>%';
```

通过输出的 `table_id` 查询索引创建失败信息，输出为空表示索引全部正常。

```
SELECT * FROM  oceanbase.__all_virtual_ddl_error_message WHERE object_id = '${tabl
e_id}';
```

- 查看某租户慢事务信息

```
SELECT /*+READ_CONSISTENCY(weak),parallel(8),QUERY_TIMEOUT(180000000)*/ usec_to_ti
me(request_time),*
  FROM oceanbase.GV$OB_SQL_AUDIT
  WHERE tenant_id = ${租户ID} and RET_CODE = '-5066' and QUERY_SQL like 'REPLACE%' a
nd usec_to_time(request_time) between '2024-05-01 15:25:00' and now() order by ELA
PSED_TIME desc;
```

您需将 `${租户ID}` 替换为待查看租户的租户 ID（可通过 `select * from oceanbase.DBA_OB_TENANTS;` 命令获取）。示例中的 `2024-05-01 15:25:00` 需根据实际情况进行替换。GV$OB_SQL_AUDIT 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 优化/SQL 执行性能监控/GV$OB_SQL_AUDIT。

- 查看长事务和悬挂事务

  此处以查询从当前时刻起，往前追溯 600s 内发生的事务为例，你可根据实际情况将下述示例中的 `600` 替换为希望追溯的时间。

  - 查看长事务

    ```
    SELECT count(1) FROM oceanbase.__all_virtual_trans_stat
      WHERE part_trans_action<=2 AND ctx_create_time < date_sub(now(), INTERV
    AL 600 SECOND) AND is_exiting != 1;
    ```

  - 查看悬挂事务

    ```
    SELECT count(1) FROM oceanbase.__all_virtual_trans_stat
      WHERE part_trans_action > 2 AND ctx_create_time < date_sub(now(), INTER
    VAL 600 SECOND) AND is_exiting != 1;
    ```

- 查看长事务

  ```
  SELECT count(1) FROM oceanbase.__all_virtual_trans_stat
    WHERE part_trans_action<=2 AND ctx_create_time < date_sub(now(), INTERVAL 600 SE
  COND) AND is_exiting != 1;
  ```

- 查看悬挂事务

  ```
  SELECT count(1) FROM oceanbase.__all_virtual_trans_stat
    WHERE part_trans_action > 2 AND ctx_create_time < date_sub(now(), INTERVAL 600 S
  ECOND) AND is_exiting != 1;
  ```

- 查看创建索引进度

  ```
  SELECT
    SUBSTRING_INDEX(SUBSTRING_INDEX(MESSAGE, 'ROW_INSERTED: ', -1), ',', 1) AS row_i
  nserted,
    SUBSTRING_INDEX(SUBSTRING_INDEX(MESSAGE, 'ROW_SCANNED: ', -1), ',', 1) AS row_sc
  anned,
    (SUBSTRING_INDEX(SUBSTRING_INDEX(MESSAGE, 'ROW_INSERTED: ', -1), ',', 1) /SUBSTR
  ING_INDEX  (SUBSTRING_INDEX(MESSAGE, 'ROW_SCANNED: ', -1), ',', 1)) AS ratio
  ```

```
  FROM oceanbase.GV$SESSION_LONGOPS;
```

GV$SESSION_LONGOPS 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/性能视图/GV$SESSION_LONGOPS。

- 查看某租户宏块利用率

```
SELECT loc.TENANT_ID,loc.DATABASE_NAME,loc.TABLE_NAME,loc.TABLE_ID,
  SUM(rep.DATA_SIZE) AS DATA_SIZE,
  SUM(rep.REQUIRED_SIZE) AS REQUIRED_SIZE,
  SUM(rep.DATA_SIZE) / NULLIF(SUM(rep.REQUIRED_SIZE), 0) AS SIZE_RATIO
  FROM oceanbase.CDB_OB_TABLE_LOCATIONS loc
  JOIN oceanbase.CDB_OB_TABLET_REPLICAS rep
  ON loc.TABLET_ID = rep.TABLET_ID
  WHERE loc.TABLE_TYPE = 'USER TABLE' and loc.TENANT_ID = ${租户ID}
  GROUP BY loc.DATABASE_NAME,loc.TABLE_NAME,loc.TABLE_ID;
```

您需将 `${租户ID}` 替换为待查看租户的租户 ID（可通过 `select * from oceanbase.DBA_OB_TENANTS;` 命令获取）。

常见问题运维

如何获取 SQL 语句对应的数据库日志信息

此类问题可以理解为如何快速找到异常 SQL 的日志信息。大部分 SQL 命令问题可能无法单独通过各种视图确认问题具体原因，需要依赖数据库系统日志进行排查。但是 OceanBase 数据库日志量比较大，而且内容刷新比较快，如果对数据库操作不熟悉，可能会因为系统日志被覆盖而无法找到异常 SQL 的信息。此运维操作将从如何调整系统日志、打印 trace_id、定位系统日志内容方面进行介绍。

问题现象：业务侧执行的 SQL 失败或任务未完成，无法通过视图信息确认问题原因。

该问题根据出现的场景不同可分为如下两种处理方法，可根据实际场景选择合适的处理方法。

场景一：SQL 不可重复执行

1. 查看 GV$OB_SQL_AUDIT 视图，找到对应异常 SQL 语句，获取到 `TRACE_ID` 和 `svr_ip` 信

息。

> **说明**
>
> 该视图记录的数据不会持久化本地，数据留存时间受租户内存大小影响，由系统变量 `ob_sql_audit_percentage` 控制，如果该视图无法查询到 SQL 信息，可以尝试调整 `ob_sql_audit_percentage` 占比。系统变量 `ob_sql_audit_percentage` 的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Global 系统变量/ob_sql_audit_percentage。

```
select query_sql,svr_ip,TRACE_ID,client_ip,TENANT_NAME,user_name,DB_NAME,ELAPSED_T
IME,RET_CODE,FROM_UNIXTIME(ROUND(REQUEST_TIME/1000/1000),'%Y-%m-%d %H:%i:%S') fro
m `GV$OB_SQL_AUDIT` WHERE REQUEST_TIME>='2024-04-05 14:34:00'  limit 10;
```

GV$OB_SQL_AUDIT 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 优化/SQL 执行性能监控/GV$OB_SQL_AUDIT。

2. 在对应 `svr_ip` 机器上，通过 `TRACE_ID` 过滤日志信息。

   不同异常场景的问题不一定都在 `observer.log` 日志中，例如执行备份命令失败，是可以在 `rootservice.log` 日志中解析出报错原因的。

```
[root@test001 ~]$ grep "YB420BA1CC68-000615A09D4CBDA0-0-0" observer.log*
[root@test001 ~]$ grep "YB420BA1CC68-000615A09D4CBDA0-0-0" rootservice.log*
```

   若根据输出的日志信息无法分析具体报错原因，可根据日志中的错误码在官方 知识库文档 中搜索相关信息。若仍旧无法解决，您也可以到官网 问答区 发帖，会有专业人员协助解决。

3. 如果 GV$OB_SQL_AUDIT 视图中的信息被淘汰、数据库系统日志被刷掉或者系统日志等级太低，可以按以下方式设置，等待 SQL 可执行复现问题时再按上述流程获取日志。

   1. 查看系统日志等级输出如下：

```
+--------+----------+-----------------+----------+---------------+-----------+-------+-
----------------------------------------------------------------------------------------
----------------+---------------+-----------+
| zone   | svr_type | svr_ip          | svr_port | name          | data_type | value |
o
| section  | scope    | source   | edit_level        | default_value | isdefault |
+--------+----------+-----------------+----------+---------------+-----------+-------+-
----------------------------------------------------------------------------------------
----------------+---------------+-----------+
```

```
| zone1 | observer | 10.10.10.1      |     2882 | syslog_level | STRING     | WDIAG |
. There are DEBUG, TRACE, WDIAG, EDIAG, INFO, WARN, ERROR, seven different log level
NAMIC_EFFECTIVE | WDIAG          |          1 |
+-------+----------+----------------+----------+--------------+-----------+-------+-
-------------------------------------------------------------------------------------
----------------+---------------+-----------+
```

2. （可选）设置日志等级syslog_level 参数默认值是 `WDIAG`，若该查看值为 `WDIAG`，可跳过本步骤。

```
alter system set syslog_level='WDIAG';
```

3. 设置日志保留个数，设置时需要注意磁盘空间是否能支撑设置的日志数量

```
alter system set max_syslog_file_count='10';
```

4. 查看系统日志等级

```
show parameters like 'syslog_level';
```

输出如下：

```
+-------+----------+----------------+----------+--------------+-----------+-------
+---------------------------------------------------------------------------------
--------------------------------------------+----------+---------+---------+-----
--------------+---------------+-----------+
| zone  | svr_type | svr_ip         | svr_port | name         | data_type | value
| inf
o
                                            | section  | scope    | source  | edit_leve
l         | default_value | isdefault |
+-------+----------+----------------+----------+--------------+-----------+-------
+---------------------------------------------------------------------------------
--------------------------------------------+----------+---------+---------+-----
--------------+---------------+-----------+
| zone1 | observer | 10.10.10.1      |     2882 | syslog_level | STRING     | WDIAG
| specifies the current level of logging. There are DEBUG, TRACE, WDIAG, EDIAG, IN
FO, WARN, ERROR, seven different log levels. | OBSERVER | CLUSTER | DEFAULT | DYNA
MIC_EFFECTIVE | WDIAG          |          1 |
+-------+----------+----------------+----------+--------------+-----------+-------
+---------------------------------------------------------------------------------
--------------------------------------------+----------+---------+---------+-----
--------------+---------------+-----------+
```

5. （可选）设置日志等级

syslog_level 参数默认值是 `WDIAG`，若该查看值为 `WDIAG`，可跳过本步骤。

```
alter system set syslog_level='WDIAG';
```

6. 设置日志保留个数，设置时需要注意磁盘空间是否能支撑设置的日志数量

```
alter system set max_syslog_file_count='10';
```

场景二：SQL 可重复执行

可以先按上述查看 GV$OB_SQL_AUDIT 视图的排查方式进行操作，或者多执行几次 SQL 语句复现。如果无法获取到需要的信息，可通过如下操作获取 trace_id 并过滤日志，通过系统租户和用户租户均可打印 trace_id，您可根据不同场景选择合适的操作租户。

- 如果 SQL 执行立刻报错，推荐使用系统租户获取 `trace_id`。

  1. 登录系统租户，打开 `enable_rich_error_msg` 参数。enable_rich_error_msg 参数用于设置是否在客户端消息中添加服务器地址、时间、追踪 ID 等调试信息，详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/集群级别配置项/enable_rich_error_msg。

  ```
  alter system set enable_rich_error_msg=true;
  ```

  2. 登录用户租户，执行报错 SQL 语句，会直接返回执行节点 IP 和 trace_id 信息。输出如下：

  ```
  ERROR 1146 (42S02): Table 'test.t2' doesn't exist
  [10.10.10.1:2882] [2024-04-13 20:10:20.292087] [YB420BA1CC68-000615A0A8EA
  5E38-0-0]
  ```

  3. 访问输出节点（本示例中为 10.10.10.1）过滤日志。

  ```
  [root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA5E38-0-0"  rootservice.l
  og*
  [root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA5E38-0-0"  observer.log*
  ```

  4. 获取日志信息后，关闭 `enable_rich_error_msg` 参数。

  ```
  alter system set enable_rich_error_msg=false;
  ```

- 登录系统租户，打开 `enable_rich_error_msg` 参数。

  enable_rich_error_msg 参数用于设置是否在客户端消息中添加服务器地址、时间、追踪 ID 等调试信息，详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/集群级别配置项/enable_rich_error_msg。

```
alter system set enable_rich_error_msg=true;
```

- 登录用户租户，执行报错 SQL 语句，会直接返回执行节点 IP 和 trace_id 信息。

```
obclient [test]> select count(*) from t2;
```

  输出如下：

```
ERROR 1146 (42S02): Table 'test.t2' doesn't exist
[10.10.10.1:2882] [2024-04-13 20:10:20.292087] [YB420BA1CC68-000615A0A8EA5E38-0-0]
```

- 访问输出节点（本示例中为 10.10.10.1）过滤日志。

```
[root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA5E38-0-0"  rootservice.log*
[root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA5E38-0-0"  observer.log*
```

- 获取日志信息后，关闭 `enable_rich_error_msg` 参数。

```
alter system set enable_rich_error_msg=false;
```

- 如果执行的 SQL 语句返回成功，但 SQL 任务一直没完成时，推荐使用用户租户获取 trace_id。例如：执行开启归档时，归档状态未 `doing` 场景，就需要使用下面这种方式。

  1. 登录用户租户，设置 `ob_enable_show_trace` 变量。ob_enable_show_trace 变量用于设置是否使用 show trace 日志，具体介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Session 系统变量/ob_enable_show_trace。

     ```
     SET ob_enable_show_trace='ON';
     ```

  2. 执行 SQL 语句，获取 trace_id。此种方式只能返回 trace_id，需要再通过 GV$OB_SQL_AUDIT 视图查到需要去哪个节点（`SVR_IP`）过滤日志。执行 SQL 语句，此处以 `select count(*) from test2;` 为例，输出如下：查看 SQL 语句执行的 trace_id：

输出如下：根据输出的 trace_id 在 GV$OB_SQL_AUDIT 视图中查看日志所在节点：

```
obclient [test]> select * from oceanbase.gv$ob_sql_audit where  trace_id=
'YB420BA1CC68-000615A0A8EA6511-0-0';
```

3. 按上述获取的 `SVR_IP` 和 trace_id 过滤日志。

```
[root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA6511-0-0" rootservice.lo
g*
[root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA6511-0-0" observer.log*
```

若根据输出的日志信息无法分析具体报错原因，可根据日志中的错误码在官方 知识库文档 中搜索相关信息。若仍旧无法解决，您也可以到官网 问答区 发帖，会有专业人员协助解决。

- 登录用户租户，设置 `ob_enable_show_trace` 变量。

  ob_enable_show_trace 变量用于设置是否使用 show trace 日志，具体介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Session 系统变量/ob_enable_show_trace。

```
SET ob_enable_show_trace='ON';
```

- 执行 SQL 语句，获取 trace_id。

  此种方式只能返回 trace_id，需要再通过 GV$OB_SQL_AUDIT 视图查到需要去哪个节点（`SVR_IP`）过滤日志。

说明

下述命令均需在同一个会话中执行。

执行 SQL 语句，此处以 `select count(*) from test2;` 为例，输出如下：

```
+----------+
| count(*) |
+----------+
|        0 |
+----------+
```

查看 SQL 语句执行的 trace_id：

```
obclient [test]> select last_trace_id();
```

输出如下：

```
+-----------------------------------+
| last_trace_id()                   |
+-----------------------------------+
| YB420BA1CC68-000615A0A8EA6511-0-0 |
+-----------------------------------+
```

根据输出的 trace_id 在 GV$OB_SQL_AUDIT 视图中查看日志所在节点：

```
obclient [test]> select * from oceanbase.gv$ob_sql_audit where  trace_id='YB420BA1
CC68-000615A0A8EA6511-0-0';
```

- 按上述获取的 `SVR_IP` 和 trace_id 过滤日志。

```
[root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA6511-0-0" rootservice.log*
[root@test001 ~]$ grep "YB420BA1CC68-000615A0A8EA6511-0-0" observer.log*
```

若根据输出的日志信息无法分析具体报错原因，可根据日志中的错误码在官方 知识库文档 中搜索相关信息。若仍旧无法解决，您也可以到官网 问答区 发帖，会有专业人员协助解决。

如何收集 SQL 性能慢的信息

数据库使用过程中经常会遇到 SQL 执行性能慢的问题，不管是硬件瓶颈还是 SQL 优化，都需要先了解当前 SQL 慢在什么地方，当然具体调优方式本节不进行介绍，第七章节会详细介绍性能调优。本节我们着重介绍如何收集性能慢的相关信息，用于自查或者社区问答区求助时提供信息。

SQL 性能慢有不同的场景，您需根据如下不同场景选择合适的解决方法。

执行 SQL 超时或者耗时长

首先推荐使用敏捷诊断工具，通过一键诊断分析对当前集群系统日志进行分析，快速发现可能得异常问题点，协助进行问题排查。具体操作可参见官网《OceanBase 敏捷诊断工具

（obdiag）》文档 一键诊断分析。

全链路诊断中的 Show Trace 功能也能轻松发现性能瓶颈并进行进一步的分析和调优。功能的详细介绍可参见官网《OceanBase 数据库》文档 管理数据库/日常巡检/全链路追踪 章节。

若以上两种方式无法解决问题，可通过如下三种方式尝试自查分析 SQL 超时或耗时长的原因。

- 通过查看 GV$OB_SQL_AUDIT 视图确认影响执行耗时的等待事件。

  ```
  select * from oceanbase.gv$ob_sql_audit where query_sql like '%<sql关键字>%'
  ```

  您需根据实际情况将 `<sql关键字>` 替换为超时或耗时长的 SQL 关键字。GV$OB_SQL_AUDIT 视图的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 优化/SQL 执行性能监控/GV$OB_SQL_AUDIT。

- 通过获取 SQL 执行计划 `EXPLAIN EXTENDED`，查看执行计划并分析，具体步骤可参见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 优化/SQL 执行性能监控/SQL 性能分析示例/查看执行计划形状并做分析。

- 通过获取 SQL Plan Monitor 信息的方法自查，具体操作如下。

  1. 登录 sys 租户，查看 enable_sql_audit 参数

     ```
     show parameters like 'enable_sql_audit';
     ```

     enable_sql_audit 参数的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/集群级别配置项/enable_sql_audit。

  2. （可选）设置 enable_sql_audit 参数若 enable_sql_audit 参数的值为 False，需执行如下命令修改其值为 true。

     ```
     alter system enable_sql_audit = true;
     ```

  3. 登录用户租户，获取 SQL 的执行计划

     ```
     EXPLAIN EXTENDED <sql语句>;
     ```

  4. 设置临时 trace 获取

     ```
     SET ob_enable_show_trace='ON';
     ```

```

```

5. 再次执行超时或耗时长的 SQL 语句

6. 执行如下命令获取上一步 SQL 语句的 trace_id 信息

```
select last_trace_id();
```

7. 在 sys 租户下执行如下命令临时关闭 Plan Monitor 数据，防止信息被覆盖

```
alter system enable_sql_audit = false;
```

8. 获取 Plan Monitor 的 SQL需将如下命令中的 `<trace_id>` 替换为第 6 步返回的 trace_id，以获取每个算子的输出信息。

```
select plan_line_id, plan_operation, sum(output_rows), sum(STARTS) rescan
, min(first_refresh_time) open_time, max(last_refresh_time) close_time, m
in(first_change_time) first_row_time, max(last_change_time) last_row_eof_
time, count(1) from oceanbase.gv$sql_plan_monitor where trace_id = '<trac
e_id>' group by plan_line_id, plan_operation order by plan_line_id;
```

9. 在 sys 租户下执行如下命令恢复 enable_sql_audit 参数

```
alter system enable_sql_audit = true;
```

- 登录 sys 租户，查看 enable_sql_audit 参数

```
show parameters like 'enable_sql_audit';
```

enable_sql_audit 参数的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/集群级别配置项/enable_sql_audit。

- （可选）设置 enable_sql_audit 参数

若 enable_sql_audit 参数的值为 False，需执行如下命令修改其值为 true。

```
alter system enable_sql_audit = true;
```

- 登录用户租户，获取 SQL 的执行计划

```
EXPLAIN EXTENDED <sql语句>;
```

- 设置临时 trace 获取

```
SET ob_enable_show_trace='ON';
```

- 再次执行超时或耗时长的 SQL 语句
- 执行如下命令获取上一步 SQL 语句的 trace_id 信息

```
select last_trace_id();
```

- 在 sys 租户下执行如下命令临时关闭 Plan Monitor 数据，防止信息被覆盖

```
alter system enable_sql_audit = false;
```

- 获取 Plan Monitor 的 SQL

  需将如下命令中的 `<trace_id>` 替换为第 6 步返回的 trace_id，以获取每个算子的输出信息。

```
select plan_line_id, plan_operation, sum(output_rows), sum(STARTS) rescan, min(first_refresh_time) open_time, max(last_refresh_time) close_time, min(first_change_time) first_row_time, max(last_change_time) last_row_eof_time, count(1) from oceanbase.gv$sql_plan_monitor where trace_id = '<trace_id>' group by plan_line_id, plan_operation order by plan_line_id;
```

- 在 sys 租户下执行如下命令恢复 enable_sql_audit 参数

```
alter system enable_sql_audit = true;
```

获取以上信息后，若无法解决 SQL 超时或耗时长的问题，可提供上述信息和对应 trace_id 的 observer.log 日志，到官网 问答区 发帖，会有专业人员协助解决。

以上的方法适应用于 SELECT 语句的排查思路，DDL 语句需要按以下方法排查。DDL 卡住场景比较复杂，不适用于用户自查，可以通过提供排查结果，在官网 问答区 发帖，会有专业人员协助解决。

DDL 语句执行卡住

1. 执行如下命令查到当前正在执行的 DDL 信息。

```
select tenant_id, gmt_create, ddl_type, status, task_id, parent_task_id, object_id
, target_object_id, execution_id, trace_id, unhex(ddl_stmt_str) from __all_virtual
_ddl_task_status;
```

根据输出结果的不同有如下两种处理方法：

- 可以查询到当前正在执行的 DDL 信息，您可直接进行第 2 步操作。
- 无法获取当前正在执行的 DDL 信息，您可直接进行第 3 步操作。

2. 可以查询到当前正在执行的 DDL 信息，您可直接进行第 2 步操作。
3. 无法获取当前正在执行的 DDL 信息，您可直接进行第 3 步操作。
4. 根据输出的 `tenant_id` 和 `task_id` 等信息，查询 DDL 任务的状态切换历史。

```
# <tenant_id> 和 <task_id> 需对应替换为第 1 步命令中输出的 `tenant_id` 和 `task_id`
set @ddl_tenant_id = <tenant_id>, @ddl_task_id = <task_id>;

select * from __all_rootservice_event_history where module = 'ddl_scheduler' and e
vent = 'switch_state' and value1 = @ddl_tenant_id and value2 = @ddl_task_id;
```

如果不知道 task_id，对于建索引任务，可以用 OBJECT_ID（主表 ID）和 TARGET_OBJECT_ID（索引表 ID）来查：

```
select * from __all_rootservice_event_history where module = 'ddl_scheduler' and e
vent = 'switch_state' and value1 = @ddl_tenant_id and value3 = '$OBJECT_ID' and va
lue4 = '$TARGET_OBJECT_ID';
```

如果输出中 `new_state` 的值是 `2`（`WAIT_TRANS_END`），大概率是在等活跃事务结束。DDL 执行阶段与输出值的对应关系如下：

```
PREPARE = 0,
LOCK_TABLE = 1,
WAIT_TRANS_END = 2,
REDEFINITION = 3,
VALIDATE_CHECKSUM = 4,
COPY_TABLE_DEPENDENT_OBJECTS = 5,
TAKE_EFFECT = 6,
WRITE_BARRIER_LOG = 7,
CHECK_CONSTRAINT_VALID = 8,
SET_CONSTRAINT_VALIDATE = 9,
MODIFY_AUTOINC = 10,
```

```
SET_WRITE_ONLY = 11,
WAIT_TRANS_END_FOR_WRITE_ONLY = 12,
SET_UNUSABLE = 13,
WAIT_TRANS_END_FOR_UNUSABLE = 14,
DROP_SCHEMA = 15,
FAIL = 99,
SUCCESS = 100
```

5. 如果通过 `__all_virtual_ddl_task_status` 表无法获取到 DDL 信息，可能是 Root Service 调度 DDL 线程卡住。

   执行如下命令获取 trace_id。

   ```
   select * from gv$ob_sql_audit where query_sql like '%DDL 语句关键字%';
   ```

   根据 trace_id 过滤 Root Service 主节点日志信息。

   ```
   grep $trace_id rootservice.log | grep "DDLQueueTh" | grep "\[DDL]"
   ```

   如果日志返回信息是表锁相关的 `6005` 或者 `4023` 错误码，可以确认下是否由 `drop/truncate` 操作锁冲突导致，或者到官网 问答区 发帖，会有专业人员协助解决。

   如果日志返回信息是 RPC 相关的 `4012` 或者 `4121` 错误码，按以下方式查看。

   ```
   # 耗时 100ms 以上的 Root Service DDL 请求
   grep "DDLQueueTh" rootservice.log | grep "\[DDL]" | grep "cost=[0-9]\{6\}"

   # 耗时 1s 以上的 Root Service DDL 请求
   grep "DDLQueueTh" rootservice.log | grep "\[DDL]" | grep "cost=[0-9]\{7\}"

   # 耗时 10s 以上的 Root Service DDL 请求
   grep "DDLQueueTh" rootservice.log | grep "\[DDL]" | grep "cost=[0-9]\{8\}"
   ```

   如果返回 DDL 耗时较多，且根据 trace_id 无法在 `__all_virtual_ddl_task_status` 查询到，大概率是业务压力太大导致，可以尝试降低压力恢复；如果返回 DDL 耗时较少且稳定，建议按当前排查过程收集的信息，到官网 问答区 发帖，会有专业人员协助解决。

如何排查数据库初始化失败问题

OceanBase 数据库初次部署启动时会有一个 Bootstrap 初始化数据库操作，此步骤只执行一

次。比如部署时对资源参数设置不合理或者服务器性能较差导致执行超时，会造成初始化失败，只能重新安装。如果对失败原因不明确，会导致重复安装试错，从而影响使用体验，因此这里将介绍数据库初始化失败的定位排查方式。

问题现象：Bootstrap 初始化阶段失败。

问题排查：

无论使用哪种工具部署 OceanBase 数据库，当出现 Bootstrap 初始化阶段失败的情况时，只需要查看创建日期最早的 observer.log 日志即可。初始化阶段 observer 进程可能不会退出的，会打印大量 observer 进程日志信息，导致日志被覆盖，干扰问题排查，因此我们需要了解数据库启动时的关键字信息进行定位。

1. 查看 `ERROR` 级别日志，通常初始化失败会打印 ERROR 日志，如果 ERROR 日志信息比较多，也可以搜索关键字 `Unexpected internal error happen`，但这个关键字并不能直观显示初始化失败原因，仍然需要往上查看 `WAIN`/`WDIAG` 级别的日志。示例如下：

   ```
   Unexpected internal error happen, please checkout the internal errcode
   ```

2. 如果系统日志初始化阶段的内容被刷掉，无法根据后续打印的日志分析出原因的，此时就需要重新安装数据库，再使用关键字 `begin server limit report` 定位到启动时的日志信息，往下查看 `WARN`/`WDIAG`/`ERROR` 级别日志内容。示例如下：

   ```
   [2024-04-11 21:32:57.748844] INFO  print_all_limits (main.cpp:368) [43378][observer][T0][Y0-0000000000000000-0-0] [lt=6] ============= *begin server limit report *=============
   ```

此处结合最简单的初始化失败的案例进行说明：

使用 obd 部署时，若设置资源参数 `memory_limit:30G，system_memory:30G`，一定会导致部署失败。obd 会进行参数预检查，`[WARN]` 提示出异常，之后 `[ERROR]` 报错启动失败。

```
obd cluster start lzq1
Get local repositories ok
Search plugins ok
Load cluster param plugin ok
Open ssh connection ok
Check before start observer ok
[WARN] obd-2010: (10.10.10.1): system_memory too large. system_memory should be less than 0.75 * memory_limit/memory_limit_percentage.
```

```
Start observer ok
observer program health check x
[WARN] obd-2002: Failed to start 10.10.10.1 observer
[ERROR] oceanbase-ce start failed
See https://www.oceanbase.com/product/ob-deployer/error-codes .
Trace ID: 12b798b0-fa48-11ee-8a58-00163e046d79
If you want to view detailed obd logs, please run: obd display-trace 12b798b0-fa48
-11ee-8a58-00163e046d79
```

此时 observer 进程是不在的，日志打印如下：

```
# observer.log 搜索首次出现 'ERROR' 日志，已经反馈出内存异常报错：update observer memory
config failed。
[2024-04-14 18:16:37.849976] ERROR issue_dba_error (ob_log.cpp:1868) [76016][obser
ver][T0][Y0-0000000000000000-0-0] [lt=18][errcode=-4388] Unexpected internal erro
r happen, please checkout the internal errcode(errcode=-4147, file="ob_server_conf
ig.cpp", line_no=365, info="update observer memory config failed")
[2024-04-14 18:16:37.849983] EDIAG [SHARE.CONFIG] reload_config (ob_server_config.
cpp:365) [76016][observer][T0][Y0-0000000000000000-0-0] [lt=7][errcode=-4147] upda
te observer memory config failed(memory_limit=32212254720, system_memory=322122547
20, hidden_sys_memory=12079595520, min_server_avail_memory=2147483648) BACKTRACE:0
x113b8115 0x6ad98b0 0x6ad9471 0x6ad90ec 0x6ad8ee5 0xf2e7697 0xf2e70c8 0x9cd0a93 0x
9cc6c55 0x6ab20e4 0x7ff08e478445 0x4e56560
[2024-04-14 18:16:37.850043] WDIAG [SHARE.CONFIG] reload_config (ob_server_config.
cpp:377) [76016][observer][T0][Y0-0000000000000000-0-0] [lt=57][errcode=-4147] th
e hold memory of tenant_500 is over the reserved memory(tenant_500_hold=6291456, t
enant_500_reserved=0)
[2024-04-14 18:16:37.850052] ERROR issue_dba_error (ob_log.cpp:1868) [76016][obser
ver][T0][Y0-0000000000000000-0-0] [lt=5][errcode=-4388] Unexpected internal error
happen, please checkout the internal errcode(errcode=-4147, file="ob_server.cpp",
line_no=1886, info="reload memory config failed")

# 通过提示内存申请信息不难看出，自适应创建的 sys 租户内存最大可申请 12G，最小 2G，而申请完 syste
m_memory 的 32G 内存后无法再支撑 2G 的内存申请导致的初始化失败。
memory_limit=32212254720, system_memory=32212254720, hidden_sys_memory=12079595520
, min_server_avail_memory=2147483648)
```

数据库升级失败如何排查日志

无论使用哪种工具升级 OceanBase 数据库，本质上均是使用 OceanBase 数据库升级脚本实现。通常情况下是通过 obd、OCP 工具来升级数据库，但是升级失败后看 obd、OCP 输出的日志很难确认失败的具体原因，需要对照脚本查看逻辑，其实升级脚本会有专属日志产生。

说明

可通过官网《OceanBase 数据库》文档 管理数据库/集群管理/集群常见操作/集群升级/OceanBase 企业版升级/升级 OceanBase 集群 中 **步骤一** 了解升级脚本。

问题现象：使用 obd 或者 OCP 升级 OceanBase 数据库失败。

问题排查：

- obd 升级 OceanBase 数据库失败有如下多个阶段，而日志生成在执行 obd 升级命令的当前目录下。
    - `Exec upgrade_checker.py x` 失败：基本可以通过执行对应的 `obd display-trace` 命令，根据输出信息确认失败原因。如果无法看出失败原因，可以查看脚本日志 `upgrade_checker.log`。

    - `Exec upgrade_pre.py x` 失败：常见失败阶段，一般执行对应 `obd display-trace` 命令后，输出信息反馈的失败原因不是很清晰，需结合脚本日志（`upgrade_pre.log`）和数据库实际情况进行分析确认。

    - `Exec upgrade_health_checker.py x` 失败：基本很少遇到此阶段的升级失败，可查看对应日志（`upgrade_cluster_health_checker.log`）分析失败原因。

    - `Exec upgrade_post.py x`：常见失败阶段，需结合脚本日志（`upgrade_post.log`）和数据库实际情况进行分析确认。

- `Exec upgrade_checker.py x` 失败：基本可以通过执行对应的 `obd display-trace` 命令，根据输出信息确认失败原因。如果无法看出失败原因，可以查看脚本日志 `upgrade_checker.log`。
- `Exec upgrade_pre.py x` 失败：常见失败阶段，一般执行对应 `obd display-trace` 命令后，输出信息反馈的失败原因不是很清晰，需结合脚本日志（`upgrade_pre.log`）和数据库实际情况进行分析确认。
- `Exec upgrade_health_checker.py x` 失败：基本很少遇到此阶段的升级失败，可查看对应日

志（`upgrade_cluster_health_checker.log`）分析失败原因。

- `Exec upgrade_post.py x`：常见失败阶段，需结合脚本日志（`upgrade_post.log`）和数据库实际情况进行分析确认。

> **说明**
>
> - obd 升级时，若遇到执行超时问题，可以尝试重新执行 obd 升级命令重试。目前 obd 已具备任意阶段升级失败可重新执行升级的能力。
> - 如果自查无法确认问题原因，建议收集以上相关日志信息，到官网 问答区 发帖，会有专业人员协助解决。

- OCP 升级 OceanBase 数据库和 obd 升级数据库逻辑类似，白屏上会实时输出升级过程的日志，比较直观，不过遇到数据库升级脚本失败时，也需要查看 OceanBase 数据库的升级日志，但是和 obd 升级时日志存放路径和名称不一样。
    - 存放路径：/tmp/{版本信息}/upgrade_*_{时间}.log，例如：/tmp/4.2.1.2-102000042023120514_upgrade_post_20240411101214.log。
    - 升级失败步骤禁止直接跳过或者通过设置为成功的方式绕过。部分升级步骤会有系统参数相关的修改操作，盲目跳过任务，可能导致集群不可用，建议收集以上日志相关信息，到官网 问答区 发帖，会有专业人员协助解决。

- 存放路径：/tmp/{版本信息}/upgrade_*_{时间}.log，例如：/tmp/4.2.1.2-102000042023120514_upgrade_post_20240411101214.log。
- 升级失败步骤禁止直接跳过或者通过设置为成功的方式绕过。部分升级步骤会有系统参数相关的修改操作，盲目跳过任务，可能导致集群不可用，建议收集以上日志相关信息，到官网 问答区 发帖，会有专业人员协助解决。

OBServer 节点 core 掉后如何收集堆栈

问题现象：正常运行的 OBServer 节点突然异常退出，本地产生 core 文件（或称 core dump 文件）。

问题排查：

1. 查看对应的 OBServer 节点日志里是否输出 `CRASH ERROR` 关键字，命令如下。

```
[root@test001 ~]$ grep "CRASH ERROR" observer.log
```

输出如下：

```
CRASH ERROR!!! sig=6, sig_code=-6, sig_addr=3eb00000dbb, timestamp=171282809927087
9, tid=4466, tname=ReplayEngine12, trace_id=0-0, extra_info=((null)),    lbt=0x9af
40d8 0x9ae4978 0x7f74fdbdd62f 0x7f74fd42a377 0x7f74fd42ba67 0x9a28638 0x873cffa 0x
8dd491f 0x8dd309b 0x8e6f22e 0x8b4e29c 0x8b4ac4b 0x8b48b04    0x9a69028 0x3426f0e 0
x2cc7671 0x985e884 0x985d271 0x9859d2e
```

2. 查看对应的 OBServer 机器 `/etc/sysctl.conf` 文件中指定的 `kernel.core_pattern` 位置是否生成对应的 core 文件，命令如下。

查看配置的 `kernel.core_pattern` 位置：

```
[root@test001 ~]$ grep "kernel.core_pattern" /etc/sysctl.conf
```

输出如下：

```
kernel.core_pattern = /obdata/data/core-%e-%p-%t
```

> **说明**
>
> 如果没有指定 `kernel.core_pattern`，默认会在 OceanBase 数据库的 home_path 路径下生成 `core.${ob_pid}` 的文件。

查看对应 core 文件：

```
[root@test001 ~]$ ls -l /obdata/data/core*
```

输出如下，若无对应 core 文件，可以通过 `ulimit -a` 或者 `ulimit -c` 查看当前资源的限制，如果设置为 `0` 或者很小，会发生节点 core 时无法生产 core 文件的情况。

```
-rw------- 1 admin admin 8723914752 4月   8 19:04 /obdata/data/core-observer-27670
-1712574296
```

3. 使用 addr2line 命令收集堆栈信息，命令如下。

```
addr2line -pCfe ./bin/observer 0x9af40d8 0x9ae4978 0x7f74fdbdd62f 0x7f74fd42a377 0x7f74fd42ba67 0x9a28638 0x873cffa 0x8dd491f 0x8dd309b 0x8e6f22e 0x8b4e29c 0x8b4ac4b 0x8b48b04 0x9a69028 0x3426f0e 0x2cc7671 0x985e884 0x985d271
```

4. 提供 core 文件、堆栈信息、故障节点事发前后的 observer.log 日志发送到官网 问答区 发帖，会有专业人员协助解决。

您也可参照如下方法进行应急处理：

数据库 core 文件通常是在数据库系统遇到异常情况而崩溃时生成，在收集或备份完所需排查分析信息后，可以尝试手动重新拉起 observer 服务进程，观察是否恢复正常。如果出现频繁 core 无法通过重启恢复的情况，在集群可用情况下可以考虑通过替换节点，或者增大永久下线时间来防止节点被踢出集群；如果集群不可用，可以考虑通过数据备份文件进行数据恢复或者切换物理备库恢复业务。

替换节点的具体操作可参见官网《OceanBase 云平台》文档 集群管理/管理 OBServer/替换 OBServer 节点。

增大永久下线时间的具体操作可参见官网《OceanBase 数据库》文档 管理数据库/集群管理/集群常见故障处理/少数派节点故障。

第六章 使用 OceanBase 数据库进行业务开发

本章介绍如何连接和访问 OceanBase 数据库以及 OceanBase 数据库的一些常用功能，指导用户如何进行常见的数据库开发。

本章目录

6.1 使用 MySQL 租户做常见数据库开发 360

6.2 通过 ODC 图形化开发工具进行 SQL 开发 418

6.3 使用 OceanBase 数据库分区表进行水平拆分 440

6.4 OceanBase 数据库在 MySQL 模式租户下的扩展功能 454

6.1 使用 MySQL 租户做常见数据库开发

本章主要介绍 MySQL 模式下连接和访问 OceanBase 数据库的方法，指导用户如何进行常见的数据库开发，如数据读取、写入、事务处理等。目前，OceanBase 支持多种连接方式，包括使用命令行客户端、图形界面工具、以及各种编程语言的数据库驱动。OceanBase 数据库 MySQL 模式兼容了 MySQL 5.7/8.0 的绝大部分功能和语法，同时存在一些 OceanBase 特有的扩展功能，后续的小节也会针对这些功能做详细介绍，如分区表、全局索引、回收站、表组、序列、闪回、复制表等等。

连接方式介绍

OceanBase 当前主要支持通过客户端、驱动或 ORM 框架连接到 OceanBase 数据库。

客户端

在连接 OceanBase 数据库的 MySQL 租户时，支持的客户端如下:

MySQL 客户端（mysql）

mysql 是 MySQL 数据库的命令行客户端，需要单独安装。OceanBase 数据库社区版仅支持 MySQL 兼容租户，当访问 MySQL 兼容租户时，可以通过 mysql 连接。此处将详细介绍 MySQL 客户端的连接方法，其余客户端、图形界面工具基本的连接配置均可参考该示例。

前提条件

通过 MySQL 客户端连接数据库前，需要确认以下信息:

- 确保本地已正确安装 MySQL 客户端。OceanBase 数据库当前版本支持的 MySQL 客户端版本包括 V5.5、V5.6 和 V5.7。
- 确保环境变量 `PATH` 中包含了 MySQL 客户端命令所在目录。
- 连接租户前，请确认当前客户端在租户白名单中，租户白名单的查询及设置具体操作请参见官网《OceanBase 数据库》文档 管理数据库/租户管理/租户常见操作/查看和设置租户白名

单。

连接操作

1. 打开一个命令行终端。
2. 使用 MySQL 命令连接 MySQL 租户。
   - 通过 ODP 连接的方式或者参数说明：示例：或者

     ```
     $mysql -h10.10.10.1 -uobdemo:obmysql:****** -P2883 -p****** -c -A -Docea
     nbase
     ```

   - 通过直连 OBServer 节点方式参数说明：
     - `-h`：提供 OceanBase 数据库连接 IP，通常是一个 OBServer 节点的 IP 地址。
     - `-u`：提供租户的连接账户，格式为：`用户名@租户名`。使用 MySQL 客户端仅支持连接 MySQL 租户，MySQL 租户的管理员用户名默认是 `root`。
     - `-P`：提供 OceanBase 数据库连接端口，默认是 `2881`，在部署 OceanBase 数据库时可自定义。
     - `-p`：提供账户密码，为了安全可以不提供，改为在后面提示符下输入，密码文本不可见。
     - `-c`：表示在 MySQL 运行环境中不要忽略注释。Hint 是特殊的注释，不受 `-c` 影响。
     - `-A`：表示在 MySQL 连接数据库时不自动获取统计信息。
     - `oceanbase`：访问的数据库的名称，可以更改为业务数据库。

说明

> 普通租户通过直连方式连接时，需要确保该租户的资源分布在该 OBServer 节点上，如果该租户的资源未分布在该 OBServer 节点上，则无法通过直连该 OBServer 节点连接到该租户。

3. 通过 ODP 连接的方式

```
$mysql -h10.10.10.1 -uusername@obmysql#obdemo -P2883 -p****** -c -A -Doceanbase
```

或者

```
$mysql -h10.10.10.1 -uobdemo:obmysql:username -P2883 -p****** -c -A -Doceanbase
```

参数说明：

- `-h`：提供 OceanBase 数据库连接 IP，通常是一个 ODP 地址。
- `-u`：提供租户的连接账户，格式有：`用户名@租户名#集群名`、`集群名:租户名:用户名`、`集群名-租户名-用户名` 或者 `集群名.租户名.用户名`。使用 MySQL 客户端仅支持连接 MySQL 租户，MySQL 租户的管理员用户名默认是 `root`。
- `-P`：提供 OceanBase 数据库连接端口，也是 ODP 的监听端口，默认为 `2883`，在部署 ODP 时可自定义。
- `-c`：表示在 MySQL 运行环境中不要忽略注释。Hint 是特殊的注释，不受 `-c` 影响。
- `-A`：表示在 MySQL 连接数据库时不自动获取统计信息。
- `oceanbase`：访问的数据库的名称，可以更改为业务数据库。

示例：

```
$mysql -h10.10.10.1 -u******@obmysql#obdemo -P2883 -p****** -c -A -Doceanbase
```

或者

```
$mysql -h10.10.10.1 -uobdemo:obmysql:****** -P2883 -p****** -c -A -Doceanbase
```

```
```

4. `-h`：提供 OceanBase 数据库连接 IP，通常是一个 ODP 地址。
5. `-u`：提供租户的连接账户，格式有：`用户名@租户名#集群名`、`集群名:租户名:用户名`、`集群名-租户名-用户名` 或者 `集群名.租户名.用户名`。使用 MySQL 客户端仅支持连接 MySQL 租户，MySQL 租户的管理员用户名默认是 `root`。
6. `-P`：提供 OceanBase 数据库连接端口，也是 ODP 的监听端口，默认为 `2883`，在部署 ODP 时可自定义。
7. `-c`：表示在 MySQL 运行环境中不要忽略注释。Hint 是特殊的注释，不受 `-c` 影响。
8. `-A`：表示在 MySQL 连接数据库时不自动获取统计信息。
9. `oceanbase`：访问的数据库的名称，可以更改为业务数据库。
10. 通过直连 OBServer 节点方式

```
$mysql -h10.10.10.1 -uusername@obmysql -P2881 -p****** -c -A -Doceanbase
```

参数说明：

- `-h`：提供 OceanBase 数据库连接 IP，通常是一个 OBServer 节点的 IP 地址。
- `-u`：提供租户的连接账户，格式为：`用户名@租户名`。使用 MySQL 客户端仅支持连接 MySQL 租户，MySQL 租户的管理员用户名默认是 `root`。
- `-P`：提供 OceanBase 数据库连接端口，默认是 `2881`，在部署 OceanBase 数据库时可自定义。
- `-p`：提供账户密码，为了安全可以不提供，改为在后面提示符下输入，密码文本不可见。
- `-c`：表示在 MySQL 运行环境中不要忽略注释。Hint 是特殊的注释，不受 `-c` 影响。
- `-A`：表示在 MySQL 连接数据库时不自动获取统计信息。

- `oceanbase`：访问的数据库的名称，可以更改为业务数据库。

11. `-h`：提供 OceanBase 数据库连接 IP，通常是一个 OBServer 节点的 IP 地址。
12. `-u`：提供租户的连接账户，格式为：`用户名@租户名`。使用 MySQL 客户端仅支持连接 MySQL 租户，MySQL 租户的管理员用户名默认是 `root`。
13. `-P`：提供 OceanBase 数据库连接端口，默认是 `2881`，在部署 OceanBase 数据库时可自定义。
14. `-p`：提供账户密码，为了安全可以不提供，改为在后面提示符下输入，密码文本不可见。
15. `-c`：表示在 MySQL 运行环境中不要忽略注释。Hint 是特殊的注释，不受 `-c` 影响。
16. `-A`：表示在 MySQL 连接数据库时不自动获取统计信息。
17. `oceanbase`：访问的数据库的名称，可以更改为业务数据库。

常见错误

- 业务数据库连接错误

```
$mysql -h10.10.10.1 -u******@obmysql#obdemo -P2883 -p****** -c -A -Doceanbaseerror
```

  输出如下：

```
mysql: [Warning] Using a password on the command line interface can be insecure.
ERROR 1049 (42000): Unknown database 'oceanbaseerror'
```

- 集群名称不对

```
$mysql -h10.10.10.1 -u******@obmysql#obdemoerror -P2883 -p****** -c -A -Doceanbase
```

  输出如下：

```
mysql: [Warning] Using a password on the command line interface can be insecure.
ERROR 4669 (HY000): cluster not exist
```

- 租户不对

```
$mysql -h10.10.10.1 -u******@obmysqlerror#obdemo -P2883 -p****** -c -A -Doceanbase
```

输出如下:

```
mysql: [Warning] Using a password on the command line interface can be insecure.
ERROR 4012 (HY000): Get Location Cache Fail
```

- 密码错误

```
$mysql -h10.10.10.1 -u******@obmysql#obdemo -P2883 -p******error -c -A -Doceanbase
```

输出如下:

```
ERROR 1045 (42000): Access denied for user 'root'@'xxx.xxx.xxx.xxx' (using passwor
d: YES)
```

- 直连 OBServer 节点但是填写了集群名称

```
$mysql -h10.10.10.1 -u******@obmysql#obdemo -P2881 -p******error -c -A -Doceanbase
```

输出如下:

```
ERROR 1045 (42000): Access denied for user 'root'@'xxx.xxx.xxx.xxx' (using passwor
d: YES)
```

- 端口不对

```
$mysql -h10.10.10.1 -u******@obmysql#obdemo -P2889 -p****** -c -A -Doceanbase
```

输出如下:

```
mysql: [Warning] Using a password on the command line interface can be insecure.
ERROR 2003 (HY000): Can't connect to MySQL server on '10.10.10.1' (111)
```

OceanBase 客户端（OBClient）

OBClient 是一个交互式和批处理查询工具，需要单独安装。它是一个命令行用户界面，在连接到数据库时充当客户端，支持 OceanBase 数据库的 MySQL 租户 和 Oracle 租户。连接上

OceanBase 数据库后，通过 OBClient 可以运行一些数据库命令（包含常用的 MySQL 命令）、SQL 语句和 PL 语句。

前提条件

- 请确认已下载并安装了 OBClient 应用。如果未下载 OBClient 应用，可以访问官网 OceanBase 软件下载中心 下载对应版本的 OBClient。
- 连接租户前，请确认当前客户端在租户白名单中，租户白名单的查询及设置具体操作请参见官网《OceanBase 数据库》文档 管理数据库/租户管理/租户常见操作/查看和设置租户白名单。

连接操作

- 通过 ODP 连接的方式

  ```
  obclient -h10.10.10.1 -uusername@obtenant#obdemo -P2883 -p****** -c -A -Doceanbase
  ```

  或者

  ```
  obclient -h10.10.10.1 -uobdemo:obtenant:username -P2883 -p****** -c -A -Doceanbase
  ```

- 通过直连 OBServer 方式

  ```
  obclient -h10.10.10.1 -uusername@obtenant -P2881 -p****** -c -A -Doceanbase
  ```

具体参数说明和前面 MySQL 客户端一致。

OceanBase 开发者中心（ODC）

OceanBase 开发者中心（OceanBase Developer Center，ODC）是为 OceanBase 数据库量身打造的企业级数据库协同开发工具。

ODC 支持连接 OceanBase 数据库的 MySQL 租户和 MySQL 数据库，ODC 有桌面版、Web 版两种产品形态。桌面版侧重数据库开发工具能力，支持 Windows、Mac、Linux 操作系统，具有轻量化和易部署的特性。Web 版在提供工具能力的同时还提供了管控、协同能力，侧重数据

库变更的安全、合规和效率。

前提条件

确保已部署 OceanBase 开发者中心（ODC），部署详细操作请参考官网《OceanBase 开发者中心》文档 部署指南 章节。

连接示例

使用 OceanBase 开发者中心连接 OceanBase 数据库租户的具体操作示例请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 ODC 连接 OceanBase 数据库。

注意事项

ODC 连接数据源页面中集群名称为可选项，如果是直连 OBServer 节点的话，不能填写集群名称，否则会连接不上。

DBeaver

DBeaver 是一款通用的数据库客户端工具，其原理是使用各个数据库提供的 JDBC 驱动连接数据库，支持常见的关系型数据库、非关系型数据库、分布式数据库等。

DBeaver 可以用于 OceanBase 数据库开发和调试、数据库管理和维护、数据分析和可视化、数据库迁移和同步以及学习和培训等多种场景。

前提条件

- 请确保已获取并安装 DBeaver，可访问 DBeaver 官方下载地址 下载系统对应的 DBeaver。
- 请确保已获取 DBeaver 连接 OceanBase 数据库的 JDBC 驱动文件，默认驱动为 `mysql-connector-java-5.1.44`，5.x 版本 MySQL 驱动文件均支持连接 OceanBase 数据库 MySQL 模式。也可访问 MySQL 官方 JDBC 驱动下载 页面下载其他 5.x 版本驱动文件。
- 请确保待连接的 OBServer 节点 IP 与 DBeaver 所在机器保持网络连通。
- 已部署 OceanBase 数据库并且创建了 MySQL 模式租户。

连接操作

使用 DBeaver 连接 OceanBase 租户的具体操作示例请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 DBeaver 连接数据库。

注意事项

当前 OceanBase 数据库 MySQL 模式不支持 event 事件，部分版本连接数据库时可能会提示 event 相关报错，可直接忽略。

Navicat

Navicat 是一款通用的数据库客户端工具，其原理是使用各个数据库提供的 JDBC 驱动连接数据库，支持常见的关系型数据库、非关系型数据库、分布式数据库等。使用 Navicat 自带的 OceanBase 驱动或 MySQL 驱动均可连接 OceanBase 数据库的 MySQL 租户。

前提条件

- 请确保已获取并安装 Navicat，可访问 Navicat Premium 官方下载地址 下载系统对应的 Navicat。
- 请确保待连接的 OBServer 节点 IP 与 Navicat 所在机器保持网络连通。
- 已部署 OceanBase 数据库并且创建了 MySQL 模式租户。

连接操作

使用 Navicat 连接 OceanBase 租户的具体操作示例请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/通过 Navicat 连接数据库。

注意事项

当前 OceanBase 数据库 MySQL 模式不支持 event 事件，部分版本连接数据库时可能会提示 event 相关报错，可直接忽略。

连接池配置

数据库连接池是一种在内存中创建和维护一组数据库连接资源的软件设计模式，目的是提高应用程序访问数据库的效率。在实际应用中，连接池的使用可以显著提升数据库操作的性能，尤其是在需要频繁访问数据库的应用程序中。

常见的数据库连接池技术包括 DBCP（数据库连接池）、Tomcat JDBC 连接池、HikariCP、Druid 等。每种技术都有其特点，开发者可以根据具体需求选择合适的连接池技术。

数据源配置建议

参数含义	说明	ZDAL 参数	Druid 参数	DBCP 参数	C3P0 参数
初始化连接数	连接池初始化时建立连接数。	prefill=true 初始化到 minConn	initialSize(0)	initialSize(0)	initialPoolSize(3)
最小连接数	最小可使用连接数，这个会让连接池日常一直保留着这么多个链接。	minConn(0)	minIdle(0)	minIdle(0)	minPoolSize(3)
最大连接数	最大可使用连接数，超过这个连接池会抛连接池已满异常。	maxConn(10)	maxActive(8)	maxActive(8)	maxActive(8)
链接空闲超时时间	设置链接空闲时间一段时间不用时，连接池主动断开链接。MySQL 默认 8 小时断开链接，主备切换时链接会变成脏连接。如果没有这个机制，可能会导致请求失败。云上 OceanBase SLB 超时时间 15 分钟，可以将超时时间设置为 12 分钟。	idleTimeoutMinutes(30min)	minEvictableIdleTimeMillis(30min)	minEvictableIdleTimeMillis(30min) 需要设置 timeBetweenEvictionRunsMillis(-1) > 0 才会生效，该参数控制异步检查周期	maxIdleTime (0 表示不超时)
连接池获取链接的	如果设置太大的话，链接池满了会导致应用响应太慢。	blockingTimeoutMillis(500ms)	maxWait (-1 表示不超时)	maxWaitMillis (-1 表示不超时)	checkoutTimeout (0 表示不超

超时时间					时）
连接获取不释放超时时间	超时这个时间的话，连接一直没有放回连接池会被直接销毁掉。可以防止连接泄露，但是会影响事务使用时间限制。	无	removeAbandonedTimeoutMillis(300s)	removeAbandonedTimeout(300s)	无

JDBC 连接池配置建议

JDBC 重要的几个参数，一定需要设置。均可以设置到连接池的 ConnectionProperties 中，或者 JdbcUrl 上。具体 JDBC 连接示例及参数说明如下所示。

```
conn=jdbc:oceanbase://xxx.xxx.xxx.xxx:3306/test?rewriteBatchedStatements=TRUE&allowMultiQueries=TRUE&useLocalSessionState=TRUE&useUnicode=TRUE&characterEncoding=utf-8&socketTimeout=3000000&connectTimeout=60000
```

- `rewriteBatchedStatements`：建议设置为 `TRUE`。
    - OceanBase 的 JDBC 驱动在默认情况下会无视 `executeBatch()` 语句，把批量执行的一组 SQL 语句拆散，一条一条地发给数据库，此时批量插入实际上是单条插入，直接造成较低的性能。要想实际执行批量插入，需要将该参数置为 `TRUE`，驱动才会批量执行 SQL。即使用 addBatch 方法把同一张表上的多条 insert 语句合在一起，做成一条 insert 语句里的多个 values 值的形式，提高 batch insert 的性能。

    - 必须使用 prepareStatement 方式来把每条 insert 做 prepare，然后再 addBatch，否则不能合并执行。

- OceanBase 的 JDBC 驱动在默认情况下会无视 `executeBatch()` 语句，把批量执行的一组 SQL 语句拆散，一条一条地发给数据库，此时批量插入实际上是单条插入，直接造成较低的性能。要想实际执行批量插入，需要将该参数置为 `TRUE`，驱动才会批量执行 SQL。即使用 addBatch 方法把同一张表上的多条 insert 语句合在一起，做成一条 insert 语句里的多个 values 值的形式，提高 batch insert 的性能。

- 必须使用 prepareStatement 方式来把每条 insert 做 prepare，然后再 addBatch，否则不能合并执行。
- `allowMultiQueries`：建议设置为 `TRUE`。

  JDBC 驱动允许应用代码把多个 SQL 用分号（`;`）拼接在一起，作为一个 SQL 发给 server 端。
- `useLocalSessionState`：建议设置为 `TRUE`，避免交易频繁向 OceanBase 数据库发送 session 变量查询 SQL。

  session 变量主要为：`autocommit`、`read_only` 和 `transaction isolation`。
- `socketTimeout`：执行 SQL 时，socket 等待 SQL 返回的时间，以毫秒为单位。值为 `0` 时，表示没有超时限制。也可以通过设置系统变量 `max_statement_time` 来限制查询时间。 默认值：`0`（标准配置）或 `10000 ms`。
- `connectTimeout`：建立连接时，等待连接的时间，以毫秒为单位。值为 `0` 时，表示没有超时限制。默认值：`30000`。
- `useCursorFetch`：建议设置为 `TRUE`。

  对于大数据量的查询语句，数据库 Server 会建立 Cursor 并根据 FetchSize 的大小向 Client 分发数据。这个属性设为 `TRUE`，会自动连带设置 `useServerPrepStms=TRUE`。
- `useServerPrepStms`：控制是否使用 PS 协议来把 SQL 发送给数据库 server。

  设置为 `TRUE` 时，SQL 在数据库内会分为两步执行：

  - 把包含 `?` 的 SQL 文本发送到数据库 Server 做 Prepare（`SQL_audit: request_type=5`）。

  - 用真实 Value 在数据库内做 Execute（`SQL_audit: request_type=6`）。

- 把包含 `?` 的 SQL 文本发送到数据库 Server 做 Prepare（`SQL_audit: request_type=5`）。
- 用真实 Value 在数据库内做 Execute（`SQL_audit: request_type=6`）。
- `cachePrepStmts`：控制 JDBC driver 是否开启 PS cache 来缓存 PreparedStatment，避免重复执行 prepare（client 端和 server 端）。`cachePrepStmts=TRUE` 对使用 `useServerPrepStms=TRUE` 并重复对同一条 SQL 做 batch execute 的场景有帮助。每一次

batch excecute 都会包含 preare 和 `executecachePrepStmts=TRUE` 可以避免重复的 prepare 操作。

- `prepStmtCacheSQLLimit`：可以放入 PS cache 的 SQL 的长度限制，超长 SQL 不可放入缓存。
- `prepStmtCacheSize`：PS cache 可以保存的 SQL 数量。
- `maxBatchTotalParamsNum`：针对 batch 操作，一条 SQL 最多能支持多少个参数（即 batch 中 ? 的个数）。如果参数个数大于限制，batch SQL 将会被拆分。

其他连接池配置示例请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/连接 OceanBase 数据库/使用数据库连接池/数据库连接池配置。

连接池设置建议

- 管控台日常 min 保持两个连接即可，具体根据业务并发及事务时间来调整。
- 设置连接空闲超时时间，推荐 30 分钟。

  MySQL 默认 8 小时连接主动断开，客户端无法感知，导致存在脏连接。连接池可以通过心跳、testOnBorrow 等机制来校验连接是否存活，当超过这个时间连接没有使用直接断开。

驱动

在连接 OceanBase 数据库的 MySQL 租户时，支持的驱动如下：

Java 驱动（MySQL Connector/J）

MySQL Connector/J 是 MySQL 官方提供的 JDBC 驱动程序。

使用 Java 应用程序连接 OceanBase 数据库的具体操作示例请参见官网《OceanBase 数据库》文档 快速上手/基于 MySQL 模式创建示例应用程序/创建 Java 示例应用程序。

C 驱动（OceanBase Connector/C）

OceanBase Connector/C 是一个基于 C/C++ 的 OceanBase 客户端开发组件，支持 C API Lib 库。

OceanBase Connector/C 允许 C/C++ 程序以一种较为底层的方式访问 OceanBase 分布式数据库集群，以进行数据库连接、数据访问、错误处理和 Prepared Statement 处理等操作。

OceanBase Connector/C 也称为 LibOBClient，用于应用程序作为独立的服务器进程通过网络连接与数据库服务器 OBServer 节点进行通信。客户端程序在编译时会引用 C API 头文件，同时可以连接到 C API 库文件。LibOBClient 生成的 so 文件为 `libobclient.so`（对应 MySQL 的 `libmysqlclient.so`），OceanBase 数据库安装后的命令行工具是 OBClient（对应 MySQL 的命令行工具）。

使用 C 应用程序连接 OceanBase 数据库的具体操作示例请参见官网《OceanBase 数据库》文档 快速上手/基于 MySQL 模式创建示例应用程序/创建 C 示例应用程序。

Python 驱动（PyMySQL）

PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库。遵循 Python 数据库 API v2.0 规范，并包含了 pure-Python MySQL 客户端库。在 OceanBase 数据库的 MySQL 模式下，用户可以使用 PyMySQL 驱动为 Python 应用程序提供与 OceanBase 数据库的连接。

使用 Python 应用程序连接 OceanBase 数据库的具体操作示例请参见官网《OceanBase 数据库》文档 快速上手/基于 MySQL 模式创建示例应用程序/创建 Python 示例应用程序。

Go 驱动（Go-SQL-Driver）

Go-SQL-Driver 是一个用于 Go 语言的 MySQL 数据库驱动程序，它实现了 Go 的 database/sql/driver 接口，在 OceanBase 数据库的 MySQL 模式下，开发者在 Go 应用程序中通过 database/sql 包的 API 来操作 OceanBase 数据库。

使用 Go 应用程序连接 OceanBase 数据库的具体操作示例请参见官网《OceanBase 数据库》文档 快速上手/基于 MySQL 模式创建示例应用程序/创建 Go 示例应用程序。

ORM 框架

ORM 对象关系映射（Object Relational Mapping，简称 ORM），是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换。从效果上说，它其实是创建了一个可在编程语言里使用的“虚拟对象数据库”。

在连接 OceanBase 数据库的 MySQL 租户时，支持的 ORM 框架及具体的操作示例可参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/示例程序/Java 章节，主要包含如下内容：

- SpringBoot 连接 OceanBase 数据库
- SpringBatch 连接 OceanBase 数据库
- SpringJDBC 连接 OceanBase 数据库
- SpringJPA 连接 OceanBase 数据库
- Hibernate 连接 OceanBase 数据库
- MyBatis 连接 OceanBase 数据库

数据库操作

数据库操作是指对数据库中的数据进行的各种处理活动，包括数据的增加、修改、查询和删除等。下面将介绍 OceanBase 数据库中常见的数据库操作，包括管理数据库对象、数据写入、数据读取、事务管理等。

数据库对象管理概述

数据库对象是数据库的组成部分，是指在数据库中可以通过 SQL 进行操作和使用的对象。下面将介绍 OceanBase 数据库 MySQL 模式下所支持的数据库对象类型、存储方式和数据库对象之间的依赖。

OceanBase 数据库 MySQL 模式下的数据库对象主要包括表（Table）、视图（View）、索引（Index）、分区（Partition）、序列（Sequence）、触发器（Trigger）、存储程序等。

用户管理

OceanBase 数据库用户分为：系统租户用户和普通租户用户。系统租户的内置系统管理员为用户 `root`，MySQL 租户的内置租户管理员为用户 `root`。创建用户时，如果当前会话的租户为系统租户，则新建的用户为系统租户用户，反之为普通租户用户。 无论是系统租户还是普通租户，租户管理员创建的用户只能用于本租户内登录。用户名称在租户内是唯一的，不同租户的用户可以

同名，通过 **用户名@租户名** 的形式在系统全局唯一定位一个用户。

前提条件

数据库在运行过程中，往往需要创建不同的用户，并为用户赋予相应的权限，一般被授予 `CREATE USER` 权限的用户可以创建用户。默认先使用业务租户的 `root` 用户创建其他的业务用户。

由于 `CREATE USER` 权限较大，默认仅集群管理员和租户管理员拥有此系统权限，其他用户如果需要创建用户，则需要被授予 `CREATE USER` 权限，授权相关操作请参见官网《OceanBase 数据库》文档 管理数据库/安全权限/访问控制/用户和权限/MySQL 模式下的权限管理/授予权限。

用户名称规则

为用户指定名称时，需要注意以下限制。

限制一：用户名的唯一性，每个用户名需要保证在租户内唯一。

- 用户名称在租户内是唯一的，不同租户下的用户可以同名，故通过 **用户名@租户名** 的形式可以在系统全局唯一定位一个租户用户。
- 由于系统租户与 MySQL 租户属于同一兼容模式，为区别系统租户和普通租户下的用户，建议对系统租户下的用户名称使用特定前缀。

限制二：用户名的命名约定。

- 使用 OBClient、ODC 等客户端创建用户时，要求用户名长度不超过 64 个字节。
- 使用 OCP 创建用户时，要求用户名以英文字母开头，可包含大写字母、小写字母、数字和下划线，且长度为 2~64 个字符。

创建用户

目前支持两种方式创建用户：

- `create user` 创建用户。
- `grant` 语句创建用户。

示例：

```
MySQL [oceanbase]> create user user01 identified by 'zf******MG';
Query OK, 0 rows affected (0.024 sec)
```

```
MySQL [oceanbase]> grant all privileges on test.* to user01 ;
Query OK, 0 rows affected (0.013 sec)

MySQL [oceanbase]> grant all privileges on test.* to user02 identified by 'dQ*****
*M8';
Query OK, 0 rows affected (0.028 sec)
```

OceanBase 数据库 MySQL 租户不支持更新用户元数据的密码字段。

用户可使用 `show grants` 语句查看用户权限。

示例：

```
MySQL [oceanbase]> show grants for user01;
+----------------------------------------------+
| Grants for user01@%                          |
+----------------------------------------------+
| GRANT USAGE ON *.* TO 'user01'               |
| GRANT ALL PRIVILEGES ON `test`.* TO 'user01' |
+----------------------------------------------+
2 rows in set (0.001 sec)

MySQL [oceanbase]> show grants for user02;
+----------------------------------------------+
| Grants for user02@%                          |
+----------------------------------------------+
| GRANT USAGE ON *.* TO 'user02'               |
| GRANT ALL PRIVILEGES ON `test`.* TO 'user02' |
+----------------------------------------------+
2 rows in set (0.001 sec)
```

更多用户管理的相关操作，请参见官网《OceanBase 数据库》文档 管理数据库/安全权限/访问控制/用户和权限/MySQL 模式下的权限管理 章节。

数据库管理

在 OceanBase 数据库中，数据库（Database）下包括若干表、索引，以及数据库对象的元数据信息。尽量避免在生产环境中使用默认的数据库，如 `test` 数据库。如果没有必要，建议使用 SQL 语句创建用户自己的数据库。

前提条件

在管理数据库前，需要确认以下事项：

- 已部署 OceanBase 集群并且创建了 MySQL 模式租户。
- 已连接到 OceanBase 数据库的 MySQL 租户。
- 已拥有 `CREATE`、`ALTER`、`DROP` 权限。

创建数据库限制

- 在 OceanBase 数据库中，每个数据库的名称必须保证全局唯一。
- 数据库名长度限制不超过 128 个字符。
- 只包含大小写字母、数字、下划线、美元符号和中文字符。
- 避免使用保留关键字作为数据库名。有关 OceanBase 数据库 MySQL 模式的保留关键字的详细信息，请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/预留关键字（MySQL 模式）。

创建数据库建议

- 建议给数据库起一个有意义的名字，尽量能够反映其用途和内容。例如，使用 `应用标识_子应用名（可选）_db` 作为数据库名称。
- 建议使用 `root` 用户创建数据库和相关用户，并且只赋予必要的权限，以确保数据库的安全性和可控性。
- 在创建数据库时，请确保设置合适的默认字符集和排序规则，以确保数据的正确存储和排序。为了适应业务的长远发展，建议在创建数据库时使用 `utf8mb4` 字符集编码，以确保能够存储绝大多数字符。
- 可以通过使用反引号 (`) 包围的方式来创建以纯数字命名的数据库名，但这种做法并不推荐，因为纯数字命名无较明显意义，且查询使用都需要加反引号 (`)，这也将导致在使用时出现不必要的复杂性，易引起混淆。

相关语法

示例一：创建数据库 `test_db`，并指定字符集为 `utf8mb4`。

```
obclient [(none)]> CREATE DATABASE test_db DEFAULT CHARACTER SET utf8mb4;
```

示例二：创建只读属性的数据库 `test_ro_db`。

```
obclient [(none)]> CREATE DATABASE test_ro_db READ ONLY;
```

示例三：创建读写属性的数据库 `test_rw_db`。

```
obclient [(none)]> CREATE DATABASE test_rw_db READ WRITE;
```

更多 database 管理相关的详细语法，请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/规划数据库对象 章节。

表管理

在 OceanBase 数据库中，表是最基础的数据存储单元。表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。

表分类

OceanBase 数据库支持主键表和无主键表。

- 主键表：有主键表即数据表中包含主键的表，在 OceanBase 数据库中需要满足以下规则。
  - 每个数据表最多拥有一个主键列集合。
  - 主键列的数量不能超过 64 列，且主键数据总长度不能超过 16 KB。

  在创建有主键表后，会自动为主键列创建一个全局唯一索引，可以通过主键快速定位到行。

  如下例所示，创建了一个以 `emp_id` 为主键的表 `emp_table`，它属于有主键表。

  ```
  CREATE TABLE emp_table (
    emp_id INT PRIMARY KEY,
    emp_name VARCHAR(100),
    emp_age INT NOT NULL
  );
  ```

- 每个数据表最多拥有一个主键列集合。
- 主键列的数量不能超过 64 列，且主键数据总长度不能超过 16 KB。
- 无主键表：数据表中未指定主键的表称为无主键表。

  如下例所示，数据表 `student_table` 未指定主键，它属于无主键表。

  ```
  CREATE TABLE student_table (
    student_id INT NOT NULL,
    student_name VARCHAR(100),
    student_age INT NOT NULL
  );
  ```

  OceanBase 数据库的无主键表采用分区级自增列作为隐藏主键。

创建表限制

- 表的名称不能超过 64 个字符。
- 表的行长度不能超过 1.5M 字节。
- 单表的列数不能超过 4096 列。
- 单表的索引个数不能超过 128 个。
- 单表的主键总列数不能超过 64 列，且主键数据总长度不能超过 16 KB。
- 单表的最大分区数有租户级配置项 `max_partition_num`（V4.2.1_CE_BP3 开始引入）控制，默认为 8192 个，最大不能超过 65536 个。

更多限制内容请参见官网《OceanBase 数据库》文档 OceanBase 简介/使用限制。

创建表建议

- 表名推荐英文字母统一采用大写或小写，不混用大小写。
- 表名推荐望文生义。示例："TEST"。
- 表名不使用系统保留字和关键字。
- 中间表用于保留中间结果集，命名规则推荐为：`tmp_表名(或缩写)列名（或缩写）创建时间`，示例：`tmp_account_tbluser_20220224`。
- 备份表用于备份或抓取源表快照，命名规则推荐为：`bak_表名(或缩写)列名（或缩写）创建`

时间，示例：`bak_ account_tbluser_20220224`。

更多表命名规范内容，可参见官网《OceanBase 数据库》文档 参考指南/数据库涉及规范和约束/对象命名规范/表命名规范。

相关语法

示例：

```
create table ware(w_id int
, 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)
, unique(w_name, w_city)
, primary key(w_id)
);

create table cust (c_w_id int NOT NULL
, c_d_id int NOT null
, c_id int NOT null
, c_discount decimal(4, 4)
, c_credit char(2)
, c_last varchar(16)
, c_first varchar(16)
, c_middle char(2)
, c_balance decimal(12, 2)
, c_ytd_payment decimal(12, 2)
, c_payment_cnt int
, c_credit_lim decimal(12, 2)
, 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 date
, c_delivery_cnt int
, c_data varchar(500)
, index icust(c_last, c_d_id, c_w_id, c_first, c_id)
, FOREIGN KEY (c_w_id) REFERENCES ware(w_id)
, primary key (c_w_id, c_d_id, c_id)
);
```

OceanBase 数据库 MySQL 租户支持外键。不过在分布式数据库里，如果读写并发很高，不建议在数据库层面使用外键约束。外键会增加不必要的阻塞和死锁，可能会给性能带来负面影响。

复制表的结构使用 `like`，表的结构中包括主键、唯一键、索引名称都会复制。在 MySQL 语法里，主键名、唯一约束和索引名在一个表内不能重复，但是不同表之间可以重复。

```
create table t1 like ware;

MySQL [test]> show create table t1\G
*************************** 1. row ***************************
Table: t1
Create Table: CREATE TABLE `t1` (
`w_id` int(11) NOT NULL,
`w_ytd` decimal(12,2) DEFAULT NULL,
`w_tax` decimal(4,4) DEFAULT NULL,
`w_name` varchar(10) DEFAULT NULL,
`w_street_1` varchar(20) DEFAULT NULL,
`w_street_2` varchar(20) DEFAULT NULL,
`w_city` varchar(20) DEFAULT NULL,
`w_state` char(2) DEFAULT NULL,
`w_zip` char(9) DEFAULT NULL,
PRIMARY KEY (`w_id`),
UNIQUE KEY `w_name` (`w_name`, `w_city`) BLOCK_SIZE 16384 GLOBAL
) DEFAULT CHARSET = utf8mb4 ROW_FORMAT = COMPACT COMPRESSION = 'zstd_1.3.8' REPLIC
A_NUM = 1 BLOC
1 row in set (0.003 sec)
```

复制表的结构和数据使用 `create table ... as select`。不过要注意的是，该语句复制的是表的基本数据类型，对于主键、唯一约束和索引信息等不会复制。

```
create table t2 as select * from ware;

MySQL [test]> show create table t2\G
*************************** 1. row ***************************
Table: t2
Create Table: CREATE TABLE `t2` (
`w_id` int(11) NOT NULL,
`w_ytd` decimal(12,2) DEFAULT NULL,
`w_tax` decimal(4,4) DEFAULT NULL,
`w_name` varchar(10) DEFAULT NULL,
`w_street_1` varchar(20) DEFAULT NULL,
`w_street_2` varchar(20) DEFAULT NULL,
`w_city` varchar(20) DEFAULT NULL,
`w_state` char(2) DEFAULT NULL,
`w_zip` char(9) DEFAULT NULL
) DEFAULT CHARSET = utf8mb4 ROW_FORMAT = COMPACT COMPRESSION = 'zstd_1.3.8' REPLIC
A_NUM = 1 BLOC
```

```
1 row in set (0.002 sec)
```

更多 table 管理相关的详细语法，请参见官网《OceanBase 数据库》文档 参考指南/数据库对象管理/MySQL 模式/创建和管理表 章节。

索引管理

索引也叫二级索引，是一种可选的表结构。OceanBase 数据库采用的是聚集索引表模型，对于用户指定的主键，系统会自动生成主键索引，而对于用户创建的其他索引，则是二级索引。用户可以根据自身业务需要来决定在哪些字段上创建索引，以便加快在这些字段上的查询速度。

更多有关 OceanBase 数据库索引的信息，请参见官网《OceanBase 数据库》文档 参考指南/系统原理/数据库对象/MySQL 模式/索引 章节。

前提条件

在创建索引前，需要确认以下事项：

- 已部署 OceanBase 集群并且创建了 MySQL 模式租户和用户。
- 已连接到 OceanBase 数据库的 MySQL 租户。
- 已创建数据库、表。
- 已拥有 `INDEX` 权限。

创建索引限制

- 在 OceanBase 数据库中，索引名称必须在表的范围内保证唯一。
- 索引名称的长度不能超过 64 字节。
- 唯一索引使用限制：
  - 在一个表中可以创建多个唯一索引，但是每个唯一索引所对应的列值都必须保持唯一。
  - 如果除了主键之外，还希望其他列的组合满足全局唯一性的要求，需要使用全局唯一索引来实现。

  - 在使用局部唯一索引时，索引必须包含表的分区函数中的所有列。

- 在一个表中可以创建多个唯一索引，但是每个唯一索引所对应的列值都必须保持唯一。
- 如果除了主键之外，还希望其他列的组合满足全局唯一性的要求，需要使用全局唯一索引来实现。
- 在使用局部唯一索引时，索引必须包含表的分区函数中的所有列。
- 在使用全局索引时，全局索引的分区规则不一定需要与表的分区规则完全相同或一致。

创建索引建议

- 建议使用能够简洁地描述索引所涵盖的列和用途的名称，例如，`idx_customer_name`。
- 如果全局索引的分区规则和主表的分区规则相同并且分区数相同，建议创建一个局部索引。
- 建议并行下发创建索引的 SQL 语句条数，不要超过租户 Unit 规格中的核数上限。例如，租户的 Unit 规格为 4 核（4C），则建议并发创建索引不超过 4 条。
- 对经常更新的表要避免对其进行过多的索引，对经常用于查询的字段应该创建索引。
- 数据量小的表建议不要使用索引，数据较少时，查询全表数据花费的时间可能比遍历索引的时间还要短，索引就可能不会产生优化效果。
- 当修改性能远远大于检索性能时，不建议创建索引。
- 创建高效索引：
  - 索引要全部包含所需查询的列，包含的列越全越好，这样可以尽可能的减少回表的行数。

  - 等值条件永远放在最前面。

  - 过滤与排序数据量大的放前面。

- 索引要全部包含所需查询的列，包含的列越全越好，这样可以尽可能的减少回表的行数。
- 等值条件永远放在最前面。

- 过滤与排序数据量大的放前面。

相关语法

示例：使用以下 SQL 语句创建一个名为 `tbl2` 的表，并为表 `tbl2` 创建一个基于 `col2` 列的索引。

1. 创建表 `tbl2`。

```
obclient [test]> CREATE TABLE tbl2(col1 INT, col2 INT, col3 VARCHAR(50), PRIMARY K
EY (col1));
```

2. 在表 `tbl2` 上基于 `col2` 列创建一个名为 `idx_tbl2_col2` 的索引。

```
obclient [test]> CREATE INDEX idx_tbl2_col2 ON tbl2(col2);
```

3. 查看表 `tbl2` 索引信息。

```
obclient [test]> SHOW INDEX FROM tbl2;
```

返回结果如下：

```
+-------+------------+---------------+--------------+-------------+-----------+---
----------+----------+--------+------+------------+-----------+---------------+---
------+------------+
| Table | Non_unique | Key_name      | Seq_in_index | Column_name | Collation | Ca
rdinality | Sub_part | Packed | Null | Index_type | Comment   | Index_comment | Vi
sible | Expression |
+-------+------------+---------------+--------------+-------------+-----------+---
----------+----------+--------+------+------------+-----------+---------------+---
------+------------+
| tbl2  |          0 | PRIMARY       |            1 | col1        | A
|        NULL | NULL     | NULL   |      | BTREE      | available |
| YES     | NULL       |
| tbl2  |          1 | idx_tbl2_col2 |            1 | col2        | A
|        NULL | NULL     | NULL   | YES  | BTREE      | available |
| YES     | NULL       |
+-------+------------+---------------+--------------+-------------+-----------+---
----------+----------+--------+------+------------+-----------+---------------+---
------+------------+
2 rows in set
```

更多索引管理相关的详细语法，请参见官网《OceanBase 数据库》文档 参考指南/数据库对象管理/MySQL 模式/创建和管理索引 章节。

子程序

子程序是包含很多 SQL 和 PL 语句的 PL 单元，以解决特定的问题或者进行一组相关的任务。 子程序可以包含参数，具体值由调用者传入。子程序可以是一个存储过程或者函数。典型的用法是使用存储过程执行一个操作，使用函数计算并返回一个值。

存储程序是存储在数据库内部的子程序，为很多不同数据库应用子程序实现复杂的逻辑运算。MySQL 模式的子程序只有独立的程序，即在 Schema 内创建的程序。MySQL 模式的子程序遵循 SQL 标准中存储程序的标准，与 Oracle 模式的子程序（PL）在语法与功能上都有显著的区别。

可以直接参见官网《OceanBase 数据库》文档 参考指南/系统原理/用户接口和查询语言/PL/PL 概念/MySQL 模式 部分，完成自学。

数据写入

插入数据

表创建后，可以使用 `INSERT` 语句或其他语句向表中插入行记录。下面介绍相关语句的使用方法和示例。

前提条件

- 已连接到数据库的 MySQL 租户。
- 已拥有待操作的表的 INSERT 权限。

使用 INSERT 语句插入数据

插入单行数据

通过 `INSERT` 语句可以插入单行数据。

假设待插入数据的表信息如下：

```
obclient [test]> CREATE TABLE t_insert(
id int NOT NULL PRIMARY KEY,
name varchar(10) NOT NULL,
value int,
```

```
gmt_create DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
Query OK, 0 rows affected
```

其中，表的 `id` 列、`name` 列不能为空，且 `id` 列为主键列，满足唯一性约束要求，不能有重复的值；`gmt_create` 列指定了默认值。

示例：使用多个单行插入语句插入多行数据。

由于 `gmt_create` 列指定了默认值，在插入数据时可以不指定默认值。

```
obclient [test]> INSERT INTO t_insert(id, name, value)
VALUES (1,'CN',10001);
Query OK, 2 rows affected

obclient [test]> INSERT INTO t_insert(id, name, value)
VALUES(2,'US', 10002);
Query OK, 2 rows affected
```

注意，如果 `gmt_create` 列未指定默认值，则在插入数据时，必须指定值，语句如下。

```
obclient [test]> INSERT INTO t_insert(id, name, value, gmt_create)
VALUES (3,'EN', 10003, current_timestamp ());
Query OK, 1 row affected
```

批量插入多行数据

在插入数据时，如果要插入多条记录，也可以用一个 `INSERT` 语句包含多个 `VALUES` 来批量插入。单个多行插入语句比多个单行插入语句要快。

示例：批量插入多行数据。

```
obclient [test]> INSERT INTO t_insert(id, name, value)
VALUES (1,'CN',10001),(2,'US', 10002);
Query OK, 2 rows affected
```

此外，当需要备份表数据或者将一个表的全部记录拷贝到另一个表时，可以使用查询语句 `INSERT INTO ... SELECT ... FROM` 充当 `INSERT` 的 `values` 子句进行批量插入。

示例：将表 `t_insert` 中的全部数据备份到 `t_insert_bak` 表中。

```
obclient [test]> SELECT * FROM t_insert;
+----+------+-------+---------------------+
```

```
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-12 15:17:17 |
|  2 | US   | 10002 | 2022-10-12 16:29:16 |
|  3 | EN   | 10003 | 2022-10-12 16:29:26 |
+----+------+-------+---------------------+
3 rows in set

obclient [test]> CREATE TABLE t_insert_bak(
id number NOT NULL PRIMARY KEY,
name varchar(10) NOT NULL,
value number,
gmt_create DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP
);
Query OK, 0 rows affected

obclient [test]> INSERT INTO t_insert_bak SELECT * FROM t_insert;
Query OK, 2 rows affected

obclient [test]> SELECT * FROM t_insert_bak;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-12 15:17:17 |
|  2 | US   | 10002 | 2022-10-12 16:29:16 |
|  3 | EN   | 10003 | 2022-10-12 16:29:26 |
+----+------+-------+---------------------+
3 rows in set
```

使用其他方式插入数据

除了 `INSERT` 语句，当表中无数据记录，或者表中有数据记录但无主键或唯一键冲突时，还可以使用 `REPLACE INTO` 语句代替 `INSERT` 语句插入数据。`REPLACE INTO` 语句的详细语法及说明请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/REPLACE。

示例：

- 创建 `t_replace` 表后，使用 `REPLACE INTO` 语句插入数据。

```
obclient [test]> CREATE TABLE t_replace(
id int NOT NULL PRIMARY KEY
, name varchar(10) NOT NULL
, value int
,gmt_create timestamp NOT NULL DEFAULT current_timestamp
);
```

```
Query OK, 0 rows affected

obclient [test]> REPLACE INTO t_replace VALUES(1,'CN',2001, current_timestamp ());
Query OK, 1 row affected

obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-11-23 09:52:44 |
+----+------+-------+---------------------+
1 row in set
```

- 在有数据记录的表 `t_replace` 中，使用 `REPLACE INTO` 语句插入数据。

```
obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-03-22 16:13:55 |
+----+------+-------+---------------------+
1 row in set

obclient [test]> REPLACE INTO t_replace values(2,'US',2002, current_timestamp ());
Query OK, 1 row affected

obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-11-23 09:52:44 |
|  2 | US   |  2002 | 2022-11-23 09:53:05 |
+----+------+-------+---------------------+
2 rows in set
```

更新数据

插入表数据后，可以使用 `UPDATE` 语句或其他语句来更新表的行记录。下面介绍相关语句的使用方法和示例。

前提条件

- 已连接到数据库的 MySQL 租户。

- 已拥有待操作的表的 UPDATE 权限。

使用 UPDATE 语句更新数据

通常使用 UPDATE 语句来更新表数据，UPDATA 语句的详细语法及说明请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/UPDATE。

简单的 UPDATE 语句的语法格式如下：

```
UPDATE table_name
SET column_name = value [, column_name = value]...
[ WHERE condition ];
```

参数	是否必填	描述
table_name	是	指定需要更新数据的表。
column_name = value [, column_name = value]	是	指定需要更新的列，等号（=）后面的 value 是更新的目标值。
[WHERE condition]	否	条件子句指定要更新的行记录必须满足的条件。如果不填表示更新表对应列的所有记录。

更新 t_insert 表中 value 列的所有值，并将所有值加 1。示例如下：

```
obclient [test]> UPDATE t_insert SET value = value+1;
Query OK, 4 rows affected
Rows matched: 4  Changed: 4  Warnings: 0

obclient [test]> SELECT * FROM t_insert;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10002 | 1970-01-01 17:18:06 |
|  2 | US   | 10003 | 1970-01-01 17:18:47 |
|  3 | EN   | 10004 | 1970-01-01 17:18:47 |
|  4 | JP   | 10005 | 1970-01-01 17:28:21 |
+----+------+-------+---------------------+
4 rows in set
```

执行 UPDATE 语句时要注意控制事务不要太大，可以通过 LIMIT 关键字来控制数量或者通过 WHERE 关键字来控制范围。这是因为不带条件更新数据时，如果记录数达到几十万或者几百万，

会有大事务产生，可能会导致执行失败。

使用其他语句更新数据

除了显式的 `UPDATE` 语句外，还有其他语句也可以更新数据。例如，通过 `INSERT` 语句插入数据时，由于约束冲突，可以使用 `ON DUPLICATE KEY UPDATE` 子句将插入数据转变为更新数据的语句来更新相关字段。

使用 `ON DUPLICATE KEY UPDATE` 子句将插入数据转变为更新数据，示例如下：

```
obclient [test]> SELECT * FROM t_insert;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-12 15:17:17 |
|  2 | US   | 10002 | 2022-10-12 16:29:16 |
|  3 | EN   | 10003 | 2022-10-12 16:29:26 |
|  4 | JP   | 10004 | 2022-10-12 17:02:52 |
+----+------+-------+---------------------+
4 rows in set

obclient [test]> INSERT INTO t_insert(id, name, value) VALUES (3,'UK', 10003),(5,
'CN', 10005
Query OK, 1 row affected

obclient [test]> SELECT * FROM t_insert;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-12 16:29:16 |
|  2 | US   | 10002 | 2022-10-12 15:17:17 |
|  3 | UK   | 10003 | 2022-10-12 16:29:26 |
|  4 | JP   | 10004 | 2022-10-12 17:02:52 |
|  5 | CN   | 10005 | 2022-10-12 17:27:46 |
+----+------+-------+---------------------+
5 rows in set
```

示例中，`ON DUPLICATE KEY UPDATE name = VALUES(name)` 即表示当插入的数据与表中的主键值有重复时，将表中冲突行原数据中 `(3,'EN', 10003)` 的 `name` 列的值更新为当前待插入的 `name` 列的值。

删除数据

表中插入数据后，可以使用 `DELETE` 语句或其他语句来删除表中的记录。本文介绍了相关语句的使用方法和示例。

前提条件

在删除表数据前，请确认以下事项：

- 已连接到数据库的 MySQL 租户。
- 已拥有待操作的表的 `DELETE` 权限。如果使用 `TRUNCATE TABLE` 语句清空表数据，则还需要拥有该表的 `CREATE` 权限。

使用 DELETE 语句删除数据

通常使用 `DELETE` 语句来删除表中的部分数据或全部数据，`DELETE` 语句的详细语法及说明请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/DELETE。

简单的 `DELETE` 语句的语法格式如下：

```
DELETE FROM table_name [ WHERE condition ] ;
```

参数	是否必填	描述
table_name	是	指定需要删除数据的表。
[WHERE condition]	否	条件子句指定要删除的数据必须满足的条件，如果没有提供就全表删除。

示例：

- 删除 `t_insert` 表中的所有行数据。

  ```
  obclient [test]> DELETE FROM t_insert;
  Query OK, 3 row affected
  ```

  当表中数据记录多达百万以上，一次性执行删除可能会出现性能问题，建议根据表内的信息使用 `WHERE` 条件进行分批删除，或者直接 **使用 TRUNCATE TABLE 语句清空表数据**。

- 筛选表 `t_insert` 中 `value` 列数据，分别执行多条语句分批删除 `value < 10000`，`value < 20000`，`value < 30000` 的数据。

```
obclient [test]> DELETE FROM t_insert WHERE value < 100000;

obclient [test]> DELETE FROM t_insert WHERE value < 200000;

obclient [test]> DELETE FROM t_insert WHERE value < 300000;
```

使用 TRUNCATE TABLE 语句清空表数据

`TRUNCATE TABLE` 语句用于完全清空指定表，但是保留表结构，包括表中定义的分区信息。从逻辑上来讲，该语句等价于删除所有行的 `DELETE FROM` 语句。

`TRUNCATE TABLE` 语句的语法格式如下：

```
TRUNCATE [TABLE] table_name;
```

通过 `TRUNCATE TABLE` 语句清空表 `t_insert` 表中的所有数据，示例如下：

```
obclient [test]> TRUNCATE TABLE t_insert;
```

更多 `TRUNCATE TABLE` 语句的使用及说明请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/TRUNCATE TABLE。

替换数据

可以使用 `REPLACE INTO` 语句来插入或者更新数据，下面介绍相关语句的使用方法和示例。

前提条件

在替换表数据前，请确认以下事项：

- 已连接到数据库的 MySQL 租户。
- 已拥有待操作的表的 `INSERT`、`UPDATE` 和 `DELETE` 权限。

使用 REPLACE INTO 语法替换数据

通常使用 REPLACE INTO 语句替换表中的一条或多条记录，`REPLACE INTO` 语句的详细语法及说明请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模

式）/SQL 语句/REPLACE。

`REPLACE INTO` 语句语法格式如下：

```
REPLACE INTO table_name VALUES(list_of_values);
```

参数	是否必填	描述	示例
table_name	是	指定需要插入数据的表	table1
(list_of_values)	是	插入的数据	(1,'CN',2001, current_timestamp ())

`REPLACE INTO` 语句会根据替换数据与表的主键或唯一键进行判断：

- 如果没有主键或唯一键冲突，则插入记录。
- 如果存在主键或唯一键冲突，则先删除已有记录，再插入新行记录。目标表建议有主键或者唯一索引，否则容易插入重复的记录。

示例：

- 创建 `t_replace` 表后，使用 `REPLACE INTO` 语句插入一行数据。

```
obclient [test]> CREATE TABLE t_replace(
id number NOT NULL PRIMARY KEY
, name varchar(10) NOT NULL
, value number
,gmt_create timestamp NOT NULL DEFAULT current_timestamp
);
Query OK, 0 rows affected

obclient [test]> REPLACE INTO t_replace values(1,'CN',2001, current_timestamp ());
Query OK, 1 row affected

obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-10-13 14:06:58 |
+----+------+-------+---------------------+
1 row in set
```

从示例结果可以看出，`t_replace` 表创建后未插入数据，执行 `REPLACE INTO` 语句后，表中插入了一条记录。

- 再次使用 `REPLACE INTO` 语句，插入一行数据。

```
obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-10-13 14:06:58 |
+----+------+-------+---------------------+
1 row in set

obclient [test]> REPLACE INTO t_replace(id, name, value, gmt_create) VALUES(2,'US'
,2002,  cur
Query OK, 1 row affected

obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-10-13 14:06:58 |
|  2 | US   |  2002 | 2022-10-13 14:17:56 |
+----+------+-------+---------------------+
2 rows in set
```

从示例结果可以看出，`t_replace` 表中已有一条记录，由于 `(2,'US',2002,current_timestamp ())` 数据与表内的记录不违反唯一性约束，故执行结果是在 `t_replace` 表中插入一条记录。

- 使用查询语句充当 `REPLACE INTO` 语句的 `VALUES` 子句来插入多条数据。将表 `t_insert` 中的数据插入到表 `t_replace` 中。

```
obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-10-13 14:06:58 |
|  2 | US   |  2002 | 2022-10-13 14:17:56 |
+----+------+-------+---------------------+
2 rows in set

obclient [test]> SELECT * FROM t_insert;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  7 | EN   |  1007 | 2022-10-13 14:36:36 |
|  8 | JP   |  1008 | 2022-10-13 14:36:36 |
+----+------+-------+---------------------+
2 rows in set

obclient [test]> REPLACE INTO t_replace
```

```
SELECT id,name,value,gmt_create FROM t_insert;
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-10-13 14:06:58 |
|  2 | US   |  2002 | 2022-10-13 14:17:56 |
|  7 | EN   |  1007 | 2022-10-13 14:36:36 |
|  8 | JP   |  1008 | 2022-10-13 14:36:36 |
+----+------+-------+---------------------+
4 rows in set
```

当表中有数据记录且插入的数据存在主键或唯一键冲突时，使用 `REPLACE INTO` 语句可以将表中已有的存在冲突的数据删除，替换为新的数据。

- 在 `t_replace` 表中插入一条记录，示例如下：

```
obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-10-13 14:06:58 |
|  2 | US   |  2002 | 2022-10-13 14:17:56 |
|  7 | EN   |  1007 | 2022-10-13 14:36:36 |
|  8 | JP   |  1008 | 2022-10-13 14:36:36 |
+----+------+-------+---------------------+
4 rows in set

obclient [test]> REPLACE INTO t_replace(id, name, value, gmt_create) VALUES(2,'EN'
,2002,  cur
Query OK, 2 rows affected

obclient [test]> SELECT * FROM t_replace;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   |  2001 | 2022-10-13 14:06:58 |
|  2 | EN   |  2002 | 2022-10-13 14:44:33 |
|  7 | EN   |  1007 | 2022-10-13 14:36:36 |
|  8 | JP   |  1008 | 2022-10-13 14:36:36 |
+----+------+-------+---------------------+
4 rows in set
```

在示例中，由于表 `t_replace` 的 `id` 列为主键列，需要满足唯一性约束，而插入的数据

`(2,'EN',2002,current_timestamp ())` 违反唯一性约束，系统删除了原来的行记录 `(2,'US',2002,current_timestamp ())`，插入 `(2,'EN',2002,current_timestamp ())` 作为新的行记录。

数据读取

单表查询

下面将介绍如何使用 SQL 语句进行 OceanBase 数据库中的单表查询操作。

前提条件

- 已连接 OceanBase 数据库 MySQL 模式租户。
- 已拥有 `SELECT` 权限。

SELECT 查询

`SELECT` 语句单表查询的一般结构如下：

```
SELECT [ALL | DISTINCT | UNIQUE | SQL_CALC_FOUND_ROWS] select_list
FROM table_name
[ WHERE query_condition ]
[ GROUP BY group_by_condition ]
[ HAVING group_condition ]
[ ORDER BY column_list ][ASC | DESC]
[ LIMIT limit_clause ]

column_list:
column_name[,column_name...]
```

参数解释：

参数	说明
select_list	要检索的列的列表，可以是列名、表达式、聚合函数等。可以使用逗号分隔多个列。
table_name	要检索数据的表的名称。
WHERE query_condition	可选参数，用于指定检索的条件。只有符合条件的行才会被返回。

GROUP BY group_by_condition	可选参数，用于按照指定的列对结果进行分组。通常与聚合函数一起使用。
HAVING group_condition	可选参数，用于过滤分组后的结果集，只返回满足条件的分组。
ORDER BY column_list	可选参数，用于对结果集进行排序。可以指定一个或多个列进行排序。
ASC|DESC	可选参数，用于指定排序的顺序。ASC 表示升序（默认），DESC 表示降序。
LIMIT limit_clause	可选参数，用于限制返回的结果集的行数。
column_list	用于指定要检索的列的参数，可以是单个列或多个列，用逗号分隔。
column_name	要检索的列的名称。

当 WHERE、GROUP BY、HAVING、ORDER BY、LIMIT 这些关键字一起使用时，先后顺序有明确的限制。关键字执行顺序如下：

1. 执行 FROM 找到表。
2. 执行 WHERE 指定约束条件。分组前对数据进行筛选。
3. 执行 GROUP BY 将取出的每条记录进行分组（聚合）。如果没有 GROUP BY，则整体作为一组。
4. 执行 HAVING 将分组后的结果进行筛选，最后返回整个 SQL 的查询结果。
5. 执行 SELECT。
6. 执行 DISTINCT 去重。
7. 执行 ORDER BY 将结果按条件升序或降序排序。
8. 执行 LIMIT 限制结果的条数。

```
SELECT * FROM student;
```

返回结果如下：

```
+----+-----------+--------+-----+-------+-----------------+-------+
| id | name      | gender | age | score | enrollment_date | notes |
+----+-----------+--------+-----+-------+-----------------+-------+
|  1 | Emma      |      0 |  20 |    85 | 2021-09-01      | NULL  |
|  2 | William   |      1 |  21 |  90.5 | 2021-09-02      | B     |
|  3 | Olivia    |      0 |  19 |  95.5 | 2021-09-03      | A     |
|  4 | James     |      1 |  20 |  87.5 | 2021-09-03      | NULL  |
|  5 | Sophia    |      0 |  20 |  91.5 | 2021-09-05      | B     |
```

```
|  6 | Benjamin  |       1 |  21 |   96.5 | 2021-09-01       | A      |
|  7 | Ava       |       0 |  22 |   89.5 | 2021-09-06       | NULL   |
|  8 | Michael   |       1 |  18 |   93.5 | 2021-09-08       | B      |
|  9 | Charlotte |       1 |  19 |     88 | 2021-09-06       | NULL   |
| 10 | Ethan     |       1 |  20 |     92 | 2021-09-01       | B      |
+----+-----------+---------+-----+--------+------------------+--------+
10 rows in set
```

数据过滤

要查询满足特定条件的数据时，可以通过在 `SELECT` 查询语句中添加一个 `WHERE` 子句来进行数据过滤。`WHERE` 子句后面可以包含一个或多个条件，这些条件用于筛选数据，只有满足 `WHERE` 条件的数据才会被返回。可以根据特定需求，通过灵活运用查询条件来过滤和检索目标数据。

在使用 `WHERE` 子句时，要确保条件正确和使用合适的运算符。

`WHERE` 子句常用的查询条件如下表所示。

查询条件类型	谓词
比较查询	`=`, `>`, `<`, `>=`, `<=`, `!=`, `<>`
逻辑查询（多重条件）	`AND`, `OR`, `NOT`
模糊查询（字符匹配）	`LIKE`, `NOT LIKE`
区间查询（确定范围）	`BETWEEN AND`, `NOT BETWEEN AND`
指定集合查询	`IN`, `NOT IN`
NULL 值查询	`IS NULL`, `IS NOT NULL`

更多有关查询条件运算符的详细信息，请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/运算符/比较运算符。

数据分组

在 SQL 查询中，使用 `GROUP BY` 子句可以对查询结果进行分组。`GROUP BY` 支持单字段分组和多字段分组。在分组前，可以使用 `WHERE` 子句对数据进行筛选，可以使用 `HAVING` 子句在分组后对数据进行筛选，还可以使用 `ORDER BY` 子句在分组后对数据进行排序。

- 在使用 `GROUP BY` 子时，`SELECT` 语句中的列必须是 `GROUP BY` 子句中的列或聚合函数。
- 在使用 `HAVING` 子句时，`HAVING` 条件是对分组结果进行筛选，而不是对原始数据进行筛选。

当查询中包含 `HAVING` 时，先获得不含 `HAVING` 子句时的 SQL 查询结果，然后在这个结果的基础

上使用 HAVING 条件筛选出符合的数据，最后返回这些数据。由此，HAVING 后是可以使用聚合函数，并且这个聚集函数不必与 SELECT 后面的聚集函数相同。

示例：查询 2019 年订单数量大于 1 的客户，并输出 user_id 和 COUNT(order_id)。

```
SELECT user_id, COUNT(order_id)
FROM fruit_order t
WHERE t.order_year = 2019
GROUP BY user_id
HAVING COUNT(order_id) >= 2;
```

返回结果如下:

```
+----------+-------------+
| 客户ID    | 下单数量      |
+----------+-------------+
|     1022 |           2 |
+----------+-------------+
1 row in set
```

聚合查询

聚合查询是一种用于对数据进行聚合操作并返回结果摘要的查询方式。它可以对一组数据进行统计、计数、求和、平均值、最大值、最小值等聚合操作。聚合查询通常与 GROUP BY 子句一起使用，以对数据进行分组并对每个组进行聚合操作。GROUP BY 子句按照指定的列对数据进行分组，然后聚合函数被应用于每个组，生成一个结果集。

分组中常用的聚合函数如下表所示。

聚合函数	描述
MAX()	查询指定列的最大值。
MIN()	查询指定列的最小值。
COUNT()	统计查询结果的行数。
SUM()	返回指定列的总和。
AVG()	返回指定列数据的平均值 。

有关使用聚合函数查询的详细信息，请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/数据读取/在查询中使用操作符和函数/查询中使用聚合函数。

示例：使用 `count()`、`sum()`、`avg()`、`round()`、`min()`、`max()` 统计玩具订单的销售额。

```
SELECT toy_id
, count(*)                          order_count
, sum(toy_amount)                   sum_amount
, round(avg(toy_amount),2)          avg_amount
, min(toy_amount)                   min_amount
,max(toy_amount)                    max_amount
FROM toys_order GROUP BY toy_id  ORDER BY toy_id;
```

返回结果如下：

```
+--------+-------------+------------+------------+------------+------------+
| toy_id | order_count | sum_amount | avg_amount | min_amount | max_amount |
+--------+-------------+------------+------------+------------+------------+
|      1 |           2 |        150 |      75.00 |         50 |        100 |
|      2 |           3 |        490 |     163.33 |        100 |        200 |
|      3 |           2 |        680 |     340.00 |        330 |        350 |
+--------+-------------+------------+------------+------------+------------+
3 rows in set
```

数据排序

数据排序是一种对查询结果按照指定的列或表达式进行排序的操作，它可以按照升序（`ASC`）或降序（`DESC`）的方式重新排列数据。在 SQL 查询中，可以使用 `ORDER BY` 子句来指定排序的方式。`ORDER BY` 子句支持单字段排序、多字段排序、按别名排序和按函数排序，多字段排序之间用逗号隔开。进行排序查询时，使用如果不添加 `ASC` 或 `DESC` 关键字，则默认查询结果为升序排列。

使用 `ORDER BY` 子句可以对结果集进行排序是消耗资源的操作，尤其是对大型数据集。在必要时，建议使用索引来优化排序操作。确保指定正确的列和排序顺序。

示例：在 `student` 表中按照 `score` 升序显示学生信息。

```
SELECT id, name, score
FROM student
ORDER BY score;
```

返回结果如下：

```
+----+------------+-------+
| id | name       | score |
+----+------------+-------+
```

```
|  1 | Emma      |    85 |
|  4 | James     |  87.5 |
|  9 | Charlotte |    88 |
|  7 | Ava       |  89.5 |
|  2 | William   |  90.5 |
|  5 | Sophia    |  91.5 |
| 10 | Ethan     |    92 |
|  8 | Michael   |  93.5 |
|  3 | Olivia    |  95.5 |
|  6 | Benjamin  |  96.5 |
+----+-----------+-------+
10 rows in set
```

LIMIT 子句限制结果集

在 SQL 查询中，可以使用 `LIMIT` 子句限制返回结果集的行数。

`LIMIT` 子句限制行数的使用格式一如下：

```
LIMIT [offset,] row_count
```

`LIMIT` 子句限制行数的使用格式二如下：

```
LIMIT row_count OFFSET offset
```

- `offset`：表示偏移量，即要跳过的行数。在格式一中 `offset` 为可选项，默认为 0，表示跳过 0 行。`offset` 取值范围为 [0,+∞)。
- `row_count`：表示要返回的行数。在格式一中如果不指定 `offset`，则默认从第一行开始。`row_count` 取值范围为 [0,+∞)。
- `offset` 和 `row_count` 的值有以下限制：
  - 不能使用表达式。
  - 只能是明确的数字，不能为负数。
- 不能使用表达式。
- 只能是明确的数字，不能为负数。

示例：在 `student` 表中查询列 `id`、`name` 第五行之后的三行数据。

```
SELECT id, name
FROM student
LIMIT 3 OFFSET 5;
```

返回结果如下：

```
+----+----------+
| id | name     |
+----+----------+
|  6 | Benjamin |
|  7 | Ava      |
|  8 | Michael  |
+----+----------+
3 rows in set
```

LIMIT 子句分页查询

在 SQL 查询中，可以使用 `LIMIT` 子句来实现分页查询。

`LIMIT` 子句分页查询的使用格式如下：

```
LIMIT (page_no - 1) * page_size, page_size;
```

- `page_no`：表示第几页，从 1 开始，范围为 [1,+∞)。
- `page_size`：表示每页显示多少条记录，范围为 [1,+∞)。例如：`page_no = 5, page_size = 10`，表示获取第 5 页 10 条数据。

示例 ：在 `student` 表中，每页显示 2 条数据，依次获取第 1 页、第 2 页的数据。

第 1 页：

```
SELECT id, name
FROM student
ORDER BY id
LIMIT 0,2;
```

返回结果如下：

```
+----+---------+
| id | name    |
+----+---------+
|  1 | Emma    |
|  2 | William |
+----+---------+
```

```
2 rows in set
```

第 2 页:

```
SELECT id, name
FROM student
ORDER BY id
LIMIT 2,2;
```

返回结果如下:

```
+----+--------+
| id | name   |
+----+--------+
|  3 | Olivia |
|  4 | James  |
+----+--------+
2 rows in set
```

子查询

子查询是指嵌套在一个上层查询中的查询。上层的查询一般被称为父查询或外层查询。子查询的结果作为输入传递回“父查询”或“外部查询”。父查询将这个值结合到计算中，以便确定最后的输出。

SQL 语言允许多层嵌套查询，即一个子查询中还可以嵌套其他子查询。同时，子查询可以出现在 SQL 语句中的各种子句中，例如 SELECT 语句，FROM 语句，WHERE 语句等。

子查询相关的详细语法，请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/查询和子查询/子查询。

事务管理

数据库事务包含了数据库上的一系列操作，事务使得数据库从一个一致的状态转化到另一个一致的状态。

数据库事务有两个作用:

- 为数据库操作序列提供一个从失败中恢复到正常状态的方法，同时提供了数据库即使在异常

状态下仍能保持一致性的方法。

- 为数据库的多个并发访问提供隔离的方法，避免多个并发操作导致数据库进入一个不一致的状态。

基本事务控制语句如下：

- BEGIN：显式开启一个事务。此语句在使用过程中有如下两种情况。
  - 当租户会话的系统变量 `autocommit` 的值为 `0` 时，表示关闭事务自动提交功能，不需要显式发出 `BEGIN` 命令来标识多个 SQL 组成一个事务。

  - 当租户会话的系统变量 `autocommit` 的值为 `1` 时，表示开启事务自动提交功能，该模式下，每条 SQL 都是一个独立的事务。如果要多个 SQL 组成一个事务，可以通过 `BEGIN` 命令显式开启一个事务，同时会禁用事务自动提交功能，直到执行 `COMMIT` 或 `ROLLBACK` 语句后，才会恢复到自动提交模式。

- 当租户会话的系统变量 `autocommit` 的值为 `0` 时，表示关闭事务自动提交功能，不需要显式发出 `BEGIN` 命令来标识多个 SQL 组成一个事务。
- 当租户会话的系统变量 `autocommit` 的值为 `1` 时，表示开启事务自动提交功能，该模式下，每条 SQL 都是一个独立的事务。如果要多个 SQL 组成一个事务，可以通过 `BEGIN` 命令显式开启一个事务，同时会禁用事务自动提交功能，直到执行 `COMMIT` 或 `ROLLBACK` 语句后，才会恢复到自动提交模式。
- SAVEPOINT：在事务过程中标记一个"保存点"，事务可以事后选择回滚到这个点。保存点是可选的，一个事务过程中也可以有多个保存点。
- COMMIT：提交并结束当前事务，让事务所有修改持久化并生效，清除所有保存点和释放事务持有的锁。
- ROLLBACK：回滚整个事务已做的修改或者只回滚某个保存点之后事务已做的修改，清除回滚部分包含的所有保存点和释放事务持有的锁。

在 `obclient` 命令行环境下，可以在 SQL 提示符后发起事务控制命令，也可以通过修改 Session 级别的 `autocommit` 变量来控制事务是否自动提交。

- 通过 `SET autocommit` 设置变量时，当前会话立即生效，断开链接之后被设置的变量会失

效。

- 通过 `SET GLOBAL autocommit` 设置租户级别的变量时，需要断开链接才生效。
- 如果当前 Session 的 `autocommit` 值为 `0`，并且没有显式的提交事务，程序异常中断时，OceanBase 数据库会自动回滚最后一个未提交的事务。

开启事务

OceanBase 数据库 MySQL 模式的事务控制语句与 MySQL 数据库兼容，OceanBase 数据库的 MySQL 模式开启事务可以通过以下方式来完成：

- 执行 `BEGIN` 命令。

```
obclient [test]> BEGIN;       //开启事务
obclient [test]> INSERT INTO table1 VALUES(1,1);
obclient [test]> COMMIT;
```

- 执行 `START TRANSACTION` 命令。

```
obclient [test]> START TRANSACTION; //开启事务
obclient [test]> INSERT INTO table1 VALUES(1,1);
obclient [test]> COMMIT;
```

- 配置 `autocommit = 0`（关闭自动提交）后，执行 `INSERT`、`UPDATE`、`DELETE`、`SELECT FOR UPDATE` 语句时，系统会默认开启一个新事务。

```
obclient [test]> SET AUTOCOMMIT=0;
obclient [test]> INSERT INTO table1 VALUES(1,1);  //开启事务
obclient [test]> COMMIT;

obclient [test]> SET AUTOCOMMIT=0;
obclient [test]> UPDATE table1 SET id = 2 WHERE id = 1;  //开启事务
obclient [test]> COMMIT;

obclient [test]> SET AUTOCOMMIT=0;
obclient [test]> DELETE FROM table1 WHERE id = 2;  //开启事务
obclient [test]> COMMIT;

obclient [test]> SET AUTOCOMMIT=0;
obclient [test]> SELECT id FROM table1 WHERE id = 1 FOR UPDATE;  //开启事务
obclient [test]> COMMIT;
```

当事务开启时，OceanBase 数据库会为事务分配一个事务 ID，用于唯一的标识一个事务。

在实际使用中，多个并发的连接下，对同一个数据表进行操作时，可能会出现两个事务对同一行数据进行操作。对于查询的读操作，可以通过 `SELECT FOR UPDATE` 语句锁定查询结果，避免其他 DML 语句对该笔记录进行同时修改。

以下示例通过 `SET autocommit=0` 关闭自动提交功能后，再通过 `UPDATE` 语句开启一个事务。

```
obclient [test]> CREATE TABLE ordr(
id INT NOT NULL PRIMARY KEY,
name VARCHAR(10) NOT NULL,
value INT,
gmt_create DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP );
Query OK, 0 rows affected

obclient [test]>  INSERT INTO ordr(id, name, value)
VALUES (1,'CN',10001),(2,'US', 10002),(3,'EN', 10003);
Query OK, 3 rows affected
 Records: 3  Duplicates: 0  Warnings: 0

obclient [test]> SELECT * FROM ordr;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   | 10003 | 2022-10-19 14:51:12 |
+----+------+-------+---------------------+
2 rows in set

obclient [test]> SET autocommit=0;
Query OK, 0 rows affected

obclient [test]> UPDATE ordr SET id=4 WHERE name='US';
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0
```

执行成功后，可以使用 `oceanbase.V$OB_TRANSACTION_PARTICIPANTS` 查询活跃事务的参与者信息。

```
obclient [test]> SELECT * FROM oceanbase.V$OB_TRANSACTION_PARTICIPANTS;
```

输出如下：

```
*************************** 1. row ***************************
```

```
       TENANT_ID: 1002
          SVR_IP: xx.xx.xx.223
        SVR_PORT: 2882
      SESSION_ID: 3221487660
  SCHEDULER_ADDR: "xx.xx.xx.223:2882"
         TX_TYPE: UNDECIDED
           TX_ID: 110352
           LS_ID: 1001
    PARTICIPANTS: NULL
 CTX_CREATE_TIME: 2022-10-19 14:55:23.763474
 TX_EXPIRED_TIME: 2022-10-19 14:55:23.763474
           STATE: ACTIVE
          ACTION: START
PENDING_LOG_SIZE: 116
FLUSHED_LOG_SIZE: 0
            ROLE: LEADER
1 row in set
```

保存点

Savepoint 是 OceanBase 数据库提供的可以由用户定义的一个事务内的执行标记。用户可以通过在事务内定义若干标记并在需要时将事务恢复到指定标记时的状态。

当用户在执行过程中在定义了某个 Savepoint 之后执行了一些错误的操作，用户不需要回滚整个事务再重新执行，而是可以通过执行 ROLLBACK TO 命令来将 Savepoint 之后的修改回滚。

如下表所示，用户可以通过创建 Savepoint sp1 来对之后插入的数据执行回滚。

命令	解释
BEGIN;	开启事务
INSERT INTO a VALUE(1);	插入行 1
SAVEPOINT sp1;	创建名为 sp1的 Savepoint
INSERT INTO a VALUE(2);	插入行 2
SAVEPOINT sp2;	创建名为 sp2的 Savepoint
ROLLBACK TO sp1;	将修改回滚到 sp1
INSERT INTO a VALUE(3);	插入行 3
COMMIT;	提交事务

在 OceanBase 数据库的实现中，事务执行过程中的修改都有一个对应的“sql sequence”，该

值在事务执行过程中是递增的（不考虑并行执行的场景），创建 Savepoint 的操作实际上是将用户创建的 Savepoint 名字对应到事务执行的当前“sql sequence”上，当执行 `ROLLBACK TO` 命令时，OceanBase 数据库内部会执行以下操作：

1. 将事务内的所有大于该 Savepoint 对应“sql sequence”的修改全部回滚，并释放对应的行锁，例如示例中的行 2。
2. 删除该 Savepoint 之后创建的所有 Savepoint，例如示例中的 `sp2`。

`ROLLBACK TO` 命令执行成功后，事务仍然可以继续操作。

保存点相关的详细语法，请参见官网《OceanBase 数据库》文档 应用开发/基于 MySQL 模式进行应用开发/事务/保存点》章节。

提交事务

OceanBase 数据库提交事务可以使用显式提交或者隐式提交。显式的提交事务，需要使用 `COMMIT` 语句或者使用提交按钮（在图形化客户端工具中）结束事务；隐式提交时不需要主动提交，当变量 `autommit` 值为 `1` 时，此时每条语句执行结束后，OceanBase 数据库将会自动的把这条语句所在的事务提交，一条语句就是一个事务。

OceanBase 数据库会在 DDL 语句前和后隐式的发起一个 `COMMIT` 语句，也会提交事务。

隐式提交是用户未发出 `COMMIT`/`ROLLBACK` 等结束事务的语句给 OceanBase 数据库，而 OceanBase 数据库自动将当前活跃的事务执行 `COMMIT` 提交的过程。

隐式提交发生在以下情形：

- 执行一个开启事务的语句
- 执行 DDL

显式提交事务

通过 `BEGIN` 开始一个事务，然后使用 `INSERT` 语句在表 `ordr` 中插入数据，最后通过 `COMMIT` 语句显示提交事务。

```
obclient [test]> SELECT * FROM ordr;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
```

```
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   | 10003 | 2022-10-19 14:51:12 |
+----+------+-------+---------------------+
3 rows in set

obclient [test]> BEGIN;
Query OK, 0 rows affected

obclient [test]> INSERT INTO ordr(id,name) VALUES(4,'JP');
Query OK, 1 row affected

obclient [test]> COMMIT;
Query OK, 0 rows affected
```

执行成功后，退出会话再重新连接，查看表数据。

```
obclient [test]> SELECT * FROM ordr;
```

输出如下，表数据已正确插入并保存成功。

```
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   | 10003 | 2022-10-19 14:51:12 |
|  4 | JP   |  NULL | 2022-10-19 14:51:44 |
+----+------+-------+---------------------+
4 rows in set
```

隐式提交事务

通过配置变量 `autocommit=1`，开启自动提交。

```
obclient [test]> SELECT * FROM ordr;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   | 10003 | 2022-10-19 14:51:12 |
|  4 | JP   |  NULL | 2022-10-19 14:51:44 |
+----+------+-------+---------------------+
4 rows in set
```

```
obclient [test]> SET autocommit=1;

obclient [test]>  INSERT INTO ordr(id,name) VALUES(5,'CN');
Query OK, 1 row affected
```

执行成功后，退出会话再重新连接，查看表数据。

```
obclient [test]> SELECT * FROM ordr;
```

输出如下，表数据已正确插入并保存成功。

```
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   | 10003 | 2022-10-19 14:51:12 |
|  4 | JP   |  NULL | 2022-10-19 14:51:44 |
|  5 | CN   |  NULL | 2022-10-19 14:53:56 |
+----+------+-------+---------------------+
5 rows in set
```

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事务，也可以回滚到事务中任意一个保存点。

回滚整个事务后：

- 所有的修改会被丢弃。
- 所有保存点被清除。
- 事务持有的所有锁被释放。

回滚整个事务的语法格式如下：

```
ROLLBACK;
```

以下示例通过 `ROLLBACK` 回滚当前事务的全部修改。

```
obclient [test]> SELECT * FROM ordr;
```

```
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   | 10003 | 2022-10-19 14:51:12 |
|  4 | JP   |  NULL | 2022-10-19 14:51:44 |
|  5 | CN   |  NULL | 2022-10-19 14:53:56 |
+----+------+-------+---------------------+
5 rows in set

obclient [test]> BEGIN;
Query OK, 0 rows affected

obclient [test]> INSERT INTO ordr(id, name, value) VALUES(6,'JP',10007);
Query OK, 1 row affected

obclient [test]> INSERT INTO ordr(id, name, value) VALUES(8,'FR',10008),(9,'RU',10
009);
Query OK, 2 rows affected
Records: 2  Duplicates: 0  Warnings: 0

obclient [test]> SELECT * FROM ordr;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   |  1003 | 2022-10-19 14:51:12 |
|  4 | JP   |  NULL | 2022-10-19 14:51:44 |
|  5 | CN   |  NULL | 2022-10-19 14:53:56 |
|  6 | JP   | 10007 | 2022-10-19 14:58:24 |
|  8 | FR   | 10008 | 2022-10-19 14:58:35 |
|  9 | RU   | 10009 | 2022-10-19 14:58:35 |
+----+------+-------+---------------------+
8 rows in set

obclient [test]> ROLLBACK;
Query OK, 0 rows affected

obclient [test]> SELECT * FROM ordr;
+----+------+-------+---------------------+
| id | name | value | gmt_create          |
+----+------+-------+---------------------+
|  1 | CN   | 10001 | 2022-10-19 14:51:12 |
|  2 | US   | 10002 | 2022-10-19 14:51:12 |
|  3 | EN   |  1003 | 2022-10-19 14:51:12 |
|  4 | JP   |  NULL | 2022-10-19 14:51:44 |
|  5 | CN   |  NULL | 2022-10-19 14:53:56 |
+----+------+-------+---------------------+
```

```
5 rows in set
```

本示例中，通过 `BEGIN` 显示开启了一个事务，在未使用 `COMMIT` 显示提交事务前，所有修改只对当前会话可见，修改未持久化，可以使用 `ROLLBACK` 语句回滚修改。

自动回滚

自动回滚是用户未发出 ROLLBACK 指令，而是 OceanBase 数据库内部发起的回滚，通常发生在以下情形：

- session 断开
- 事务执行超时（ob_trx_timeout，事务超时时间，单位为微秒）
- 活跃事务的 Session 超过一定时长没有语句执行（ob_trx_idle_timeout，事务空闲超时时间，即事务中两条语句之间的执行间隔超过该值时超时，单位为微秒）

事务被中断

当事务执行过程中发生内部错误，如参与者节点宕机或者其它导致事务无法继续时，当前事务无法继续成功地执行语句，只能回滚。

此种情况发生时，用户执行 SQL 语句将会收到 `transaction need rollback` 的错误，用户需要执行 `ROLLBACK` 来结束当前事务。

事务隔离级别

隔离级别用于描述事务并发执行时互相干扰的程度。ANSI/ISO SQL 标准（SQL 92）基于事务执行过程中必须避免的异常情况定义了四种隔离级别，隔离级别越高，事务间的相互影响越小，允许出现的异常情况越少；在最高的隔离级别可串行化（Serializable）中，不允许出现任何异常情况。 其中这些需要避免的异常情况包括：

- 脏读（Dirty Read）：一个事务读到其他事务尚未提交的数据。
- 不可重复读（Non Repeatable Read）：曾经读到的某行数据，再次查询发现该行数据已经被修改或者删除。例如：`select c2 from test where c1=1;` 第一次查询 `c2` 的结果为 `1`，再次查询由于其他事务修改了 `c2` 的值，因此 `c2` 的结果为 `2`。

- 幻读（Phantom Read）：在执行请求中，当再次执行相同的搜索条件时，结果集中读取到了另一个已提交事务新插入的满足条件的行。

隔离级别	脏读	不可重复读	幻读
读未提交	可能	可能	可能
读已提交	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

OceanBase 数据库 MySQL 模式下目前支持了以下几种隔离级别:

- 读已提交（Read Committed）：一个事务执行的查询，只能看到这次查询开始之前提交的数据。读已提交无法防止不可重复读和幻读两种异常情况。
- 可重复读（Repeatable Read）：事务内不同时间读到的同一批数据是一致的。
- 可串行化（Serializable）：该隔离级别类似 Oracle 数据库的 Serializable，并非严格意义上的 Serializable。

OceanBase 数据库默认的隔离级别为 **读已提交（Read Committed）**。

实际在 OceanBase 数据库中只实现了两种隔离级别，即 **读已提交** 和 **可串行化**，当用户指定可重复读的隔离级别时，实际使用的是可串行化。也就是说，OceanBase 数据库的可重复读的隔离级别更加严格，不会出现幻读的异常情况。OceanBase 数据库的读已提交不会出现脏读的异常情况，可能会出现不可重复读和幻读的异常情况；而可串行化不会出现脏读、不可重复读以及幻读的异常情况。

OceanBase 数据库隔离级别与其他数据库对比

数据库	读未提交（Read Uncommitted）	读已提交（Read Committed）	可重复读（Repeatable Read）	可串行化（Serializable）
OceanBase	不支持语法	支持，与 SQL 标准一致	支持，且不存在幻读	支持，但不保证严格可串行化
MySQL	支持，有可能读到脏数据	支持，与 SQL 标准一致	支持，且不存在幻读	支持，可以保证严格可串行化

Oracle	不支持语法	支持，与 SQL 标准一致	不支持语法	支持，但不保证严格可串行化
PostgreSQL 9.1 之前版本	支持语法，但实际是 Read Committed	支持，与 SQL 标准一致	支持，且不存在幻读	支持，但不保证严格可串行化
PostgreSQL 9.1 及之后版本	支持语法，但实际是 Read Committed	支持，与 SQL 标准一致	支持，且不存在幻读	支持，可以保证严格可串行化

OceanBase 数据库 MySQL 模式与 MySQL 数据库事务隔离级别的具体差异如下：

- 读未提交：MySQL 数据库支持读未提交，而 OceanBase 数据库不支持读未提交。
- 读已提交：在读已提交隔离级别下判断行是否满足更新条件时，MySQL 数据库使用半一致性读，如果一行已经被并发事务更新，MySQL 数据库会等待并发事务的结束，然后在最新版本上判断是否需要更新；而在 OceanBase 数据库中，无论并发事务是否更新，总是基于语句快照中的版本来判断一行是否满足更新条件。
- 可重复读：在可重复读隔离级别下发生写冲突时，MySQL 数据库中后写的事务会等待先写的事务结束，如果先写的事务回滚，后写的事务则在原版本上直接更新，如果先写的事务提交，后写的事务会在新提交的版本上再进行更新；OceanBase 数据库中后写的事务会等待先写的事务结束，如果先写的事务回滚，后写的事务则在原版本上直接更新，如果先写的事务提交，后写的事务会回滚并返回错误。
- 可串行化：MySQL 数据库的可串行化隔离级别使用两阶段锁（2PL），能够保证严格的可串行化；OceanBase 数据库的可串行化隔离级别使用快照隔离（Snapshot Isolation），不能保证严格的可串行化。

读已提交

在 OceanBase 数据库的读已提交隔离级别下，每条 `SELECT` 语句执行时仅能够读到在此之前所有已经提交事务的数据，而不会读到在语句执行过程中新提交或被并发事务修改的数据，就像是在每条语句执行之前获取了当前数据库的最新快照。快照中只记录了已提交的数据，因此不会出现脏读的异象。但由于每条语句执行前会获取新的快照，在同一个事务中连续的两条 `SELECT` 语句有可能看到不同的数据，即读已提交隔离级别下不能避免不可重复读和幻读的异常情况。

对于 `UPDATE`、`DELETE`、`SELECT FOR UPDATE` 等更新操作，它们在搜索目标行时的行为与 `SELECT`

相同，即只能找到在语句开始前已经提交的行版本，如果该版本不满足更新操作的谓词条件，则直接跳过该行，如果满足条件，才尝试更新该行。然而，当前事务（简称事务 A）在尝试更新一个满足谓词条件的目标行时，该行可能已经被另一个并发事务（简称事务 B）更新。在这种情况下，若事务 B 还没有结束，事务 A 需要等待事务 B 的提交或回滚。如果事务 B 回滚，事务 A 可以继续更新目标行；如果事务 B 提交，事务 A 将重新执行该语句，从而重新获取语句快照，以读到事务 B 更新后的版本，如果该版本依旧满足谓词条件，则在此基础上再进行更新。

可重复读或可串行化

在 OceanBase 数据库的可串行化（或可重复读）隔离级别下，事务的第一条语句会获取当前数据库的快照作为事务快照，后续 `SELECT` 语句都会基于事务快照读取数据，只能读到在事务快照之前所有已经提交事务的数据，而不会读到在事务执行过程中新提交或被并发事务修改的数据。由于每条语句使用同一个事务快照，所以事务内总能看到一致的数据，不会出现不可重复读和幻读的异常情况。

对于 `UPDATE`、`DELETE`、`SELECT FOR UPDATE` 等更新操作，它们在搜索目标行时的行为与 `SELECT` 相同，即只能找到在获取事务快照前已经提交的行版本，如果该版本不满足更新操作的谓词条件，则直接跳过该行，如果满足条件，才尝试更新该行。然而，当前事务（简称事务 A）在搜索到一个目标行时，该行可能已经被另一个并发事务（简称事务 B）更新。在这种情况下，若事务 B 还没有结束，事务 A 需要等待事务 B 的提交或回滚。如果事务 B 回滚，事务 A 可以继续更新最初找到的行；如果事务 B 提交，事务 A 不能基于旧的快照进行更新，否则会出现丢失更新（Lost Update）的情况，因此事务 A 只能进行回滚，此时 OceanBase 数据库 MySQL 模式会返回下列错误信息：

```
ERROR 6235 (25000): can't serialize access for this transaction
```

说明

业务层需要考虑到事务可能会因为写冲突而回滚，并准备事务重试逻辑。如果事务复杂，重试代价较大，且业务并不要求事务所有语句看到一致的数据情况，建议使用读已提交的隔离级别。

隔离级别设置方法

设置隔离级别有两种方式，分别为 Global 级别和 Session 级别。

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL [READ COMMITTED | REPEATABLE RE
AD | SERIALIZABLE];
```

隔离级别使用限制

- 不允许在事务执行过程中设置隔离级别，否则会报如下错误：

```
ERROR 1568 (25001): Transaction characteristics can't be changed while a transacti
on is in progress
```

- Global 级别的隔离级别可以被 Session 级别的隔离级别覆盖。
- 可串行化隔离级别的限制：

  在 SQL 标准中，可串行化隔离级别只需要避免脏读、不可重复读和幻读的异常情况，但是可串行化的严格定义为：任何两个成功提交的并发事务按顺序执行，一个事务在另一个事务之后，即事务的并行执行结果需要和某种串行执行的结果相同。与 Oracle、PostgreSQL 9.0 及之前的版本类似，OceanBase 数据库的可串行化隔离级别下并不能保证严格的可串行化，即事务执行的结果可能不与任何串行执行模式的结果相同。其中写偏斜（Write Skew）是一个经典例子： 假设存在表 `T1(num int)` 和 `T2(num int)`，两个表初始都没有数据。此时在事务 1（`Trx1`）和事务 2（`Trx2`）中按照以下顺序执行命令：

```
Trx1                                           Trx2
BEGIN;
INSERT INTO T2 SELECT COUNT(*) FROM T1;

                                               BEGIN;
                                               INSERT INTO T1 SELECT COUNT(
*) FROM T2;
COMMIT;
                                               COMMIT:
```

  由于 `Trx1` 和 `Trx2` 获取的快照中两个表的 `COUNT` 都为 `0`，最终表 `T1` 和表 `T2` 中都会插入 `num=0` 的一行，而假设事务 1 和事务 2 串行执行，无论是 `Trx1->Trx2` 还是 `Trx2->Trx1`，最终两个表应该会分别插入 `num=0` 和 `num=1`。OceanBase 数据库目前无法保证严格可串行化的原因在于其读操作不会上锁，读写不互斥，并且也没有在事务提交时检查读写冲突是否成

环。在大部分现实应用场景下，只要不出现脏读、幻读和不可重复读就能够满足业务的需求，如果业务一定需要严格的可串行化，可以显式地为读操作加锁，如 `SELECT FOR UPDATE`。

弱一致性读

OceanBase 数据库提供了两种一致性级别（Consistency Level）：STRONG、WEAK。STRONG 指强一致性，读取最新数据，请求路由给主副本；WEAK 指弱一致性，不要求读取最新数据，请求优先路由给从副本。OceanBase 数据库的写操作始终是强一致性的，即始终由主副本提供服务；读操作默认是强一致性的，由主副本提供服务，用户也可以指定弱一致性读，由从副本优先提供服务。

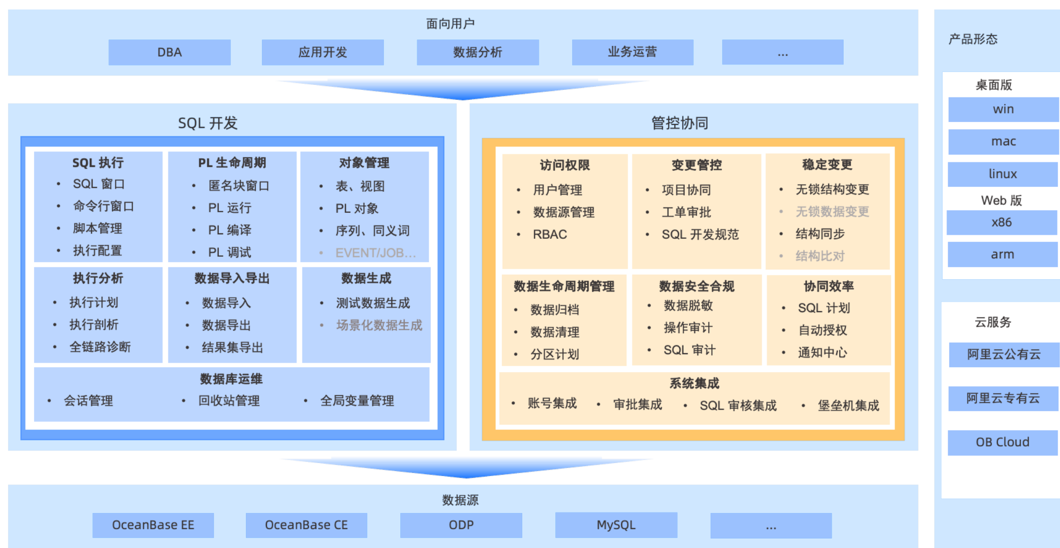
关于弱一致性的详细介绍，请参见官网《OceanBase 数据库》文档 参考指南/系统原理/事务管理/事务并发和一致性/弱一致性读。

6.2 通过 ODC 图形化开发工具进行 SQL 开发

ODC 介绍

OceanBase 开发者中心（OceanBase Developer Center，ODC）是数据库图形化开发工具，也是数据研发和生产变更管控协同平台。

产品功能架构



ODC 使用限制

数据源版本支持限制

ODC 支持的数据源类型	支持版本
OceanBase 数据库 MySQL 模式	OceanBase 数据库社区版
OB Sharding MySQL	ODP V3.2.8 及之后的版本
MySQL	MySQL 5.7
Oracle	Oracle 11g
Doris	Doris 2.0.0 及之后的版本

更多 ODC 的功能限制，请参见官网《OceanBase 开发者中心》文档 产品简介/使用限制。

ODC 部署说明

ODC 有桌面版、Web 版两种产品形态，而 Web 版 ODC 又支持单节点和高可用两种部署形态。建议用户使用 Web 版 ODC。

Web 版

ODC Web 版提供了个人空间和团队空间两种工作模式。个人空间适合个人开发者使用，为用户在浏览器端提供桌面版的体验，可以自由访问，新建数据源并使用平台提供的各种窗口、工具对数据库进行开发。团队空间适合开发者和 DBA 协同使用，既是开发工具也是管控协同平台，ODC 通过项目协同、稳定变更、数据安全、冷热数据分离等一系列功能，为用户提供数据库开发管控一站式服务。

单节点部署

Web 版 ODC 单节点部署方式详见官网《OceanBase 开发者中心》文档 部署指南/部署单节点 ODC。

高可用部署

Web 版 ODC 高可用部署方式详见官网《OceanBase 开发者中心》文档 部署指南/部署高可用 ODC。

检查部署状态

部署 Web 版 ODC 以后，可以在浏览器中通过宿主机（高可用部署为 Nginx 代理所在宿主机）的 IP 和端口号（`<http://IP:PORT>` 或 `<http://DOMAIN:PORT>`）访问 Web 版 ODC。

访问后展示登录界面即证明访问成功，若是初次访问请先注册账号后进行登录。若在 ODC 中创建连接成功，且在 ODC 中可对数据库进行正常操作，则证明 Web 版 ODC 部署成功。

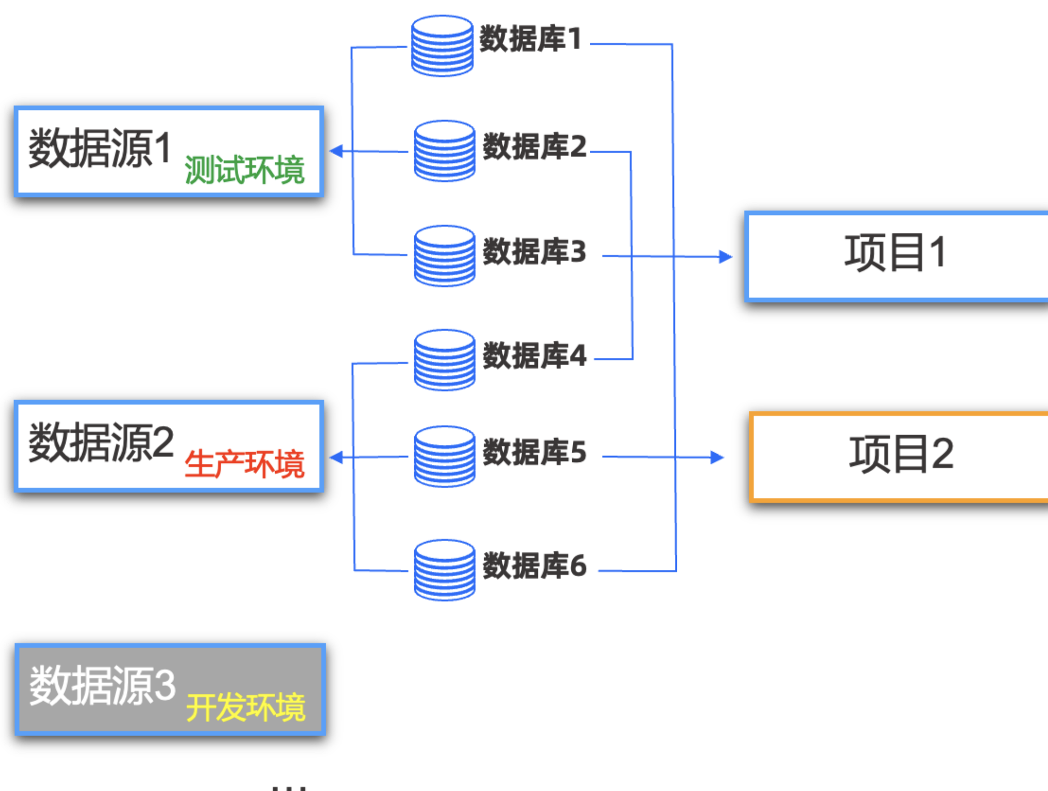
桌面版

ODC 桌面版侧重数据库开发工具能力，支持 Windows、Mac、Linux 操作系统，具有轻量化和易部署的特性。Web 版在提供工具能力的同时还提供了管控、协同能力，侧重数据库变更的安全、合规和效率。部署方式详见官网《OceanBase 开发者中心》文档 快速入门/桌面版 ODC/安装桌面版 ODC。

使用 ODC 进行 SQL 开发

数据源

数据源用于指代一个远端的数据库环境，比如一个独立部署的 OceanBase 集群或其他数据库系统实例。



配置

- 支持的数据源类型：OB-MySQL，OB-Oracle，OB-MySQL-Cloud，OB-Oracle-Cloud，ODP-Sharding-MySQL，MySQL。
- 主要可配置信息

  - 主机 IP：数据源所在的 IP 地址或域名。

  - 端口：数据源所在的端口号。

  - 数据库用户名：数据库的用户名。

  - 数据库密码：数据库用户的密码。

- 主机 IP：数据源所在的 IP 地址或域名。
- 端口：数据源所在的端口号。
- 数据库用户名：数据库的用户名。
- 数据库密码：数据库用户的密码。
- OceanBase 数据库独有选项：集群名，租户名。
- OceanBase-Cloud 特殊之处：不含有集群名，租户名，主机 IP 必须填数据源所在域名。
- 环境：用于配置该数据源所属的运行环境，如开发环境，生产环境等，用于配置不同的管控策略。

新建数据源

数据源类型： OceanBase Oracle

智能解析 （可选）

粘贴连接串信息，自动识别连接信息，如：obclient -h 10.210.2.51 -P2883 -uroot@tenantname#clustername -p'oBpasswORd'

智能解析

连接地址

主机 IP/域名 | 端口
请输入主机地址 | 请输入端口

集群名 （可选） | 租户名
请输入集群名 | 请输入租户名

数据库账号

数据库用户名 | 数据库密码
请输入数据库用户名 | 请输入密码

测试连接

环境

高级设置

取消 确定

使用

管理数据库

- 入口：左侧导航栏 -> 数据源 -> 数据源xxx -> 数据库。
- 功能：
    - 新建数据库。
    - 分配数据库到项目。
- 新建数据库。
- 分配数据库到项目。

数据库　会话　回收站　命令行窗口

新建数据库　同步数据库

数据库名称	字符集
SHANLU_TEST	ZHS
SHANLU	ZHS
ODC_UPFW56V0230LOCAL_856180	ZHS

管理数据源会话

- 入口：左侧导航栏 -> 数据源 -> 数据源xxx -> 会话。
- 功能：
    - 查看所有该数据源下的会话。

    - 关闭某个会话。

    - 关闭某个会话上正在运行的查询。

- 查看所有该数据源下的会话。
- 关闭某个会话。
- 关闭某个会话上正在运行的查询。

数据库　会话　回收站　命令行窗口

关闭会话　关闭查询

会话 ID	用户	来源	数据库名	状态	命令
3221511193	SYS		SYS	SLEEP	Sleep
3221511191	SYS		SYS	ACTIVE	Query
3221511021	SYS		SHANLU	SLEEP	Sleep
3221508412	SYS		SHANLU	SLEEP	Sleep

回收站

- 入口：左侧导航栏 -> 数据源 -> 数据源xxx -> 回收站。
- 功能：
    - 还原已经删除的表，视图等数据库对象。

    - 清空回收站，删除回收站中某个特定对象。

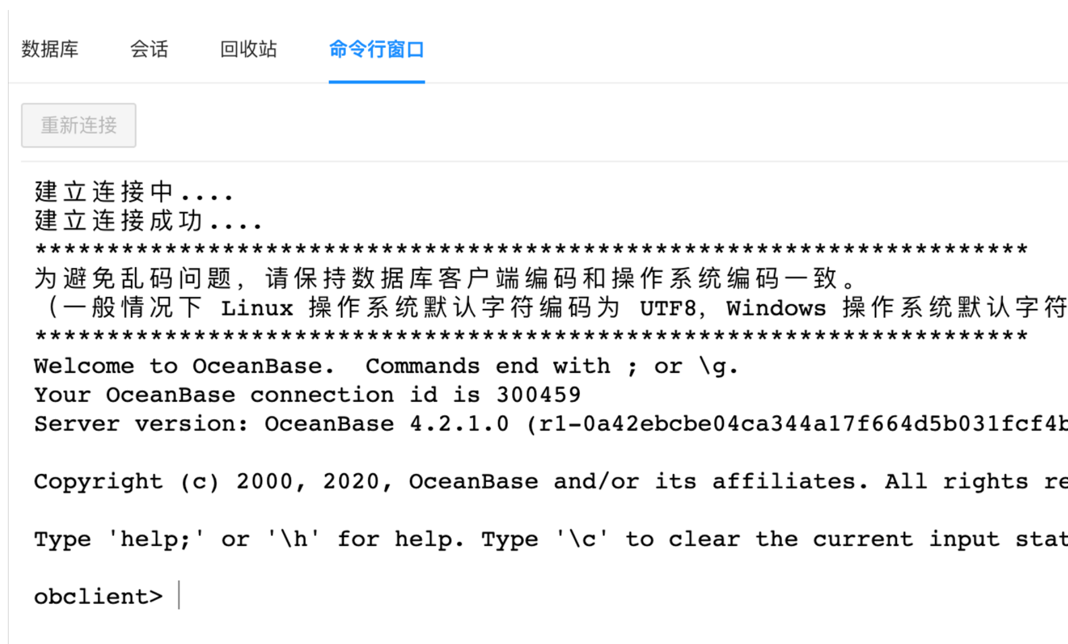
- 还原已经删除的表，视图等数据库对象。
- 清空回收站，删除回收站中某个特定对象。

注意

该功能是租户级别的，谨慎开启。

命令行窗口

- 入口：左侧导航栏 -> 数据源 -> 数据源xxx -> 命令行窗口。
- 功能：通过 obclient 的方式连接到目标数据源。

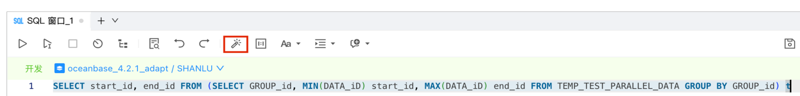


SQL 开发

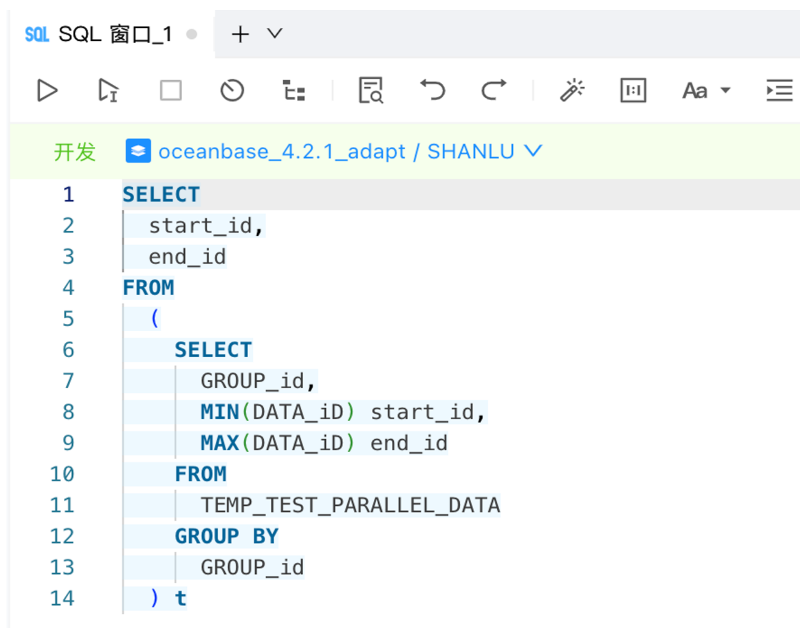
SQL 编辑与执行

- 语法高亮与美化
  - 语法高亮，关键字高亮，一眼就能抓住关键信息。
  - SQL 美化：
    - 根据语法结构格式化代码，让用户的 SQL 赏心悦目。
    - 根据语法结构展开，让用户的 SQL 结构一目了然，更易于理解。

美化前：

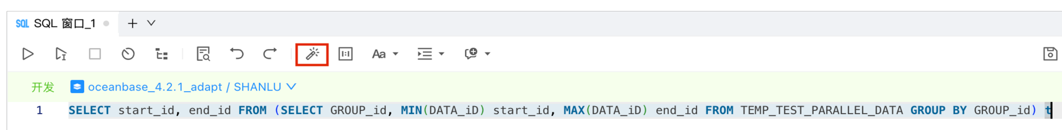


美化后：

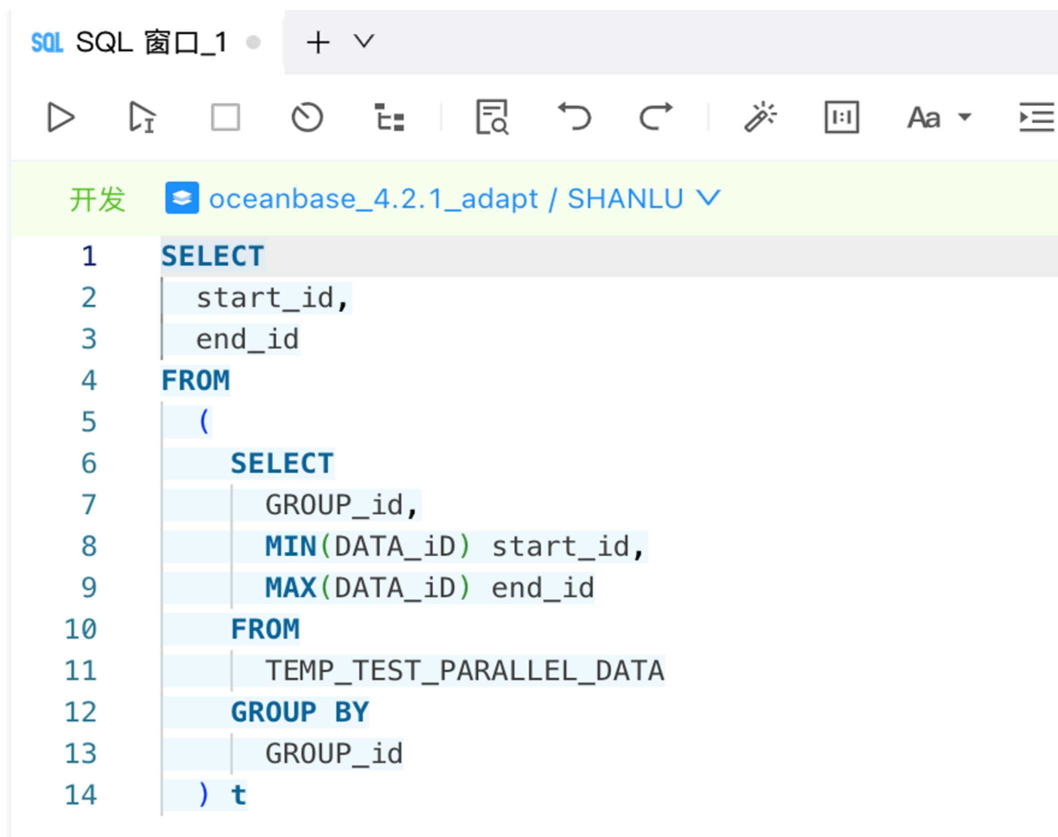


- 语法高亮，关键字高亮，一眼就能抓住关键信息。
- SQL 美化：
  - 根据语法结构格式化代码，让用户的 SQL 赏心悦目。
  - 根据语法结构展开，让用户的 SQL 结构一目了然，更易于理解。

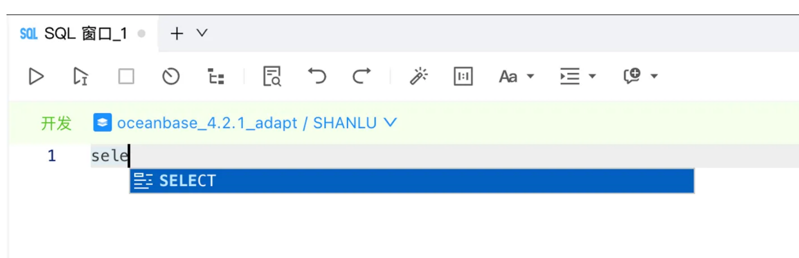
美化前：



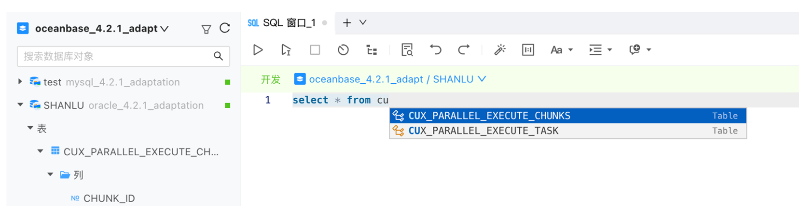
美化后：



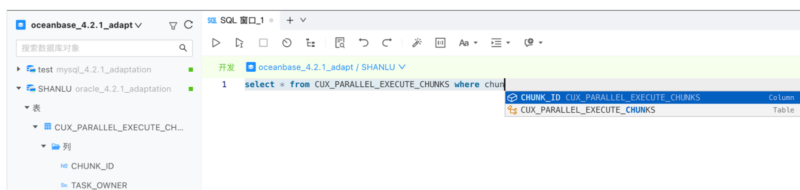
- 根据语法结构格式化代码，让用户的 SQL 赏心悦目。
- 根据语法结构展开，让用户的 SQL 结构一目了然，更易于理解。
- 语法提示
    - 关键字提示，让用户不再为拼错关键字而烦恼。



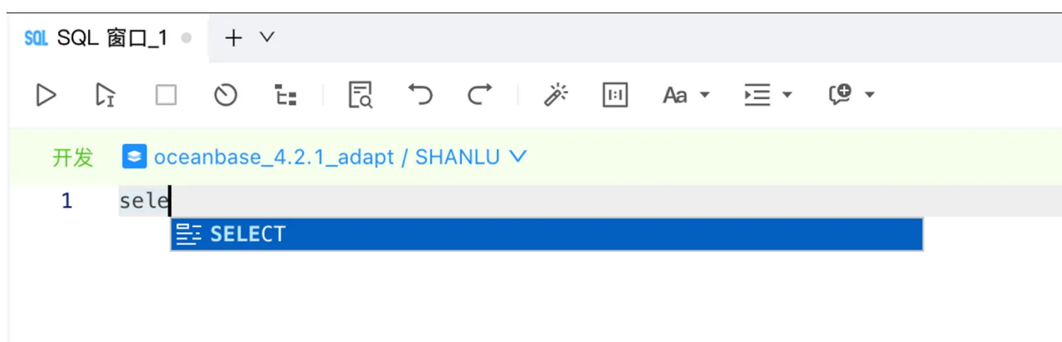
    - 数据库对象名提示，让用户在茫茫数据库对象列表中快速匹配到目标。



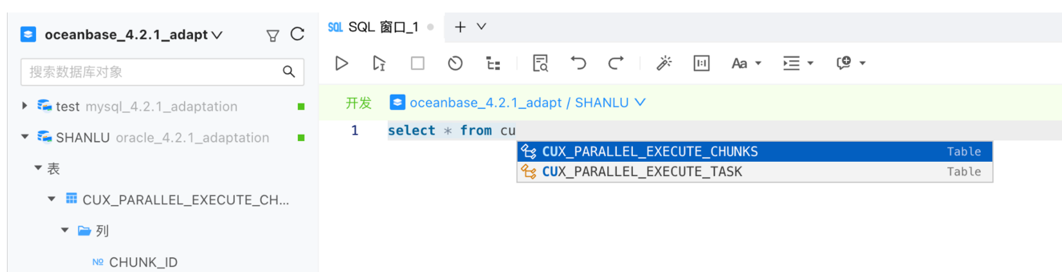
- 列名提示，智能列名提示，能根据用户当前操作的表智能给出列建议。



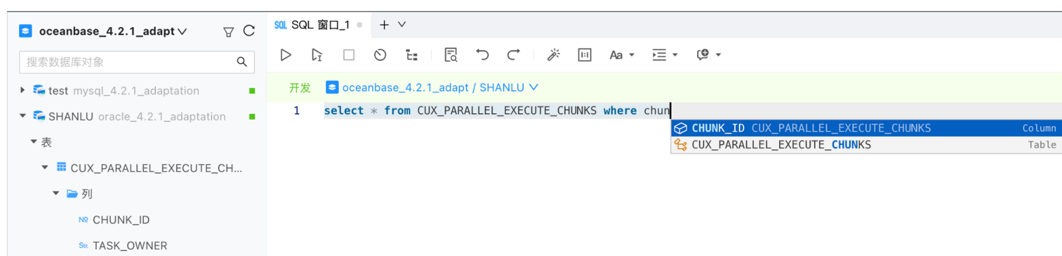
- 关键字提示，让用户不再为拼错关键字而烦恼。



- 数据库对象名提示，让用户在茫茫数据库对象列表中快速匹配到目标。



- 列名提示，智能列名提示，能根据用户当前操作的表智能给出列建议。



- SQL 检查
    - 规则数量多，覆盖面广。
    - 规则来自 OceanBase 数据库最佳实践，参考价值高。

- 检测方便，一键点击。
- 风险等级智能提示，帮助您更好规避风险。

内置检查规则：

规则名称	规则类型	支持数据源	配置值	改进等级	是否启用	操作
不能对列设置字符集	ALTER,DDL,TABLE	OB_MYSQL,OB_OR...	-	无需改进		编辑
不能对列设置排序规则	ALTER,DDL,TABLE	OB_MYSQL,OB_OR...	-	无需改进		编辑
限制列不可空（NOT NULL）	ALTER,DDL,TABLE	OB_MYSQL,OB_OR...	-	无需改进		编辑
列名不能处于黑名单	ALTER,DDL,TABLE	OB_MYSQL,OB_OR...	-	无需改进		编辑
表不能使用外键	ALTER,DDL,TABLE	OB_MYSQL,OB_OR...	-	需要审批		编辑
列条件上存在计算	DQL,SELECT	OB_MYSQL,OB_OR...	-	无需改进		编辑
LIKE 运算存在左模糊匹配	DELETE,DML,DQL,S...	OB_MYSQL,OB_OR...	-	无需改进		编辑
表缺少必要的列	DDL,TABLE	OB_MYSQL,OB_OR...	-	无需改进		编辑
列要有注释	ALTER,DDL,TABLE	OB_MYSQL,OB_OR...	-	无需改进		编辑

共 45 条　< 1 2 3 4 5 >

执行 SQL 检查：

SQL 窗口_1

开发　oceanbase_4.2.1_adapt / SHANLU

SQL 检查

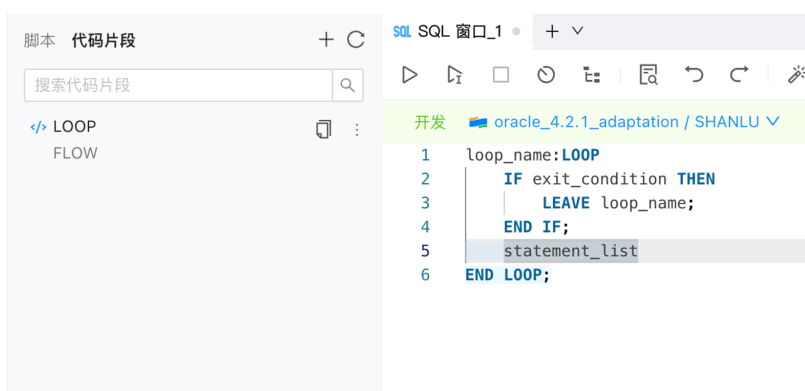
```sql
create table test_tbl(
    id integer,
    name varchar2(64)
)
```

执行记录　问题

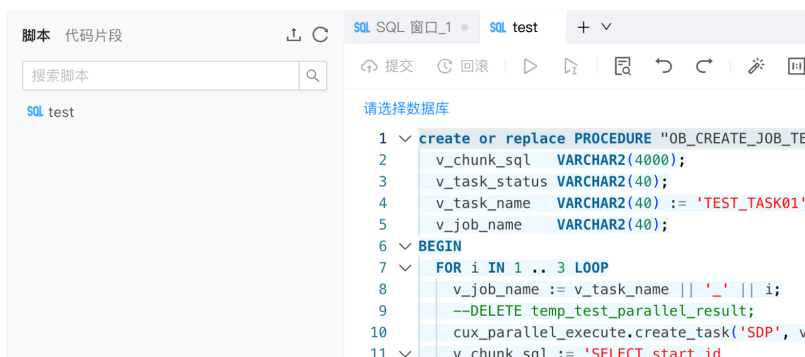
create table test_tbl(id integer, name varchar2(64))

必须改进	表要有主键
需要审批	表对象 test_tbl 必须含有注释
无需改进	类型为 integer 的列不允许为空，允许为空的列的数据类型：N/A
无需改进	类型为 varchar2(64) 的列不允许为空，允许为空的列的数据类型：N/A
无需改进	数据类型为 integer 的列不能没有默认值，允许没有默认值的列类型：N/A
无需改进	数据类型为 varchar2(64) 的列不能没有默认值，允许没有默认值的列类型：N/A
无需改进	列 id 没有注释
无需改进	列 name 没有注释

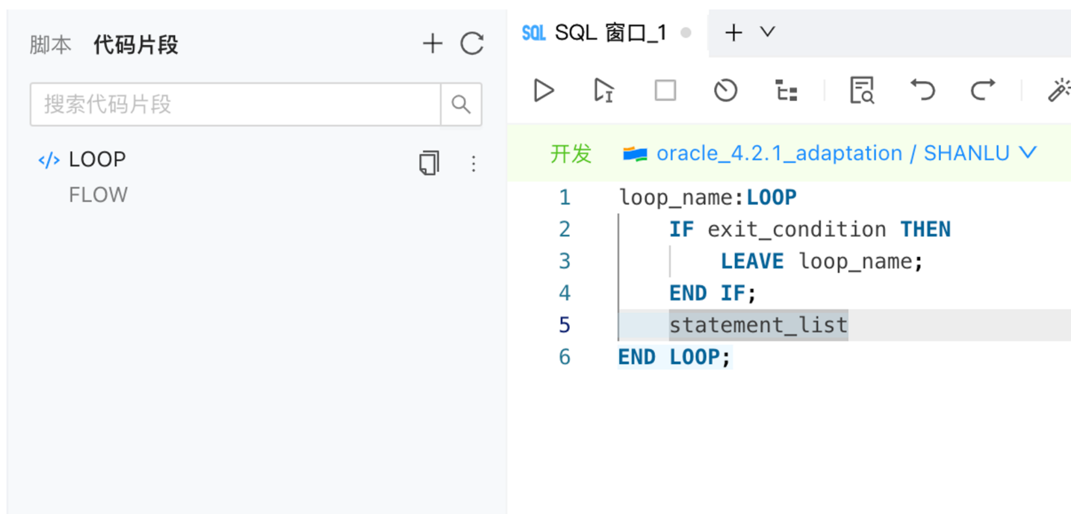
- 规则数量多，覆盖面广。
- 规则来自 OceanBase 数据库最佳实践，参考价值高。
- 检测方便，一键点击。
- 风险等级智能提示，帮助您更好规避风险。
- SQL 脚本
  - 代码片段
    - 保存常用的语法块，例如循环控制，分支控制。
    - 拖动唤起，操作方便快捷，省去记忆诸多 SQL/PL 语法结构的苦恼。



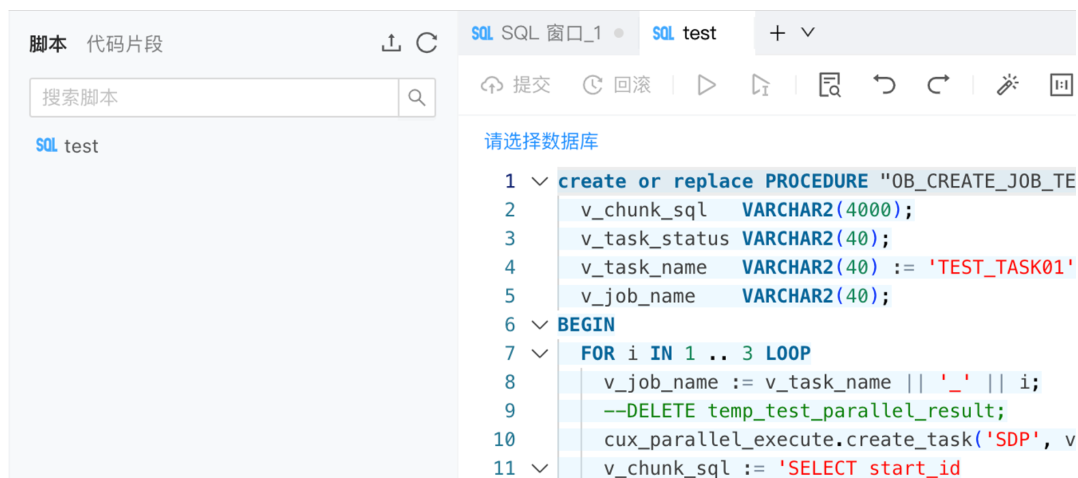
  - SQL 脚本
    - 保存常用的 SQL 脚本，例如常用的匿名块，常用的查询逻辑。
    - 检索高效，帮用户管理好众多常用的 SQL 脚本。



- 代码片段
  - 保存常用的语法块，例如循环控制，分支控制。
  - 拖动唤起，操作方便快捷，省去记忆诸多 SQL/PL 语法结构的苦恼。



- 保存常用的语法块，例如循环控制，分支控制。
- 拖动唤起，操作方便快捷，省去记忆诸多 SQL/PL 语法结构的苦恼。
- SQL 脚本
  - 保存常用的 SQL 脚本，例如常用的匿名块，常用的查询逻辑。
  - 检索高效，帮用户管理好众多常用的 SQL 脚本。



- 保存常用的 SQL 脚本，例如常用的匿名块，常用的查询逻辑。
- 检索高效，帮用户管理好众多常用的 SQL 脚本。

结果集

- 结果集展示
    - 行模式展示数据，总揽全局。

SQL 窗口_1

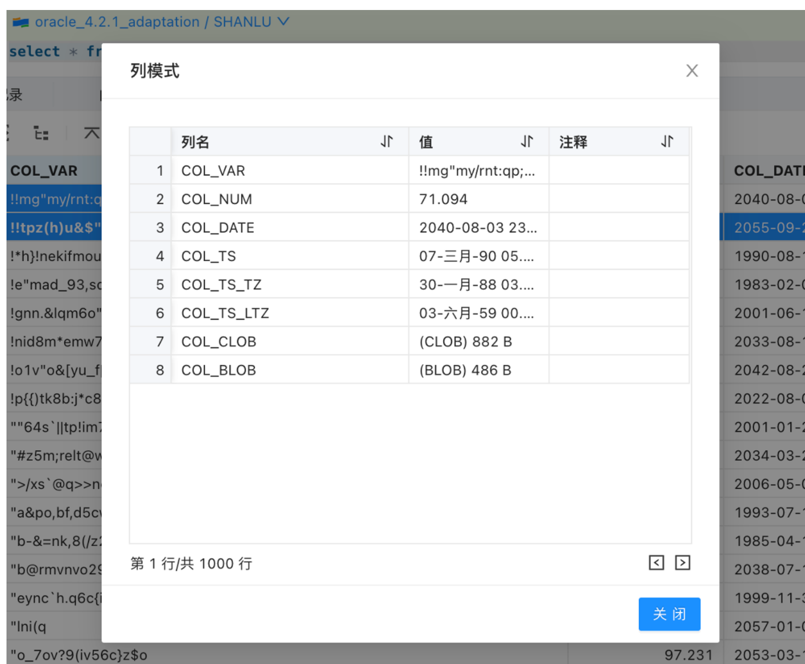
开发　oracle_4.2.1_adaptation / SHANLU

```sql
1 select * from show_tbl;
```

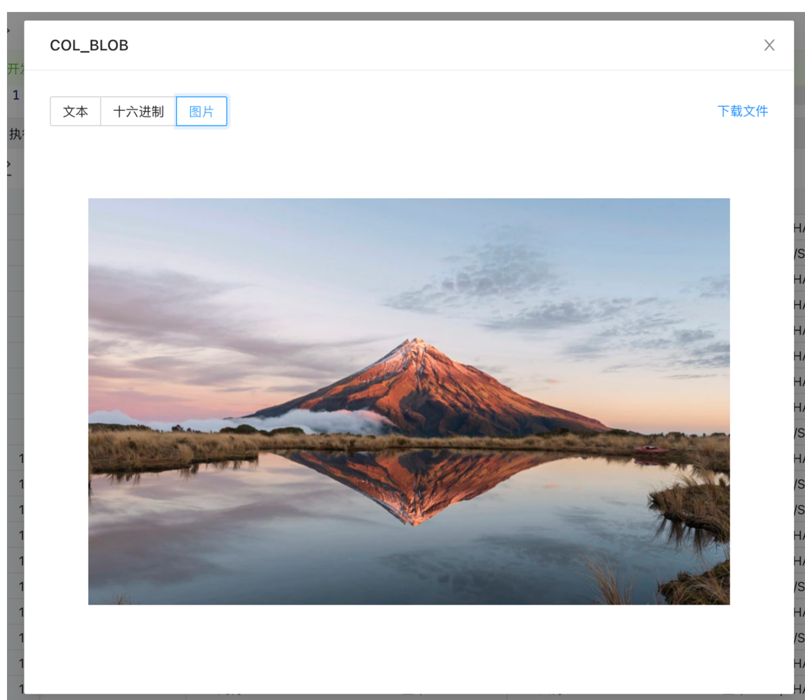
执行记录　问题　日志　结果1

		COL_TS	COL_TS_TZ
1	23:39:27	07-三月-90 05.46.19000001000 ...	30-一月-88 03.17.19000000000 上午 GMT+08:...
2	22:41:34	16-一月-87 08.31.44000001000 ...	11-二月-84 07.32.19000001000 上午 GMT+08:...
3	04:46:34	06-四月-57 02.49.47000000000 ...	21-三月-90 03.55.38000001000 上午 GMT+08:...
4	14:18:22	25-五月-01 10.54.24000000000 ...	28-一月-47 01.07.39000001000 上午 GMT+08:...
5	10:17:42	01-十二月-50 05.42.35000000000...	10-五月-40 01.00.25000000000 上午 GMT+08:...
6	22:38:15	01-十一月-44 01.18.07000001000...	25-二月-86 05.50.42000000000 上午 GMT+08:...
7	21:25:42	02-三月-34 07.30.09000001000 ...	22-七月-48 10.56.14000000000 上午 GMT+08:...
8	20:14:06	22-九月-46 09.58.09000000000 ...	16-七月-02 06.17.41000001000 下午 GMT+08:...
9	22:35:35	14-一月-87 00.16.14000001000 ...	17-七月-14 02.11.59000001000 下午 GMT+08:...

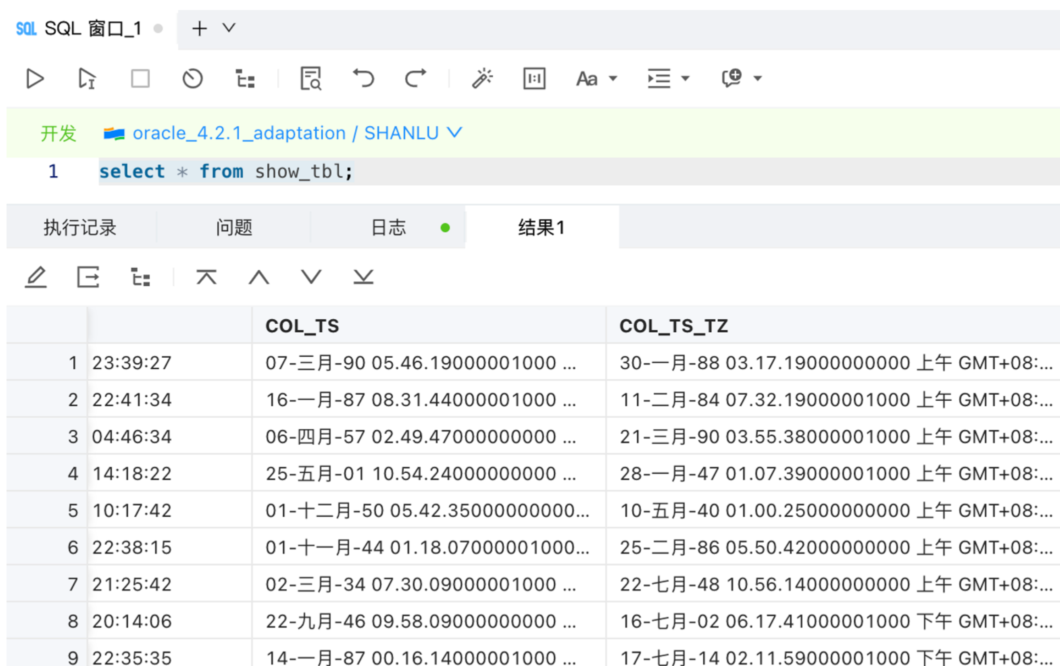
    - 列模式展示数据，让用户聚焦细节。



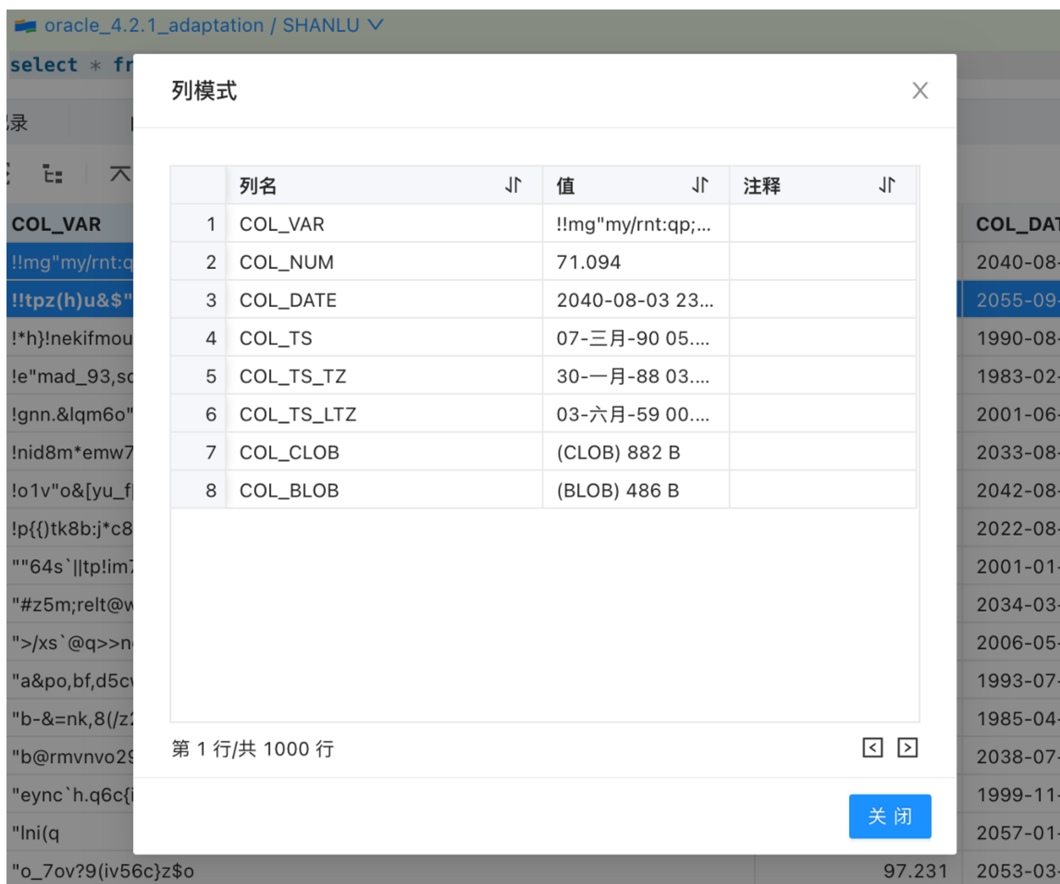
• 大字段数据可选择以文本，十六进制，图片模式展示，让用户还原数据真实面貌。



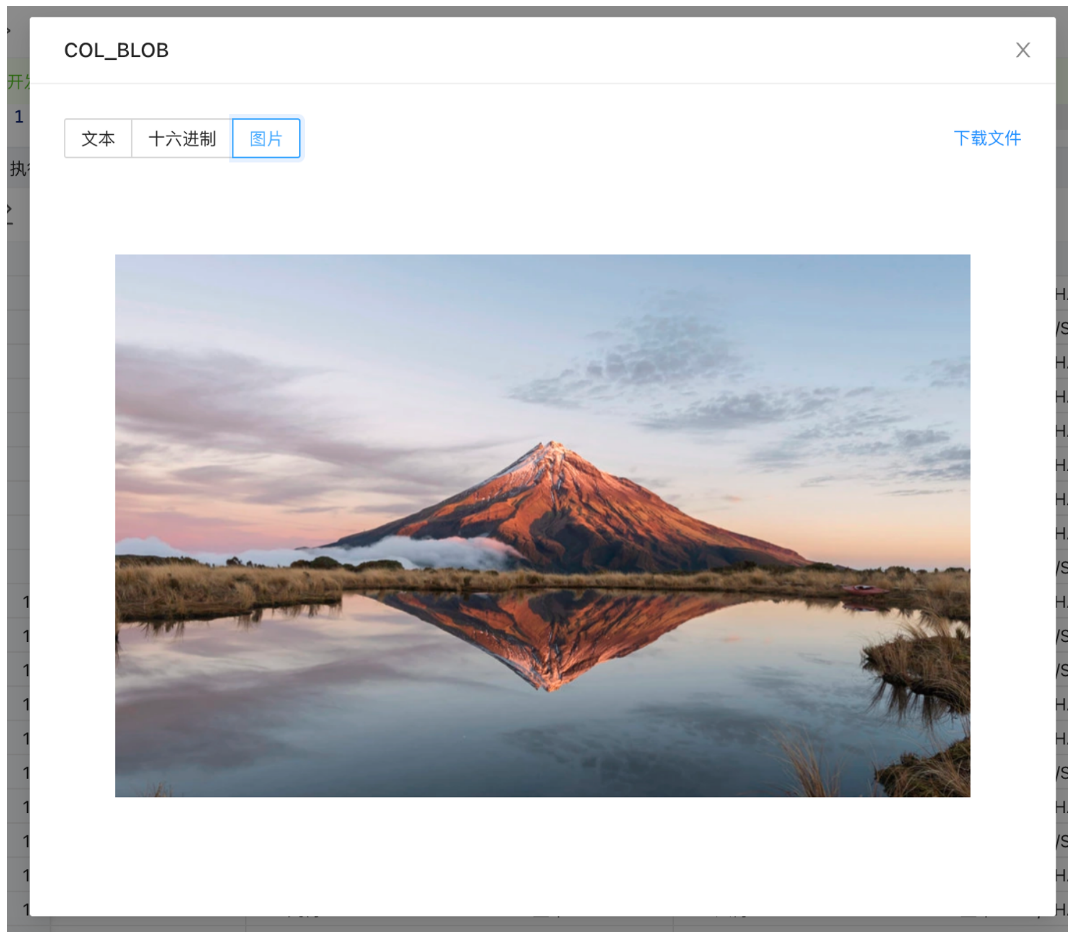
• 行模式展示数据，总揽全局。



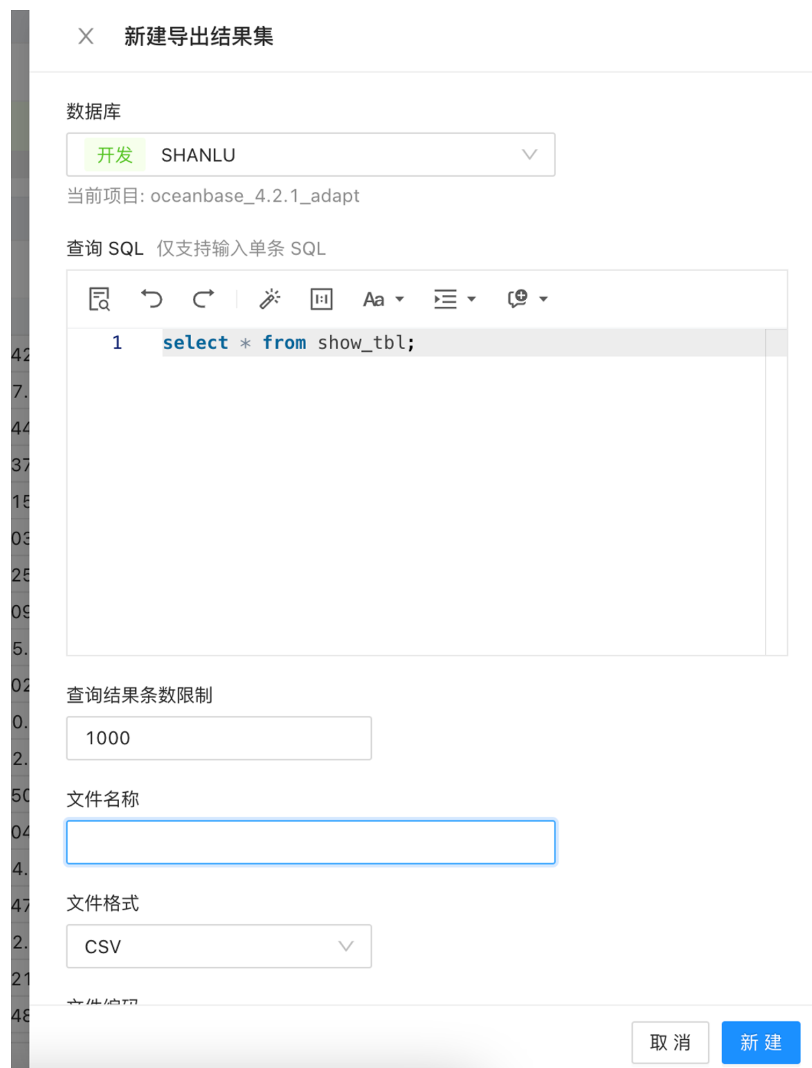
• 列模式展示数据，让用户聚焦细节。



• 大字段数据可选择以文本，十六进制，图片模式展示，让用户还原数据真实面貌。



- 结果集导出与编辑
  - 结果集编辑
    - 编辑选定行。
    - 不同数据类型定制不同的修改方式。
    - 支持多行修改，一键提交。
  - 结果集导出
    - 导出 SQL 执行结果。支持多种格式：CSV，EXCEL，SQL。
    - 支持多种编码格式。



- 结果集编辑
  - 编辑选定行。
  - 不同数据类型定制不同的修改方式。
  - 支持多行修改，一键提交。
- 编辑选定行。
- 不同数据类型定制不同的修改方式。
- 支持多行修改，一键提交。
- 结果集导出

    - 导出 SQL 执行结果。支持多种格式：CSV，EXCEL，SQL。

    - 支持多种编码格式。

- 导出 SQL 执行结果。支持多种格式：CSV，EXCEL，SQL。
- 支持多种编码格式。

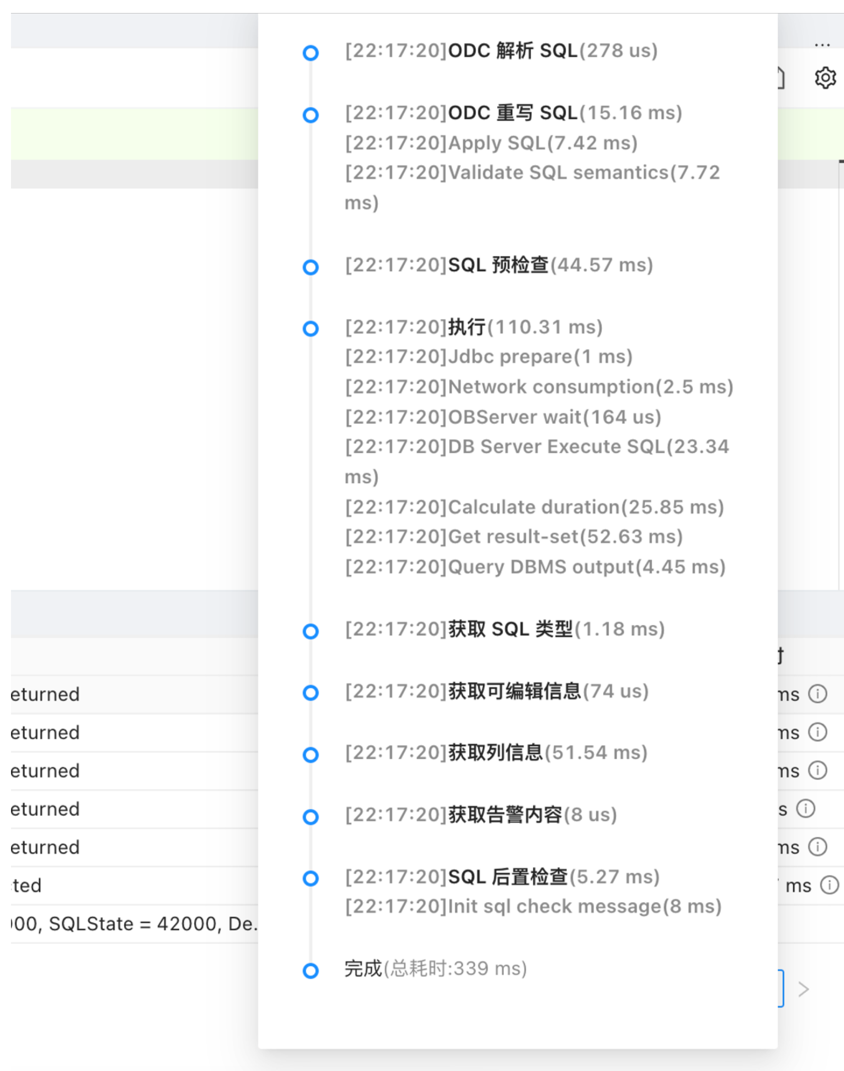
SQL 执行详情与执行计划

- 执行耗时统计
    - 对每一条执行的 SQL 进行耗时统计。

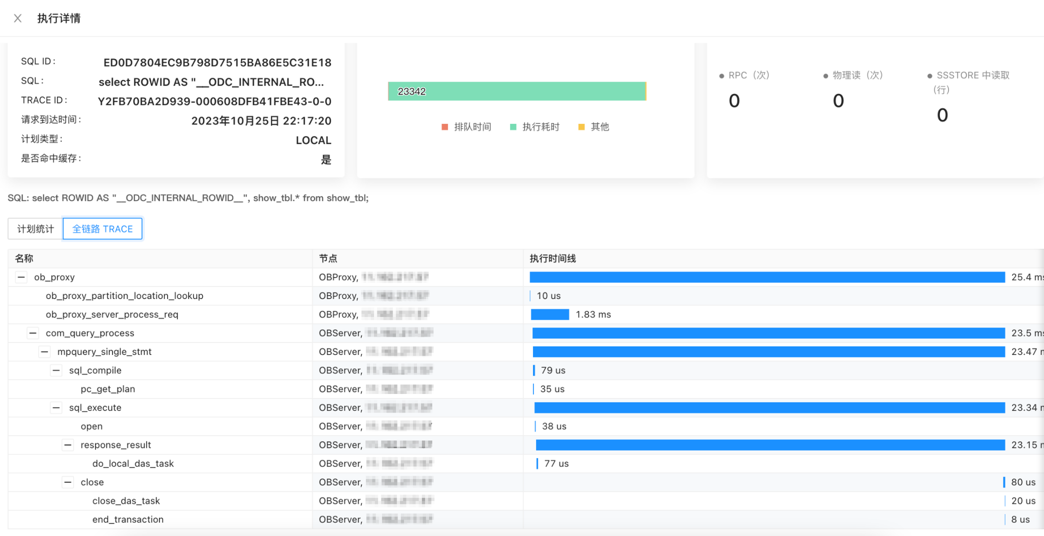
    - 分段统计，让用户知道 ODC 执行 SQL 经历了哪些阶段。

    - 内核执行耗时也纳入进来，让用户不必查 SQL-Audit 就能知道具体耗时。

    - 记录每条 SQL 的 trace-id，排查问题更简单。



- 对每一条执行的 SQL 进行耗时统计。
- 分段统计，让用户知道 ODC 执行 SQL 经历了哪些阶段。
- 内核执行耗时也纳入进来，让用户不必查 SQL-Audit 就能知道具体耗时。
- 记录每条 SQL 的 trace-id，排查问题更简单。
- 执行详情
    - 展示 SQL 的 trace-id，耗时，执行计划等信息。
    - 全链路诊断，全方位展示 driver，ob-proxy，observer 整条链路上 SQL 执行的全过程。



- 展示 SQL 的 trace-id，耗时，执行计划等信息。
- 全链路诊断，全方位展示 driver，ob-proxy，observer 整条链路上 SQL 执行的全过程。

数据库对象

ODC 支持管理所有 OceanBase 数据库支持的数据库对象，如表、视图、函数、存储过程、序列、程序包、触发器、同义词等，详细的图形化管理对象操作，请参见官网《OceanBase 开发者中心》文档 SQL 开发/数据库对象 章节。

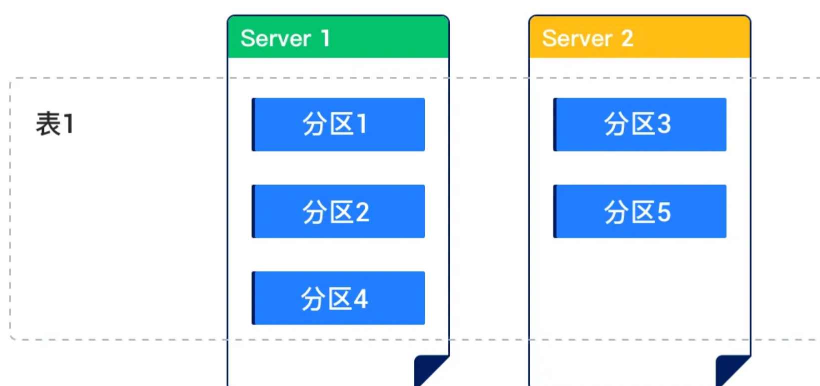
更为详细的 SQL 开发指南可参见官网《OceanBase 开发者中心》文档 SQL 开发 章节。

6.3 使用 OceanBase 数据库分区表进行水平拆分

分区表

功能定义

在 OceanBase 数据库中，分区是指根据一定的规则，把一个表分解成多个更小的、更容易管理的部分。每个分区都是一个独立的对象，具有自己的名称和可选的存储特性。如下图所示，一张表被划分成了 5 个分区，分布在 2 台机器上：



对于访问数据库的应用而言，在逻辑上的访问只有一个表或一个索引，但是实际上这个表可能由多个物理分区对象组成，每个分区都是一个独立的对象，可以独自处理访问，也可以作为表的一部分处理访问。分区对应用来说是完全透明的，不影响应用的业务逻辑。

从应用程序的角度来看，只存在一个 Schema 对象。访问分区表不需要修改 SQL 语句。分区对于许多不同类型的数据库应用程序非常有用，尤其是那些管理大量数据的应用程序。

业务拆分简介

很多业务误以为用上分布式数据库后，性能就一定会很好，或者扩容后业务的性能也能相应的提升；但事与愿违，实际使用经验并不一定如此。在使用了分布式数据库之后，业务的性能或者说扩展性是否能够提升，很大程度上取决于业务上是否能够拆分，让数据分区分布在更多的节点上，利用上更多节点的能力。

分布式数据库的优势在于对于空间问题和请求访问问题分而治之。针对每个分区的访问，由该分区所在的节点响应即可。即使该 SQL 并发很高，由于访问的是不同的分区，分别由不同的节点提供服务。每个节点自身也有一定能力满足一定的 QPS，所有节点集中在一起就能提供更大的 QPS。这个时候如果扩容节点数量，该 SQL 总的 QPS 也能获得相应的提升，这是分布式数据库里最好的情形。

分区的目标是将大量数据和访问请求均匀分布在多个节点上。如果每个节点均匀承担数据和请求，那么理论上 10 个节点就应该能承担 10 倍于单节点的数据量和访问量。然而如果分区是不均匀的，一些分区的数据量或者请求量会相对比较高，出现数据偏斜（skew），这个可能导致节点资源利用率和负载也不均衡。偏斜集中的数据我们又称为热点数据。避免热点数据的直接方法就是数据存储时随机分配（没有规则）给节点，缺点是读取的时候不知道去哪个分区找该记录，只有扫描所有分区了，所以这个方法意义不大。实际常用的分区策略都是有一定的规则。这个规则可以是业务规则，也可以不是。

使用分区的好处如下:

- 提高了可用性

  分区不可用并不意味着对象不可用。查询优化器自动从查询计划中删除未引用的分区。因此，当分区不可用时，查询不受影响。

- 更轻松地管理对象

  分区对象具有可以集体或单独管理的片段。DDL 语句可以操作分区而不是整个表或索引。因此，可以对重建索引或表等资源密集型任务进行分解。例如，可以一次只移动一个分区。如果出现问题，只需要重做分区移动，而不是表移动。此外，对分区进行 TRUNCATE 操作可以避免大量数据被 DELETE。

- 减少 OLTP 系统中共享资源的争用

  在 TP 场景中，分区可以减少共享资源的争用。例如，DML 分布在许多分区而不是一个表上。

- 增强数据仓库中的查询性能

  在 AP 场景中，分区可以加快即时查询的处理速度。分区键有天然的过滤功能。例如，查询一个季度的销售数据，当销售数据按照销售时间进行分区时，仅仅需要查询一个分区或者几个分区，而不是整个表。

- 提供更好的负载均衡效果

  OceanBase 数据库的存储单位和负载均衡单位都是分区。不同的分区可以存储在不同的节点。因此，一个分区表可以将不同的分区分布在不同的节点，这样可以将一个表的数据比较均匀的分布在整个集群。

分区表注意事项

- 关于分区表创建时的注意事项。
  - 如果数据量很大并且访问比较集中时，可以在创建表时使用分区表。一般千万级别以上数据量就可以考虑分区，分区数量上限为 8192 个。

  - 分区表约束注意事项。
    - 建分区表时，表上的每一个主键、唯一键所对应的字段里都必须至少有一个字段包含在表的分区键字段中。如果该表没有主键或者唯一键，分区键可以是任意一个字段。

    - 分区表中的全局唯一性建议能通过主键实现的都通过主键实现。

    - 分区表的唯一索引必须包含表分区的拆分键。

- 如果数据量很大并且访问比较集中时，可以在创建表时使用分区表。一般千万级别以上数据量就可以考虑分区，分区数量上限为 8192 个。
- 分区表约束注意事项。
  - 建分区表时，表上的每一个主键、唯一键所对应的字段里都必须至少有一个字段包含在表的分区键字段中。如果该表没有主键或者唯一键，分区键可以是任意一个字段。

  - 分区表中的全局唯一性建议能通过主键实现的都通过主键实现。

  - 分区表的唯一索引必须包含表分区的拆分键。

- 建分区表时，表上的每一个主键、唯一键所对应的字段里都必须至少有一个字段包含在表的分区键字段中。如果该表没有主键或者唯一键，分区键可以是任意一个字段。
- 分区表中的全局唯一性建议能通过主键实现的都通过主键实现。
- 分区表的唯一索引必须包含表分区的拆分键。
- 关于分区策略，推荐从表的实际用途和应用场景方面进行设计。
    - 实际用途：历史表，流水表。
    - 应用场景：存在明显访问热点的表。
- 实际用途：历史表，流水表。
- 应用场景：存在明显访问热点的表。
- 关于分区键的选择，使用分区表时要选择合适的拆分键以及拆分策略。
    - `HASH` 分区：选择区分度较大、在查询条件中出现频率最高的字段作为 `HASH` 分区的分区键。
    - `RANGE` 和 `LIST` 分区：根据业务规则选择合适的字段作为分区键，但分区数量不宜过少。示例：如果是日志类型的大表，根据时间类型的列做 `RANGE` 分区。
    - `RANGE` 分区：最后一列不建议设成 `MAXVALUE`。
    - `HASH` 分区下，不适合基于分区字段进行范围查询。
- `HASH` 分区：选择区分度较大、在查询条件中出现频率最高的字段作为 `HASH` 分区的分区键。
- `RANGE` 和 `LIST` 分区：根据业务规则选择合适的字段作为分区键，但分区数量不宜过少。示例：如果是日志类型的大表，根据时间类型的列做 `RANGE` 分区。
- `RANGE` 分区：最后一列不建议设成 `MAXVALUE`。
- `HASH` 分区下，不适合基于分区字段进行范围查询。

创建和管理分区表

OceanBase 数据库的分区表是内建功能，只需要在建表的时候指定分区策略和分区数即可。分区表的查询 SQL 跟普通表是一样的，OceanBase 的 OBProxy 或 OBServer 会自动将用户 SQL 路由到相应节点内。因此，分区表的分区细节对业务是透明的。

如果知道要读取的数据所在的分区号，可以通过 SQL 直接访问分区表的某个分区。简单语法格式如下：

```
part_table partition ( p[0,1,...][sp[0,1,...]] )
```

除了表定义了分区名这一特殊情况，默认情况下，分区名都是按一定规则编号，例如：

一级分区名为：p0, p1, p2, ... 二级分区名为：p0sp0, p0sp1, p0sp2, ... ; p1sp0, p1sp1, p1sp2, ...

示例：访问分区表的具体分区。

```
select * from t1 partition (p0) ;

select * from t1 partition (p5sp0) ;
```

创建分区表

OceanBase 数据库支持多种分区策略：

- 范围（`RANGE`）分区
- `RANGE COLUMNS` 分区
- 列表（`LIST`）分区
- `LIST COLUMNS` 分区
- 哈希（`HASH`）
- KEY 分区
- 组合分区

范围（RANGE）分区

`RANGE` 分区根据分区表定义时为每个分区建立的分区键值范围，将数据映射到相应的分区中。它是常见的分区类型，经常和日期类型一起使用。比如，可以将业务日志表按日/周/月分区。

`RANGE` 分区的语法格式如下：

```
CREATE TABLE table_name (
    column_name1        column_type
    [, column_nameN     column_type]
) PARTITION BY RANGE ( expr(column_name1) )
(
    PARTITION   p0      VALUES LESS THAN ( expr )
    [, PARTITION pN     VALUES LESS THAN (expr ) ]
 [, PARTITION pX    VALUES LESS THAN (maxvalue) ]
);
```

`RANGE` 分区规则：

- `PARTITION BY RANGE ( expr )` 中的 expr 表达式的结果必须为整型。
- 每个分区都有一个 `VALUES LESS THAN` 子句，它为分区指定一个非包含的上限值。分区键中等于或大于这个上限值将被映射到下一个分区中。
- 除第一个分区外，所有分区都隐含一个下限值，即上一个分区的上限值。
- 允许且只允许最后一个分区上限定义为 `MAXVALUE`，该值没有具体的数值，比其他所有分区的上限都要大，也包含空值。

注意

RANGE 分区可以新增、删除分区。如果最后一个 RANGE 分区指定了 MAXVALUE，则不能新增分区。所以建议不要使用 MAXVALUE 定义最后一个分区。

`RANGE` 分区要求表拆分键表达式的结果必须为整型，如果要按时间类型列做 `RANGE` 分区，则必须使用 `timestamp` 类型，并且使用函数 `UNIX_TIMESTAMP` 将时间类型转换为数值。这个需求也可以使用 `RANGE COLUMNS` 分区实现，则没有整型的限制。

示例：

```
CREATE TABLE test_range(id INT, gmt_create TIMESTAMP, info VARCHAR(20), PRIMARY KE
Y (gmt_create))
PARTITION BY RANGE(UNIX_TIMESTAMP(gmt_create))
(PARTITION p0 VALUES LESS THAN (UNIX_TIMESTAMP('2015-01-01 00:00:00')),
PARTITION p1 VALUES LESS THAN (UNIX_TIMESTAMP('2016-01-01 00:00:00')),
```

```
PARTITION p2 VALUES LESS THAN (UNIX_TIMESTAMP('2017-01-01 00:00:00')));
```

RANGE COLUMNS 分区

`RANGE COLUMNS` 分区与 `RANGE` 分区类似，但不同点在于 `RANGE COLUMNS` 分区可以按一个或多个分区键向量进行分区，并且每个分区键的类型除了 INT 类型还可以支持其他类型，比如 `VARCHAR`、`DATETIME` 等。

`RANGE COLUMNS` 分区的语法格式如下：

```
CREATE TABLE table_name (column_name column_type[, column_name column_type])
  PARTITION BY { RANGE COLUMNS(column_name [,column_name])
               }
    (
     PARTITION partition_name VALUES LESS THAN(expr)
     [, PARTITION partition_name VALUES LESS THAN (expr )...]
     [, PARTITION partition_name VALUES LESS THAN (MAXVALUE)]
     );
```

`RANGE COLUMNS` 和 `RANGE` 的区别：

- `RANGE COLUMNS` 分区不要求是 INT 类型，可以是任意类型。
- `RANGE COLUMNS` 分区不能写表达式。
- `RANGE COLUMNS` 分区支持向量。

示例：

```
CREATE TABLE tbl1_log_rc (log_id BIGINT NOT NULL,log_value VARCHAR(50),log_date DA
TE NOT NULL)
       PARTITION BY RANGE COLUMNS(log_date)
        (PARTITION M202001 VALUES LESS THAN('2020/02/01')
       , PARTITION M202002 VALUES LESS THAN('2020/03/01')
       , PARTITION M202003 VALUES LESS THAN('2020/04/01')
       , PARTITION M202004 VALUES LESS THAN('2020/05/01')
       , PARTITION M202005 VALUES LESS THAN('2020/06/01')
       , PARTITION M202006 VALUES LESS THAN('2020/07/01')
       , PARTITION M202007 VALUES LESS THAN('2020/08/01')
       , PARTITION M202008 VALUES LESS THAN('2020/09/01')
       , PARTITION M202009 VALUES LESS THAN('2020/10/01')
       , PARTITION M202010 VALUES LESS THAN('2020/11/01')
       , PARTITION M202011 VALUES LESS THAN('2020/12/01')
       , PARTITION M202012 VALUES LESS THAN('2021/01/01')
       , PARTITION MMAX VALUES LESS THAN MAXVALUE
```

```
        );
Query OK, 0 rows affected
```

列表（LIST）分区

`LIST` 分区是根据枚举类型的值来划分分区的，主要用于枚举类型。

`LIST` 分区可以显式的控制记录行如何映射到分区，具体方法是为每个分区的分区键指定一组离散值列表，这点跟 `RANGE` 分区和 `HASH` 分区都不同。`LIST` 分区的优点是可以方便的对无序或无关的数据集进行分区。

`LIST` 分区的语法格式如下：

```
CREATE TABLE table_name (column_name column_type[,column_name column_type])
  PARTITION BY { LIST ( expr(column_name) | column_name )
               }
    (PARTITION partition_name VALUES IN ( v01 [, v0N])
     [,PARTITION partition_name VALUES IN ( vN1 [, vNN])]
     [,PARTITION partition_name VALUES IN (DEFAULT)]
    );
```

`LIST` 分区规则：

- 分区键可以是列名，也可以是一个表达式，分区键必须是 INT 类型或者是返回 INT 类型的表达式。
- 分区表达式只能引用一列，不能有多列（即列向量），例如不允许 `partition by list (c1, c2)`。

示例：

```
CREATE TABLE tbl1_l (col1 BIGINT PRIMARY KEY,col2 VARCHAR(50))
       PARTITION BY LIST(col1)
        (PARTITION p0 VALUES IN (1, 2, 3),
         PARTITION p1 VALUES IN (5, 6),
         PARTITION p2 VALUES IN (DEFAULT)
        );
Query OK, 0 rows affected
```

LIST COLUMNS 分区

`LIST COLUMNS` 分区是 `LIST` 分区的一个扩展，支持多个分区键，并且支持 INT 数据、DATE 类型和 DATETIME 类型。

如果要使用多列的 `LIST` 分区，或者其他数据类型的 `LIST` 分区，可以使用 `LIST COLUMNS` 分区。

`LIST COLUMNS` 分区的语法格式如下：

```
CREATE TABLE table_name (column_name column_type[,column_name column_type])
  PARTITION BY { LIST COLUMNS ( column_name [,column_name])
                }
    (PARTITION partition_name VALUES IN ( v01 [, v0N])
     [,PARTITION partition_name VALUES IN ( vN1 [, vNN])]
     [,PARTITION partition_name VALUES IN (DEFAULT)]
    );
```

`LIST COLUMNS` 和 `LIST` 的区别：

- `LIST COLUMNS` 分区不要求是 INT 类型，可以是任意类型。
- `LIST COLUMNS` 分区不能写表达式。
- `LIST COLUMNS` 分区支持向量。

示例：

```
CREATE TABLE tbl1_lc (id INT,partition_id VARCHAR(2))
       PARTITION BY LIST COLUMNS(partition_id)
        (PARTITION p0 VALUES IN ('00','01'),
         PARTITION p1 VALUES IN ('02','03'),
         PARTITION p2 VALUES IN (DEFAULT)
        );
Query OK, 0 rows affected
```

哈希（HASH）分区

`HASH` 分区适合于不能使用 `RANGE` 分区、`LIST` 分区方法的场景。`HASH` 分区的实现方法简单，通过对分区键上的 HASH 函数值来散列记录到不同分区中。通常用于给定分区键的点查询，例如按照用户 id 来分区。 `HASH` 分区通常能消除热点查询。

如果表的数据符合下列特点，使用 `HASH` 分区是个很好的选择：

- 不能指定数据的分区键的列表特征。
- 不同范围内的数据大小相差非常大，并且很难手动调整均衡。

- 使用 RANGE 分区后数据聚集严重。
- 并行 DML、分区剪枝和分区连接等性能非常重要。

`HASH` 分区的语法格式如下：

```
CREATE TABLE table_name (column_name column_type[,column_name column_type])
  PARTITION BY { HASH(expr) }
    PARTITIONS partition_count;
```

`HASH` 分区规则：

- 分区键为 INT 类型，或者是返回为 INT 类型的表达式。
- 不能写向量，例如 `partition by hash(c1, c2)`。
- 对于 `HASH` 分区，创建时如果没有指定分区的名字，分区的命名由系统根据命名规则完成。对于一级分区表，则每个分区分别命名为 p0、p1、...、pn。
- `HASH` 分区不支持新增或删除分区。

示例：

```
create table t1 (
    c1 int,
    c2 int
) partition by hash(c1 + 1) partitions 5;
```

KEY 分区

`KEY` 分区与 `HASH` 分区类似，也是通过对分区个数取模的方式来确定数据属于哪个分区。

`KEY` 分区的语法格式如下：

```
CREATE TABLE table_name (column_name column_type[,column_name column_type])
  PARTITION BY { KEY([column_name_list]) }
    PARTITIONS partition_count;
```

`KEY` 分区规则：

- 系统会对 `KEY` 分区键做一个内部默认的 HASH 函数后再取模。
- 用户通常没有办法自己通过简单的计算来得知某一行属于哪个分区。

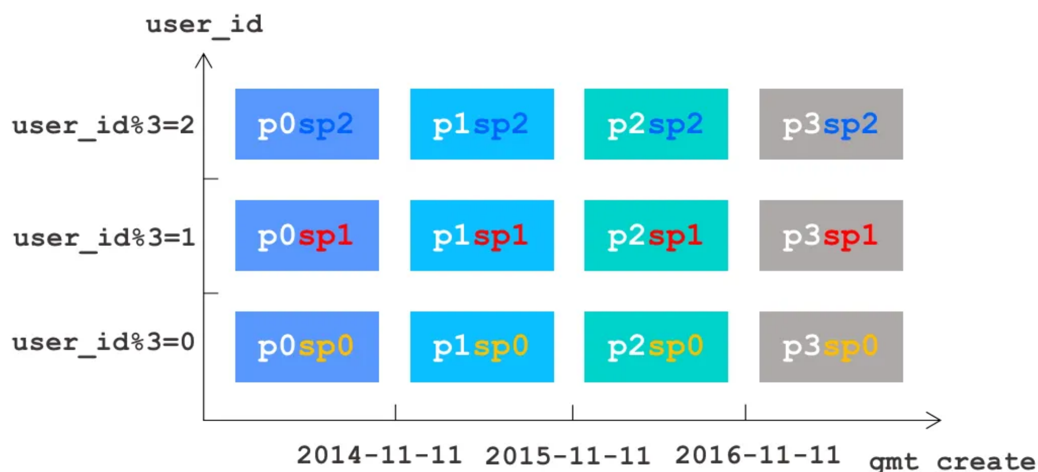
- KEY 分区不要求是 INT 类型，可以是任意类型。
- KEY 分区不能写表达式，只能是列。
- KEY 分区支持向量。
- KEY 分区分区键不写任何 column 时，表示 KEY 分区的列是主键。

示例：

```
CREATE TABLE tbl1_k(id INT,gmt_create DATETIME,info VARCHAR(20))
       PARTITION BY KEY(id,gmt_create) PARTITIONS 10;
Query OK, 0 rows affected
```

组合分区（二级分区）

二级分区是指按照两个维度来把数据拆分成分区。最常用的地方就是类似用户账单领域，会按照 user_id 做 HASH 分区，按照账单创建时间做 RANGE 分区。



OceanBase 数据库 MySQL 模式目前支持 HASH、RANGE、LIST、KEY、RANGE COLUMNS 和 LIST COLUMNS 六种分区类型，二级分区为任意两种分区类型的组合。

组合分区有如下优点：

- 根据 SQL 语句，在一维或二维上进行分区修剪可能会提高性能。
- 查询可以在任一维度上使用全分区或部分分区连接。
- 可以对单个表执行并行备份和恢复。
- 分区的数量大于单层分区，这可能有利于并行执行。

- 可以实现一个滚动窗口来支持历史数据，如果许多语句可以从分区修剪或分区连接中受益，则仍然可以在另一个维度上进行分区。
- 可以根据分区键的标识以不同方式存储数据。例如，可能决定以只读的压缩格式存储特定产品类型的数据，并保持其他产品类型的数据不压缩。

示例：

```
CREATE TABLE t_log_part_by_range_hash (
    log_id      int NOT NULL
    , log_value varchar(50)
    , log_date  TIMESTAMP NOT NULL
    , PRIMARY key(log_id, log_date)
) PARTITION BY RANGE(UNIX_TIMESTAMP(log_date))
SUBPARTITION BY HASH(log_id) SUBPARTITIONS 16
(
    PARTITION M202001 VALUES LESS THAN(UNIX_TIMESTAMP('2020/02/01'))
    , PARTITION M202002 VALUES LESS THAN(UNIX_TIMESTAMP('2020/03/01'))
    , PARTITION M202003 VALUES LESS THAN(UNIX_TIMESTAMP('2020/04/01'))
    , PARTITION M202004 VALUES LESS THAN(UNIX_TIMESTAMP('2020/05/01'))
    , PARTITION M202005 VALUES LESS THAN(UNIX_TIMESTAMP('2020/06/01'))
    , PARTITION M202006 VALUES LESS THAN(UNIX_TIMESTAMP('2020/07/01'))
    , PARTITION M202007 VALUES LESS THAN(UNIX_TIMESTAMP('2020/08/01'))
    , PARTITION M202008 VALUES LESS THAN(UNIX_TIMESTAMP('2020/09/01'))
    , PARTITION M202009 VALUES LESS THAN(UNIX_TIMESTAMP('2020/10/01'))
    , PARTITION M202010 VALUES LESS THAN(UNIX_TIMESTAMP('2020/11/01'))
    , PARTITION M202011 VALUES LESS THAN(UNIX_TIMESTAMP('2020/12/01'))
    , PARTITION M202012 VALUES LESS THAN(UNIX_TIMESTAMP('2021/01/01'))
);
```

尽管 OceanBase 数据库在组合分区上支持 `RANGE + HASH` 和 `HASH + RANGE` 两种组合，但是对于 `RANGE` 分区的分区操作 `add`/`drop`，必须是 `RANGE` 分区做为一级分区的方式。所以针对例如数据量较大的流水表，为了维护方便（新增和删除分区），建议使用 `RANGE + HASH` 组合方式。

管理分区表

分区表创建成功后，可以根据业务需要，修改分区规则、添加分区、删除分区、Truncate 分区，具体的语法可参见官网《OceanBase 数据库》参考指南/数据库对象管理/MySQL 模式/创建和管理分区 章节。

分区表的索引

分区表的查询性能跟 SQL 的条件有关。当 SQL 带上拆分键时，OceanBase 数据库会根据条件做分区检索，只搜索特定的分区；如果没有拆分键，则要扫描所有分区。

分区表也可以通过创建索引来提升性能。跟分区表一样，分区表的索引也可以分区或者不分区。

- 如果分区表的索引不分区，就是一个全局索引（`GLOBAL`），是一个独立的分区，索引数据覆盖整个分区表。
- 如果分区表的索引分区，根据分区策略又可以分为两类。
  - 一类是跟分区表保持一致的分区策略，则每个索引分区的索引数据覆盖相应的分区表的分区，这个索引又叫本地索引（`LOCAL`）。
  - 另一类则是分区策略和分区表不一致的全局索引（`GLOBAL`），称为全局分区索引。
- 一类是跟分区表保持一致的分区策略，则每个索引分区的索引数据覆盖相应的分区表的分区，这个索引又叫本地索引（`LOCAL`）。
- 另一类则是分区策略和分区表不一致的全局索引（`GLOBAL`），称为全局分区索引。

注意

- 分区表创建索引时默认创建本地索引。
- 建议尽可能的使用本地索引，只在必要时使用全局索引。因为全局索引会降低 DML 的性能，DML 可能会因此产生分布式事务。

示例：创建分区表的本地索引和全局索引。

```
CREATE INDEX idx_log_date ON t_log_part_by_range_hash(log_date) LOCAL;

CREATE INDEX idx_log_date2 ON t_log_part_by_range_hash(log_value, log_date) GLOBAL
;
```

注意

OceanBase 数据库的分区表主键和唯一键不需要单独建索引。OceanBase 数据库分区表的一个功能限制是如果分区表有主键，主键必须包含分区键，并且本地唯一索引也必须包含分区键。

分区表使用建议

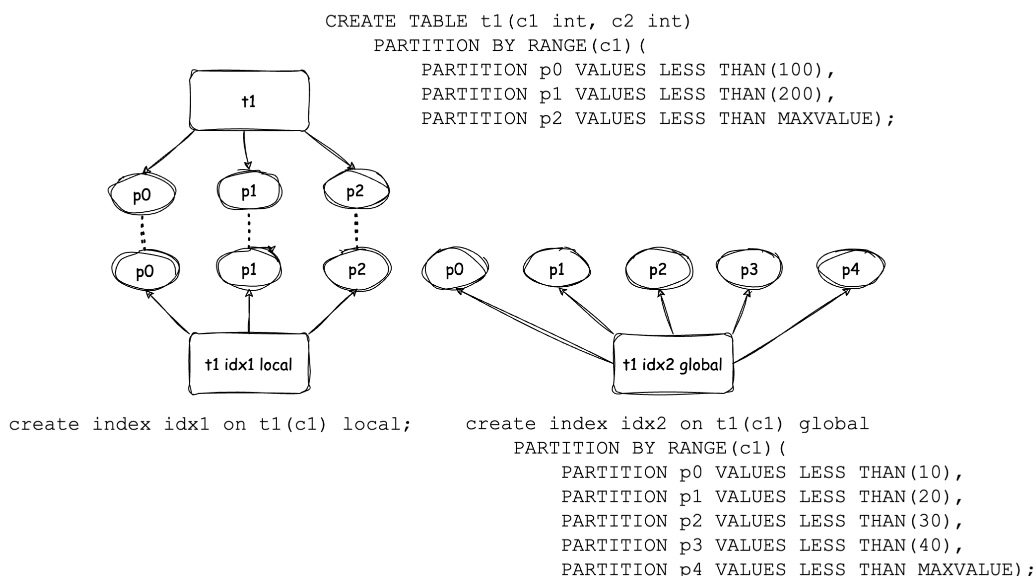
- 根据业务形态（热点数据打散、历史数据维护便利性、业务 SQL 的条件形态）来做分区。
- 关于分区键在多维业务查询场景下的选择，如账号和卡号同时存在情况下，需根据业务使用频率和业务重要性等维度来考虑分区键的选择。
- 在业务查询条件明确的情况下根据业务场景进行分区规划，分区目的是要利用分区裁剪的能力提高查询效率，禁止在场景不明确的情况下随意规划分区规则。如果查询条件部分场景下仅能覆盖一级分区，建议按照一级分区规划，不需要强行规划为二级分区。
- 对于分区表的索引创建，按照本地索引 -> 全局分区索引 -> 全局索引的顺序进行选择，只有在有必要的时候才使用全局索引，原因是全局索引会降低 DML 的性能，可能会因此产生分布式事务。
- 分区表的查询或修改尽量带上分区键。
- `RANGE` 分区不建议指定 `MAXVALUE`，否则后续无法新增分区。对于 `RANGE` 分区，建议业务自行做分区管理，定时增加分区，避免分区越界问题。可以考虑使用 ODC 分区管理功能。
- 为了避免写入放大问题，选择表的自定义主键时，不要使用随机生成的值，要尽量有序，比如时序递增的。
- 有历史数据清理的表，需要根据业务使用场景和清理周期进行分区表设计。如交易流水表，可按日分区并按日删除旧分区。

6.4 OceanBase 数据库在 MySQL 模式租户下的扩展功能

全局索引（Global Index）

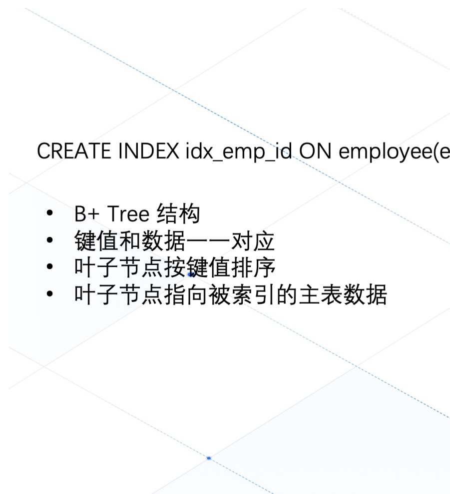
功能定义

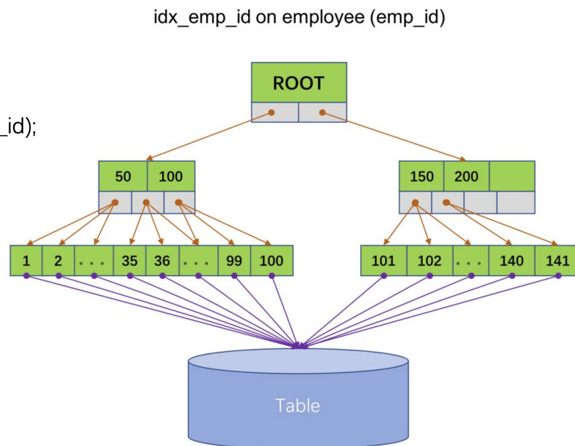
MySQL 中没有全局索引，只有局部索引。全局索引与局部索引相比，最大的特点是全局索引的分区规则跟表分区是相互独立的，全局索引允许指定自己的分区规则和分区个数，不需要跟表分区规则保持一致。



引入原因

在关系型数据库中，索引通常被组织成一颗 B+ Tree 的形式，叶子节点按照键值的大小被有序存放，索引的键值跟主表上的数据一一对应，当用户指定索引条件访问数据的时候，可以通过搜索 B+ Tree 上的对应关系，快速定位到被访问到的数据在主表中的位置。





当主表数据被拆分成多个分片的时候（在 OceanBase 数据库中叫分区表），索引应该如何拆分？一个思路是让索引跟主表一起拆分，拆分后的索引只检索当前分区中的部分数据，这样的索引我们一般称为本地索引（Local Index，MySQL 就只支持这种索引）。在 OceanBase 数据库中要创建这样的索引也很简单，只需要在语句最后指定一个 `local` 关键字即可。这样的索引有什么缺点呢？

首先第一个缺点是查询需要指定分区键，否则，数据库将不知道检索的数据位于哪个分片中，只能枚举出所有的数据分片，让查询效率变低。由下图示例可见（主表 `employ` 是分区表，分区键是 `emp_id`），如果查询中的过滤条件没有指定主表的分区键 `emp_id`，从执行计划的红框部分可以看出来，数据库就会扫描所有的数据分片。

语法：

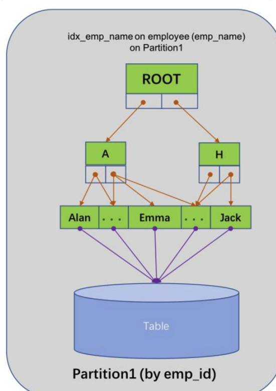
CREATE INDEX idx_emp_name ON employ(emp_name) LOCAL;

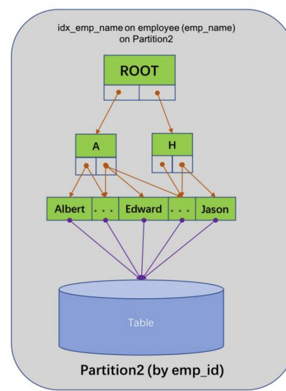
特点：

- 和主表的分片信息一一对应
- 每个子树只索引分片内的数据

缺点：

- 查询需要指定分区键
  select * from employee where emp_name='Edward';

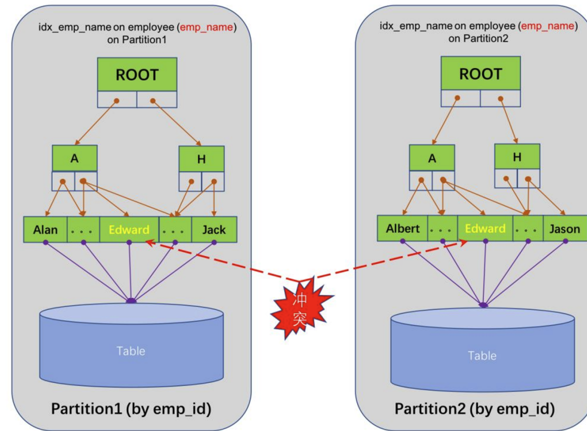




另一个缺点就是由于本地索引是创建在数据分片内部，因此无法保证索引键值的全局唯一性。例如下图的例子中，本地索引的两部分都可能出现 `Edward` 这个重复键值，因此数据库要求要创建带唯一性约束的本地索引必须指定数据分片的分区键。

缺点：
- 无法保证全局唯一性

```
CREATE UNIQUE INDEX idx_emp_name
 ON employ(emp_name) LOCAL;

ERROR 1503 (HY000): A UNIQUE INDEX
must include all columns in the
table's partitioning function
```



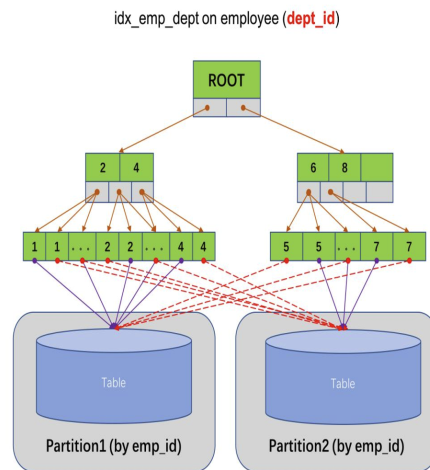
为了避免本地索引的诸多限制带来使用上的复杂度，OceanBase 数据库在 MySQL 模式下又推出了一种新的索引形式，那就是全局索引（Global Index），它和本地索引（Local Index）的本质区别是：全局索引的索引结构并不跟主表的分片信息一一对应，它们数据的分片位置信息是各自独立的。同时，一个索引键值可能会对应到不同的主表分片中，例如下图右侧的这个索引结构中，索引键值 `1`，`2`，`5` 都同时对应了两个不同分区里的数据，因此在使用全局索引进行数据检索的时候，我们不再需要指定分区键，同时由于全局索引的索引结构是独立于主表的，索引键值的全局唯一性在这里也可以得到保证。

语法：

```
CREATE INDEX idx_emp_dept ON
   employ(dept_id) GLOBAL;
```

特点：
- 索引和主表的分片信息各自独立
- 一个键值可能对应多个主表分片中的数据

优点：
- 索引检索无需指定分区键
- 可以保证键值的全局唯一性



相关语法

要创建一个全局索引也很简单，只需要我们在创建索引的语法后面指定一个 `global` 关键字就好了，示例如下：

```
-- 创建一张表
create table t1(
  c1 int primary key,
```

```
  c2 int
  ) partition by hash(c1) partitions 5;

-- 创建分区方式和主表不同的分区索引
create index g_idx on t1(c2) global
  partition by range(c2)
    (partition p0 values less than(100),
     partition p1 values less than(200),
     partition p2 values less than(300)
    );

-- 上面两条 SQL，等价于这一条 SQL
CREATE TABLE `t1` (
  `c1` int(11) NOT NULL,
  `c2` int(11) DEFAULT NULL,
  PRIMARY KEY (`c1`),
  KEY `g_idx` (`c2`) GLOBAL
   partition by range(c2) -- 索引的分区方式
    (partition `p0` values less than (100),
     partition `p1` values less than (200),
     partition `p2` values less than (300)
    )
  ) partition by hash(c1) -- 主表的分区方式
    (partition `p0`,
     partition `p1`,
     partition `p2`,
     partition `p3`,
     partition `p4`);
```

注意

如果在创建索引时，不加 global 或者 local 关键字，但是索引后面有分区信息的话，会被默认当作 Global Index；如果索引后面没有分区信息的话，会被默认当作 Local Index。

适用场景

刚才我们了解了全局索引的两个好处，那么在 OceanBase 数据库这样的分布式数据库中，是不是应该无条件使用全局索引呢？由于全局索引可能会出现跨节点的数据访问，因此数据检索过程中，RPC 的代价是无法忽略的，因此并不是所有情况下用全局索引都比本地索引表现更好。

那么哪些场景推荐使用全局索引呢？

- 主表是分区表，但是索引键不包含主表分区键的场景。如果这种场景使用局部索引，由于某

一个索引键值在所有分区的本地索引上都可能存在，任何索引扫描都必须在索引的所有的分区上都做一遍，否则就造成数据遗漏。这会导致索引扫描效率低下，带来资源浪费。对于全局索引来说，可以为索引的数据指定自己的分区方式，并且索引的分区键一定是索引键的子集，因此可以很容易解决这个问题。

- 如果需要保证索引键满足唯一性约束，并且索引键不包含主表分区键信息的场景，这种情况下，由于本地索引自身的唯一性缺陷，只能选择全局索引。

> **说明**
>
> 很多数据库在分区表中都会增加一些限制，例如要求主键和唯一索引的定义中，必须包含主表所有的分区键字段。有了这个限制，索引中的某一个键所对应的索引数据只可能存在于一个分区中，因此只要在每一个分区内保证唯一性约束，即可在全表范围内保证唯一性约束。
>
> 这个限制虽然解决了数据库的难题，却增加了开发人员的烦恼。因为这会导致主键和唯一索引的可选范围大大缩小（只能是主表分区键的“超集”），很多业务的需求因此无法满足，这也是本地索引最为被大家诟病的地方。

总的来说，全局索引相比于本地索引实现会更加复杂，尤其在分布式数据库中，只有少数的商业数据库支持这种索引形式，但是它对用户的使用约束更小，体验也会更好。

常见问题

为什么主键索引不能设置成 Global 属性？

OceanBase 数据库的表目前都是索引组织表（Index Organized Table，简称 IOT 表），暂时还没有支持堆表。索引组织表是一种数据库表的存储方式，它的特点是根据表的主键索引来组织数据，而不是按照数据的物理顺序来组织。因为每张表都是根据主键索引来组织的，所以主键和主表的组织形式一定是一致的，不能设置成 Global 属性。

顺带一提，OceanBase 数据库中的无主键表其实也有隐藏主键，是个自增列，`column id` 为 `1`，`column name` 叫 `__pk_increment`，有兴趣的同学可以去查查 `oceanbase.__all_column` 看一下。

为什么分区表的分区键一定要被包含在本地唯一索引和主键索引里？

OceanBase 数据库中，如果需要在分区表上创建局部分区唯一索引（Local Partitioned

Unique Index），则该索引键需要包含主表的分区键，而对于全局分区唯一索引（Global Partitioned Unique Index）并没有这个限制，详细介绍可参见官网《OceanBase 数据库》文档 参考指南/数据库对象管理/MySQL 模式/创建和管理分区/在分区表上建立索引 章节。

```
create table t1(c1 int unique key, c2 int) partition by hash(c2);
ERROR 1503 (HY000): A UNIQUE INDEX must include all columns in the table's partiti
oning function

create table t1(c1 int primary key, c2 int) partition by hash(c2);
ERROR 1503 (HY000): A PRIMARY KEY must include all columns in the table's partitio
ning function
```

上文提到：唯一索引只在各个分区内进行数据的唯一性检查。假设唯一索引不包含全部分区键，并且让 `create table t1(c1 int unique key, c2 int) partition by hash(c2);` 执行成功的话，主表上的数据可能会如下所示。

c1	c2
1	1
1	2
2	1
2	2
3	1
3	2

这时第一个分区的数据如下所示，在这个分区中 `c1` 是满足唯一性的，唯一性检查会成功。

c1	c2
1	1
2	1
3	1

第二个分区的数据如下所示，在这个分区中 `c1` 也是满足唯一性的，唯一性检查也会成功。

c1	c2
1	2
2	2

3	2

这就会出现：所有分区在对 `c1` 做唯一性检查时都成功了，数据库认为 `c1` 列已经满足了唯一性，但实际上 `c1` 列的数据却并没有满足唯一性。主键同理。当然，MySQL 和 Oracle 数据库也有相同的要求和限制。

如果唯一索引被打上了 Global 的属性，不再使用主表的分区规则进行分区，自然也就可以避免这个唯一性检查出错的问题了。

回收站（recyclebin）

功能定义

回收站是一种存放被用户删除的数据库对象的功能。

在 OceanBase 数据库中，支持进入回收站的对象包括库（database）、表（table）、索引（index）和租户（tenant）。用户删除的信息被放入回收站后，其实仍然占据着物理空间，除非手动清除（PURGE）或者配置数据库定期清理回收站对象。在回收站对象被清理之前，可以通过命令进行恢复。

引入原因

回收站的功能是为了防止用户误删除数据库对象的时候，无法对其进行恢复。

MySQL 数据库缺失这个重要功能，OceanBase 数据库参考了 Oracle 数据库中回收站的功能和使用方法，在 OceanBase 数据库的 MySQL 租户下也支持了回收站的功能。

相关语法

设置回收站开关及定时清理周期

- 开启/关闭回收站功能

  回收站功能默认是关闭的。

  ```
  -- 对当前 session 生效
  ```

```
SET recyclebin = on;
SET recyclebin = off;
SET session recyclebin = on;
SET session recyclebin = off;
```

可增加 `global` 关键字，让开启/关闭回收站功能的命令在当前租户内生效。

```
-- 设置后，对当前 session 无效，对后续建立新的 session 生效。
SET global recyclebin = on;
SET global recyclebin = off;
```

- 查看回收站开关状态

```
show variables like 'recyclebin';
```

输出如下：

```
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| recyclebin    | ON    |
+---------------+-------+
```

- 设置回收站中对象的定期清理周期

默认值为 `0s`，表示不开启该功能。如下命令只能通过 sys 租户执行，对整个集群生效。

```
alter system set recyclebin_object_expire_time= '7d';
```

设置为 `0s`，表示关闭该功能。

```
ALTER SYSTEM SET recyclebin_object_expire_time = "0s";
```

- 查看集群中的 `recyclebin_object_expire_time` 参数

```
show parameters like 'recyclebin_object_expire_time'\G
```

输出如下：

```
*************************** 1. row ***************************
         zone: zone1
     svr_type: observer
```

```
       svr_ip: 1.2.3.4
     svr_port: 12345
         name: recyclebin_object_expire_time
    data_type: NULL
        value: 0s
         info: recyclebin object expire time, default 0 that means auto purge recy
clebin off. Range: [0s, +∞)
      section: ROOT_SERVICE
        scope: CLUSTER
       source: DEFAULT
   edit_level: DYNAMIC_EFFECTIVE
default_value: 0s
    isdefault: 1
```

查看回收站对象

说明

- 单独删除索引时，该索引不会进入回收站。
- 删除表时，表上的索引会随主表一起进入回收站。

1. 创建 database、table、index

```
create database test_db;
create table t1(c1 int, c2 int, index idx(c2));
```

2. 删除 database、table、index

```
drop database test_db;
drop table t1;
```

3. 查看回收站对象

```
show RECYCLEBIN;
```

输出如下，`OBJECT_NAME` 表示回收站对象进入回收站之后的新名字，为了唯一标识回收站内重名的同类型对象，`ORIGINAL_NAME` 表示回收站对象被删除前的原名。

```
+--------------------------------+------------------+----------+------------------
----------+
```

```
| OBJECT_NAME                      | ORIGINAL_NAME    | TYPE      | CREATETIM
E                  |
+----------------------------------+------------------+-----------+------------------
----------+
| __recycle_$_1_1713173706419784 | __idx_500044_idx | INDEX     | 2024-04-15 17:35:
06.419840 |
| __recycle_$_1_1713173706464688 | t1               | TABLE     | 2024-04-15 17:35:
06.465056 |
| __recycle_$_1_1713173712877712 | test_db          | DATABASE | 2024-04-15 17:35:
12.877862 |
+----------------------------------+------------------+-----------+------------------
----------+
3 rows in set
```

4. 查询内部表 `oceanbase.__all_recyclebin`

```
select * from oceanbase.__all_recyclebin;
```

输出如下，可以看到回收站对象在被删除前的详细信息，如被删除前的 `database_id`、`table_id` 等。

```
+----------------------------+-----------+---------------------------------+-------
-------------+----------+--------------+------------------+
| gmt_create                 | tenant_id | object_name                     | type
| database_id | table_id | tablegroup_id | original_name    |
+----------------------------+-----------+---------------------------------+-------
-------------+----------+--------------+------------------+
| 2024-04-15 17:35:06.419840 |         0 | __recycle_$_1_1713173706419784 |     2
|      500001 |    500045 |            -1 | __idx_500044_idx |
| 2024-04-15 17:35:06.465056 |         0 | __recycle_$_1_1713173706464688 |     1
|      500001 |    500044 |            -1 | t1               |
| 2024-04-15 17:35:12.877862 |         0 | __recycle_$_1_1713173712877712 |     4
|      500046 |        -1 |            -1 | test_db          |
+----------------------------+-----------+---------------------------------+-------
-------------+----------+--------------+------------------+
3 rows in set
```

恢复回收站对象

下面以恢复 table 为例，恢复 table 的语法是：`FLASHBACK TABLE object_name TO BEFORE DROP [RENAME To new_table_name];`。

```
-- 创建一个和刚刚被删除表的同名表
create table t1(c1 int);
```

```
Query OK, 0 rows affected

-- 从回收站中恢复表，需要到表被删除前的原库下执行，恢复后的表名默认为进入回收站前的表名，可以通过 R
ENAME To new_table_name 进行修改
-- 执行该语句后，恢复后的表名为进入回收站前的名称，和表一起进入回收站的索引也会一同恢复出来
-- 如果该表进入回收站前的名称与已有表重名，则系统会报错。
flashback table __recycle_$_1_1713173706464688 to before drop;
ERROR 1050 (42S01): Table 't1' already exists

-- 通过 RENAME To new_table_name 进行修改
flashback table __recycle_$_1_1713173706464688 to before drop rename to old_t1;
Query OK, 0 rows affected

-- 可以看到回收站中的表和表上的索引都一起被恢复出来了
obclient [test]> show RECYCLEBIN;
+---------------------------------+--------------+----------+---------------------
-------+
| OBJECT_NAME                     | ORIGINAL_NAME | TYPE     | CREATETIM
E                  |
+---------------------------------+--------------+----------+---------------------
-------+
| __recycle_$_1_1713173712877712 | test_db      | DATABASE | 2024-04-15 17:35:12.
877862 |
+---------------------------------+--------------+----------+---------------------
-------+
```

database 和 tenant 只需要把命令中的 table 关键字改为 database 或 tenant 即可，这里不再详述。

说明

1. 从回收站中恢复表前，如果该表从属的数据库已删除，则需要先恢复数据库再恢复表。
2. 不支持直接恢复索引。在通过 FLASHBACK 语句恢复表时，表上的索引也会一起恢复。
3. 恢复回收站对象时可修改待恢复对象的名称，但是不能与已有对象重名，否则系统会报错。
4. 如果一张表在进入回收站前属于某个表组，从回收站中恢复后该表默认会恢复到原表组中；如果原表组已经被删除，则该表恢复后会不属于任何一个表组。

清空回收站对象

下面展示清空回收站对象的命令。

```
show recyclebin;
+--------------------------------+------------------+----------+------------------
----------+
| OBJECT_NAME                    | ORIGINAL_NAME    | TYPE     | CREATETIM
E                 |
+--------------------------------+------------------+----------+------------------
----------+
| __recycle_$_1_1713173712877712 | test_db          | DATABASE | 2024-04-15 17:35:
12.877862 |
| __recycle_$_1_1713177112706600 | __idx_500048_idx | INDEX    | 2024-04-15 18:31:
52.706664 |
| __recycle_$_1_1713177112725848 | t1               | TABLE    | 2024-04-15 18:31:
52.727735 |
+--------------------------------+------------------+----------+------------------
----------+
3 rows in set

-- 清理指定 index 的语法是: PURGE INDEX object_name;
purge index __recycle_$_1_1713177112706600;

-- 清理指定 table 的语法是: PURGE TABLE object_name;
purge table __recycle_$_1_1713177112725848;

-- 清理指定 table 的语法是: PURGE DATABASE object_name;
purge database __recycle_$_1_1713173712877712;

-- 类似地，清空租户的语法是: PURGE TENANT object_name;（只有通过 sys 租户才能管理租户级别的对
象）

-- 还可以通过 purege recyclebin 一次性清理回收站中的所有对象。
purge recyclebin;

-- 展示清理之后的回收站中的对象
show recyclebin;
Empty set
```

说明

Purge 操作会删除对象和从属于该对象的对象（即 Database -> Table -> Index）。例如，Purge 数据库会删除数据库和从属于该数据库的表和表索引。

适用场景

在生产环境中，当某个特殊租户中存放了重要的数据，为了防止 DBA 同学的误操作，建议开启

回收站功能。

为了减少回收站中数据的额外磁盘开销，可以适当调小回收站定时清理的周期。

常见问题

禁用回收站是否会清空回收站中已经存在的对象？

通过设置 `recyclebin = off` 禁用回收站时，不会清空或影响回收站中已经存在的对象。

是否支持通过数据库对象的原名进行回收站对象的 flashback 和 purge？

对于 table 对象，支持通过被删除前的原名（ORIGINAL_NAME）进行 flashback 和 purge。规则如下：

- 当回收站中出现同原名的表对象时，flashback 会还原最晚进回收站的那张表。所以 flashback original_name 的操作可以理解成是一个栈，后进先还原。
- 当回收站中出现同原名的表对象时，purge 则会清空最早进回收站的那张表，所以 purge original_name 的操作可以理解成是一个队列，先进先清空。

上述规则参考了 Oracle 数据库中的回收站行为，均和 Oracle 的回收站表现一致。

对于 tenant 对象，也支持过被删除前的原名（ORIGINAL_NAME）进行 flashback 和 purge，和上述规则相同。

对于 index 和 database 对象，则不支持该行为。

> **说明**
>
> 通过数据库对象原名进行的回收站操作，为什么不在上面语法部分给大家介绍，而是放到了常见问题里？是因为虽然通过原名进行 flashback 和 purge 看上去更易用，但还是建议大家尽量通过可以作为唯一标识的新名字 OBJECT_NAME 对回收站对象进行还原和清空，这样可以避免由于记错通过对象原名进行操作的规则而造成的损失。

为什么不支持索引单独进回收站？

在 OceanBase 数据库中直接删除索引，该索引不会进入回收站。删除表时，表上的索引会随主

表一起进入回收站。

因为如果单独删除索引时，支持被删除的索引进入回收站的话，那么在主表后续发生表结构变更或者表数据变更时，回收站中的索引已经被删除，不能跟随主表同步更新，就会出现回收站中的索引和未被删除的主表数据不一致的问题，因此该单独删除的索引不能通过 flashback 命令进行还原，也就失去了进入回收站的意义。

为什么 4.x 版本的 truncate table 不进回收站了？

在 3.x 版本中，当 session 级别的系统变量 `ob_enable_truncate_flashback` 被置为 `on` 时，如果进行了一个 truncate table 的误操作，可以通过 flashback 还原执行 truncate table 之前的表和数据。

在 4.x 版本出现了一个功能回退，原因说来话长，简单来说是 truncate table 的实现方式发生变化导致的。对这个问题感兴趣的同学，可以通过阅读官网博客《浅析 4.x 版本的 truncate table 为什么不进回收站了？》来了解，这里不再详细分析这个问题。

表组（tablegroup）

功能定义

表组（tablegroup）是一个逻辑概念，表示一组表的集合。默认情况下，不同表之间的数据是随机分布的，没有关系。通过定义表组，可以控制一组表在物理存储上的邻近关系。

引入原因

OceanBase 数据库是原生的分布式数据库，当用户部署的环境中存在每个 Zone 中有多台 OBServer 节点的情况时，在同一个 Zone 中，为了负载均衡，不同表中的数据可能会分布在不同的机器上，同一张表中不同分区的数据可能也会分布在不同的机器上。

这样会有两个问题：

- 当两张分区表的分区定义（分区类型、分区个数、分区值等）完全相同时，两张表中对应的相同分区可能会在两个不同的节点上。每次对这两张表的分区键进行 JOIN 操作时，都需要

对相同分区的数据做跨机通信，可能会有较大的网络开销。

- 类似地，当经常出现在同一个事务里的表（或分区）不能分布到同一个节点上时，就会出现分布式事务，可能会影响事务性能。

为了提高数据库性能，应该尽量避免上述的跨机动作，OceanBase 数据库可以利用表组功能，把一批表聚集在同一台机器上，或者把有相同分区属性表中的相同分区聚集在同一台机器上。

举个例子：

1. 创建两张分区规则相同的表 `t1` 和 `t2`

```
obclient [test]> CREATE TABLE t1(col1 int, col2 int)
                     PARTITION BY RANGE(col1)(
                     PARTITION p0 VALUES LESS THAN(100),
                     PARTITION p1 VALUES LESS THAN(200),
                     PARTITION p2 VALUES LESS THAN(300));
```

```
obclient [test]> CREATE TABLE t2(col1 int, col2 int)
                     PARTITION BY RANGE(col1)(
                     PARTITION p0 VALUES LESS THAN(100),
                     PARTITION p1 VALUES LESS THAN(200),
                     PARTITION p2 VALUES LESS THAN(300));
```

2. 创建 tablegroup，并把 `t1` 和 `t2` 加入进这个表组

```
obclient [test]> CREATE TABLEGROUP tg1 sharding = 'ADAPTIVE';
obclient [test]> ALTER TABLEGROUP tg1 add t1, t2;
```

查看表组 `tg1`。

```
obclient [test]> SHOW TABLEGROUPS WHERE tablegroup_name = 'tg1';
```

输出如下：

```
+-----------------+------------+---------------+----------+
| Tablegroup_name | Table_name | Database_name | Sharding |
+-----------------+------------+---------------+----------+
| tg1             | t1         | test          | ADAPTIVE |
| tg1             | t2         | test          | ADAPTIVE |
+-----------------+------------+---------------+----------+
2 rows in set
```

3. 创建一张分区规则不同的表

```
obclient [test]> CREATE TABLE t3(col1 int, col2 int)
                     PARTITION BY RANGE(col1)(
                     PARTITION p0 VALUES LESS THAN(400),
                     PARTITION p1 VALUES LESS THAN(500),
                     PARTITION p2 VALUES LESS THAN(600));
```

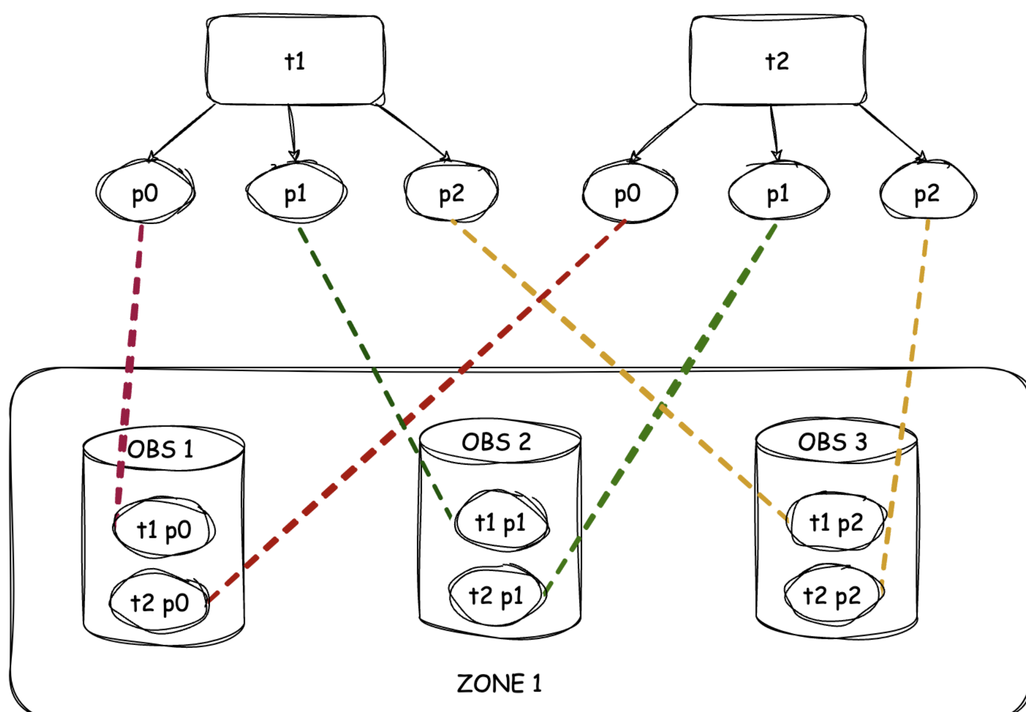
4. 尝试把分区规则不同的表加入 TABLEGROUP

```
obclient [test]> ALTER TABLEGROUP tg1 add t3;
```

输出如下，会失败报错。

```
ERROR 4179 (HY000): range_part partition value not equal, add table to tablegroup
not allowed
```

加入 tablegroup 之后，`t1` 和 `t2` 表的相同分区的数据，会被分布到同一个节点上，如下图所示：



在分区规则完全相同，且对应分区也分布在相同节点的场景下，对分区键进行 join 计算时，不需要对各个分区的数据进行分区数据的重分布，无需走网络。OceanBase 数据库内部把这种特殊的 join 称为 partition wise join，可以节约大量的网络传输开销。

```
obclient [test]> explain select * from t1, t2 where t1.col1 = t2.col1;
```

输出如下：

```
+--------------------------------------------------------------------------------
--------------+
| Query Pla
n
   |
+--------------------------------------------------------------------------------
--------------+
| ============================================================
=                                |
| |ID|OPERATOR                   |NAME    |EST.ROWS|EST.TIME(us)
|                                |
| ------------------------------------------------------------
-                                |
| |0 |PX COORDINATOR             |        |1        |26
|                                |
| |1 | └─EXCHANGE OUT DISTR      |:EX10000|1        |25
|                                |
| |2 |   └─PX PARTITION ITERATOR|        |1        |24
|                                |
| |3 |     └─HASH JOIN           |        |1        |24
|                                |
| |4 |       ├─TABLE FULL SCAN   |t1      |1        |12
|                                |
| |5 |       └─TABLE FULL SCAN   |t2      |1        |12
|                                |
| ============================================================
=                                |
| Outputs & filters
:                                                                           |
| -------------------------------------
-                                                        |
|   0 - output([INTERNAL_FUNCTION(t1.col1, t1.col2, t2.col1, t2.col2)]), filter(ni
l), rowset=16 |
|   1 - output([INTERNAL_FUNCTION(t1.col1, t1.col2, t2.col1, t2.col2)]), filter(ni
l), rowset=16 |
|       dop=
1
  |
|   2 - output([t1.col1], [t2.col1], [t1.col2], [t2.col2]), filter(nil), rowset=1
6             |
|       partition wise, force partition granul
e                                        |
|   3 - output([t1.col1], [t2.col1], [t1.col2], [t2.col2]), filter(nil), rowset=1
6             |
|       equal_conds([t1.col1 = t2.col1]), other_conds(nil
)                                       |
```

```
|   4 - output([t1.col1], [t1.col2]), filter(nil), rowset=1
6                                     |
|       access([t1.col1], [t1.col2]), partitions(p[0-2]
)                                         |
|       is_index_back=false, is_global_index=false
,                                              |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                              |
|   5 - output([t2.col1], [t2.col2]), filter(nil), rowset=1
6                                     |
|       access([t2.col1], [t2.col2]), partitions(p[0-2]
)                                         |
|       is_index_back=false, is_global_index=false
,                                              |
|       range_key([t2.__pk_increment]), range(MIN ; MAX)always tru
e                              |
+-------------------------------------------------------------------
--------------+
27 rows in set
```

大家可以根据如下示例，对比一下分区规则不同的两张表之间做 join，计划上有什么区别？

```
explain select * from t1, t3 where t1.col1 = t3.col1;
```

输出如下，`EXCHANGE OUT DISTR (PKEY)` 这个算子的含义是把 `t1` 的数据，按照 `t3` 的分区规则，进行数据重分布，会带来网络传输的开销。

```
+-------------------------------------------------------------------
--------------+
| Query Pla
n
   |
+-------------------------------------------------------------------
--------------+
| ====================================================================
=                          |
| |ID|OPERATOR                    |NAME    |EST.ROWS|EST.TIME(us)
|                          |
| --------------------------------------------------------------------
-                          |
| |0 |PX COORDINATOR              |        |1       |27
|                          |
| |1 | └─EXCHANGE OUT DISTR        |:EX10001|1       |26
|                          |
| |2 |   └─HASH JOIN               |        |1       |25
|                          |
| |3 |     ├─EXCHANGE IN DISTR     |        |1       |13
|                          |
```

```
| |4 |     |  └─EXCHANGE OUT DISTR (PKEY)|:EX10000|1       |13
|                          |
| |5 |     |    └─PX PARTITION ITERATOR  |         |1       |12
|                          |
| |6 |     |      └─TABLE FULL SCAN      |t1       |1       |12
|                          |
| |7 |     └─PX PARTITION ITERATOR      |         |1       |12
|                          |
| |8 |       └─TABLE FULL SCAN          |t3       |1       |12
|                          |
| ===================================================================
=                          |
| Outputs & filters
:                                                                             |
| ------------------------------------
-                                                                |
|   0 - output([INTERNAL_FUNCTION(t1.col1, t1.col2, t3.col1, t3.col2)]), filter(ni
l), rowset=16 |
|   1 - output([INTERNAL_FUNCTION(t1.col1, t1.col2, t3.col1, t3.col2)]), filter(ni
l), rowset=16 |
|       dop=
1
  |
|   2 - output([t1.col1], [t3.col1], [t1.col2], [t3.col2]), filter(nil), rowset=1
6            |
|       equal_conds([t1.col1 = t3.col1]), other_conds(nil
)                                              |
|   3 - output([t1.col1], [t1.col2]), filter(nil), rowset=1
6                                            |
|   4 - output([t1.col1], [t1.col2]), filter(nil), rowset=1
6                                            |
|       (#keys=1, [t1.col1]), dop=
1                                                                     |
|   5 - output([t1.col1], [t1.col2]), filter(nil), rowset=1
6                                            |
|       force partition granul
e                                                                         |
|   6 - output([t1.col1], [t1.col2]), filter(nil), rowset=1
6                                            |
|       access([t1.col1], [t1.col2]), partitions(p[0-2]
)                                                |
|       is_index_back=false, is_global_index=false
,                                                     |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                                      |
|   7 - output([t3.col1], [t3.col2]), filter(nil), rowset=1
6                                            |
|       affinitize, force partition granul
e                                                              |
|   8 - output([t3.col1], [t3.col2]), filter(nil), rowset=1
6                                            |
```

```
|       access([t3.col1], [t3.col2]), partitions(p[0-2]
)                                        |
|       is_index_back=false, is_global_index=false
,                                             |
|       range_key([t3.__pk_increment]), range(MIN ; MAX)always tru
e                             |
+--------------------------------------------------------------------------------
--------------+
35 rows in set
```

相关语法

表组的属性和语法，详见官网《OceanBase 数据库》文档 参考指南/数据库对象管理/MySQL 模式/创建和管理表组》章节，本文中不再赘述。

SHARDING 属性

OceanBase 数据库从 1.x 版本开始就引入了 Table Group 的概念和能力。多张具有关联关系的表往往遵守相同的分区规则，通过将相同规则的分区聚集分布在一起，可以实现 Partition Wise Join，极大地优化读写性能。Table Group 的引入就是为了让普通的 Join 操作变为 Partition Wise Join。

OceanBase 数据库 4.2 版本为 Table Group 引入了 `SHARDING` 属性，用于控制 Table Group 内表数据的聚集和打散关系。默认情况下，不同表之间数据是随机分布的。如果希望多张具有关联关系的表数据分布相同，则需要明确不同表分区之间的对齐规则，从而将不同表的分区聚集在同一台 Server，实现 Partition Wise Join，提升性能。采用 `SHARDING` 不为 `NONE` 的 Table Group 可以满足该需求。

- `SHARDING = NONE` 时，Table Group 描述了所有表的所有分区聚集在同一台 Server，并且不限制表的分区类型，可以通过这种属性实现让经常出现在同一个事务中的表分布在相同的节点上，从而让分布式事务简化成为单机事务。
- `SHARDING` 不为 `NONE` 时，Table Group 内每一张表的数据打散分布在多台机器上。为了保证所有表的数据分布相同，Table Group 要求所有表的分区方式一致，包括分区类型、分区个数、分区值。系统会调度具有相同分区属性的分区聚集（对齐）分布在同一台 Server，从而实现 Partition Wise Join。

下面描述不同 `SHARDING` 取值的含义以及对 Table Group 中表的影响。

- `PARTITION`：按一级分区打散，如果是二级分区表，一级分区下的所有二级分区聚集在一起。
    - 分区方式要求：一级分区的分区方式相同，如果是二级分区表，也只校验一级分区的分区方式。因此，一级分区表和二级分区表可以同时存在，只要他们的一级分区的分区方式相同即可。

    - 分区对齐规则：相同一级分区 Value 的分区聚集在一起，包括一级分区表的一级分区和二级分区表的对应一级分区下的所有二级分区。

- 分区方式要求：一级分区的分区方式相同，如果是二级分区表，也只校验一级分区的分区方式。因此，一级分区表和二级分区表可以同时存在，只要他们的一级分区的分区方式相同即可。
- 分区对齐规则：相同一级分区 Value 的分区聚集在一起，包括一级分区表的一级分区和二级分区表的对应一级分区下的所有二级分区。
- `ADAPTIVE`：自适应打散方式。如果 Table Group 内是一级分区表，则按一级分区打散；如果 Table Group 内是二级分区表，则每个一级分区下的二级分区打散。
    - 分区方式要求：要么全部是一级分区表，要么全部是二级分区表。如果是一级分区表，则要求一级分区的分区方式相同；如果是二级分区表，则要求一级和二级分区方式都相同。

    - 分区对齐规则：对于一级分区表，一级分区 Value 相同的分区聚集在一起；对于二级分区表，一级分区 Value 相同，并且二级分区 Value 相同的分区聚集在一起。

- 分区方式要求：要么全部是一级分区表，要么全部是二级分区表。如果是一级分区表，则要求一级分区的分区方式相同；如果是二级分区表，则要求一级和二级分区方式都相同。
- 分区对齐规则：对于一级分区表，一级分区 Value 相同的分区聚集在一起；对于二级分区表，一级分区 Value 相同，并且二级分区 Value 相同的分区聚集在一起。

了解了 `SHARDING` 属性的取值和意义后，我们以具体示例来看如何使用。

示例一：SHARDING = NONE

`SHARDING = NONE` 的 Table Group 内，无论何种分区方式的表的分区，都会分布在同一个 Partition Group 中（Partition Group 是一个物理概念，可以简单理解成同一个 Partition Group 的数据一定分布在同一台机器上）。

```
SQL> CREATE TABLEGROUP TG1 SHARDING = 'NONE';

# 非分区表
SQL> CREATE TABLE T_NONPART (pk int primary key) tablegroup = TG1;

# 一级分区表
SQL> CREATE TABLE T_PART_2 (pk int primary key)  tablegroup = TG1
     partition by hash(pk)
     partitions 2;

# 二级分区表
SQL> CREATE TABLE T_SUBPART_2_2 (pk int, c1 int, primary key(pk, c1)) tablegroup
= TG1
     partition by hash(pk)
     subpartition by hash(c1) subpartitions 2
     partitions 2;
```

Table	Partition Group
	0
T_NONPART	P0
T_PART_2	P0, P1
T_SUBPART_2_2	P0SP0, P0SP1, P1SP0, P1SP1

示例二：SHARDING = PARTITION

`SHARDING = PARTITION` 的 Table Group 里所有表都会看成是一级分区表，要求所有表的一级分区方式相同，而一级分区属性相同的分区会聚集成一个 Partition Group。

```
SQL> CREATE TABLEGROUP TG1 SHARDING = 'PARTITION';

# 一级分区表
SQL> CREATE TABLE T_PART_2 (pk int primary key) tablegroup = TG1
     partition by hash(pk) partitions 2;

# 二级分区表
SQL> CREATE TABLE T_SUBPART_2_2 (pk int, c1 int, primary key(pk, c1)) tablegroup
= TG1
     partition by hash(pk)
```

```
     subpartition by hash(c1) subpartitions 2
     partitions 2;
```

Table	Partition Group	
	0	1
T_PART_2	P0	P1
T_SUBPART_2_2	P0SP0, P0SP1	P1SP0, P1SP1

示例三：SHARDING = ADAPTIVE

Table Group 要求所有表的一级和二级分区方式完全一致。一级分区表和二级分区表不支持在一个 Table Group 中。

一级分区表的 Table Group：

```
SQL> CREATE TABLEGROUP TG_PART SHARDING = 'ADAPTIVE';

# 一级分区表
SQL> CREATE TABLE T1_PART_2 (pk int primary key) tablegroup = TG_PART
     partition by hash(pk) partitions 2;

# 一级分区表
SQL> CREATE TABLE T2_PART_2 (pk int primary key, c1 int) tablegroup = TG_PART
     partition by hash(pk) partitions 2;
```

Table	Partition Group	
	0	1
T1_PART_2	P0	P1
T2_PART_2	P0	P1

二级分区表的 Table Group：

```
SQL> CREATE TABLEGROUP TG_SUBPART SHARDING = 'ADAPTIVE';

# 二级分区表
SQL> CREATE TABLE T1_SUBPART_2_2 (pk int, c1 int, primary key(pk, c1)) tablegroup
= TG_SUBPAR
     partition by hash(pk)
     subpartition by hash(c1) subpartitions 2
     partitions 2;

# 二级分区表
```

```
SQL> CREATE TABLE T2_SUBPART_2_2 (pk int, c1 int, c2 int, primary key(pk, c1)) tab
legroup = T
     partition by hash(pk)
     subpartition by hash(c1) subpartitions 2
     partitions 2;
```

Table	Partition Group			
	00	01	10	11
T1_SUBPART_2_2	P0SP0	P0SP1	P1SP0	P1SP1
T2_SUBPART_3_3	P0SP0	P0SP1	P1SP0	P1SP1

常见问题

表组（tablegroup）和库（database）是否存在从属关系？

表组（tablegroup）和库（database）不存在从属关系，都是租户下的对象。

常见数据库对象的从属关系大致如下：

- tenant
  - database / tablegroup
    - table
      - index / partition / constraint
- database / tablegroup
  - table
    - index / partition / constraint
- table
  - index / partition / constraint
- index / partition / constraint

序列（sequence）

功能定义

在 OceanBase 数据库中，序列（Sequence）是数据库按照一定规则生成的唯一且通常是递增

的数值。通常被用于生成唯一标识符。

引入原因

OceanBase 数据库存在很多用户，业务原本跑在 DB2 / Oracle 上，但是后续准备选择 MySQL 的技术路线，需要从 DB2 / Oracle 向 Oceanbase 数据库 MySQL 模式的租户进行迁移。

为了降低客户对之前在 DB2 / Oracle 中大量使用 sequence 的业务进行改造的复杂度，OceanBase 数据库在 MySQL 模式下支持了和 Oracle 行为兼容的 sequence 功能。

相关语法

详细介绍可参见官网《OceanBase 数据库》中 参考指南/数据库对象管理/MySQL 模式/创建和管理序列》章节，语法和 Oracle 保持兼容，本文中不再赘述。

适用场景

1. 从 DB2 / Oracle 向 Oceanbase 数据库 MySQL 模式的租户进行迁移的场景。
2. 自增列（increment column）和表绑定的特性无法满足业务使用要求。序列（sequence）不和表绑定，可独立创建，也可跨表使用。
3. 自增列（increment column）没有 CYCLE 能力，达到 MAXVALUE 后会罢工的特性无法满足业务使用要求。序列（sequence）支持循环序列，有 CYCLE 能力。

常见问题

序列（sequence）和自增列（increment column）的异同是什么？

1. 自增列（increment column）和表绑定。序列（sequence）不和表绑定，可独立创建，也可跨表使用。
2. 自增列（increment column）没有 CYCLE 能力。序列（sequence）支持循环序列，有 CYCLE 能力。

```
-- 创建一张含有自增列 id 的表，自增列（increment column）和表强绑定
obclient [test]> CREATE TABLE t1(id bigint not null auto_increment primary key, na
```

```
me varchar(
Query OK, 0 rows affected

obclient [test]> INSERT INTO t1(name) VALUES('A'),('B'),('C');
Query OK, 3 rows affected

obclient [test]> SELECT * FROM t1;
+----+------+
| id | name |
+----+------+
|  1 | A    |
|  2 | B    |
|  3 | C    |
+----+------+
3 rows in set


-- 创建一个序列，起始值是 1，最小值是 1，最大值是 5，步长是 2，序列的值不循环生成
obclient [test]> CREATE SEQUENCE seq1 START WITH 1 MINVALUE 1 MAXVALUE 5 INCREMEN
T BY 2 NOCYC
Query OK, 0 rows affected

obclient [test]> SELECT seq1.nextval FROM DUAL;
+---------+
| nextval |
+---------+
|       1 |
+---------+
1 row in set

obclient [test]> SELECT seq1.nextval FROM DUAL;
+---------+
| nextval |
+---------+
|       3 |
+---------+
1 row in set

obclient [test]> SELECT seq1.nextval FROM DUAL;
+---------+
| nextval |
+---------+
|       5 |
+---------+
1 row in set

-- 如果设置 NOCYCLE，达到 MAXVALUE 后，无法继续生成更大的序列
obclient [test]> SELECT seq1.nextval FROM DUAL;
ERROR 4332 (HY000): sequence exceeds MAXVALUE and cannot be instantiated
```

```
-- 再创建一个序列，起始值是 1，最小值是 1，最大值是 5，步长是 2，序列的值循环生成（在内存中预分配
的自增值个数是 2）
obclient [test]> CREATE SEQUENCE seq7 START WITH 1 MINVALUE 1 MAXVALUE 5 INCREMEN
T BY 2 CYCLE
Query OK, 0 rows affected

obclient [test]> SELECT seq7.nextval FROM DUAL;
+---------+
| nextval |
+---------+
|       1 |
+---------+
1 row in set

obclient [test]> SELECT seq7.nextval FROM DUAL;
+---------+
| nextval |
+---------+
|       3 |
+---------+
1 row in set

obclient [test]> SELECT seq7.nextval FROM DUAL;
+---------+
| nextval |
+---------+
|       5 |
+---------+
1 row in set

obclient [test]> SELECT seq7.nextval FROM DUAL;
+---------+
| nextval |
+---------+
|       1 |
+---------+
1 row in set


-- 序列除了可用于顶层 SELECT，还可用在 INSERT 与 UPDATE 中

obclient [test]> create table t2(c1 int);
Query OK, 0 rows affected

obclient [test]> insert into t2 values(seq7.nextval);
Query OK, 1 row affected

obclient [test]> select * from t2;
+------+
```

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

obclient [test]> update t2 set c1 = seq7.nextval;
Query OK, 1 row affected
Rows matched: 1  Changed: 1  Warnings: 0

obclient [test]> select * from t2;
+-------+
| c1    |
+-------+
|     5 |
+-------+
1 row in set
```

序列和自增列的 ORDER | NOORDER 属性会带来什么影响？

序列和自增列的 ORDER | NOORDER 属性带来的影响相同，下面以仅自增列为例进行说明。

OceanBase 数据库作为分布式数据库，其数据库表通常分布在多台不同的机器上。因为需要保证分布式多机场景下自增列生成的性能，从而会出现自增值生成过程中的跳变问题。

在 OceanBase 数据库中，自增列支持两种自增模式，即 `NOORDER` 模式和 `ORDER` 模式，默认为 `ORDER` 模式。

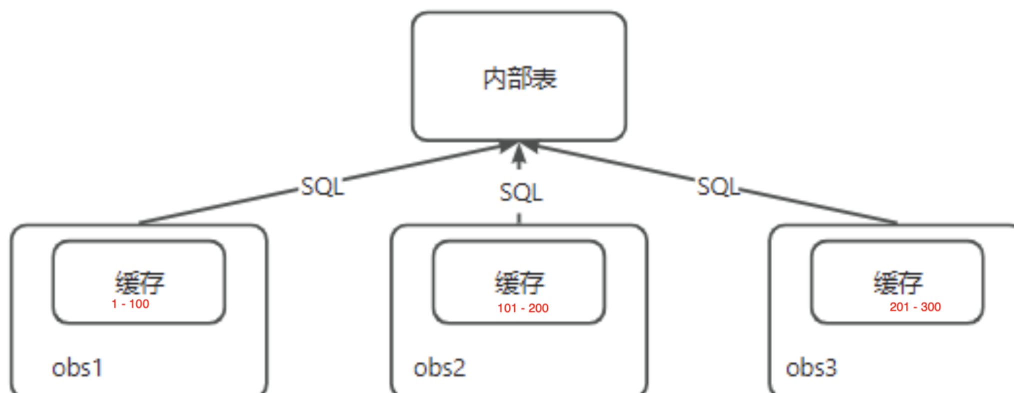
NOORDER 模式

基于分布式缓存的自增列，设置为该模式后，仅保证值的全局唯一。自增值只在节点内局部递增，不保证全局递增。

`NOORDER` 模式的自增列，其数据结构分为以下两部分：

内部表：负责持久化当前已经使用的自增值位点，可以理解成中心节点。

缓存：在各个 OBServer 节点内部的一段自增值区间，OBServer 节点通过向内部表申请获得。

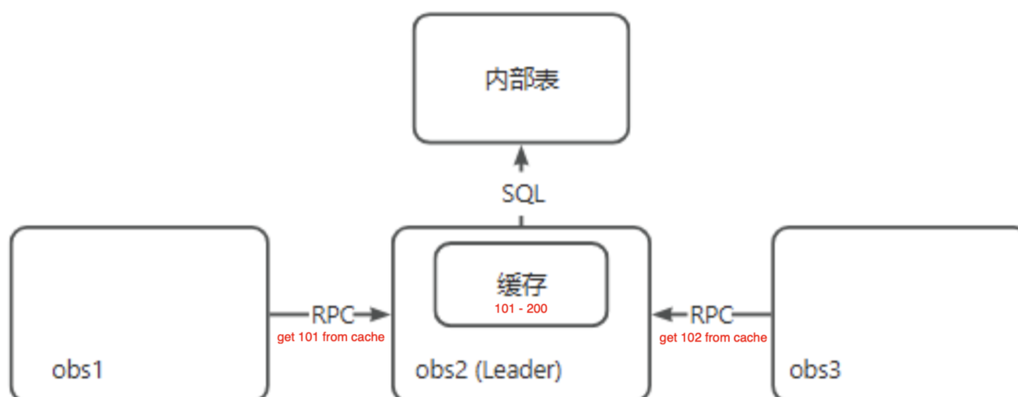


`NOORDER` 模式的自增列中的每一个 OBServer 节点之间均保持独立，各节点可以自主从内部表获取自增区间并记录到机器缓存中。在生产自增值时不需要访问中心节点，直接从本机缓存中获取自增值即可。

ORDER 模式

基于集中缓存的自增列。设置为该模式后，序列和自增列的值全局递增。

对于 `ORDER` 模式的自增列，其内部原理如下图所示：



`ORDER` 模式的自增列会在所有 OBServer 节点中选取当前集群的 Leader（Rootservice 服务所在的节点） 作为自增列服务的 Leader，其它 OBServer 节点需要通过发送 RPC 请求来从作为 Leader 的 OBServer 节点处申请自增值，而作为 Leader 的 OBServer 节点会从内部表申请自增区间来作为自增缓存。

如果为了避免 `NOORDER` 模式中所提到的分布式场景下，在不同的的节点上申请自增值导致的自增值跳变的问题，需要将该属性调整为 `ORDER`。

说明

序列和自增列这里还存在一个区别：

- 创建序列时，默认为 NOORDER 属性（为了和 Oracle 行为兼容）。
- 创建自增列时，默认为 ORDER 属性（为了和 MySQL 行为兼容）。

如果基于性能开销考虑，创建序列时需要如何设置相关属性？

创建序列时，如果设置 `ORDER` 属性，为了保证全局有序，每一次获取 NEXTVALUE 的操作都需要到中心节点去更新一张特定的内部表，在高并发场景下，可能会存在较高的锁冲突。如果不要求序列值递增，只要求唯一，建议将序列的属性设置为 `NOORDER`。

同时，对性能要求较高时，还应该关注 CACHE / NOCACHE 这个属性。

- `NOCACHE`：表示 OBServer 节点内不缓存自增值。这种模式下每次调用 NEXTVAL 都会触发一次内部表 SELECT 与 UPDATE，会影响数据库的性能。
- `CACHE`：用来指定每个 OBServer 节点内存中缓存的自增值个数，默认值为 `20`。

说明

在创建序列时，由于默认的 CACHE 值过小，需要手动声明。单机 TPS 为 100 时，CACHE SIZE 建议设置为 360000。

闪回查询（自学内容）

OceanBase 数据库提供了记录级别的闪回查询（Flashback Query）功能，该功能允许用户获取某个历史版本的数据。

请大家直接参见官网《OceanBase 数据库》中 管理数据库/高可用/闪回查询 部分，完成自学。

复制表（自学内容）

OceanBase 数据库在建表时提供了复制表属性的选项，复制表属性表示可以在任意一个健康的

副本上读取到数据的最新修改。对于写入频率较低、更关心读操作延迟和负载均衡的用户来说，复制表是一个很好的选择。

请大家直接参见社区博客《OceanBase v4.2 复制表特性说明》部分，完成自学。

第七章 OceanBase 数据库的诊断和调优

本章介绍如何对 OceanBase 数据库进行性能诊断和调优。

本章目录

7.1 诊断调优概述 486

7.2 ODP SQL 路由原理 487

7.3 管理 OceanBase 数据库连接 509

7.4 分析 SQL 监控视图 524

7.5 阅读和管理 OceanBase 数据库 SQL 执行计划 551

7.6 常见的 SQL 调优方式 609

7.7 SQL 性能问题的典 675

7.8 通过 SQL Diagnoser 工具进行 SQL 性能诊断和分析 700

7.9 通过 obdiag 工具进行诊断和分析 706

7.1 诊断调优概述

OceanBase 数据库的诊断和调优目的是：对 OceanBase 数据库问题进行诊断分析，优化数据库性能提高数据库的资源利用率，降低业务成本，降低应用系统的运行风险，提高系统稳定性，提供更高效的服务。

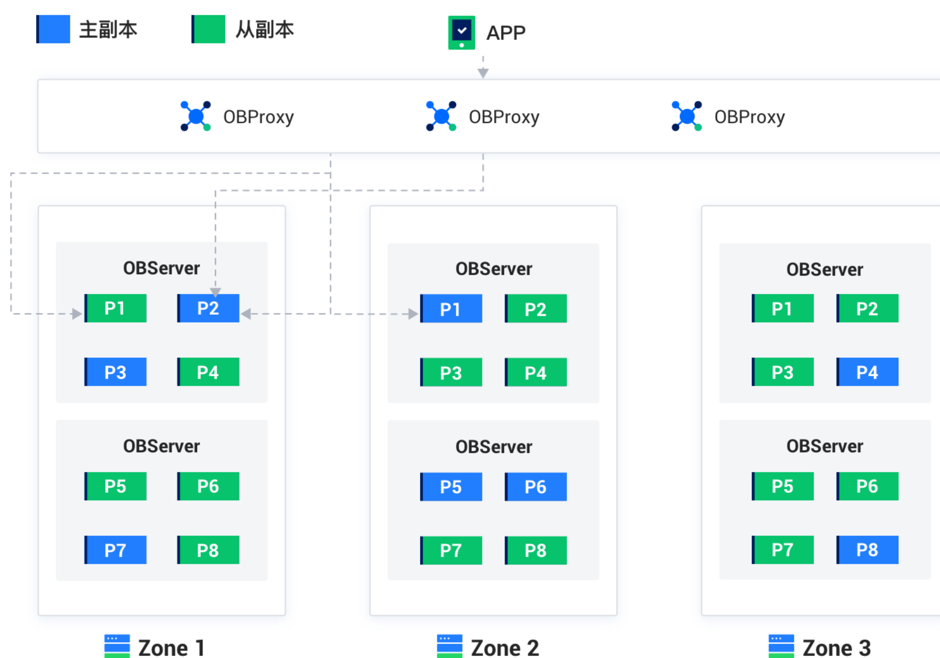
本章主要介绍 OceanBase 诊断调优的相关内容，其中诊断包括了性能诊断/故障诊断，调优主要是指 SQL 调优。具体内容包括：ODP 的 SQL 路由原理、分析 SQL 监控视图、阅读和管理 SQL 执行计划，常见的 SQL 调优方式，SQL 性能问题的典型场景分析以及如果通过 SQL Diagnoser 工具和 obdiag 工具进行诊断调优分析。

7.2 ODP SQL 路由原理

OceanBase 数据库代理 ODP（OceanBase Database Proxy）是 OceanBase 数据库专用的代理服务器，OceanBase 数据库用户的数据会以多副本的形式存放在各个 OBServer 节点上，ODP 接收用户发出的 SQL 请求，并将 SQL 请求转发至最佳目标 OBServer 节点，最后将执行结果返回给用户。

ODP 的功能

ODP 是代理服务器，代理服务器会让访问数据库的链路多一跳，那为什么需要 ODP 呢？我们以下图为例进行说明。



图中 APP 是我们的业务程序，APP 下面有三台 ODP（ODP 的进程名叫做 obproxy），在实际部署中，ODP 和 APP 之间一般会有一个负载均衡（如 F5、LVS 或 Nginx 等）将请求分散到多台 ODP 上面，ODP 下面是 OBServer 节点，图中有 6 个 OBServer 节点。

需要使用 ODP 的原因如下：

- 数据路由

  ODP 可以获取到 OBServer 节点中的数据分布信息，可以将用户 SQL 高效转发到数据所在

机器，执行效率更高。如：表 `t1` 数据在图中 `P1` 内，表 `t2` 数据在图中 `P2` 内，表 `t3` 数据在图中 `P3` 内。对于 `insert into t1` 语句 ODP 可以将 SQL 转发到 `Zone 2` 中含有 `P1` 主副本的机器上。对于 `update t2` 语句 ODP 可以将 SQL 转发到 `Zone 1` 中含有 `P2` 主副本的机器上。

- 连接管理

  如果一个 OceanBase 集群的规模比较大，那么运维机器上、下线以及机器出现问题的概率也会相应增大。如果直连 OBServer 节点，遇到上面的情况，客户端就会发生断连。ODP 屏蔽了 OBServer 节点本身分布式的复杂性，客户连接 ODP，ODP 可以保证连接的稳定性，自身对 OBServer 节点的复杂状态进行处理。

也就是说，ODP 可以实现像使用单机数据库一样使用分布式数据库。

ODP 的特性

作为 OceanBase 数据库的关键组件，ODP 具有如下特性：

- 高性能转发

  ODP 完整兼容 MySQL 协议，并支持 OceanBase 自研协议，采用多线程异步框架和透明流式转发的设计，保证了数据的高性能转发，同时确保了自身对机器资源的最小消耗。

- 最佳路由

  ODP 会充分考虑用户请求涉及的副本位置、用户配置的读写分离路由策略、OceanBase 数据库多地部署的最优链路，以及 OceanBase 数据库各机器的状态及负载情况，将用户的请求路由到最佳的 OBServer 节点上，最大程度的保证了 OceanBase 数据库整体的高性能运转。

- 连接管理

  针对一个客户端的物理连接，ODP 维持自身到后端多个 OBServer 节点的连接，采用基于版本号的增量同步方案维持了每个 OBServer 节点连接的会话状态，保证了客户端高效访问各个 OBServer 节点。

- 安全可信

  ODP 支持使用 SSL 访问数据，并和 MySQL 协议做了兼容，满足客户安全需求。

- 易运维

ODP 本身无状态支持无限水平扩展，支持同时访问多个 OceanBase 集群。可以通过丰富的内部命令实现对自身状态的实时监控，提供极大的运维便利性。

ODP 社区版完全开源，使用 MulanPubL - 2.0 许可证，用户可以免费复制和使用源代码，修改或分发源代码时，请遵守木兰协议。

ODP 路由的性能因素

高性能是 OceanBase 数据库的重要特性，路由对性能的影响主要在网络通信开销方面。ODP 通过感知数据分布和机器地理位置降低网络通信开销，提高整体性能。第七章是介绍 SQL 性能诊断和调优的章节，所以我们这一小节主要会从性能方面对路由策略进行介绍。

我们首先会介绍三个背景知识：

- ODP 路由的实现逻辑是什么？
- SQL 主要的计划类型有哪些？
- 如何查看资源分布信息？

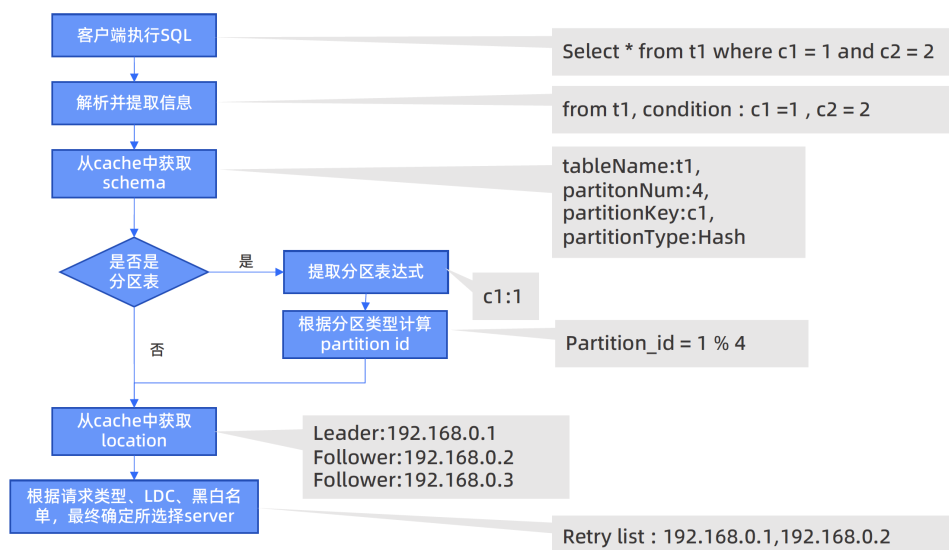
ODP 路由的实现逻辑

路由是 OceanBase 分布式数据库中的一个重要功能，是分布式架构下，实现快速访问数据的利器。

Partition 是 OceanBase 数据存储的基本单元。当我们创建一张 Table 时，就会存在表和 Partition 的映射。非分区表中，一张 Table 对应一个 Partition；分区表中一个 Table 会对应多个 Partition。

路由实现了根据 OBServer 节点的数据分布精准访问到数据所在的机器。同时还可以根据一定的策略将一致性要求不高的读请求发送给副本机器，充分利用机器的资源。路由选择输入的是用户的 SQL、用户配置规则、和 OBServer 节点状态，路由选择输出的是一个可用 OBServer 地址。

其路由实现逻辑如下图所示：



1. 解析 SQL 并提取信息

   解析 SQL 模块使用的是 ODP 自己定制的 Parser 模块，只需要解析出 DML 语句中的数据库名、表名和 Hint，不需要通过其他复杂的表达式推演。

2. 通过 location cache（路由表）获取位置信息

   ODP 根据用户的请求 SQL 获取该 SQL 涉及的副本位置。ODP 每次首先会尝试从本地缓存中获取路由表，其次是全局缓存，如果都没有获取到，最后会发起异步任务去向 OBServer 节点查询路由表。对于路由表的更新，ODP 采用触发更新机制。ODP 每次会把 SQL 根据路由表转发给相应的 OBServer 节点，当 OBServer 节点发现该 SQL 不能在本地执行时，会在回包时反馈给 ODP。ODP 根据反馈决定下次是否强制更新本地缓存路由表。通常是在 OBServer 节点合并或者负载均衡导致切主时，路由表才会发生变化。

3. 确定路由规则

   ODP 需要根据不同情况确定最佳的路由规则。比如：强一致性读的 DML 请求期望发到分区主副本所在的 OBServer 节点上，弱一致性读的 DML 请求和其他请求则不要求，主副本和从副本均衡负载即可。如果 OceanBase 集群是多地部署，ODP 还提供了 LDC 路由，优先发给同机房的 OBServer 节点，其次是同城的 OBServer 节点，最后才是其他城市的 OBServer 节点。如果 OceanBase 集群是读写分离部署，ODP 还提供了读 Zone 优先、只限读 Zone、非合并优先等规则供业务按照自身特点配置。上述的几种情况在路由选择中是组合关系，输出是一个确定的路由规则。

4. 选择目标 OBServer 节点

根据确定的路由规则从获取的路由表中选择最佳的 OBServer 节点，经过黑名单、灰名单检查后，对目标 OBServer 节点进行请求转发。

SQL 的计划类型

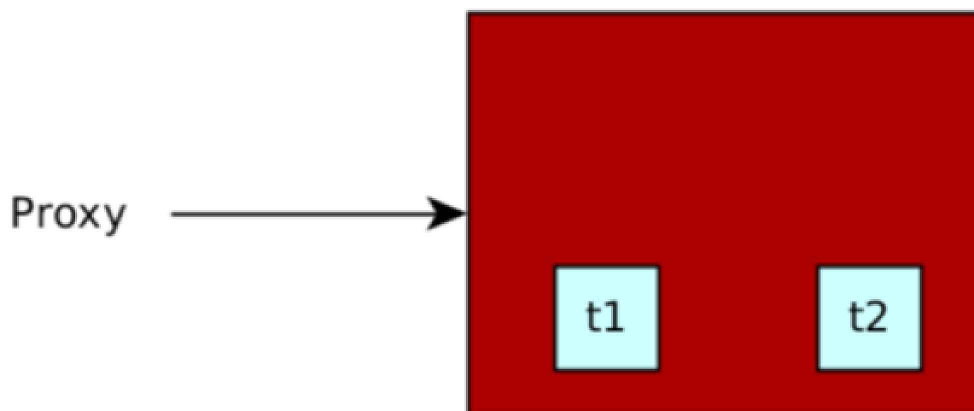
分区（Partition）是数据存储的基本单元，当我们创建表时，就会存在表和分区的映射。如果是非分区表，一张表仅对应一个分区，如果是分区表，一张表可能会对应多个分区。每个分区根据副本角色又分为一个主副本和多个备副本，默认读写的都是分区的主副本。

在 OceanBase 数据库中，SQL 在执行前会生成一个执行计划，执行器根据这个执行计划来调度不同的算子完成计算。计划主要分为三种不同的类型，分别是本地（Local）计划、远程（Remote）计划和分布式（Distributed）计划。

下面以 OceanBase 数据库默认的强一致性读为例，介绍这三种不同的计划类型，即查询过程中需要访问分区的主（Leader）副本。

本地计划

本地计划表示语句所涉及的所有分区的主副本都在当前 Session 所在的 OBServer 节点上，整条 SQL 在执行过程中，不需要和其他 OBServer 节点再进行额外地交互。



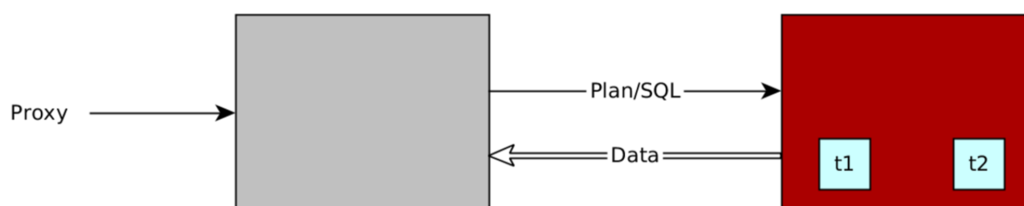
一般来说，本地执行通过 explain 命令看到的计划会长这样：

```
========================================
|ID|OPERATOR   |NAME|EST. ROWS|COST    |
----------------------------------------
|0 |HASH JOIN  |    |98010000 |66774608|
|1 | TABLE SCAN|T1  |100000   |68478   |
|2 | TABLE SCAN|T2  |100000   |68478   |
```

```
========================================
```

远程计划

远程（Remote）计划表示当前语句所涉及的所有分区主副本都与当前 Session 所在的 OBServer 节点不同，且都集中在另外一台 OBServer 节点上，需要 OBServer 节点再对 SQL 或者子计划进行一次转发。



一般来说，远程执行通过 explain 命令看到的计划会长这样（存在 `EXCHANGE REMOTE` 算子）：

```
==================================================
|ID|OPERATOR            |NAME|EST. ROWS|COST     |
--------------------------------------------------
|0 |EXCHANGE IN REMOTE  |    |98010000 |154912123|
|1 | EXCHANGE OUT REMOTE|    |98010000 |66774608 |
|2 |  HASH JOIN         |    |98010000 |66774608 |
|3 |   TABLE SCAN       |T1  |100000   |68478    |
|4 |   TABLE SCAN       |T2  |100000   |68478    |
==================================================
```

上面的计划中，0 号 `EXCHANGE IN REMOTE` 算子和 1 号 `EXCHANGE OUT REMOTE` 算子用于切分在不同 OBServer 节点上做的工作。

0 号算在所在的 OBServer 节点接收到 ODP 路由到的 SQL，负责生成这个远程计划，并把完整 SQL 或者 1 - 4 号算子构成的子计划转发到分区主副本所在的另一个 OBServer 节点。

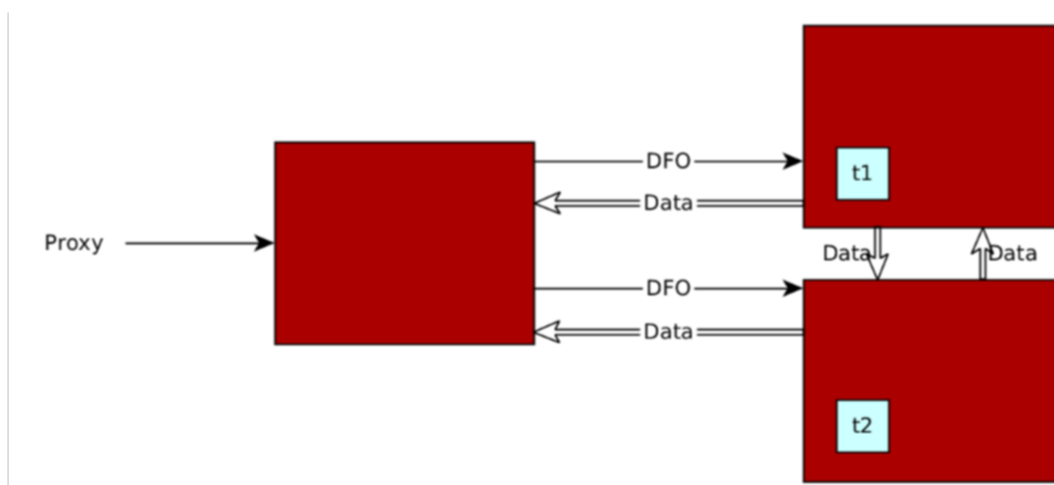
1 - 4 号算子所在的 OBServer 节点负责对 `HASH JOIN` 进行计算，并通过 1 号算子 `EXCHANGE OUT REMOTE` 把计算结果返回给 0 号算在所在的 OBServer 节点。

最后 0 号算在所在的 OBServer 节点通过 `EXCHANGE IN REMOTE` 算子接收这个结果，并把计算结果继续向上层返回。

分布式计划

分布式计划不能确定当前语句涉及到的分区主副本和当前 Session 的关系，往往都是 SQL 需要访问多个分区且多个分区的主副本分布在多个不同的 OBServer 节点上。

分布式计划会使用并行执行的方式进行调度，调度过程中会将其切分成多个操作步骤，每个操作步骤称之为一个 DFO（Data Flow Operation）。



分布式执行通过 explain 命令看到的计划（存在 `EXCHANGE DISTR` 算子）：

```
=================================================================
|ID|OPERATOR                        |NAME    |EST. ROWS|COST      |
-----------------------------------------------------------------
|0 |PX COORDINATOR                  |        |980100000|1546175452|
|1 | EXCHANGE OUT DISTR             |:EX10002|980100000|664800304 |
|2 |  HASH JOIN                     |        |980100000|664800304 |
|3 |   EXCHANGE IN DISTR            |        |200000   |213647    |
|4 |    EXCHANGE OUT DISTR (HASH)|:EX10000|200000   |123720    |
|5 |     PX BLOCK ITERATOR          |        |200000   |123720    |
|6 |      TABLE SCAN                |T1      |200000   |123720    |
|7 |   EXCHANGE IN DISTR            |        |500000   |534080    |
|8 |    EXCHANGE OUT DISTR (HASH)|:EX10001|500000   |309262    |
|9 |     PX BLOCK ITERATOR          |        |500000   |309262    |
|10|      TABLE SCAN                |T2      |500000   |309262    |
=================================================================
```

总的来说，Local 计划和 Remote 计划涉及到的分区主副本都分布在单节点上，ODP 的作用就是尽量消除性能较差的 Remote 计划，将路由尽可能的变为性能更优的 Local 计划。

如果通过 OBServer 节点生成 Remote 计划后再进行转发，在转发前需要对 SQL 进行完整的 parser 和 resolver 流程，还需要在优化器模块对 SQL 进行改写和选择最佳的执行计划等等，而且转发的可能还是一个网络传输代价较大的子计划。

而 ODP 转发 SQL 只需要进行 SQL 解析和简单的分区信息获取即可（参见上面的 ODP 路由的实现逻辑），整体流程相比通过 OBServer 节点生成 Remote 计划后再进行转发要轻量的多。

如果表路由类型为 Remote 计划的 SQL 过多，说明该 SQL 的路由可能存在问题（可通过查看 oceanbase.GV$OB_SQL_AUDIT 视图中 `plan_type` 字段来确认）。查看 SQL 计划类型的相关 SQL 如下:

```
MySQL [oceanbase]> select plan_type, count(1) from gv$ob_sql_audit where
request_time > time_to_usec('2021-08-24 18:00:00') group by plan_type;
```

输出如下:

```
+-----------+----------+
| plan_type | count(1) |
+-----------+----------+
|         1 |    17119 |
|         0 |     9614 |
|         3 |     4400 |
|         2 |    23429 |
+-----------+----------+
4 rows in set
```

其中，plan_type = 1、2、3 分别表示 Local、Remote、Distribute 执行计划。一般来讲，`0` 代表无 plan 的 SQL 语句，比如：`set autocommit=0/1`，`commit` 等。

资源分布信息

ODP 的主要功能是提供 SQL 路由。所以要先了解数据的位置，再了解 SQL 路由策略。

查看租户资源单元位置

ODP 为了把 SQL 准确地路由到最佳的节点上，首先需要知道的就是租户资源所在的节点位置信息（LOCATION CACHE）。有如下 2 种方法可以确认租户的位置:

- 系统（sys）租户直接查询

  ```
  select
    t1.name resource_pool_name,
    t2.`name` unit_config_name,
    t2.max_cpu,
    t2.min_cpu,
  ```

```
  round(t2.memory_size / 1024 / 1024 / 1024) max_mem_gb,
  round(t2.memory_size / 1024 / 1024 / 1024) min_mem_gb,
  t3.unit_id,
  t3.zone,
  concat(t3.svr_ip, ':', t3.`svr_port`) observer,
  t4.tenant_id,
  t4.tenant_name
from
  __all_resource_pool t1
  join __all_unit_config t2 on (t1.unit_config_id = t2.unit_config_id)
  join __all_unit t3 on (t1.`resource_pool_id` = t3.`resource_pool_id`)
  left join __all_tenant t4 on (t1.tenant_id = t4.tenant_id)
order by
  t1.`resource_pool_id`,
  t2.`unit_config_id`,
  t3.unit_id;
```

输出如下：

```
+------------------------------+------------------------------------+---------+----
-----+------------+------------+---------+-------+--------------------+-----------
+---------------+
| resource_pool_name           | unit_config_name                   | max_cpu | min
_cpu | max_mem_gb | min_mem_gb | unit_id | zone  | observer           | tenant_id
| tenant_name   |
+------------------------------+------------------------------------+---------+----
-----+------------+------------+---------+-------+--------------------+-----------
+---------------+
| sys_pool                     | config_sys_zone1_xiaofeng_sys_lpj |       3
|       3 |          6 |          6 |       1 | zone1 | xx.xxx.xx.20:22602
|         1 | sys           |
| pool_for_tenant_mysql        | 2c2g                               |       2
|       2 |          2 |          2 |    1001 | zone1 | xx.xxx.xx.20:22602 |
1002 | mysql         |
| pool_mysql_standby_zone1_xcl | config_mysql_standby_zone1_S1_xic |     1.5
|     1.5 |          6 |          6 |    1002 | zone1 | xx.xxx.xx.20:22602 |
1004 | mysql_standby |
+------------------------------+------------------------------------+---------+----
-----+------------+------------+---------+-------+--------------------+-----------
+---------------+
3 rows in set
```

- 普通用户租户直接查询

```
select * from GV$OB_UNITS where tenant_id=1002;
```

在业务租户里，条件 `tenant_id=1002` 也可以不加，因为每个业务租户只能查看自己的租户

资源单元信息，命令输出如下：

```
+---------------+----------+---------+-----------+-------+-----------+------------+
---------+---------+-------------+---------------------+---------------------+----
---------+--------------+----------------+-------------------+--------+----------
------------------+
| SVR_IP         | SVR_PORT | UNIT_ID | TENANT_ID | ZONE  | ZONE_TYPE | REGION
| MAX_CPU | MIN_CPU | MEMORY_SIZE | MAX_IOPS            | MIN_IOPS            | IO
PS_WEIGHT | LOG_DISK_SIZE | LOG_DISK_IN_USE | DATA_DISK_IN_USE | STATUS | CREATE_T
IME                        |
+---------------+----------+---------+-----------+-------+-----------+------------+
---------+---------+-------------+---------------------+---------------------+----
---------+--------------+----------------+-------------------+--------+----------
------------------+
| 1.2.3.4        |    22602 |    1001 |      1002 | zone1 | ReadWrite | sys_region
|       2 |       2 |  1073741824 | 9223372036854775807 | 9223372036854775807
|           2 |    5798205850 |      4607930545 |         20971520 | NORMAL | 2023
-11-20 11:09:55.668007 |
+---------------+----------+---------+-----------+-------+-----------+------------+
---------+---------+-------------+---------------------+---------------------+----
---------+--------------+----------------+-------------------+--------+----------
------------------+
1 row in set
```

查看分区副本位置

根据上面介绍的 ODP 路由的实现逻辑，ODP 在解析 SQL 获得涉及到的表和分区信息后，需要根据表和分区的信息获得对应分区的主副本的位置信息。

OceanBase 数据库 4.x 版本的副本管理策略是租户级的，即同一个租户下所有表的 Primary Zone 的规则是统一的。在系统租户下可以通过查询 `oceanbase.DBA_OB_TENANTS` 表确定租户信息。

```
select * from oceanbase.DBA_OB_TENANTS;
```

输出如下：

```
+-----------+-------------+-------------+----------------------------+------------
----------------+-------------+--------------+-------------------+-------------
------+--------+---------------+--------+-------------+-------------------+-------
-----------+--------------------+--------------------+----
-----------------+----------------------------+-----------+---------
---+------------+
| TENANT_ID | TENANT_NAME | TENANT_TYPE | CREATE_TIME                | MODIFY_TIM
E                 | PRIMARY_ZONE | LOCALITY      | PREVIOUS_LOCALITY | COMPATIBILIT
```

```
Y_MODE | STATUS | IN_RECYCLEBIN | LOCKED | TENANT_ROLE | SWITCHOVER_STATUS | SWITC
HOVER_EPOCH | SYNC_SCN              | REPLAYABLE_SCN        | READABLE_SCN          | RE
COVERY_UNTIL_SCN   | LOG_MODE      | ARBITRATION_SERVICE_STATUS | UNIT_NUM | COMPATI
BLE | MAX_LS_ID |
+-----------+-------------+-------------+----------------------------+------------
----------------+--------------+---------------+-------------------+-------------
------+--------+--------------+--------+--------------+------------------+-------
-----------+---------------------+---------------------+---------------------+----
-----------------+-------------+----------------------------+----------+---------
---+------------+
|         1 | sys         | SYS         | 2024-04-10 10:48:59.526612 | 2024-04-10
10:48:59.526612 | RANDOM        | FULL{1}@zone1 | NULL              | MYSQ
L               | NORMAL | NO            | NO      | PRIMARY      | NORMA
L             |                 0 |                 NULL |                 NULL
|                NULL |               NULL | NOARCHIVELOG | DISABLE
D                |         1 | 4.2.3.0     |          1 |
|      1001 | META$1002   | META        | 2024-04-10 10:49:30.029481 | 2024-04-10
10:50:27.254959 | zone1         | FULL{1}@zone1 | NULL              | MYSQ
L               | NORMAL | NO            | NO      | PRIMARY      | NORMA
L             |                 0 |                 NULL |                 NULL
|                NULL |               NULL | NOARCHIVELOG | DISABLE
D                |         1 | 4.2.3.0     |          1 |
|      1002 | mysql       | USER        | 2024-04-10 10:49:30.048284 | 2024-04-10
10:50:27.458529 | zone1         | FULL{1}@zone1 | NULL              | MYSQ
L               | NORMAL | NO            | NO      | PRIMARY      | NORMA
L             |                 0 | 1717384184174664001 | 1717384184174664001 | 1717
384184174664001 | 4611686018427387903 | NOARCHIVELOG | DISABLED
|         1 | 4.2.3.0     |       1001 |
+-----------+-------------+-------------+----------------------------+------------
----------------+--------------+---------------+-------------------+-------------
------+--------+--------------+--------+--------------+------------------+-------
-----------+---------------------+---------------------+---------------------+----
-----------------+-------------+----------------------------+----------+---------
---+------------+
3 rows in set
```

在系统租户下，可以通过查询 `oceanbase.cdb_ob_table_locations` 获得各个租户中所有表的各分区位置信息。

```
select * from oceanbase.cdb_ob_table_locations where table_name = 't1';
```

输出如下：

```
+-----------+---------------+------------+----------+------------+---------------
+-------------------+------------+---------------+-----------+-------+-------+----
----------+----------+--------+---------------+------------------+------------+------
-----------+-------------------+----------+
| TENANT_ID | DATABASE_NAME | TABLE_NAME | TABLE_ID | TABLE_TYPE | PARTITION_NAME
```

```
| SUBPARTITION_NAME | INDEX_NAME | DATA_TABLE_ID | TABLET_ID | LS_ID | ZONE  | SVR
_IP         | SVR_PORT | ROLE   | REPLICA_TYPE | DUPLICATE_SCOPE | OBJECT_ID | TABLE
GROUP_NAME | TABLEGROUP_ID | SHARDING |
+-----------+---------------+-------------+----------+-------------+----------------
+-------------------+-------------+---------------+-----------+-------+-------+----
----------+----------+--------+---------------+------------------+-----------+------
-----------+----------------+----------+
|         1 | oceanbase      | t1          |   500010 | USER TABLE | NULL
| NULL              | NULL        |           NULL |     200003 |      1 | zone1 | xx.
xxx.xx.20 |     22602 | LEADER | FULL         | NONE              |     500010 | NUL
L            |           NULL | NULL      |
|      1002 | test           | t1          |   500087 | USER TABLE | NULL
| NULL              | NULL        |           NULL |     200049 |   1001 | zone1 | xx.
xxx.xx.20 |     22602 | LEADER | FULL         | NONE              |     500087 | NUL
L            |           NULL | NULL      |
|      1004 | test           | t1          |   500003 | USER TABLE | NULL
| NULL              | NULL        |           NULL |     200001 |   1001 | zone1 | xx.
xxx.xx.20 |     22602 | LEADER | FULL         | NONE              |     500003 | NUL
L            |           NULL | NULL      |
+-----------+---------------+-------------+----------+-------------+----------------
+-------------------+-------------+---------------+-----------+-------+-------+----
----------+----------+--------+---------------+------------------+-----------+------
-----------+----------------+----------+
3 rows in set
```

在普通租户下，可以通过查询 `oceanbase.dba_ob_table_locations` 获得各个租户中所有表的各分区位置信息。

```
select database_name, table_name, table_id, table_type, zone, svr_ip, role from oc
eanbase.dba_ob_table_locations where table_name = 't1';
```

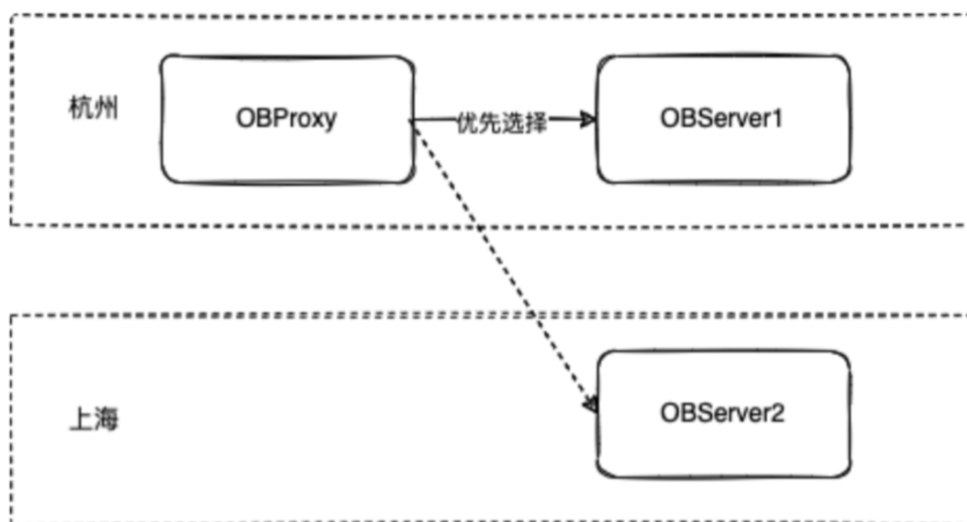
输出如下：

```
+---------------+------------+----------+------------+-------+-------------+-----
---+
| database_name | table_name | table_id | table_type | zone  | svr_ip       | rol
e   |
+---------------+------------+----------+------------+-------+-------------+-----
---+
| test          | t1         |   500087 | USER TABLE | zone1 | xx.xxx.xx.20 | LEAD
ER |
+---------------+------------+----------+------------+-------+-------------+-----
---+
1 row in set
```

查看和调整 LDC 设置

逻辑数据中心（Logical Data Center，LDC）路由可用于解决分布式关系型数据库中多地多中心部署时产生的异地路由延迟问题。

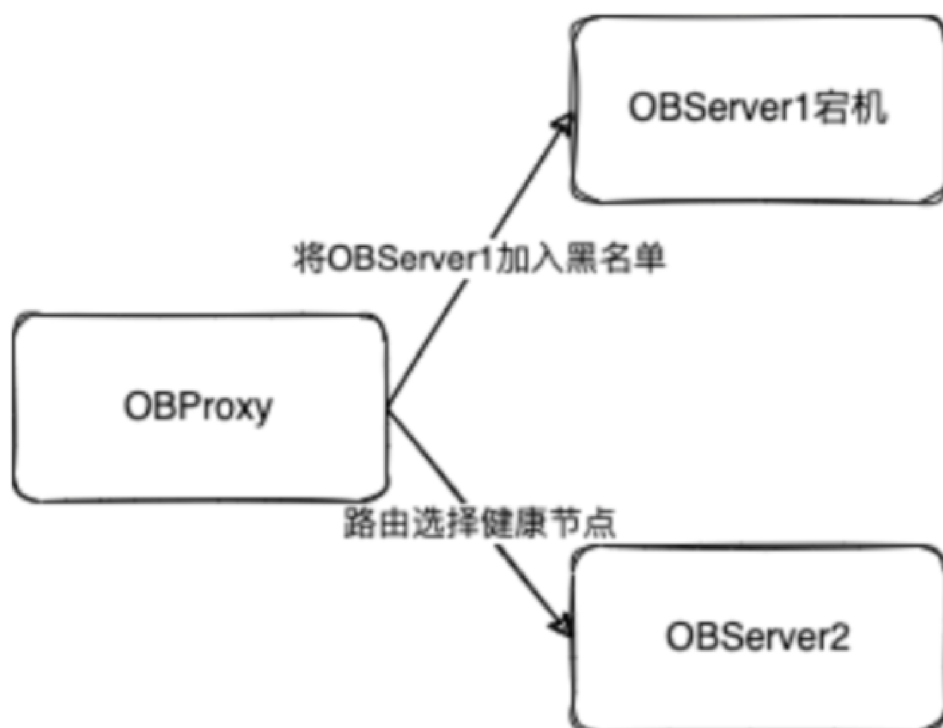
OceanBase 数据库作为典型的高可用分布式关系型数据库，使用 Paxos 协议进行日志同步，天然支持多地多中心的部署方式以提供高可靠的容灾保证。但当真正多地多中心部署时，任何数据库都会面临异地路由延迟问题。逻辑数据中心（Logical Data Center，LDC）路由正是为了解决这一问题而设计的。在为 OceanBase 集群的每个 Zone 设置地区（Region）属性和机房（IDC）属性，并为 ODP 指定机房（IDC）名称配置项的情况下，当数据请求发到 ODP 时，ODP 将按 “同机房 > 同地区 > 异地” 的优先级顺序进行 OBServer 节点的选取。具体的设置方法详见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/路由策略路由。



同时，OceanBase 数据库还支持通过调整系统变量的方式改变默认的路由策略，详见官网《OceanBase 数据库》文档 参考指南/系统原理/数据链路/数据库代理/SQL 路由。

ODP 路由的高可用因素

高可用因素是指 OceanBase 数据库对机器故障有容忍能力，让故障对应用透明无感知，ODP 发现 OBServer 节点故障后，路由时会排除故障节点，选择健康节点，对于正在执行的SQL也有一定的重试能力。高可用涉及故障探测、黑名单机制、重试逻辑等内容。如图所示，ODP 发现 OBServer1 故障后，将该节点加入黑名单。路由时从健康节点选择。



了解了数据路由的影响因素和路由原则后，我们就可以更高效地进行路由策略设计了。不过，现实情况会复杂很多，原则上我们要实时感知 OBServer 节点状态、数据分布等，但在工程实践中很难做到，便引发出许多问题。因此，我们在考虑路由时需要兼顾功能、性能和高可用，让 OceanBase 数据库“更好用”。

ODP 路由的功能和策略

ODP 实现了集群路由、租户路由和租户内路由，通过 ODP 可以访问不同集群的不同租户的不同机器。接下来我们将围绕这三部分介绍 ODP 的路由功能。

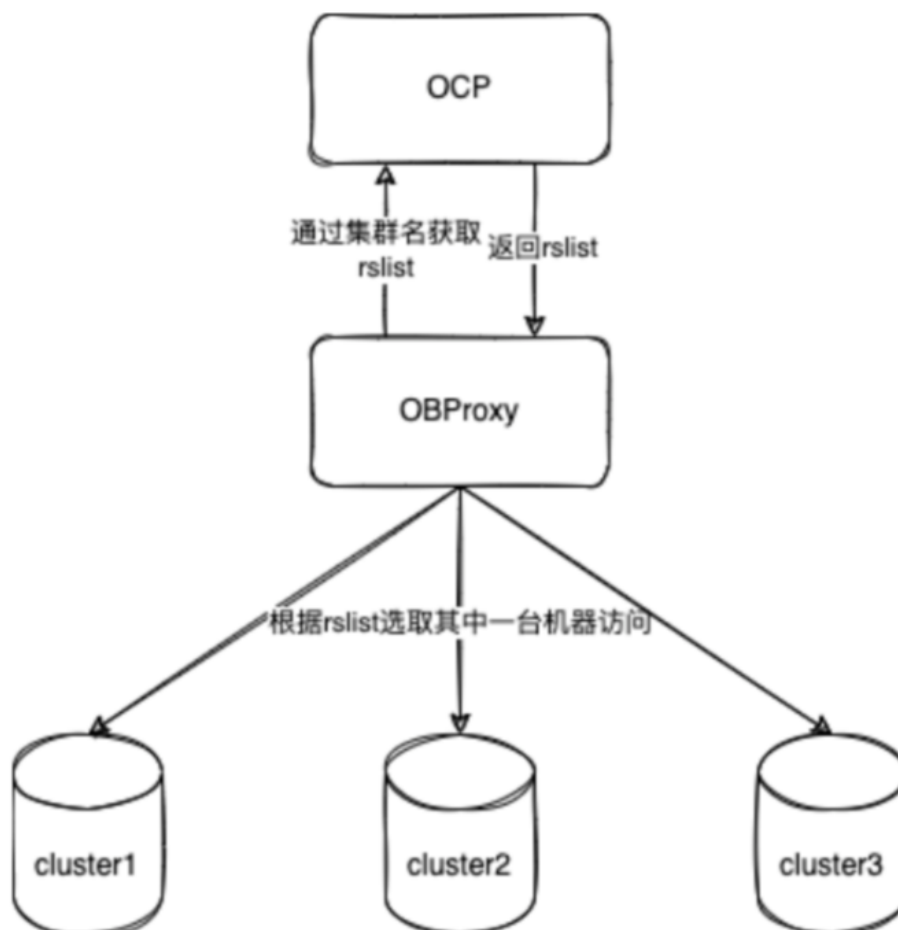
集群路由

集群路由是指 ODP 路由功能支持访问不同的集群，它的关键点在于获取集群名和 rslist（rootservice_list）的映射关系：

- 对于启动参数指定 rslist 的启动方式，集群名和 rslist 的映射关系通过启动参数指定。
- 对于指定 `config_server_url` 的启动方式，集群名和 rslist 的映射关系通过访问 URL 获取。

需要注意的是，这里的 rslist 不需要包含所有的集群机器列表，ODP 会通过访问视图获取集群所

有机器，一般 rslist 为 RootServer（OceanBase 数据库的总控服务）所在的机器。



从上图中可以看到，OCP 是集群路由时非常重要的一个模块。当生产环境中出现集群路由问题时，需着重排查是否是 OCP 模块出现了问题。

ODP 是在用户登录首次访问集群时获取 rslist，并保存到内存中，后续再访问该集群，从 ODP 的内存中获取就可以了。

需要注意的是，因为 ODP 路由功能支持访问不同的集群，所以在命令行中通过 ODP（假设 ODP 的 ip 和 port 分别为 `127.0.0.1` 和 `2883`）进行连接时，除了需要指定用户名 user_name 和租户名 tenant_name 外，还需要额外指定要连接的集群名 cluster_name。如果一些周边工具例如 ODC 等，说明必须通过 ODP 进行连接，在 ODC 等工具中配置连接串时，也需要在连接串中加上集群名 cluster_name，例如：

```
mysql -h 127.0.0.1 -u user_name@tenant_name#cluster_name -P2883 -Ddatabase_name -p
PASSWORD
```

如果在命令行中不通过 ODP 进行连接，而是对 OBServer 节点（假设 OBServer 节点的 ip 和 port 分别为 `1.2.3.4` 和 `12345`）进行直连，那么就只需要指定用户名和租户名，不能再额外指定对应的集群名了。例如：

```
mysql -h 1.2.3.4 -u user_name@tenant_name -P12345 -Ddatabase_name -pPASSWORD
```

租户路由

OceanBase 数据库中，一个集群有多个租户，租户路由是指 ODP 路由功能支持访问不同的租户。在众多租户中，sys 租户比较特殊，类似于管理员租户，和集群管理相关。我们将分开讨论 sys 租户路由和普通租户路由。

sys 租户路由

ODP 完成上述的集群路由后，可获得集群的 rslist，此时 ODP 会通过 proxyro@sys 账号登录 rslist 中的一台机器，并通过视图 `DBA_OB_SERVERS` 获取集群的所有机器节点。

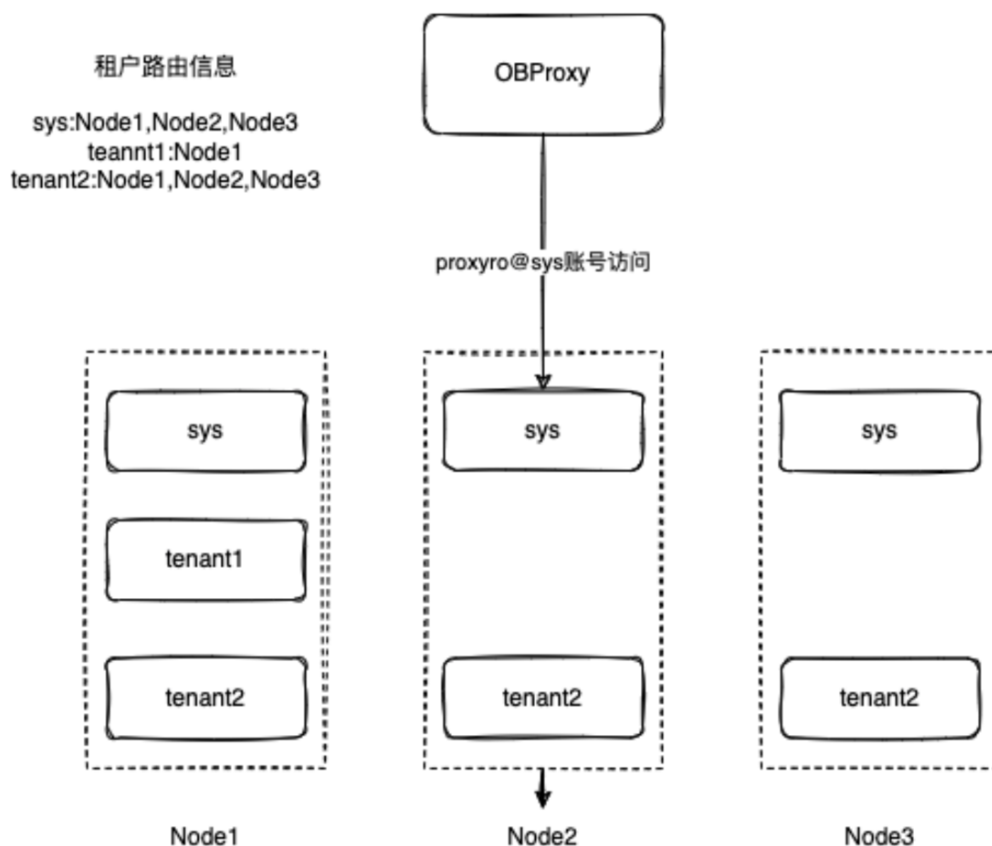
在 OceanBase 数据库的现有实现中，sys 在每个节点都有分布，因此，`DBA_OB_SERVERS` 返回的结果也就是 sys 租户的路由信息。

ODP 会每 15 秒访问一次 `DBA_OB_SERVERS`，维护最新的路由信息，这样可以感知到集群发生的节点变更。除了集群机器列表，ODP 还会通过 sys 租户获取 partition 分布信息、Zone 信息、租户信息等。

普通租户路由

与 sys 租户的路由信息就是集群的机器列表不同，普通租户路由信息是租户资源所在的机器。

ODP 查询租户路由信息并不是通过 unit 相关的表，而是通过特殊表名 `__all_dummy` 表示查询租户信息。ODP 需要通过内部表 `__all_virtual_proxy_schema` 获取租户的机器列表，在访问 `__all_virtual_proxy_schema` 时，ODP 指定表名（`__all_dummy`）和指定租户名获取租户的节点信息。



ODP 会将获取到的租户信息保存在本地内存中，并根据一定策略进行缓存信息的更新。对于 sys 租户，通过每 15 秒一次的拉取任务获得最新的信息；对于普通租户，ODP 的刷新频率并不高，普通租户的路由缓存策略如下：

- 创建：首次访问租户时，通过 `__all_virtual_proxy_schema` 获得普通租户路由信息并创建。
- 淘汰：当 OBServer 节点返回错误码 `OB_TENANT_NOT_IN_SERVER` 时设置缓存失效。
- 更新：当缓存失效后重新访问 `__all_virtual_proxy_schema` 获得普通租户路由信息。

总的来说，在多租户架构下，ODP 通过 sys 租户获得元数据信息（sys 租户本身路由信息就是集群的机器列表），然后通过元数据信息获得租户的路由信息。通过租户路由功能，ODP 支持了 OceanBase 数据库的多租户架构。

租户内路由

强一致性读路由策略

路由策略很多，这里针对部分主要的路由策略由简单到复杂进行列举。

- 强一致性读路由策略将 SQL 路由到访问表的分区的主副本所在节点。这一条理解起来比较简单，但实际 SQL 情形很复杂。
- 如果 SQL 访问了两个表，会依据第一个表及其条件判断出该分区主副本节点。如果无法判断就随机发。所以 SQL 里多表连接时，表的前后顺序对路由策略是有影响的，间接对性能有影响。
- 如果要判断的表是分区表，会判断条件是否是分区键等值条件。如果不是，则不能确定是哪个分区，就随机发到该表的所有分区所在的节点任意一个。
- 如果开启事务了，则事务里开启事务的 SQL 的路由节点会作为事务后面其他 SQL 路由的目标节点，直到事务结束（提交或者回滚）为止。
- 当 SQL 被 ODP 路由分配到一个节点上时:
  - 如果要访问的数据分区的主副本恰好在那个节点上，SQL 就在该节点执行，这个 SQL 的执行类型是本地 SQL（`plan_type` 为 `1`）。
  - 如果要访问的数据分区的主副本不在这个节点上，SQL 会被 OBServer 节点再次转发。这个 SQL 的执行类型是远程 SQL（`plan_type` 为 `2`）。
  - 如果 SQL 执行计划要访问的数据分区是跨越多个节点，则这个 SQL 的执行类型是分布式 SQL（`plan_type` 为 `3`）。
- 如果要访问的数据分区的主副本恰好在那个节点上，SQL 就在该节点执行，这个 SQL 的执行类型是本地 SQL（`plan_type` 为 `1`）。
- 如果要访问的数据分区的主副本不在这个节点上，SQL 会被 OBServer 节点再次转发。这个 SQL 的执行类型是远程 SQL（`plan_type` 为 `2`）。
- 如果 SQL 执行计划要访问的数据分区是跨越多个节点，则这个 SQL 的执行类型是分布式 SQL（`plan_type` 为 `3`）。
- 如果事务中有复制表的读 SQL，只要 SQL 被路由到的节点上有该复制表的备副本，则该 SQL 可以读取本地备副本，因为复制表的所有备副本跟主副本是强一致的。这个 SQL 的执行

类型是本地 SQL。

实际 SQL 类型很复杂，ODP 的路由策略也变得很复杂，有时候会出现路由不准的情形。如果不符合设计预期就可能会产生 BUG，但很可能也是设计如此（BY DESIGN）。毕竟当前版本的 ODP 只能做简单的 SQL 解析，不像 OceanBase 数据库那样做完整的执行计划解析。

当业务 SQL 很多很复杂时，远程 SQL 和分布式 SQL 将会无法避免，这时主要观察远程 SQL 和分布式 SQL 在业务 SQL 中的占比。如果比例很高，整体上业务 SQL 性能都不会很好。此时应尽可能地减少远程 SQL 和分布式 SQL。比如，通过表分组、复制表和 PRIMARY_ZONE 设置等。

弱一致性读路由策略

OceanBase 数据库默认是强一致性读，即写后读立即可见（READ AFTER WRITER）。使用强一致性读策略时，ODP 会优先路由到访问表的分区的主副本节点上。

和强一致性读对立的就是弱一致性读，弱一致性读不要求写后读立即可见。弱一致性读也可以路由到分区的主副本和备副本节点，通常有三副本所在节点可以选。

但是开启弱一致性读后，如果 OceanBase 数据库和 ODP 都开启了 LDC 特性，那么弱一致性读语句的路由策略会改变，弱一致性读语句将按下述顺序进行路由：

- 同一个机房或者同一个 Region 状态不是合并中（merging）的节点。
- 同一 Region 中正在合并的节点。
- 其他 Region 的不在合并的或在合并的节点。

即 ODP 会尽力避开合并中的节点。不过若 OceanBase 集群关闭了轮转合并（参数 `enable_merge_by_turn` 设置为 `false`），合并（major freeze）则是所有节点都开始合并，那么 ODP 也就无法避开合并中的节点。

还有一些 SQL 不是访问数据，而是查看或者设置变量值等。如：

```
set @@autocommit=off
show variables like 'autocommit';
```

这类 SQL 的路由策略则是随机路由。在随机路由策略中，如果 OceanBase 数据库和 ODP 开启了 LDC 设置，也会按照上文的路由顺序进行路由。

弱一致性读通常用在读写分离场景中。不过当租户 PRIMARY_ZONE 为 `RANDOM` 时，租户的所有

分区的主副本也是分散在所有 Zone 下，这时弱一致性读备副本的意义也不是很大。

但是，如果使用了只读副本，只读副本设置为独立的 IDC，然后单独的 ODP 设置为同一个 IDC，则这个 ODP 可以用于只读副本的路由。

其他路由策略

ODP 的路由策略非常丰富，本小节只做大概的介绍。大家仅需了解上述的两个最基本的 ODP 路由策略（强度策略和弱读策略）即可。

ODP 租户内路由的完整策略如下。

- 指定 IP 路由

  通过 ODP 配置项（`target_db_server`）或语句注释指定 OBServer 节点，ODP 会将语句准确路由至指定的 OBServer 节点。此路由功能优先级最高，当指定 IP 时，ODP 会忽略其他的路由功能。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/指定 IP 路由。

- 强读分区表路由

  在强读分区表的语句中提供分区键值、表达式或分区名称，ODP 会将语句准确路由到数据所在分区的主副本 OBServer 节点执行。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/强读分区表路由。

- 强读全局索引表路由

  在强读主表的语句中提供全局索引表的列值、表达式或索引分区名称，ODP 会将语句准确路由到数据所在的索引分区的主副本 OBServer 节点执行。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/强读全局索引表路由。

- 强读复制表路由

  在强读复制表时，ODP 将语句路由至与 ODP 位置关系最近的 OBServer 节点执行。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/强读复制表路由。

- 强读 Primary Zone 路由

  通过 ODP 配置项配置 Primary Zone，ODP 将无法计算路由的强读语句路由至 Primary Zone 内的 OBServer 节点。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路

由/租户内路由/强读 Primary Zone 路由。

- 路由策略路由

  ODP 按照用户配置的路由策略规则进行路由。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/路由策略路由。

- 分布式事务路由

  通过 ODP 配置项（enable_transaction_internal_routing）开启，开启后事务内的语句不受事务开启节点限制，无需强制路由至事务开启的 OBServer 节点。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/分布式事务路由。

- 二次路由

  通过 ODP 配置项开启后，当语句路由至某 OBServer 节点，但未命中分区或者分布式事务无法在该 OBServer 节点执行时，ODP 可以重新进行一次准确路由。详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/二次路由。

- 强制路由

  用户无法控制此行为，由 ODP 决定是否强制路由，主要有以下几种情况。详详细介绍请参见官网《OceanBase 数据库代理》文档 数据路由/租户内路由/强制路由。

  - 非分布式事务路由，事务内语句强制路由至事务开启 OBServer 节点。

  - 会话级临时表路由，对会话级临时表进行查询时，会强制路由至第一次查询临时表的 OBServer 节点。

  - 复用会话路由，当计算路由失败且 enable_cached_server 配置项开启时，ODP 会强制路由到上一次会话所在的 OBServer 节点。

  - CURSOR/PIECES 路由，客户端使用 CURSOR/PIECES 流式获取/上传数据时，所有请求会强制路由至同一 OBServer 节点。

- 非分布式事务路由，事务内语句强制路由至事务开启 OBServer 节点。
- 会话级临时表路由，对会话级临时表进行查询时，会强制路由至第一次查询临时表的

OBServer 节点。

- 复用会话路由，当计算路由失败且 enable_cached_server 配置项开启时，ODP 会强制路由到上一次会话所在的 OBServer 节点。
- CURSOR/PIECES 路由，客户端使用 CURSOR/PIECES 流式获取/上传数据时，所有请求会强制路由至同一 OBServer 节点。

ODP 为租户内路由提供了丰富的功能，用户可选择需要的路由功能进行使用。如在使用过程中遇到问题，可通过路由诊断功能查看 ODP 的路由选取过程，路由诊断的详细介绍可参见官网《OceanBase 数据库代理》文档 运维/路由诊断 章节。

7.3 管理 OceanBase 数据库连接

本小节的内容分为原理介绍、客户端连接（Client Session，即 Client 和 ODP 间的连接）、服务端连接（Server Session，即 ODP 和 OBServer 节点间的连接）三个部分。

原理介绍

背景介绍

ODP 为用户提供了数据库接入和路由功能，直接连接 ODP 就可以正常使用 OceanBase 数据库。在使用数据库功能时，ODP 和 OBServer 节点进行交互，且交互流程透明，连接管理就是该交互过程中的关键点之一。

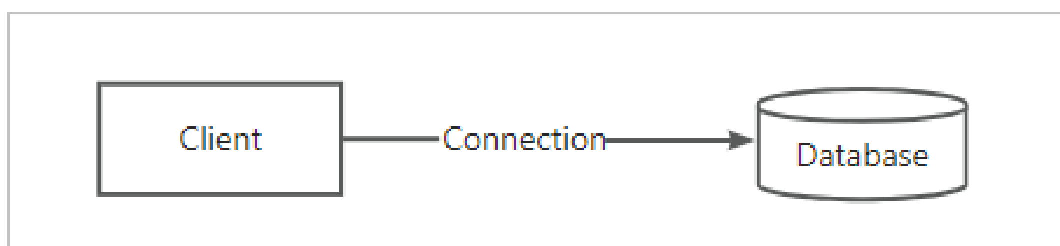
特性

ODP 的连接管理有如下三个特性。

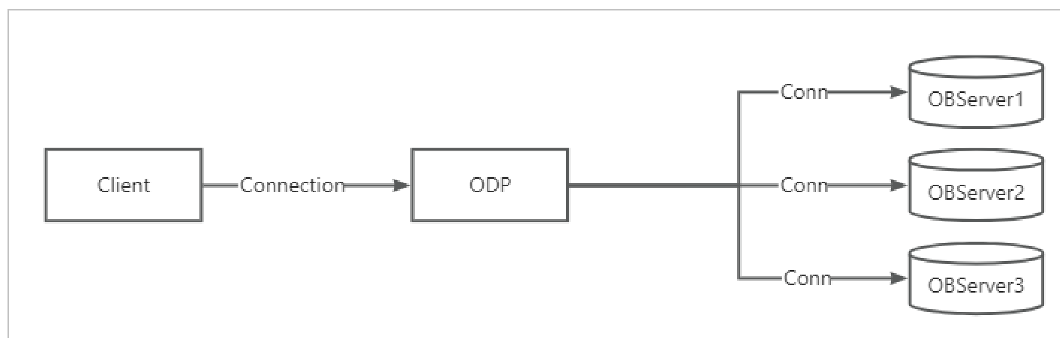
1. 代理特性：ODP 既是客户端，也是服务端，还需要保证交互行为符合 MySQL 协议规范。
2. 功能特性：ODP 实现了很多的连接功能特性，如访问不同集群、不同租户，再如支持主备库、分布式下的 PS 功能，以及兼容 `kill`、`show processlist` 等命令。
3. 高可用特性：ODP 可以处理超时、机器状态变化、网络状态变化等问题，屏蔽后端异常，让用户无感知。

连接映射关系

OceanBase 数据库不同于单机数据库，当通过客户端与单机数据库建立连接时，客户端与数据库之间只有一个物理连接，如下图所示。



当通过 ODP 与 OceanBase 数据库建立连接时，客户端与 ODP 之间存在一个物理连接，而 ODP 与 OBServer 节点间可能存在多个物理连接，如下图所示。其中 Client 与 ODP 的连接称为客户端连接，ODP 与 OBServer 节点间的连接称为服务端连接。



当客户端所访问的数据存在于不同 OBServer 节点上时，ODP 会与 OceanBase 数据库建立多个物理连接并管理复用这些连接，在客户端视角看来仅存在一个逻辑连接。ODP 基于此可以提供多种功能特性，比如主备库分离、读写分离、分区表数据路由、分布式 PS、屏蔽后端异常等。

连接功能特性

ODP 将连接的映射关系改变为 M:N，因此有些连接功能需要额外处理。举例说明：通过 `show processlist` 命令查看连接数时，希望看到的是客户端和 ODP 之间的连接数，而不是 ODP 与 OceanBase 数据库间的连接数。

下面我们对常见的连接功能展开详细介绍。

- 连接粘性：ODP 还未实现所有功能的状态同步，如事务状态、临时表状态、cursor 状态等。对于这些功能，ODP 只会将后续请求都发往状态开始的节点，这样就不需要进行状态同步，而缺点是无法充分发挥分布式系统的优势。因此，ODP 将根据功能重要程度，逐步支持相关功能的分布式化。
- `show processlist` 和 `kill` 命令配套使用：`show processlist` 命令用于展示客户端和服务端之间的连接，对于 ODP 来说，`show processlist` 命令只展示客户端和 ODP 之间的连接，不展示 ODP 和 OBServer 节点之间的连接。`kill` 命令用于断开一个客户端连接，客户端连接关闭后，ODP 也会关闭对应的服务端连接。对于 ODP 的 `kill` 命令，需要先获取对应的 ID（使用 `show processlist` 命令即可获取 ID）。
- 负载均衡影响：因为 ODP 对 `show processlist` 和 `kill` 命令做了处理，所以 `show`

processlist 和 kill 命令只有都发往同一台 ODP 才能正常工作。在公有云等环境，ODP 前面有负载均衡，负载均衡后面挂在多个 ODP 上，此时，如果执行 show prcesslist 和 kill 命令是两个不同的连接，负载均衡组件可能将请求发往不同的 ODP，在这种情况下，建议不要使用相关命令。

客户端连接

这一部分会介绍客户端连接（Client Session，即 Client 和 ODP 间的连接）的一些常用操作。

查看客户端连接

通过 ODP 连接时，在 sys 租户下可以通过 SHOW PROXYSESSION 语句可以查看当前 ODP 上所有租户连接的全部客户端连接的内部状态（sys 租户下执行该命令可以看到和这个集群相关的所有连接，普通租户下可以看到当前连接），示例如下：

```
obclient> show proxysession;
```

输出如下：

```
+--------------------+---------+-------------------------------+--------+------+--
-----------------+-----------+-------------+-------------------+-----------------
--+-------+-------+-----------+
| proxy_sessid       | Id      | Cluster                       | Tenant | User | H
ost              | db        | trans_count | svr_session_count | stat
e            | tid    | pid    | using_ssl |
+--------------------+---------+-------------------------------+--------+------+--
-----------------+-----------+-------------+-------------------+-----------------
--+-------+-------+-----------+
| 838175694068187151 | 1048577 | obn.xiaofeng.lby.123.456.78.9 | sys    | root | 1
23.456.78.9:39012 | oceanbase |           0 |                 1 | MCS_ACTIVE_READE
R | 73180 | 73104 |         0 |
| 838175694068187149 |       5 | obn.xiaofeng.lby.123.456.78.9 | mysql  | root | 1
23.456.78.9:38027 | test      |           0 |                 1 | MCS_ACTIVE_READE
R | 73104 | 73104 |         0 |
| 838175694068187150 |  524297 | obn.xiaofeng.lby.123.456.78.9 | mysql  | root | 1
23.456.78.9:38270 | oceanbase |           0 |                 1 | MCS_ACTIVE_READE
R | 73179 | 73104 |         0 |
+--------------------+---------+-------------------------------+--------+------+--
-----------------+-----------+-------------+-------------------+-----------------
--+-------+-------+-----------+
3 rows in set
```

各字段含义如下表所示：

字段	字段
proxy_sessid	OceanBase 数据库内标记每个与 ODP 的会话的 ID 号
Id	ODP 内标记每个 Client Session 的 ID 号，即下文的cs_id
Cluster	连接所属的 OceanBase 集群名
Tenant	连接 OceanBase 集群所使用的租户名
User	连接 OceanBase 集群所使用的用户名
Host	客户端 IP 地址及其端口号
db	执行命令时所处的数据库
trans_count	客户端会话已完成的事务数量
svr_session_count	ODP 与 OceanBase 数据库之间维持的会话总数量
state	客户端会话状态，存在如下几个状态： • MCS_INIT（初始化） • MCS_ACTIVE_READER（激活） • MCS_KEEP_ALIVE（保活） • MCS_HALF_CLOSE（半关闭） • MCS_CLOSED（已关闭）
tid	线程 ID
pid	进程 ID
using_ssl	客户端会话是否使用 SSL 协议传输

查看客户端连接详细信息

通过 ODP 连接时，可以通过 `SHOW PROXYSESSION ATTRIBUTE` 语句查看指定 Client Session 的详细内部状态，包括该 Client Session 上涉及的相关 Server Session（sys 租户下执行该命令可以看到和这个集群相关的所有连接，普通租户下可以看到当前连接）。

SQL 语句如下：

```
SHOW PROXYSESSION ATTRIBUTE [id [like 'xxx']]
```

参数说明：

- 不指定 `id` 时，显示当前 Session 的详细状态（ODP 1.1.0 版本起开始支持），支持模糊查询当前 Session 指定属性名称的 value（ODP 1.1.2 版本起开始支持）。
- 指定 `id` 时， 支持模糊查询指定属性名称的 value（ODP 1.1.0 版本起开始支持）。
- `id` 既可以是 `cs_id`，也可以是 `CONNECTION_ID`，显示结果相同。

  `cs_id` 为 ODP 内部标记的每个 Client 的 id 号，`CONNECTION_ID` 为整个 OceanBase 数据库标记的每个 Client 的 id 号。MySQL 模式下的 `CONNECTION_ID` 通过 `SELECT CONNECTION_ID();` 语句获取，有关 `CONNECTION_ID` 的详细介绍，请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/函数/信息函数/CONNECTION_ID。
- like 模糊匹配，支持 `%` 和 `_`。

示例如下：

```
obclient> SHOW PROXYSESSION;
+---------------------+------+---------+--------+------+-----------------+------+--
-----------+--
| proxy_sessid        | Id   | Cluster | Tenant | User | Host            | db   | t
rans_count | s
+---------------------+------+---------+--------+------+-----------------+------+--
-----------+--
| 756006681247547396 |    2 |  ob1.cc | sys    | root | 127.0.0.1:22540 | NULL
|           0 |
+---------------------+------+---------+--------+------+-----------------+------+--
-----------+--
1 row in set

obclient> SHOW PROXYSESSION ATTRIBUTE;
+-----------------------------------+----------------------+-----------------+
| attribute_name                    | value                | info            |
+-----------------------------------+----------------------+-----------------+
| proxy_sessid                      | 756006681247547396   | cs common       |
| cs_id                             | 2                    | cs common       |
| cluster                           | ob1.cc               | cs common       |
| tenant                            | sys                  | cs common       |
| user                              | root                 | cs common       |
| host_ip                           | 127.0.0.1            | cs common       |
| host_port                         | 22540                | cs common       |
| db                                | NULL                 | cs common       |
| total_trans_cnt                   | 0                    | cs common       |
| svr_session_cnt                   | 1                    | cs common       |
| active                            | true                 | cs common       |
| read_state                        | MCS_ACTIVE_READER    | cs common       |
……
```

```
# 省略后续输出
39 rows in set

obclient> SHOW PROXYSESSION ATTRIBUTE 2 like '%id%';
+-------------------------+--------------------+----------------+
| attribute_name          | value              | info           |
+-------------------------+--------------------+----------------+
| proxy_sessid            | 756006681247547396 | cs common      |
| cs_id                   | 2                  | cs common      |
| tid                     | 2230520            | cs common      |
| pid                     | 2230520            | cs common      |
| last_insert_id_version  | 0                  | cs var version |
| server_sessid           | 2147549201         | last used ss   |
| ss_id                   | 4                  | last used ss   |
| last_insert_id_version  | 0                  | last used ss   |
+-------------------------+--------------------+----------------+
8 rows in set

obclient> SHOW PROXYSESSION ATTRIBUTE 2147549201 like '%id%';
+-------------------------+--------------------+----------------+
| attribute_name          | value              | info           |
+-------------------------+--------------------+----------------+
| proxy_sessid            | 756006681247547396 | cs common      |
| cs_id                   | 2                  | cs common      |
| tid                     | 2230520            | cs common      |
| pid                     | 2230520            | cs common      |
| last_insert_id_version  | 0                  | cs var version |
| server_sessid           | 2147549201         | last used ss   |
| ss_id                   | 4                  | last used ss   |
| last_insert_id_version  | 0                  | last used ss   |
+-------------------------+--------------------+----------------+
8 rows in set
```

各字段含义如下表所示：

字段	说明
attribute_name	属性名称
value	属性值
info	基本信息

常见的属性及其说明如下表所示：

字段	字段
proxy_sessid	OceanBase 数据库内标记每个与 ODP 的会话的 ID 号
cs_id	ODP 内标记每个 Client Session 的 ID 号，即上文的Id

cluster	连接所属的 OceanBase 集群名
tenant	连接 OceanBase 集群所使用的租户名
user	连接 OceanBase 集群所使用的用户名
host_ip	客户端 IP
host_port	客户端端口号
db	执行命令时所处的数据库
total_trans_cnt	ODP 传输事务的总数量
svr_session_cnt	ODP 与 OceanBase 数据库之间维持的会话总数量
active	是否存活
read_state	客户端会话的状态，存在如下几个状态： • MCS_INIT（初始化） • MCS_ACTIVE_READER（激活） • MCS_KEEP_ALIVE（保活） • MCS_HALF_CLOSE（半关闭） • MCS_CLOSED（已关闭）
tid	线程 ID
pid	进程 ID
modified_time	历史修改时间
reported_time	历史报告时间
hot_sys_var_version	热更新的系统变量版本
sys_var_version	系统变量版本
user_var_version	用户变量版本
last_insert_id_version	最后插入 ID 版本
db_name_version	数据库名的版本
server_ip	OBServer 节点的 IP 地址
server_port	OBServer 节点的端口号
server_sessid	OBServer 节点的会话 ID 号
ss_id	ODP 标记 Server Session 的 ID 号

查看客户端会话变量

通过 `SHOW PROXYSESSION VARIABLES [all] id [like 'xx']` 语句可以查看指定 Client Session 的 Session 变量。

- 不带 `all` 参数时，展示指定 Client Session 的本地 Session 变量（包括：修改过的系统变量和用户变量）。
- 带 `all` 参数时，展示指定 Client Session 的全部 Session 变量（包括：所有系统变量和用户变量）。

参数说明：

- `id` 既可以是 `cs_id`，也可以是 `connection_id`，显示结果相同。
- `cs_id` 为 ODP 内部标记的每个 client 的 id 号，`connection_id` 为整个 OceanBase 数据库标记的每个 client 的 id 号。MySQL 模式下的 `connection_id` 通过 `SELECT CONNECTION_ID();` 语句获取，有关 `CONNECTION_ID` 的详细介绍，请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/函数/信息函数/CONNECTION_ID。
- like 模糊匹配，支持 `%` 和 `_`。

示例如下。

1. 通过 cs_id 查询 Session 变量。

```
obclient> SHOW PROXYSESSION VARIABLES 3;
```

输出如下：

```
+-----------------------------------+-------------------+-----------------+--------
------------+--------------------------------------------+
| variable_name                     | value             | info            | modifie
d_type      | sys_variable_flag                          |
+-----------------------------------+-------------------+-----------------+--------
------------+--------------------------------------------+
| ob_proxy_global_variables_version | 1461742173142100 | changed sys var | cold mo
dified vars |  && invisible && session_scope && readonly |
| ob_proxy_user_privilege           | 65534             | changed sys var | cold mo
dified vars |  && invisible && session_scope && readonly |
| ob_capability_flag                | 654159            | changed sys var | cold mo
dified vars |  && invisible && session_scope && readonly |
| ob_enable_transmission_checksum   | 1                 | changed sys var | cold mo
```

```
dified vars |  && global_scope && session_scope           |
| _min_cluster_version                | '4.1.0.1'          | user var          | cold mo
dified vars |                                             |
+-------------------------------------+------------------+------------------+--------
------------+---------------------------------------------+
```

2. 通过 connection_id 查询 Session 变量。

```
obclient> SELECT CONNECTION_ID();
+-----------------+
| CONNECTION_ID() |
+-----------------+
|      2147549231 |
+-----------------+
1 row in set

obclient> SHOW PROXYSESSION VARIABLES 2147549231;
+-------------------------------------+------------------+------------------+--------
------------+-
| variable_name                       | value            | info             | modifie
d_type      |
+-------------------------------------+------------------+------------------+--------
------------+-
| ob_proxy_global_variables_version | 1461742173142100 | changed sys var | cold mo
dified vars |
| ob_proxy_user_privilege             | 65534            | changed sys var | cold mo
dified vars |
| ob_capability_flag                  | 654159           | changed sys var | cold mo
dified vars |
| ob_enable_transmission_checksum    | 1                | changed sys var | cold mo
dified vars |
| _min_cluster_version                | '4.1.0.1'        | user var         | cold mo
dified vars |
+-------------------------------------+------------------+------------------+--------
------------+-
5 rows in set
```

3. 带 `all` 参数查询 Session 变量。

```
obclient> SHOW PROXYSESSION VARIABLES all 3;
```

输出如下：

```
+-------------------------------------+-----------------------+---------+-----------
-------------------+---------------------------------------------------+
| variable_name                       | value                 | info    | modified_t
ype               | sys_variable_flag                                  |
```

```
+---------------------------------+---------------------+---------+-----------
-------------------+-------------------------------------------------+
| ob_proxy_global_variables_version | 1461742173142100      | sys var | cold modif
ied vars             |  && invisible && session_scope && readonly    |
| ob_proxy_user_privilege           | 65534                 | sys var | cold modif
ied vars             |  && invisible && session_scope && readonly    |
| ob_capability_flag                | 654159                | sys var | cold modif
ied vars             |  && invisible && session_scope && readonly    |
| ob_enable_transmission_checksum   | 1                     | sys var | cold modif
ied vars             |  && global_scope && session_scope             |
| auto_increment_increment          | 1                     | sys var | cold modif
ied vars             |  && global_scope && session_scope             |
| auto_increment_offset             | 1                     | sys var | cold modif
ied vars             |  && global_scope && session_scope             |
......
# 省略后续输出
```

各字段含义如下表所示：

字段	说明
variable_name	变量名
value	变量值
info	变量类型（用户变量或系统变量）
modified_type	变量类型（根据修改频率区分）
sys_variable_flag	系统变量范围

有关系统变量的详细介绍请参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量 章节。有关用户变量设置请参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/SET。

终止客户端连接

用户可以使用 `KILL (cs_id | connection_id)` 命令终止客户端连接，结合示例介绍如何通过 Client Session ID 或 Connection ID 终止连接。

通过指定 `cs_id` 或 `CONNECTION_ID` 来 KILL 当前的 Session 时，当前的连接中断，命令执行成功。使用 `SHOW PROXYSESSION` 查看时，客户端会重新建立 Session 连接，并执行 SQL 显示结果。

通过 Client Session ID 终止连接

1. 通过命令 `SHOW PROXYSESSION` 查询需要终止的客户端会话的 ID（Client Session ID），也就是 `cs_id`。

```
obclient> show proxysession;
```

输出如下，第二列数据（`Id`）表示 Client Session ID。

```
+---------------------+--------+---------+--------+------+------------------+------
+-------------+-------------------+-------------------+------+------+-----------+
| proxy_sessid        | Id     | Cluster | Tenant | User | Host             | db
| trans_count | svr_session_count | state             | tid  | pid  | using_ssl |
+---------------------+--------+---------+--------+------+------------------+------
+-------------+-------------------+-------------------+------+------+-----------+
| 7230691418559283266 |     68 |  ob1.cc | sys    | root | 127.0.0.1:50260 | NULL
|           0 |                 1 | MCS_ACTIVE_READER | 8728 | 8728 |         0 |
+---------------------+--------+---------+--------+------+------------------+------
+-------------+-------------------+-------------------+------+------+-----------+
1 rows in set
```

2. 执行如下 SQL 终止会话。

```
obclient> kill 68;
ERROR 1317 (70100): Query execution was interrupted
```

3. 验证是否终止会话，执行如下 SQL。

```
obclient> select 88;
```

输出显示已失去连接。

```
ERROR 2013 (HY000): Lost connection to MySQL server during query
```

通过 Connection ID 终止连接

1. 执行以下命令获取当前会话的 Connection ID。

```
obclient> select CONNECTION_ID();
```

输出结果如下：

```
+-----------------+
| CONNECTION_ID() |
+-----------------+
|      3221766868 |
+-----------------+
1 row in set
```

2. 执行如下命令终止该会话。

```
obclient> kill 3221766868;
ERROR 1317 (70100): Query execution was interrupted
```

3. 验证是否终止会话，执行如下 SQL。

```
obclient> select 88;
```

输出显示已失去连接。

```
ERROR 2013 (HY000): Lost connection to MySQL server during query
```

服务端连接

这一部分会介绍服务端连接（Server Session，即 ODP 和 OBServer 节点间的连接）的一些常用操作。

查看服务端连接

目前暂不支持直接通过 ODP 查看服务端连接。可以使用 root 用户登录 OceanBase 数据库的 sys 租户查看集群内的所有 Server，之后直连对应的 OBServer 节点查看连接。

详细的操作和输出介绍可参见官网《OceanBase 数据库》文档 参考指南/数据库代理/查看租户会话，此处仅简单介绍一种查看方法。

1. 使用 root 用户登录 OceanBase 数据库的 sys 租户查看集群内的所有 OBServer 节点。

```
obclient> select * from oceanbase.__all_server;
```

输出如下：

```
+----------------------------+----------------------------+------------+----------
+----+------+------------+---------------+--------+-------------------------+-----
-----------------------------------------------------------------------+------------+
--------------------+--------------+----------------+
| gmt_create                 | gmt_modified               | svr_ip     | svr_port
| id | zone | inner_port | with_rootserver | status | block_migrate_in_time | buil
d_version                                                                | stop_time
| start_service_time | first_sessid | with_partition |
+----------------------------+----------------------------+------------+----------
+----+------+------------+---------------+--------+-------------------------+-----
-----------------------------------------------------------------------+------------+
--------------------+--------------+----------------+
| 2023-02-28 15:45:53.230044 | 2023-02-28 15:46:25.577180 | 10.10.10.1 |     2882
|  3 | z3   |       2881 |               1 | ACTIVE |                     0 | 4.1.
0.0_1-703037f0b023c8ffa880258463b25b1735cf27b3(Feb 28 2023 13:21:21) |          0
|   1677570376568330 |            0 |              1 |
| 2023-02-28 15:45:53.197477 | 2023-02-28 15:46:25.534448 | 10.10.10.2 |     2882
|  2 | z2   |       2881 |               0 | ACTIVE |                     0 | 4.1.
0.0_1-703037f0b023c8ffa880258463b25b1735cf27b3(Feb 28 2023 13:21:21) |          0
|   1677570376522994 |            0 |              1 |
| 2023-02-28 15:45:53.113870 | 2023-02-28 15:46:25.098607 | 10.10.10.3 |     2882
|  1 | z1   |       2881 |               0 | ACTIVE |                     0 | 4.1.
0.0_1-703037f0b023c8ffa880258463b25b1735cf27b3(Feb 28 2023 13:21:21) |          0
|   1677570378084150 |            0 |              1 |
+----------------------------+----------------------------+------------+----------
+----+------+------------+---------------+--------+-------------------------+-----
-----------------------------------------------------------------------+------------+
--------------------+--------------+----------------+
3 rows in set
```

2. 选择想要查看的 OBServer 节点，直连该 OBServer 节点，以直连 `10.10.10.1`，SQL 端口为 `2881` 为例。

```
[admin@test001 ~]$ obclient -h 10.10.10.1 -P2881 -uroot@sys -p -Doceanbase -A
```

3. 执行 `show processlist;` 命令查看当前 OBServer 节点的所有连接。

```
obclient> show processlist;
```

输出如下：

```
+------------+---------+------------------+----------+---------+------+--------+-
-----------------+
| Id         | User    | Host             | db       | Command | Time | State  |
```

```
Info             |
+------------+---------+-------------------+-----------+---------+------+--------+-
-----------------+
| 3221812197 | root    | 10.10.10.1:48563 | oceanbase | Query   |    0 | ACTIVE |
show processlist |
| 3222117829 | proxyro | 10.10.10.1:37876 | oceanbase | Sleep   |    6 | SLEEP  |
NULL             |
| 3221709618 | root    | 10.10.10.1:51390 | oceanbase | Sleep   |  831 | SLEEP  |
NULL             |
+------------+---------+-------------------+-----------+---------+------+--------+-
-----------------+
3 rows in set
```

终止服务端连接

ODP 目前暂时未提供可以终止服务端连接的命令，可以通过直连 OBServer 节点终止服务端连接。

1. 使用 root 用户登录 OceanBase 数据库的 sys 租户查看集群内的所有 Server 节点。

```
obclient> select * from oceanbase.__all_server;
```

输出如下：

```
+-----------------------------+-----------------------------+------------+----------
+----+------+------------+-----------------+--------+------------------------+-----
---------------------------------------------------------------------+-----------+
--------------------+--------------+-----------------+
| gmt_create                  | gmt_modified                | svr_ip     | svr_port
| id | zone | inner_port | with_rootserver | status | block_migrate_in_time | buil
d_version                                                            | stop_time
| start_service_time | first_sessid | with_partition |
+-----------------------------+-----------------------------+------------+----------
+----+------+------------+-----------------+--------+------------------------+-----
---------------------------------------------------------------------+-----------+
--------------------+--------------+-----------------+
| 2023-02-28 15:45:53.230044 | 2023-02-28 15:46:25.577180 | 10.10.10.1 |     2882
|  3 | z3   |       2881 |               1 | ACTIVE |                      0 | 4.1.
0.0_1-703037f0b023c8ffa880258463b25b1735cf27b3(Feb 28 2023 13:21:21) |         0
|   1677570376568330 |            0 |               1 |
| 2023-02-28 15:45:53.197477 | 2023-02-28 15:46:25.534448 | 10.10.10.2 |     2882
|  2 | z2   |       2881 |               0 | ACTIVE |                      0 | 4.1.
0.0_1-703037f0b023c8ffa880258463b25b1735cf27b3(Feb 28 2023 13:21:21) |         0
|   1677570376522994 |            0 |               1 |
| 2023-02-28 15:45:53.113870 | 2023-02-28 15:46:25.098607 | 10.10.10.3 |     2882
|  1 | z1   |       2881 |               0 | ACTIVE |                      0 | 4.1.
```

```
0.0_1-703037f0b023c8ffa880258463b25b1735cf27b3(Feb 28 2023 13:21:21) |         0
|   1677570378084150 |             0 |                1 |
+-----------------------------+-----------------------------+------------+----------
+----+------+------------+-----------------+--------+-----------------------+-----
-------------------------------------------------------------------+-----------+
--------------------+--------------+----------------+
3 rows in set
```

2. 直连 OBServer 节点，以直连 `10.10.10.1`，SQL 端口为 `2881` 为例。

```
[admin@test001 ~]$ obclient -h10.10.10.1 -P2881 -uroot@sys -p -Doceanbase -A
```

3. 执行 `show processlist;` 命令查看当前 OBServer 节点的所有连接。

```
obclient> show processlist;
```

输出如下：

```
+------------+---------+------------------+-----------+---------+------+--------+-
-----------------+
| Id         | User    | Host             | db        | Command | Time | State  |
Info             |
+------------+---------+------------------+-----------+---------+------+--------+-
-----------------+
| 3221812197 | root    | 10.10.10.1:48563 | NULL      | Query   |    0 | ACTIVE |
show processlist |
| 3222117829 | proxyro | 10.10.10.1:37876 | oceanbase | Sleep   |    6 | SLEEP  |
NULL             |
| 3221709618 | root    | 10.10.10.1:51390 | NULL      | Sleep   |  831 | SLEEP  |
NULL             |
+------------+---------+------------------+-----------+---------+------+--------+-
-----------------+
3 rows in set
```

4. 以终止当前连接（`Id` 为 `3221812197`）为例，使用 `kill <id>` 命令。

```
obclient> kill 3221812197;
ERROR 2013 (HY000): Lost connection to MySQL server during query
```

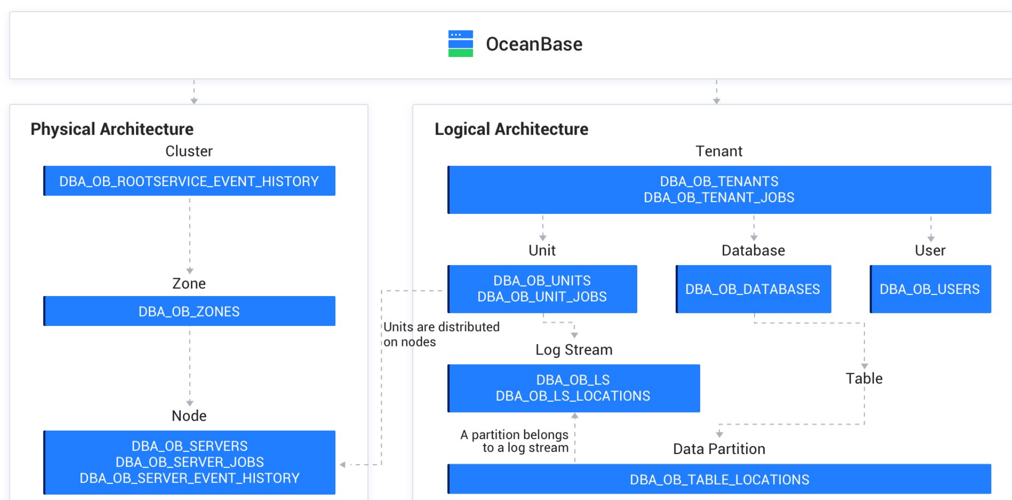
7.4 分析 SQL 监控视图

系统视图概述

OceanBase 数据库 4.x 版本中有着非常丰富的视图，通过这些视图可以获取 OceanBase 集群各种数据库对象的基本信息和实时状态信息。这些视图分为两大类：数据字典视图和动态性能视图。

- 数据字典视图展示系统租户管理的数据库对象的基本信息。命名特点以 `DBA_` 和 `CDB_` 开头。`DBA_` 开头的视图展示的是各个租户内的信息，比如 `DBA_OB_LS` 视图展示的是租户内的日志流信息，系统租户内的 `DBA_OB_LS` 表只展示系统租户自身的日志流信息。`CDB_` 开头的视图是系统租户专用的，用于在系统租户内查看集群所有租户的数据库状态，比如 `CDB_OB_LS` 视图展示的是集群所有租户的所有日志流信息。一般每个 `DBA_` 开头的视图在系统租户下都有对应的 `CDB_` 开头的视图。
- 动态性能视图展示系统动态变化的状态信息。命名特点以 `GV$` 和 `V$` 开头。`V$` 只展示登录到的节点上的信息，`GV$` 展示所有节点的信息。用户租户下的动态性能视图可以查看本租户的信息，比如 `GV$OB_UNITS` 视图可以在用户租户内查询到本租户的 Unit 资源分配信息。系统租户的动态性能视图可以查看整个集群所有租户的信息。

丰富的视图展示了 OceanBase 数据库的内部架构，及系统运行的详细状态。通过视图可以方便的查看 OceanBase 数据库的系统组成及实时状态，了解组件之间的关系，内部视图是学习 OceanBase 数据库的最好途径之一，其相应的数据字典视图如下图所示：



监控指标相关的数据来源于 OceanBase 数据库内部的动态性能视图，所有监控指标都可以通过 SQL 语句进行访问。动态性能视图分为 `GV$` 视图和 `V$` 视图，外部监控系统（例如 OCP）通过在每个数据库服务器上部署代理进程，通过 SQL 接口定期拉取本机上的监控信息（`V$` 视图），部分全局信息（例如 Root Service 相关）通过中心节点采集。监控数据统一汇报给监控系统数据库，并按照各种维度聚合（集群维度、租户维度、节点维度、Unit 维度），从而构建整个监控大盘。

说明

- 监控指标的数据来源于 OceanBase 数据库的视图，但日常工作中应该尽量使用可视化的外部监控系统，可以更加方便地分析监控指标的变化趋势，提高效率。
- 由于 GV$ 视图和 V$ 视图的同质性，本小节不再专门对其进行区分，将根据具体场景选择合适的视图。

OceanBase 数据库常用的动态性能视图见如下列表：

视图	展示内容
GV$SYSSTAT	系统统计信息。
GV$SYSTEM_EVENT	租户级等待事件统计。
GV$SESSION_EVENT	会话级等待事件统计。
GV$LATCH	锁统计信息。
GV$OB_MEMORY	内存信息。
GV$OB_PROCESSLIST	会话信息。

GV$OB_TRANSACTION_PARTICIPANTS	活跃事务的参与者信息。
GV$OB_SQL_AUDIT	SQL 统计信息。
GV$SQL_PLAN_MONITOR	SQL 算子级统计信息。
GV$OB_PLAN_CACHE_STAT	Plan Cache 统计信息。
GV$OB_PLAN_CACHE_PLAN_STAT	执行计划统计信息。
GV$OB_PLAN_CACHE_PLAN_EXPLAIN	执行计划详细信息。
GV$OB_SERVERS	节点的信息。
GV$OB_UNITS	Unit 的信息。
GV$OB_PARAMETERS	配置项信息。
GV$OB_KVCACHE	KVCache 信息。
GV$OB_SSTABLES	SSTable 信息。
GV$OB_MEMSTORE	MemStore 信息。
GV$OB_LOG_STAT	日志流的信息。

监控指标可以从多个角度进行分类。

- 根据监控指标的类型分类，可以分为系统监控和 SQL 监控。系统监控描述系统的运行状况，从而对系统（集群、租户、会话）的健康状态作出判断，系统监控又可以细分为监控项、等待事件、锁事件等。SQL 监控描述 SQL 相关的监控信息，记录 SQL 在执行过程中的资源消耗和等待事件，从而对一个具体的 SQL 问题进行诊断。
- 根据监控指标的粒度分类，可以分为租户级监控和会话级监控。租户级监控指标以租户 ID 和节点 IP 来索引，外部监控系统定期采集每台服务器上按租户粒度汇总的监控数据，并根据不同场景的需要，按照不同维度聚合展示（例如集群维度、租户维度、节点维度、Unit 维度）。租户级监控是监控大盘的数据来源。会话级监控指标以会话 ID 和 Trace ID 来索引，可以对特定的会话进行全链路诊断，是全链路跟踪的重要基础。
- 根据监控指标描述的对象分类，可以分为若干个大类的监控，包括事务、KVCache、CLog、存储、资源、SSTable、MemStore 等内部组件。基于 OceanBase 数据库丰富的监控指标，可以在故障诊断时逐步逼近根因；可以实时跟踪各组件的健康状况，及时发现风险隐患，避免发酵到影响数据库 SLA。

本小节主要介绍的监控手段均属于 SQL 监控，即根据特定 SQL 的多次执行或特定执行中收集的监控信息对系统的性能问题进行诊断。系统监控部分的内容详见官网《OceanBase 数据库》文档 管理数据库/监控指标/监控指标/系统监控 章节。

SQL 监控概述

SQL 相关的监控信息，记录 SQL 在执行过程中的资源消耗，这些信息通常在一条 SQL 执行的前后，通过收集监控项和等待事件的统计信息，并计算差值来获得。除此之外，SQL 监控还包含 SQL 文本、SQL 计划、执行反馈等信息，这些信息在诊断一个具体的 SQL 问题时会非常有帮助。

OceanBase 数据库提供了多种视图用于 SQL 诊断，主要视图如下：

视图	展示内容
GV$OB_SQL_AUDIT	SQL 统计信息。
GV$OB_PLAN_CACHE_PLAN_STAT	执行计划统计信息。
GV$OB_PLAN_CACHE_PLAN_EXPLAIN	执行计划详细信息。
DBA_OB_OUTLINES	普通 Outline 信息。
DBA_OB_CONCURRENT_LIMIT_SQL	限流 Outline 信息。
GV$OB_PLAN_CACHE_STAT	Plan Cache 统计信息。
GV$SQL_PLAN_MONITOR	SQL 算子级统计信息。
GV$SESSION_EVENT	会话级等待统计。
GV$OB_TRANSACTION_PARTICIPANTS	活跃事务的参与者信息。

SQL Audit 监控视图

`GV$OB_SQL_AUDIT` 是最常用的 SQL 监控视图，能够记录每一次 SQL 请求的来源、执行状态、资源消耗及等待事件，无论 SQL 执行是成功还是失败。除此之外还记录了 SQL 文本、执行计划等关键信息。该视图是诊断 SQL 问题的利器。

`GV$OB_SQL_AUDIT` 视图的数据存放在一个可配置的内存空间中，每个租户在每个节点上都有一块

独立的缓存，当内存使用或记录数达到淘汰上限时会触发自动淘汰，最久的数据优先淘汰，类似于一个内存中的 FIFO 队列。有经验的 DBA 在排查 SQL 性能抖动问题时，往往第一件事就是关闭 SQL Audit 功能以保存现场，避免抖动现场的监控数据被淘汰。

SQL Audit 相关设置

通过以下参数可以控制 SQL Audit 功能的行为：

- `enable_sql_audit`：集群配置项，默认值为 `True`，控制全部租户的 SQL Audit 功能是否开启，动态生效。详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/集群级别配置项/enable_sql_audit。
  - 查看 `enable_sql_audit` 的值输出如下：

    ```
    *************************** 1. row ***************************
             zone: zone1
         svr_type: observer
           svr_ip: 1.2.3.4
         svr_port: 12345
             name: enable_sql_audit
        data_type: NULL
            value: True
             info: specifies whether SQL audit is turned on. The default valu
    e is TRUE. Value: TRUE: turned on FALSE: turned off
          section: OBSERVER
            scope: CLUSTER
           source: DEFAULT
       edit_level: DYNAMIC_EFFECTIVE
    default_value: true
        isdefault: 1
    1 row in set
    ```

  - 修改 `enable_sql_audit` 的值

    ```
    -- 开启整个集群的 SQL Audit 功能
    ALTER SYSTEM SET enable_sql_audit = true;

    -- 关闭整个集群的 SQL Audit 功能
    ALTER SYSTEM SET enable_sql_audit = false;
    ```

- 查看 `enable_sql_audit` 的值

  ```
  show parameters like 'enable_sql_audit'\G
  ```

输出如下：

```
*************************** 1. row ***************************
         zone: zone1
     svr_type: observer
       svr_ip: 1.2.3.4
     svr_port: 12345
         name: enable_sql_audit
    data_type: NULL
        value: True
         info: specifies whether SQL audit is turned on. The default value is TRUE
. Value: TRUE: turned on FALSE: turned off
      section: OBSERVER
        scope: CLUSTER
       source: DEFAULT
   edit_level: DYNAMIC_EFFECTIVE
default_value: true
    isdefault: 1
1 row in set
```

- 修改 `enable_sql_audit` 的值

> **说明**
>
> enable_sql_audit 仅支持在 sys 租户下修改，对整个集群生效。

```
-- 开启整个集群的 SQL Audit 功能
ALTER SYSTEM SET enable_sql_audit = true;

-- 关闭整个集群的 SQL Audit 功能
ALTER SYSTEM SET enable_sql_audit = false;
```

- `ob_enable_sql_audit`：租户级别的系统变量，默认值为 `ON`，控制当前租户是否开启 SQL Audit 功能，动态生效。详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Global 系统变量/ob_enable_sql_audit。
  - 查看 `ob_enable_sql_audit` 的值输出如下：

```
+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| ob_enable_sql_audit | ON    |
+---------------------+-------+
1 row in set
```

```
```

  - 修改 `ob_enable_sql_audit` 的值

```
-- 关闭当前租户的 SQL Audit 功能，对命令执行成功之后新创建的连接生效
SET global ob_enable_sql_audit = OFF;

-- 开启当前租户的 SQL Audit 功能，对命令执行成功之后新创建的连接生效
SET global ob_enable_sql_audit = ON;
```

- 查看 `ob_enable_sql_audit` 的值

```
show variables like 'ob_enable_sql_audit';
```

输出如下：

```
+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| ob_enable_sql_audit | ON    |
+---------------------+-------+
1 row in set
```

- 修改 `ob_enable_sql_audit` 的值

```
-- 关闭当前租户的 SQL Audit 功能，对命令执行成功之后新创建的连接生效
SET global ob_enable_sql_audit = OFF;

-- 开启当前租户的 SQL Audit 功能，对命令执行成功之后新创建的连接生效
SET global ob_enable_sql_audit = ON;
```

- `ob_sql_audit_percentage`：租户级别的系统变量，控制当前租户 SQL Audit 功能占用租户内存的百分比，默认值为 `3`（即百分之三），动态生效。为防止 SQL Audit 占用过高内存，系统设置其内存上限为 1 GB。详细介绍可参见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/系统变量/Global 系统变量/ob_sql_audit_percentage。
  - 查看 `ob_sql_audit_percentage` 的值输出如下：

```
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| ob_sql_audit_percentage | 3     |
+-------------------------+-------+
1 row in set
```

```

```

- 调整当前租户的 `ob_sql_audit_percentage`，对命令执行成功之后新创建的连接生效

```
SET global ob_sql_audit_percentage = 5;
```

- 查看 `ob_sql_audit_percentage` 的值

```
show variables like 'ob_sql_audit_percentage';
```

输出如下：

```
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| ob_sql_audit_percentage | 3     |
+-------------------------+-------+
1 row in set
```

- 调整当前租户的 `ob_sql_audit_percentage`，对命令执行成功之后新创建的连接生效

```
SET global ob_sql_audit_percentage = 5;
```

SQL Audit 淘汰机制

租户的后台任务每隔 1 s 会根据 OBServer 节点和 SQL Audit 的内存使用情况来决定是否触发淘汰 SQL。SQL Audit 最大可用内存是通过 `ob_sql_audit_percentage` 控制的。当 SQL Audit 的实际内存使用达到指定阈值时，满足触发淘汰的条件，开启淘汰；当 SQL Audit 的实际内存使用降到指定阈值时，满足停止淘汰的条件，停止淘汰。

SQL Audit 淘汰机制如下表所示：

触发机制	SQL Audit 内存上限	触发淘汰的条件	停止淘汰的条件
内存使用	[0, 64 MB)	内存上限 * 50%	0 MB
内存使用	[64 MB, 100 MB)	内存上限 - 20 MB	内存上限 - 40 MB
内存使用	[100 MB, 5 GB)	内存上限 * 80%	内存上限 * 60%
内存使用	[5 GB, +∞)	内存上限 - 1 GB	内存上限 - 2 GB
记录数	无	900 万条	800 万条

此外，SQL Audit 支持手动淘汰，粒度为租户级别，手动淘汰设置方式如下：

```
alter system flush sql audit tenant = <tenant_name>;
```

GV$OB_SQL_AUDIT 视图字段介绍

`GV$OB_SQL_AUDIT` 视图的字段很多，我们在本小节中只介绍其中一部分比较重要的字段。`GV$OB_SQL_AUDIT` 视图的完整字段介绍，详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 优化/SQL 执行性能监控 中 **GV$OB_SQL_AUDIT 字段说明**。

- `TENANT_ID`：请求的租户 ID。
- `SVR_IP`：接受请求的服务端节点 IP。
- `CLIENT_IP`：发送请求的客户端 IP。
- `REQUEST_TIME`：请求到达时间。
- `REQUEST_ID`：请求的 ID，标识请求的一次执行，随时间而递增。外部监控系统可以将该字段作为游标拉取审计信息。
- `IS_INNER_SQL`：是否内部 SQL 请求。
- `IS_EXECUTOR_RPC`：当前请求是否 RPC 请求。
- `SQL_ID`：标识一条具体的 SQL，同一条 SQL 的多次执行具有相同的 `SQL_ID` 和不同的 `REQUEST_ID`。
- `QUERY_SQL`：SQL 语句完整的文本。OceanBase 数据库支持通过 `SQL_ID` 和 `SQL_TEXT` 绑定执行计划。
- `SID`：标识一个 Session，可以关联该连接上的所有 SQL，及等待事件。对应 `GV$OB_PROCESSLIST` 视图中的 `ID` 字段。
- `TX_ID`：标识一个事务，可以关联该事务的所有 SQL。如果一个事务中连续执行的两条 SQL 的起止时间存在较大差距，说明链路上的耗时较大。
- `TRACE_ID`：该 SQL 请求的 trace ID，可以关联查询其他监控指标，也可以关联查询日志。
- `IS_HIT_PLAN`：是否命中 Plan Cache 中的 Plan。SQL 优化比较耗时，为了避免反复执行 SQL 优化，会将生成的计划放进 Plan Cache，再次执行时直接从 Plan Cache 获取 Plan。

未命中 Plan Cache 的情况称为硬解析，命中 Plan Cache 的情况称为软解析。硬解析会影响 SQL 性能，增加耗时。如果租户 Plan Cache 命中率过低，需要确认 Plan Cache 空间是否过小导致 Plan 频繁被淘汰。

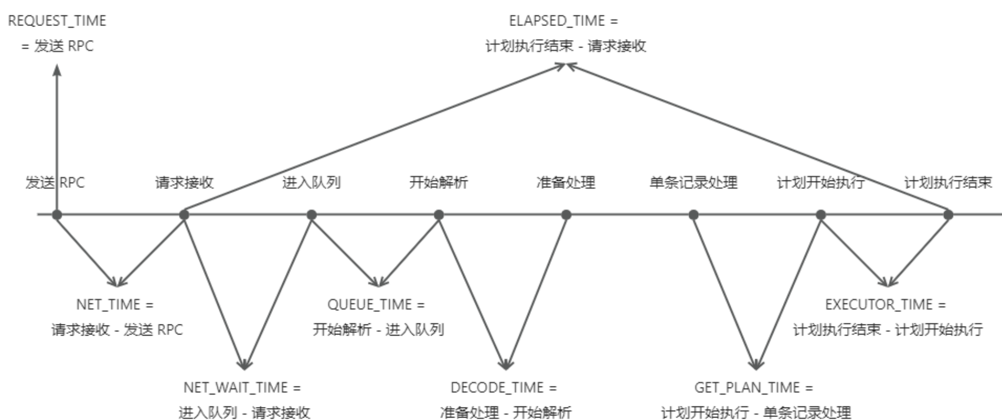
- `PLAN_ID`：执行计划 ID，可以关联查询该计划的详细信息和统计信息。
- `PLAN_HASH`：执行计划的 Hash 值。
- `PLAN_TYPE`：执行计划的类型，取值为 `0`、`1`、`2`、`3`，其中：`1`、`2`、`3` 分别对应本地计划、远程计划、分布式计划；`0` 表示无执行计划，例如 commit 语句。
- `AFFECTED_ROWS`：影响行数。
- `RETURN_ROWS`：返回行数。
- `RET_CODE`：执行结果返回码。
- `EVENT`：最长等待事件名称。
- `P1TEXT ~ P3TEXT`：等待事件参数 1 ~ 3。
- `P1 ~ P3`：等待事件参数的值 1~3。
- `LEVEL`：等待事件的 level 级别。
- `WAIT_CLASS_ID`：等待事件所属的类别 ID。
- `WAIT_CLASS`：等待事件所属的类别名称。
- `STATE`：等待事件的状态。
- `WAIT_TIME_MICRO`：该等待事件所等待的时间（微秒）。
- `TOTAL_WAIT_TIME_MICRO`：执行过程所有等待的总时间（微秒）。
- `TOTAL_WAITS`：执行过程总等待的次数。
- `ELAPSED_TIME`：本次执行的总时间（从请求到达到执行结束），由多个子阶段组成。
- `NET_TIME`：发送 RPC 到接收到请求的时间。
- `NET_WAIT_TIME`：接收到请求到进入队列时间。
- `QUEUE_TIME`：队列时间，反映当前租户的请求积压情况。
- `DECODE_TIME`：出队列后 Decode 时间。
- `GET_PLAN_TIME`：生成执行计划的时间，反映当前租户 Plan Cache 的健康状况。

- `EXECUTE_TIME`：计划的执行时间，由 CPU 时间和 `TOTAL_WAIT_TIME_MICRO` 组成。`TOTAL_WAIT_TIME_MICRO` 由 `APPLICATION_WAIT_TIME`、`CONCURRENCY_WAIT_TIME`、`USER_IO_WAIT_TIME`、`SCHEDULE_TIME` 等几个部分组成。`EXECUTE_TIME` 与 `TOTAL_WAIT_TIME_MICRO` 的差值即为 `CPU_TIME`。
- `APPLICATION_WAIT_TIME`：所有 application 类事件的总时间。
- `CONCURRENCY_WAIT_TIME`：所有 concurrency 类事件的总时间。
- `USER_IO_WAIT_TIME`：所有 user_io 类事件的总时间。
- `SCHEDULE_TIME`：所有 Schedule 类事件的时间，单位：微秒。
- 逻辑读：请求执行中读取数据时会首先读取各级缓存（对应 `ROW_CACHE_HIT`、`BLOOM_FILTER_CACHE_HIT`、`BLOCK_CACHE_HIT` 等字段），如果全部没有命中会产生一次实际的磁盘读取（对应 `DISK_READS` 字段）。通过统计缓存读取次数和磁盘读取次数，可以得出该请求执行过程中扫描行数的多少（扫描行数不等于实际物理读，会首先扫描各级缓存），从而判断该 SQL 是否需要优化。
- `ROW_CACHE_HIT`：行缓存命中次数。
- `BLOOM_FILTER_CACHE_HIT`：bloom filter 缓存命中次数。
- `BLOCK_CACHE_HIT`：块缓存命中次数。
- `BLOCK_INDEX_CACHE_HIT`：块索引缓存命中次数。
- `DISK_READS`：物理读次数。
- `RETRY_CNT`：重试次数。
- `TABLE_SCAN`：判断该请求是否含全表扫描。
- `CONSISTENCY_LEVEL`：事务一致性级别，取值如下：
  - `-1`：无效
  - `1`：指定读存储在 SSTable 中的数据
  - `2`：弱一致性读

    - `3`：强一致性读

- `-1`：无效
- `1`：指定读存储在 SSTable 中的数据
- `2`：弱一致性读
- `3`：强一致性读
- `MEMSTORE_READ_ROW_COUNT`：MemStore 中读的行数。
- `SSSTORE_READ_ROW_COUNT`：SSStore 中读的行数。
- `REQUEST_MEMORY_USED`：该请求消耗的内存。

一些重要的事件间隔



GV$OB_SQL_AUDIT 视图使用示例

通过 `GV$OB_SQL_AUDIT` 视图，我们可以方便的查询 SQL 执行的各种维度信息，下面就列举几个常用的相关 SQL。

- 慢 SQL 统计

  查询一段时间内，耗时超过某个阈值的 SQL，后续可以根据查询的结果对特定 SQL 进行调优。此处以查询 `tenant id` 为 `1002` 的租户，在 `2024-02-20 12:00:00` 之后，query 执行时间超过 100 ms 的前五条 SQL 为例：

```
select
  tenant_id,
```

```
  request_id,
  usec_to_time(request_time),
  elapsed_time,
  queue_time,
  execute_time,
  query_sql
from
  oceanbase.GV$OB_SQL_AUDIT
where
  tenant_id = 1002
  and elapsed_time > 100000
  and request_time > time_to_usec('2024-02-20 12:00:00')
order by
  elapsed_time desc
limit
  5;
```

输出如下：

```
+-----------+------------+----------------------------+--------------+------------
+--------------+-------------------------------+
| tenant_id | request_id | usec_to_time(request_time) | elapsed_time | queue_time
| execute_time | query_sql                     |
+-----------+------------+----------------------------+--------------+------------
+--------------+-------------------------------+
|      1002 |   21371247 | 2024-02-20 16:14:10.008111 |       153118 |         34
|       139873 | select * from xxxx where xxxx |
+-----------+------------+----------------------------+--------------+------------
+--------------+-------------------------------+
1 row in set
```

- 事务内涉及的 SQL 统计

sql_audit 里的每条 SQL 预期都记录了当前 SQL 所涉及的事务唯一标识 transaction_hash，可以根据该字段找到当前事务内所有的 SQL 信息，进而基于该信息确定业务压测的事务模型是否符合预期等。此处以查询 `tenant_id` 为 `1` 的 sys 租户，在 `tx_id = '27592485'` 这个事务中，执行了哪些 query 为例：

```
select
  tenant_id,
  tx_id,
  query_sql
from
  oceanbase.GV$OB_SQL_AUDIT
where
  tenant_id = 1
```

```
  and tx_id = '27592485'
order by
  request_time asc;
```

输出如下:

```
+-----------+----------+------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------+
| tenant_id | tx_id    | query_sq
l

     |
+-----------+----------+------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------+
|         1 | 27592485 | START TRANSACTIO
N
                                                                                 |
|         1 | 27592485 | SELECT column_value FROM __all_core_table WHERE TABLE_NAM
E = '__all_global_stat' AND COLUMN_NAME = 'snapshot_gc_scn' FOR UPDAT
E                                                        |
|         1 | 27592485 | UPDATE __all_core_table SET column_value = 17084168438966
58521 WHERE table_name = '__all_global_stat' AND column_name = 'snapshot_gc_scn' A
ND column_value < 1708416843896658521 |
|         1 | 27592485 | COMMI
T

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

- 同一个 Session 上执行的 SQL

基于 SID 可以查询出某一个 Session 上所有的业务请求，通常被用来分析业务模型。此处以查询 `tenant id` 为 `1002` 的租户，`2024-02-20 12:00:00` 之后，在 `sid = 3221652146` 这个 session 中执行过哪些 query 为例：

```
select
  tenant_id,
  sid,
  query_sql
from
  oceanbase.GV$OB_SQL_AUDIT
where
```

```
  tenant_id = 1002
  and sid = 3221652146
  and request_time > time_to_usec('2024-02-20 12:00:00')
order by
  request_time asc;
```

输出如下:

```
+-----------+------------+--------------------------------------------------------------------------------------------------------------------------------------------+
| tenant_id | sid        | query_sql                                                                                                                                  |
+-----------+------------+--------------------------------------------------------------------------------------------------------------------------------------------+
|      1002 | 3221652146 | set _show_ddl_in_compat_mode = 1;                                                                                                          |
|      1002 | 3221652146 | set autocommit=1                                                                                                                           |
|      1002 | 3221652146 | set names utf8                                                                                                                             |
|      1002 | 3221652146 | SELECT @@max_allowed_packet,@@system_time_zone,@@time_zone,@@auto_increment_increment,@@tx_isolation AS tx_isolation,@@session.tx_read_only AS tx_read_only |
|      1002 | 3221652146 | select @@version_comment, @@version, ob_version() limit 1                                                                                  |
|      1002 | 3221652146 | show variables like 'version_comment'                                                                                                      |
|      1002 | 3221652146 | SHOW DATABASES                                                                                                                             |
+-----------+------------+--------------------------------------------------------------------------------------------------------------------------------------------+
7 rows in set
```

- 查询最近 1000 条 SQL 的平均排队时间。

```
SELECT
  --+ query_timeout(30000000)
  avg(queue_time)
```

```
FROM
  oceanbase.GV$OB_SQL_AUDIT
WHERE
  request_id > (
    SELECT
      max(request_id)
    FROM
      v$OB_SQL_AUDIT
  ) - 1000;
```

输出如下:

```
+------------------+
| avg(queue_time) |
+------------------+
|          0.2960 |
+------------------+
1 row in set
```

- 查询占用租户资源最多的 SQL

如果该租户当前存在容量不足（租户 CPU 使用率被打爆），可以通过如下语句判断是否为 SQL 问题，及具体的疑点 SQL 是什么。输出按照 `执行时间*执行次数` 降序排序。

```
select
  SQL_ID,
  avg(ELAPSED_TIME),
  avg(QUEUE_TIME),
  avg(ROW_CACHE_HIT + BLOOM_FILTER_CACHE_HIT + BLOCK_CACHE_HIT + DISK_READS) avg_l
ogical_read,
  avg(execute_time) avg_exec_time,
  count(*) cnt,
  avg(execute_time - TOTAL_WAIT_TIME_MICRO) avg_cpu_time,
  avg(TOTAL_WAIT_TIME_MICRO) avg_wait_time,
  WAIT_CLASS,
  avg(retry_cnt),
  QUERY_SQL
from
  oceanbase.GV$OB_SQL_AUDIT
group by
  SQL_ID
order by
  avg_exec_time * cnt desc
limit
  10;
```

输出如下：

```
+-----------------------------------+-------------------+-----------------+--------
----------+---------------+------+--------------+---------------+------------+----
------------+--------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
----------------------------+
| SQL_ID                            | avg(ELAPSED_TIME) | avg(QUEUE_TIME) | avg_log
ical_read | avg_exec_time | cnt  | avg_cpu_time | avg_wait_time | WAIT_CLASS | avg
(retry_cnt) | QUERY_SQ
L





      |
+-----------------------------------+-------------------+-----------------+--------
----------+---------------+------+--------------+---------------+------------+----
------------+--------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
----------------------------+
| 1532BA78C664771E7113567D8E951B51 |         4330.2720 |          0.0000
|          0.0000 |     4138.0498 |  261 |    4138.0498 |          0.0000 | OTHE
R      |          0.0000 | SELECT FIELD FROM `oceanbase`.`__tenant_virtual_table_co
lumn` WHERE TABLE_ID = 2000
1




             |
| 1D0BA376E273B9D622641124D8C59264 |          323.2321 |          0.0000
|          0.0000 |      224.9437 |  711 |     224.9437 |          0.0000 | OTHE
R      |          0.0000 | COMMI
T





         |
| 9050622DE000F09344C9A882DAF34A75 |          472.4209 |          0.0000
|          0.0000 |      265.2209 |  575 |     265.2209 |          0.0000 | OTHE
R      |          0.0000 | select * from __all_ls where 1=1 and status != 'DROPPED
' and status != 'CREATE_ABORT' order by ls_i
d
```

```
                                                                                 |
| 19A6B821DB3DE80D69EE8880D3A45C5F |       160857.0000 |         38.0000
|          0.0000 |   137214.0000 |    1 |  137214.0000 |         0.0000 | OTHE
R      |         0.0000 | select * from GV$OB_SQL_AUDIT where SQL_ID = '1D0BA376E2
73B9D622641124D8C59264
'


                  |
| 37B3D4D55DC217FA1BCA9031DC3AF1DC |        34309.2500 |         37.2500
|          0.0000 |    28792.0000 |    4 |   28792.0000 |         0.0000 | OTHE
R      |         0.0000 | select   SQL_ID,   avg(ELAPSED_TIME),   avg(QUEUE_TIME)
,   avg(ROW_CACHE_HIT + BLOOM_FILTER_CACHE_HIT + BLOCK_CACHE_HIT + DISK_READS) avg
_logical_read,   avg(execute_time) avg_exec_time,   count(*) cnt,   avg(execute_ti
me - TOTAL_WAIT_TIME_MICRO) avg_cpu_time,   avg(TOTAL_WAIT_TIME_MICRO) avg_wait_ti
me,   WAIT_CLASS,   avg(retry_cnt) from   GV$OB_SQL_AUDIT group by   SQL_ID order
by   avg_exec_time * cnt desc limit   10 |
| 735537F7B5DB7C4E0E946C9B26108560 |          615.7653 |          0.0000
|          0.0000 |      395.2419 |  277 |     395.2419 |         0.0000 | OTHE
R      |         0.0000 | SELECT * FROM __all_spatial_reference_systems WHERE (SRS
_ID < 70000000 AND SRS_ID != 0) OR SRS_ID > 200000000
0


                                                                       |
| 2EAAA3C495632AB97F13659E429FC9CD |          464.6715 |          0.0000
|          0.0000 |      240.4224 |  277 |     240.4224 |         0.0000 | OTHE
R      |         0.0000 | select * from __all_balance_task where parent_list is nu
ll or parent_list = '
'


                   |
| B7A6FA97FEC98C06F9586D23935AC4C6 |          276.7934 |          0.0000
|          0.0000 |      104.0799 |  576 |     104.0799 |         0.0000 | OTHE
R      |         0.0000 | START TRANSACTIO
N


                                                                                  |
| 556CEFD22F28DDDCB6E283DF89BD9348 |         2263.9643 |          0.0000
|          0.0000 |     2103.5714 |   28 |    2103.5714 |         0.0000 | OTHE
R      |         0.0000 | UPDATE __all_core_table SET column_value = 1708415231707
450537 WHERE table_name = '__all_global_stat' AND column_name = 'snapshot_gc_scn'
AND column_value < 170841523170745053
7

   |
```

```
| 17605A1DA6B6A2150E9FBCA5D4C7653A |         797.5532 |          0.0000
|           0.0000 |       594.5532 |   94 |      594.5532 |          0.0000 | OTHE
R       |          0.0000 | SELECT row_id, column_name, column_value FROM __all_core
_table WHERE table_name = '__all_global_stat' ORDER BY row_id, column_nam
e

                                                  |
+-----------------------------------+-------------------+-----------------+--------
----------+----------------+------+--------------+---------------+------------+----
------------+-----------------------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
------------------------------------------------------------------------------------
----------------------------+
10 rows in set
```

租户抖动时，表现一般是“租户 CPU 被打满”和“所有 SQL RT 飙升”，此时首要判断 SQL RT 飙升是否是第一案发现场（第一案发现场指的是 SQL 自己的问题导致 RT 飙升，除此以外，还可能是其他问题导致 SQL RT 飙升）。

上面介绍的这条 SQL 是分析这类问题一大利器，基于 `SQL_ID` 做聚合，按照资源占用量倒序排序（资源占用量可以认为是 `avg_exec_time * cnt`），观察最上面的几条 top SQL 是否存在明显异常，从而可以判断出是否由于某条 SQL 导致的租户抖动。

通过上面这条 SQL 进行问题定位的案例，详见官网《OceanBase 数据库》文档 管理数据库/监控指标/案例。

执行计划监控视图

OceanBase 数据库为了避免反复对同一条 SQL 生成执行计划，会将生成的计划放进 Plan Cache，再次对相同的 SQL 生成计划时，会直接从 Plan Cache 获取之前已经生成好的 Plan。每个租户在每个节点上都有一个独立的 Plan Cache，用以缓存在此节点上处理过的 SQL Plan。

Plan Cache 相关视图包含 Plan Cache 的统计信息、执行统计的统计信息及计划信息，这些视图的数据均来自 Plan Cache。相关视图包括：

- `GV$OB_PLAN_CACHE_STAT`：每个租户在每个节点上都有一个独立的 Plan Cache，用以缓存在此节点上处理过的 SQL Plan。该视图记录每个 Plan Cache 的统计信息，每个 Plan Cache

都有一条记录。

- GV$OB_PLAN_CACHE_PLAN_STAT：记录 Plan Cache 中计划的详细信息及其执行统计信息，每个计划都一条记录。
- GV$OB_PLAN_CACHE_PLAN_EXPLAIN：记录 Plan Cache 中计划的算子（Operator）信息，每个算子都有一条记录。**需要注意以下两点。**
  - 查询 GV$OB_PLAN_CACHE_PLAN_EXPLAIN 视图时，需要指定 TENANT_ID、SVR_IP、SVR_PORT、PLAN_ID 的等值条件，否则查询结果为空。
  - 查询 V$OB_PLAN_CACHE_PLAN_EXPLAIN 视图时，需要指定 TENANT_ID、PLAN_ID 的等值条件，否则查询结果为空。
- 查询 GV$OB_PLAN_CACHE_PLAN_EXPLAIN 视图时，需要指定 TENANT_ID、SVR_IP、SVR_PORT、PLAN_ID 的等值条件，否则查询结果为空。
- 查询 V$OB_PLAN_CACHE_PLAN_EXPLAIN 视图时，需要指定 TENANT_ID、PLAN_ID 的等值条件，否则查询结果为空。

GV$OB_PLAN_CACHE_PLAN_STAT 视图的主要字段如下。

- TENANT_ID：租户 ID。
- SVR_IP：节点 IP。
- PLAN_ID：执行计划 ID，对应 GV$OB_SQL_AUDIT 视图中的 PLAN_ID 字段。
- SQL_ID：SQL ID，对应 GV$OB_SQL_AUDIT 视图中的 SQL_ID 字段。
- TYPE：执行计划的类型，1 表示本地计划、2 表示远程计划、3 表示分布式计划。
- STATEMENT：参数化后的 SQL 语句。
- QUERY_SQL：第一次加载计划时的原始 SQL 语句。
- FIRST_LOAD_TIME：第一次被加载时间。
- SCHEMA_VERSION：Schema 版本号。
- LAST_ACTIVE_TIME：最近一次被执行时间。
- AVG_EXE_USEC：平均执行时间。

- `SLOWEST_EXE_TIME`：最慢执行的时间戳。
- `SLOWEST_EXE_USEC`：最慢执行的耗时。
- `SLOW_COUNT`：判断为慢查询次数，阈值通过集群配置项 `trace_log_slow_query_watermark` 控制。
- `HIT_COUNT`：计划命中 plan cahce 的次数。
- `LARGE_QUERYS`：判断为大查询次数，阈值通过集群配置项 `large_query_threshold` 控制。
- `DELAYED_LARGE_QUERYS`：被判断为大查询且被丢入大查询队列的次数。
- `TIMEOUT_COUNT`：超时次数。
- `EXECUTIONS`：执行次数。
- `DISK_READS`：所有执行物理读次数。
- `DIRECT_WRITES`：所有执行写盘的次数。
- `BUFFER_GETS`：所有执行逻辑读次数。
- `APPLICATION_WAIT_TIME`：执行所有 application 类事件的总时间。
- `CONCURRENCY_WAIT_TIME`：执行所有 concurrency 类事件的总时间。
- `USER_IO_WAIT_TIME`：所有执行 user_io 类事件的总时间。
- `ROWS_PROCESSED`：所有执行选择的结果行数或执行更改表中的行数。
- `ELAPSED_TIME`：执行接收到请求到执行结束消耗时间。
- `CPU_TIME`：所有执行消耗的 CPU 时间。
- `OUTLINE_ID`：是否命中用户创建的 Outline ID，为 `-1` 表示不是通过绑定 Outline 生成的计划。
- `OUTLINE_DATA`：计划对应的 Outline 信息。
- `TABLE_SCAN`：该查询是否为主键扫描。

`GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 视图记录了执行计划的详细信息，其主要字段介绍如下。

- `TENANT_ID`：租户 ID。
- `SVR_IP`：节点 IP。
- `PLAN_ID`：执行计划 ID。

- `PLAN_DEPTH`：Operator 的深度。
- `OPERATOR`：Operator 的名称。
- `NAME`：Operator 对应的表名。
- `ROWS`：预估的结果行数。
- `COST`：预估的代价。
- `PROPERTY`：Operator 的详细信息。

需要注意的是，在反复执行同一条相同的 SQL 时，执行计划可能会随着一些信息的变更而发生变化，例如数据库对象的变更、统计信息的变更、OBServer 版本的变更等等。所以为了确认某一条 SQL 在执行时的真实计划，不应该直接执行 explain 语句，而是要通过查询上述的两个计划监控视图 `gv$ob_sql_audit` 和 `GV$OB_PLAN_CACHE_PLAN_EXPLAIN`。

下面通过举例说明如何通过这两个计划监控视图确认具体某一条 SQL 在执行时的真实计划是什么。

1. 执行一条 SQL

```
select * from oceanbase.DBA_OB_DATABASES;
```

输出如下：

```
+--------------------+---------------+--------------------+-----------+-----------
---------+
| DATABASE_NAME      | IN_RECYCLEBIN | COLLATION          | READ_ONLY | COMMEN
T            |
+--------------------+---------------+--------------------+-----------+-----------
---------+
| obproxy            | NO            | utf8mb4_general_ci | NO
|                    |
| test               | NO            | utf8mb4_general_ci | NO        | test schem
a        |
| __public           | NO            | utf8mb4_general_ci | YES       | public sch
ema      |
| __recyclebin       | NO            | utf8mb4_general_ci | YES       | recyclebi
n schema   |
| mysql              | NO            | utf8mb4_general_ci | NO        | MySql sche
ma       |
| information_schema | NO            | utf8mb4_general_ci | NO        | informatio
n_schema |
| oceanbase          | NO            | utf8mb4_general_ci | NO        | system dat
abase    |
+--------------------+---------------+--------------------+-----------+-----------
```

```
---------+
8 rows in set
```

2. 查看 SQL 的 `trace_id`

```
select last_trace_id();
```

输出如下：

```
+-------------------------------------+
| last_trace_id()                     |
+-------------------------------------+
| Y584A0B9E1F14-0006104BFEB3B549-0-0 |
+-------------------------------------+
1 row in set
```

3. 通过 `trace_id` 查询 SQL 对应的 `TENANT_ID`、`SVR_IP`、`SVR_PORT`、`PLAN_ID` 字段

```
select TENANT_ID,
  SVR_IP,
  SVR_PORT,
  PLAN_ID,
  QUERY_SQL
from oceanbase.gv$ob_sql_audit
where trace_id = 'Y584A0B9E1F14-0006104BFEB3B549-0-0';
```

输出如下：

```
+-----------+--------------+----------+---------+---------------------------------
---------+
| TENANT_ID | SVR_IP       | SVR_PORT | PLAN_ID | QUERY_SQ
L                                 |
+-----------+--------------+----------+---------+---------------------------------
---------+
|      1002 | 1.2.3.4      |    12345 |   67890 | select * from oceanbase.DBA_OB_D
ATABASES |
+-----------+--------------+----------+---------+---------------------------------
---------+
1 row in set
```

4. 根据查询的信息，查询对应 SQL 在执行时的真实计划

```
SELECT
  *
```

```
FROM
  oceanbase.GV$OB_PLAN_CACHE_PLAN_EXPLAIN
WHERE
  tenant_id = 1002
  AND SVR_IP = '1.2.3.4'
  AND SVR_PORT = 12345
  AND PLAN_ID = 67890;
```

输出如下：

```
+-----------+--------------+----------+---------+------------+--------------+-----
-----------+------+------+------+-----------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
----------------------------------------------------------+
| TENANT_ID | SVR_IP       | SVR_PORT | PLAN_ID | PLAN_DEPTH | PLAN_LINE_ID | OPER
ATOR         | NAME | ROWS | COST | PROPERT
Y


               |
+-----------+--------------+----------+---------+------------+--------------+-----
-----------+------+------+------+-----------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
----------------------------------------------------------+
|      1002 | 1.2.3.4      |    12345 |   67890 |          0 |            0 | PHY_
HASH_JOIN    | NULL |    8 |   56 | NUL
L


                   |
|      1002 | 1.2.3.4      |    12345 |   67890 |          1 |            1 |  PHY
_TABLE_SCAN | D    |    8 |    5 | table_rows:8, physical_range_rows:8, logical_ra
nge_rows:8, index_back_rows:0, output_rows:8, avaiable_index_name[idx_db_name,__al
l_database], pruned_index_name[idx_db_name], estimation info[table_id:104, (table_
type:10, version:-1--1--1, logical_rc:8, physical_rc:8)] |
|      1002 | 1.2.3.4      |    12345 |   67890 |          1 |            2 |  PHY
_TABLE_SCAN | C    |   18 |   45 | table_rows:18, physical_range_rows:18, logical_
range_rows:18, index_back_rows:0, output_rows:18, avaiable_index_name[__tenant_vir
tual_collation
]
                                            |
+-----------+--------------+----------+---------+------------+--------------+-----
-----------+------+------+------+-----------------------------------------------------
---------------------------------------------------------------------------------
---------------------------------------------------------------------------------
----------------------------------------------------------+
3 rows in set
```

最后再介绍一个 `GV$OB_SQL_PLAN` 系统视图，这个视图可以展示计划在第一次执行时真实的执行反馈信息，例如会同时展示优化器估算当前算子的输出行数，以及计划第一次执行时当前算子的真实输出行数。通过对比这些数值，可以反映出优化器的代价评估是否存在明显的问题。`GV$OB_SQL_PLAN` 系统视图的介绍详见官网《OceanBase 数据库》文档 参考指南/系统试图/MySQL 租户系统视图/性能视图/GV$OB_SQL_PLAN。

下面举一个简单的例子：

1. 执行一条 SQL

```
select * from oceanbase.DBA_OB_DATABASES;
```

输出如下：

```
+--------------------+---------------+---------------------+-----------+-----------
---------+
| DATABASE_NAME      | IN_RECYCLEBIN | COLLATION           | READ_ONLY | COMMEN
T            |
+--------------------+---------------+---------------------+-----------+-----------
---------+
| obproxy            | NO            | utf8mb4_general_ci | NO
|                    |
| test               | NO            | utf8mb4_general_ci | NO        | test schem
a         |
| __public           | NO            | utf8mb4_general_ci | YES       | public sch
ema       |
| __recyclebin       | NO            | utf8mb4_general_ci | YES       | recyclebi
n schema   |
| mysql              | NO            | utf8mb4_general_ci | NO        | MySql sche
ma        |
| information_schema | NO            | utf8mb4_general_ci | NO        | informatio
n_schema  |
| oceanbase          | NO            | utf8mb4_general_ci | NO        | system dat
abase     |
+--------------------+---------------+---------------------+-----------+-----------
---------+
8 rows in set
```

2. 查看 SQL 的 `trace_id`

```
select last_trace_id();
```

输出如下：

```
+-----------------------------------+
| last_trace_id()                   |
+-----------------------------------+
| Y584A0B9E1F14-0006104BFEB3B549-0-0 |
+-----------------------------------+
1 row in set
```

3. 通过 `trace_id` 查询 SQL 对应的 `TENANT_ID`、`SVR_IP`、`SVR_PORT`、`PLAN_ID` 字段

```
select TENANT_ID,
  SVR_IP,
  SVR_PORT,
  PLAN_ID,
  QUERY_SQL
from oceanbase.gv$ob_sql_audit
where trace_id = 'Y584A0B9E1F14-0006104BFEB3B549-0-0';
```

输出如下：

```
+-----------+--------------+----------+---------+---------------------------------
---------+
| TENANT_ID | SVR_IP       | SVR_PORT | PLAN_ID | QUERY_SQ
L                                 |
+-----------+--------------+----------+---------+---------------------------------
---------+
|      1002 | 1.2.3.4      |    12345 |   67890 | select * from oceanbase.DBA_OB_D
ATABASES |
+-----------+--------------+----------+---------+---------------------------------
---------+
1 row in set
```

4. 根据查询的信息，查询对应 SQL 在执行时的真实计划

```
select
  OPERATOR,
  ID,
  PARENT_ID,
  DEPTH,
  POSITION,
  OBJECT_NAME,
  COST,
  REAL_COST,
  CARDINALITY,
  REAL_CARDINALITY,
  IO_COST,
  CPU_COST
from oceanbase.GV$OB_SQL_PLAN
```

```
where plan_id = 67890;
```

输出如下:

```
+------------------+----+-----------+-------+----------+-------------------------
--+------+----------+-------------+-----------------+---------+----------+
| OPERATOR         | ID | PARENT_ID | DEPTH | POSITION | OBJECT_NAM
E                | COST | REAL_COST | CARDINALITY | REAL_CARDINALITY | IO_COST | C
PU_COST |
+------------------+----+-----------+-------+----------+-------------------------
--+------+----------+-------------+-----------------+---------+----------+
| HASH OUTER JOIN  |  0 |        -1 |     0 |        1
|                             |   57 |      2110 |           8 |                  8
|        0 |          0 |
| TABLE FULL SCAN  |  1 |         0 |     1 |        1 | __all_databas
e              |    6 |         0 |            8 |                 8 |         0
|          0 |
| TABLE FULL SCAN  |  2 |         0 |     1 |        2 | __tenant_virtual_collatio
n |   46 |      2110 |          18 |                18 |         0 |          0 |
+------------------+----+-----------+-------+----------+-------------------------
--+------+----------+-------------+-----------------+---------+----------+
3 rows in set
```

7.5 阅读和管理 OceanBase 数据库 SQL 执行计划

执行计划（Execution Plan）是对一条 SQL 查询语句在数据库中执行过程的描述。

用户可以通过 EXPLAIN 命令查看优化器针对指定 SQL 生成的执行计划。如果要分析某条 SQL 的性能问题，通常需要先查看 SQL 的执行计划，排查每一步 SQL 执行是否存在问题。所以读懂执行计划是 SQL 优化的先决条件，而了解执行计划的算子是理解 EXPLAIN 命令的关键。

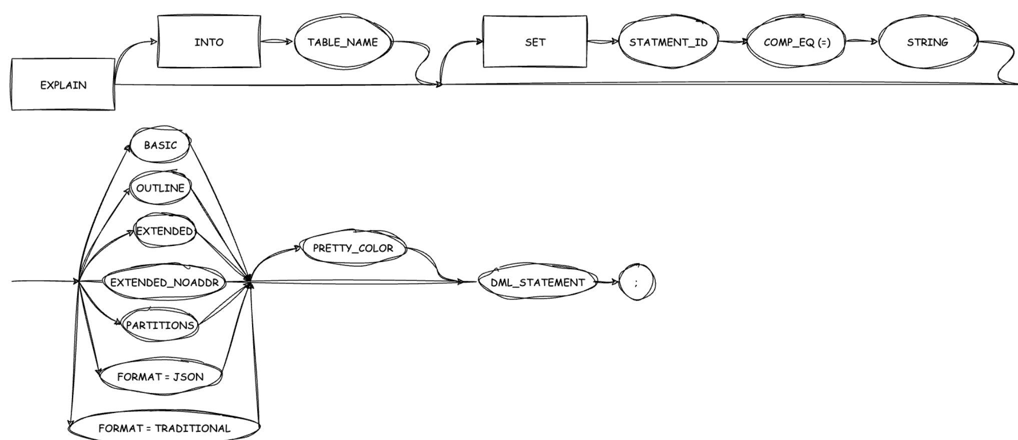
EXPLAIN 命令格式

该语句用于解释 SQL 语句的执行计划，可以是 SELECT、DELETE、INSERT、REPLACE 或 UPDATE 语句。

EXPLAIN 与 DESCRIBE、DESC 互为同义词。

语法

```
{EXPLAIN [INTO table_name ] [SET statement_id = string]}
[explain_type] [PRETTY | PRETTY_COLOR] dml_statement;

explain_type:
    BASIC
  | OUTLINE
  | EXTENDED
  | EXTENDED_NOADDR
  | PARTITIONS
  | FORMAT = {TRADITIONAL| JSON}

dml_statement:
    SELECT statement
  | DELETE statement
  | INSERT statement
  | REPLACE statement
```



参数解释

参数	描述
INTO table_name	表示将 EXPLAIN 的计划信息保存到指定表内。如果没有指定INTO table_name，默认查询到 PLAN_TABLE 表内。
SET statement_id	表示当前查询使用字符串标记，以方便后续查询该 SQL 的计划信息。如果没有指定 SET statement_id，默认使用空字符串作为信息标记。
PRETTY | PRETTY_COLOR	将计划树中的父节点和子节点使用树线或彩色树线连接起来，使得执行计划展示更方便阅读。
BASIC	指定输出计划的基础信息，如算子 ID、算子名称、所引用的表名。
OUTLINE	指定输出的计划信息包含 OUTLINE 信息。
EXTENDED	展示附加信息。
EXTENDED_NOADDR	以简约的方式展示附加信息。
PARTITIONS	显示分区相关信息。
FORMAT = {TRADITIONAL| JSON}	指定 EXPALIN 的输出格式。 • TRADITIONAL：表格输出格式； • JSON：KEY:VALUE 输出格式，JSON 显示为 JSON 字符串，包括 EXTENDED 和 PARTITIONS 信息。
dml_statement	DML 语句。

对于 OceanBase 数据库的使用者来说，最常用的是 EXPLAIN 命令和 EXPLAIN EXTENDED_NOADDR 命令。

- EXPLAIN 所展示的信息可以快速帮助普通用户了解整个计划的执行方式。例如：

```
create table t1(c1 int, c2 int);

create table t2(c1 int, c2 int);

-- 向 t1 表插入 10 行测试数据，c1 列的值是从 1 到 1000 的连续整数。
insert into t1 with recursive cte(n) as (select 1 from dual union all select n +
1 from cte wher

-- 向 t2 表插入 10 行测试数据，c1 列的值是从 1 到 1000 的连续整数。
insert into t2 with recursive cte(n) as (select 1 from dual union all select n +
1 from cte wher

-- 收集指定表 t1 的统计信息
analyze table t1 COMPUTE STATISTICS for all columns size 128;

-- 收集指定表 t2 的统计信息
analyze table t2 COMPUTE STATISTICS for all columns size 128;

explain select * from t1, t2 where t1.c1 = t2.c1 and t1.c1 < 500;
+-----------------------------------------------------------------------------------
---+
| Query Pla
n                                                                          |
+-----------------------------------------------------------------------------------
---+
| ================================================
=                                  |
| |ID|OPERATOR        |NAME|EST.ROWS|EST.TIME(us)
|                                  |
| ------------------------------------------------
-                                  |
| |0 |HASH JOIN       |    |498     |315
|                                  |
| |1 |├─TABLE FULL SCAN|t1  |499     |76
|                                  |
| |2 |└─TABLE FULL SCAN|t2  |499     |76
|                                  |
| ================================================
=                                  |
| Outputs & filters
:                                                                  |
| ------------------------------------
-                                                 |
|   0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil), rowset=25
6          |
```

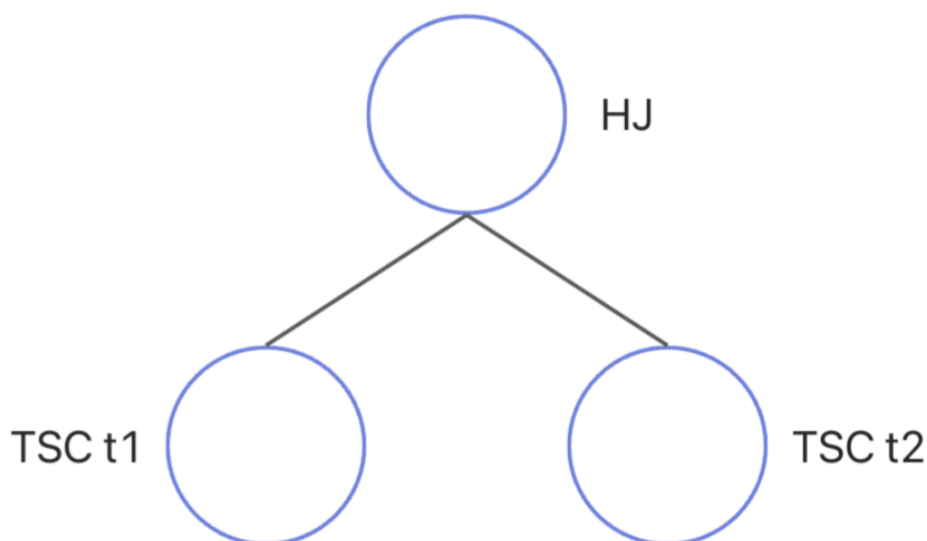
```
|       equal_conds([t1.c1 = t2.c1]), other_conds(nil
)                               |
|   1 - output([t1.c1], [t1.c2]), filter([t1.c1 < 500]), rowset=25
6                   |
|       access([t1.c1], [t1.c2]), partitions(p0
)                                     |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                   |
|   2 - output([t2.c1], [t2.c2]), filter([t2.c1 < 500]), rowset=25
6                   |
|       access([t2.c1], [t2.c2]), partitions(p0
)                                     |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t2.__pk_increment]), range(MIN ; MAX)always tru
e                   |
+-----------------------------------------------------------------------------------
---+
```

OceanBase 数据库执行计划中的各列的含义如下表所示。

列名	含义
ID	执行树按照前序遍历的方式得到的编号（从 0 开始）。
OPERATOR	操作算子的名称。
NAME	对应表操作的表名（索引名）。
EST. ROWS	优化器估算该操作算子的输出行数，仅作为参考。 例如上图中最下方的 1 号算子 `TABLE FULL SCAN`，因为过滤条件有 `t1.c1 < 500`，所以优化器根据这 1000 行数据的统计信息，可以估算出过滤之后需要输出的数据行数为 499 行。
EST.TIME	优化器估算该操作算子的执行代价（微秒），仅作为参考。

在表操作中，`NAME` 字段会显示该操作涉及的表的名称（别名），如果是使用索引访问，还会在名称后的括号中展示该索引的名称，例如 `t1(t1_c2)` 表示使用了索引 `t1_c2`。如果扫描的顺序是逆序，还会在后面使用 RESERVE 关键字标识，例如 `t1(t1_c2,RESERVE)`。

OceanBase 数据库 EXPLAIN 命令输出的第一部分是执行计划的树形结构展示。其中每一个操作在树中的层次通过其在算子中的缩进予以展示。树的层次关系用缩进来表示，层次最深的优先执行，层次相同的算子以指定算子的执行顺序为标准来执行。上述示例查询的计划展示树如下图所示：



其中，0 号算子是一个 hash join（HJ） 算子，它有两个子节点，分别是 1 和 2 号 table scan 算子（TSC）。0 号 hash join 算子的执行逻辑如下：

1. 读取左子节点的数据，根据联接列计算哈希值，构建一张哈希表。

2. 读取右子节点的数据，根据联接列计算哈希值，尝试使用通过左支数据构建的哈希表进行哈希探测，完成联接计算。

OceanBase 数据库 EXPLAIN 命令输出的第二部分是各操作算子的详细信息，包括输出表达式、过滤条件、分区信息以及各算子的独有信息（包括排序键、联接键、下压条件等）。即上述计划中的下半部分：

```
 Outputs & filters
:
 ------------------------------------
-
   0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil), rowset=1
6
       equal_conds([t1.c1 = t2.c1]), other_conds(nil
)
   1 - output([t1.c1], [t1.c2]), filter([t1.c1 > 10]), rowset=1
6
       access([t1.c1], [t1.c2]), partitions(p0
)
       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,
```

```
      range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e
   2 - output([t2.c1], [t2.c2]), filter([t2.c1 > 10]), rowset=1
6
      access([t2.c1], [t2.c2]), partitions(p0
)
      is_index_back=false, is_global_index=false, filter_before_indexback[false]
,
      range_key([t2.__pk_increment]), range(MIN ; MAX)always true
```

第二部分的内容跟第一部分有关，主要是描述第一部分算子的具体信息。其中一些公共的信息如下。

1. output：该算子的输出表达式。

2. filter：该算子的过滤谓词。 如果算子没有设置过滤条件，则为 nil。

为了更好地读懂 OceanBase 数据库 EXPLAIN 计划中各个算子的第二部分，需要用户对各个算子的作用有一个初步的了解，具体介绍可参见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 执行计划/执行计划算子 章节。

例如上面这个计划，用户需要通过官网《OceanBase 数据库》文档中 参考指南/性能调优/SQL 调优指南/SQL 执行计划/执行计划算子/TABLE SCAN 内容和 参考指南/性能调优/SQL 调优指南/SQL 执行计划/执行计划算子/JOIN 一文中的 **HASH JOIN** 内容的介绍来读懂上面这部分内容，大家不妨一试。

说明

上述计划中的 rowset=16 和 OceanBase 数据库执行引擎的向量化执行技术相关。表示在特定算子中，会按照 16 行一组分批进行计算。

用户可以通过 _rowsets_enabled 对向量化开关进行打开或关闭，例如通过 alter system set _rowsets_enabled = 0; 关闭向量化开关。用户也可以通过 _rowsets_max_rows 对向量化执行中一批处理的行数进行设置，例如通过 alter system set_rowsets_max_rows = 4; 将向量化执行中一批数据的行数改为 4。

向量化更详细的内容不在这里展开，感兴趣的用户可以阅读 OceanBase 社区博客《OceanBase 轻量级数仓关键技术解读》中的 **向量化执行技术** 部分。

- 读取左子节点的数据，根据联接列计算哈希值，构建一张哈希表。
- 读取右子节点的数据，根据联接列计算哈希值，尝试使用通过左支数据构建的哈希表进行哈希探测，完成联接计算。
- output : 该算子的输出表达式。
- filter : 该算子的过滤谓词。 如果算子没有设置过滤条件，则为 nil。
- EXPLAIN EXTENDED_NOADDR 命令用于进行详细的计划展示。通常用户在排查 SQL 性能问题时，会使用这种展示模式。

```
CREATE TABLE `t1` (
  `c1` int, `c2` int,
  KEY `idx` (`c1`));

insert into t1 values(1, 2);

explain EXTENDED_NOADDR select * from t1 where c1 > 10;
+-----------------------------------------------------------------------------+
| Query Plan                                                                  |
+-----------------------------------------------------------------------------+
| ===================================================                         |
| |ID|OPERATOR        |NAME   |EST.ROWS|EST.TIME(us)|                         |
| ---------------------------------------------------                         |
| |0 |TABLE RANGE SCAN|t1(idx)|1       |7           |                         |
| ===================================================                         |
| Outputs & filters:                                                          |
| -------------------------------------                                       |
|   0 - output([t1.c1], [t1.c2]), filter(nil), rowset=16                      |
|       access([t1.__pk_increment], [t1.c1], [t1.c2]), partitions(p0)         |
|       is_index_back=true, is_global_index=false,                            |
|       range_key([t1.c1], [t1.__pk_increment]), range(10,MAX ; MAX,MAX),     |
|       range_cond([t1.c1 > 10])                                              |
| Used Hint:                                                                  |
| -------------------------------------                                       |
|   /*+                                                                       |
|                                                                             |
|   */                                                                        |
| Qb name trace:                                                              |
| -------------------------------------                                       |
|   stmt_id:0, stmt_type:T_EXPLAIN                                            |
|   stmt_id:1, SEL$1                                                          |
| Outline Data:                                                               |
| -------------------------------------                                       |
|   /*+                                                                       |
|       BEGIN_OUTLINE_DATA                                                    |
|       INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx")                             |
|       OPTIMIZER_FEATURES_ENABLE('4.0.0.0')                                  |
```

```
|       END_OUTLINE_DATA                                                 |
|   */                                                                   |
| Optimization Info:                                                     |
| -------------------------------------                                  |
|   t1:                                                                  |
|       table_rows:1                                                     |
|       physical_range_rows:1                                            |
|       logical_range_rows:1                                             |
|       index_back_rows:1                                                |
|       output_rows:1                                                    |
|       table_dop:1                                                      |
|       dop_method:Table DOP                                             |
|       avaiable_index_name:[idx, t1]                                    |
|       unstable_index_name:[t1]                                         |
|       stats version:0                                                  |
|       dynamic sampling level:0                                         |
|   Plan Type:                                                           |
|       LOCAL                                                            |
|   Note:                                                                |
|       Degree of Parallelisim is 1 because of table property            |
+------------------------------------------------------------------------+
47 rows in set
```

上图中的 Optimization Info 往往可以用于分析一些 SQL 的性能问题，其中重要信息的介绍如下。

Optimization Info	含义解释
table_rows	上一个合并版本 SSTable 中的表的行数，可以简单理解为 `t1` 表的行数，只具有参考意义。
physical_range_rows	`t1` 表需要扫描的物理行数。如果走了索引的话，含义为 `t1` 表在索引上需要扫描的物理行数。

logical_range_rows	`t1`
	表在需要扫描的逻辑行数。如果走了索引的话，含义为
	`t1`
	表在索引上需要扫描的逻辑行数。在上面这个计划中，因为扫描了
	`idx`
	索引，所以能看到扫描范围都是
	`range(10,MAX ; MAX,MAX),range_cond([t1.c1 > 10])`
	。如果没有索引，则需要扫描全表，这时扫描的范围会变为
	`range(MIN ; MAX)`
	。

	注意 • physical_range_rows 和 logical_range_rows 这两个指标一般来说是相近的，看任意一个都可以。仅在一些特殊的 Buffer 表场景下，physical_range_rows 可能会远大于 logical_range_rows。 • Buffer 表表示频繁插入、删除的表。当 LSM tree 中的增量数据中积累了大量标记删除的数据时，从上层应用视角实际存在的行很少，但范围查询时可能需要处理较多的标记删除的数据，这种场景下，physical_range_rows 可能会远大于 logical_range_rows，从而导致 SQL 耗时不够理想。同时 Buffer 表场景下也容易导致优化器生成非最优执行计划。 • Buffer 表的介绍、检测逻辑，以及规避方法详见官网《OceanBase 数据库》文档 管理数据库/性能调优/识别组件内的瓶颈/OBServer 端性能瓶颈/非最优计划/Buffer 表。
index_back_rows	如果索引需要回表，这个值表示索引回表的行数。当全表扫描或者索引扫描但不需要回表时，该值为 0。索引回表的概念详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 执行计划/执行计划算子/TABLE SCAN。简单来说，这个计划中索引需要回主表的原因是索引 idx 中只有 c1 列的数据，需要根据索引中过滤出来的数据对应到主表中的隐藏主键信息，回到主表中查出 c2 列的数据。如果这个查询不是 select * from t1 where c1 > 10，而是 select c1 from t1 where c1 > 10，因为索引中已经包含了需要查询的全部信息，则不需要索引回表。
output_rows	预估输出行数。在上面的计划中，表示 t1 表经过过滤之后的行数。
table_dop	t1 表扫描时的并行度（并行执行时所使用的工作线程数）。
dop_method	决定表扫描并行度的原因。可以是 TableDOP（表在定义时指定的并行度）、AutoDop（优化器基于代价选择的并行度，需要打开 auto dop 功能）、global parallel（parallel hint 或系统变量设置的并行度）。
avaiable_index_name	t1 表可用的索引列表。这里除了索引表以外，还包括主表。如果没有合适的索引，计划会选择在主表上进行全表扫描。
pruned_index_name	当前的查询基于优化器的规则，认为不应该使用的索引列表。OceanBase 数据库优化器的索引剪枝规则详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 优化/查询优化/访问路径/基于规则的路径选择。

unstable_index_name	这个如果存在，则一定是被裁剪的主表路径，被裁剪通常是因为存在其他的索引路径 range 行数较小。
stats version	`t1` 表统计信息版本号，如果值为 `0`，表示该表没有收集统计信息。为了保证计划生成正确，可以自动或者手动收集对应表的统计信息。
dynamic sampling level	动态采样等级，如果值为 `0`，表示该表没有使用动态采样。 动态采样是一种优化器的优化工具，详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 优化/优化器统计信息/统计信息收集方式/优化器动态采样。
estimation method	`t1` 表行数估计方式，可以是 DEFAULT（使用的默认统计信息，这种情况行数估计可能会不准，需要 DBA 介入优化）、STORAGE（使用存储层实时估行）、STATS（使用统计信息估行）。
Plan Type	当前计划类型，可以是 LOCAL、REMOTE、DISTRIBUTED。具体含义详见本教程 ODP SQL 路由原理 中的 **SQL 的计划类型** 部分。
Note	生成该计划的一些备注信息。例如上面这个计划中的：`Degree of Parallelisim is 1 because of table property` 表示由于当前表的并行度设置为 `1`，所以当前查询的并行度被设置为 `1`。

- `physical_range_rows` 和 `logical_range_rows` 这两个指标一般来说是相近的，看任意一个都可以。仅在一些特殊的 Buffer 表场景下，`physical_range_rows` 可能会远大于 `logical_range_rows`。
- Buffer 表表示频繁插入、删除的表。当 LSM tree 中的增量数据中积累了大量标记删除的数据时，从上层应用视角实际存在的行很少，但范围查询时可能需要处理较多的标记删除的数据，这种场景下，`physical_range_rows` 可能会远大于 `logical_range_rows`，从而导致 SQL 耗时不够理想。同时 Buffer 表场景下也容易导致优化器生成非最优执行计划。
- Buffer 表的介绍、检测逻辑，以及规避方法详见官网《OceanBase 数据库》文档 管理数据库/性能调优/识别组件内的瓶颈/OBServer 端性能瓶颈/非最优计划/Buffer 表。

- EXPLAIN EXTENDED 在 EXPLAIN EXTENDED_NOADDR 的基础上，还会额外展示各个算子中涉及到的各个表达式的数据所存放的地址信息。通常技术支持同学和研发同学在排查 SQL 正确性问题时，会使用到这种展示模式。一般用户不需要关心各个表达式后面对应的地址信息，可以直接忽略。

```
explain EXTENDED select * from t1 where c1 > 10;
```

输出如下：

```
+-----------------------------------------------------------------------
```

```
-------------------------------------+
| Query Pla
n
                         |
+-----------------------------------------------------------------------------------
-------------------------------------+
| ==================================================
=                                                                 |
| |ID|OPERATOR        |NAME    |EST.ROWS|EST.TIME(us)
|                                                                 |
| --------------------------------------------------
-                                                                 |
| |0 |TABLE RANGE SCAN|t1(idx)|1       |7
|                                                                 |
| ==================================================
=                                                                 |
| Outputs & filters
:
                 |
| ------------------------------------
-                                                                               |
|   0 - output([t1.c1(0x7fed4fe0df50)], [t1.c2(0x7fed4fe0e4c0)]), filter(nil), row
set=16                              |
|       access([t1.__pk_increment(0x7fed4fe0e9d0)], [t1.c1(0x7fed4fe0df50)], [t1.c
2(0x7fed4fe0e4c0)]), partitions(p0) |
|       is_index_back=true, is_global_index=false
,                                                                    |
|       range_key([t1.c1(0x7fed4fe0df50)], [t1.__pk_increment(0x7fed4fe0e9d0)]), r
ange(10,MAX ; MAX,MAX),             |
|       range_cond([t1.c1(0x7fed4fe0df50) > 10(0x7fed4fe0d800)]
)                                                      |

|
                                    |
|       .....
.
                       |

|
                                    |
+-----------------------------------------------------------------------------------
-------------------------------------+
```

执行计划中的算子介绍

常见的算子详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 执行计划/执行计划算子 章节。因为官网内容足够详细，所以本文档不再对各个算子逐一进行介绍。

说明

强烈推荐用户去了解一下其中最常见的 TABLE SCAN 算子 和 JOIN 算子，以及 TABLE SCAN 中的 DAS 执行。

本文只会重点对一个被称为 EXCHANGE 的算子进行介绍。这个 EXCHANGE 算子在 OceanBase 数据库的分布式计划中很常见，且作用和含义不像其他算子那么容易理解，因此会在官网内容的基础上，对 EXCHANGE 算子进行一些补充。

EXCHANGE 算子是用于线程间进行数据交互的算子，一般都是成对出现的，数据源端有一个 OUT 算子，目的端会有一个 IN 算子。作用主要是：gather（数据汇聚）、transmit（数据转发）、re-partition（数据重分区），下面会对这三种作用逐一进行介绍。

EXCH-IN/OUT

EXCHANGE IN/EXCHANGE OUT 被用于将多个分区上的数据汇聚到一起，发送到查询所在的主节点上。

下面的查询中访问了 5 个分区（p0-p4）的数据：

```
CREATE TABLE t3 (c1 INT, c2 INT) PARTITION BY HASH(c1) PARTITIONS 5;
```

执行 explain 命令查看执行计划

```
explain select * from t3 where c1 > 10;
```

输出如下：

```
+------------------------------------------------------------------------------------
---+
| Query Pla
n                                                                          |
+------------------------------------------------------------------------------------
---+
| =============================================================
=                      |
| |ID|OPERATOR                |NAME     |EST.ROWS|EST.TIME(us)
|                      |
| -------------------------------------------------------------
-                      |
| |0 |PX COORDINATOR          |         |1       |20
|                      |
```

```
| |1 | └─EXCHANGE OUT DISTR     |:EX10000|1       |20
|                      |
| |2 |   └─PX PARTITION ITERATOR|        |1       |19
|                      |
| |3 |     └─TABLE FULL SCAN    |t3      |1       |19
|                      |
| ============================================================
=                      |
| Outputs & filters
:                                                                   |
| ------------------------------------
-                                                |
|   0 - output([INTERNAL_FUNCTION(t3.c1, t3.c2)]), filter(nil), rowset=1
6            |
|   1 - output([INTERNAL_FUNCTION(t3.c1, t3.c2)]), filter(nil), rowset=1
6            |
|       dop=
1                                                                          |
|   2 - output([t3.c1], [t3.c2]), filter(nil), rowset=1
6                           |
|       force partition granul
e                                                         |
|   3 - output([t3.c1], [t3.c2]), filter([t3.c1 > 10]), rowset=1
6                    |
|       access([t3.c1], [t3.c2]), partitions(p[0-4]
)                                 |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t3.__pk_increment]), range(MIN ; MAX)always tru
e                  |
+----------------------------------------------------------------------------------
---+
```

其中：

- 2 号算子 `PX PARTITION ITERATOR` 负责按照分区粒度迭代数据。granule iterator 算子的介绍详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 执行计划/执行计划算子/GI。
- 1 号算子 `EXCHANGE OUT DISTR` 接受 2 号算子 `PX PARTITION ITERATOR` 产生的输出，并将数据传出。
- 0 号算子 `PX COORDINATOR` 接收多个分区上 1 号算子产生的输出，并将结果汇总输出给用户。`PX COORDINATOR` 是一种特殊的 EXCHANGE IN 算子，除了能够拉回远程的数据外，还负责调度子计划的执行。

EXCH-IN/OUT (REMOTE)

我们在 ODP SQL 路由原理 中介绍了执行计划主要分为三种不同的类型，分别是本地（Local）计划、远程（Remote）计划和分布式（Distributed）计划。EXCHANGE IN REMOTE 和 EXCHANGE OUT REMOTE 算子就是在远程计划中用于将远程的数据（单个分区的数据）拉回本地。

例如我们创建了一个 1 - 1 - 1（3 Zone 3 节点）的环境，三个可用区分别是：`zone1`、`zone2`、`zone3`，对应的三个节点分别叫做 `A`、`B`、`C`。某个租户的 primary_zone 是 `zone1`，即这个租户下的所有表的主副本都会在 `zone1` 的 `A` 节点上。

本地计划

如果我们直连 `A` 节点，对两张非分区表进行计算，因为两表的主副本都在本机，就会生成如下的本地计划：

```
create table t1(c1 int, c2 int);

create table t2(c1 int, c2 int);

explain select * from t1, t2 where t1.c1 = t2.c1 and t1.c1 > 10;
+--------------------------------------------------------------------------------------
---+
| Query Pla
n                                                                            |
+--------------------------------------------------------------------------------------
---+
| ================================================
=                                  |
| |ID|OPERATOR         |NAME|EST.ROWS|EST.TIME(us)
|                                  |
| ------------------------------------------------
-                                  |
| |0 |HASH JOIN        |    |1       |9
|                                  |
| |1 |├─TABLE FULL SCAN|t1  |1       |4
|                                  |
| |2 |└─TABLE FULL SCAN|t2  |1       |4
|                                  |
| ================================================
=                                  |
| Outputs & filters:
略                                                                |
```

远程计划

如果我们直连 B 节点，对两张非分区表进行计算，因为两表的主副本都不在本机，就会生成如下的远程计划：

```
explain select * from t1, t2 where t1.c1 = t2.c1 and t1.c1 > 10;
```

输出如下：

```
+------------------------------------------------------------------------------------
---+
| Query Pla
n                                                                              |
+------------------------------------------------------------------------------------
---+
| =====================================================
=                               |
| |ID|OPERATOR                 |NAME|EST.ROWS|EST.TIME(us)
|                               |
| -----------------------------------------------------
-                               |
| |0 |EXCHANGE IN REMOTE    |    |1       |11
|                               |
| |1 | └─EXCHANGE OUT REMOTE|    |1       |10
|                               |
| |2 |   └─HASH JOIN        |    |1       |9
|                               |
| |3 |     ├─TABLE FULL SCAN|t1  |1       |4
|                               |
| |4 |     └─TABLE FULL SCAN|t2  |1       |4
|                               |
| =====================================================
=                               |
| Outputs & filters
:                                                                      |
| ------------------------------------
-                                                  |
|   0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil
)                        |
|   1 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil
)                        |
|   2 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil), rowset=1
6            |
|       equal_conds([t1.c1 = t2.c1]), other_conds(nil
)                                |
|   3 - output([t1.c1], [t1.c2]), filter([t1.c1 > 10]), rowset=1
6                      |
|       access([t1.c1], [t1.c2]), partitions(p0
```

```
)                                      |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                  |
|   4 - output([t2.c1], [t2.c2]), filter([t2.c1 > 10]), rowset=1
6                    |
|       access([t2.c1], [t2.c2]), partitions(p0
)                                      |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t2.__pk_increment]), range(MIN ; MAX)always tru
e                  |
+-----------------------------------------------------------------------------------
---+
23 rows in set
```

远程计划中的 `EXCHANGE IN REMOTE` 和 `EXCHANGE OUT REMOTE` 算子用于将远程的数据（单个分区的数据）拉回本地。

由于待读取的数据在远程，执行计划中分配了 0 号算子和 1 号算子来拉取远程的数据。其中：

- 2 号 - 4 号算子在 A 机器上执行，负责读取存储层的数据，并完成 HASH JOIN 的计算。
- 1 号算子 `EXCHANGE OUT REMOTE` 也在 `A` 机器上执行，读取 2 号算子 HASH JOIN 计算的结果数据，并将数据传出给 0 号算子。
- 0 号算子 `EXCHANGE IN REMOTE` 在 `B` 机器上执行，接收 1 号算子产生的输出。

EXCH-IN/OUT (PKEY)

EXCH-IN/OUT (PKEY) 算子用于数据重分区。它通常用于二元算子（例如 JOIN 算子）中，将算子其中一个孩子节点（左支）的数据，按照另外一个孩子节点（右支）的分区方式进行重分区，然后将左支重分区后的数据发送给右支对应分区所在的节点，最终完成相应的计算。

如下示例中，该查询是对两个分区表的数据进行联接。

```
CREATE TABLE t1 (c1 INT, c2 INT) PARTITION BY HASH(c1) PARTITIONS 5;

CREATE TABLE t2 (c1 INT PRIMARY KEY, c2 INT) PARTITION BY HASH(c1) PARTITIONS 4;

EXPLAIN SELECT * FROM t1, t2 WHERE t1.c1 = t2.c1;
+-----------------------------------------------------------------------------------
------+
| Query Pla
```

```
n                                                                              |
+-----------------------------------------------------------------------------------
------+
| ====================================================================
=                  |
| |ID|OPERATOR                        |NAME     |EST.ROWS|EST.TIME(us)
|                  |
| --------------------------------------------------------------------
-                  |
| |0 |PX COORDINATOR                  |         |1        |38
|                  |
| |1 | └─EXCHANGE OUT DISTR             |:EX10001|1        |37
|                  |
| |2 |   └─HASH JOIN                    |         |1        |36
|                  |
| |3 |     ├─EXCHANGE IN DISTR          |         |1        |20
|                  |
| |4 |     │ └─EXCHANGE OUT DISTR (PKEY)|:EX10000|1        |20
|                  |
| |5 |     │   └─PX PARTITION ITERATOR  |         |1        |19
|                  |
| |6 |     │     └─TABLE FULL SCAN      |t1       |1        |19
|                  |
| |7 |     └─PX PARTITION ITERATOR      |         |1        |16
|                  |
| |8 |       └─TABLE FULL SCAN          |t2       |1        |16
|                  |
| ====================================================================
=                  |
| Outputs & filters
:                                                                     |
| ------------------------------------
-                                                     |
|   0 - output([INTERNAL_FUNCTION(t1.c1, t1.c2, t2.c1, t2.c2)]), filter(nil), rows
et=16 |
|   1 - output([INTERNAL_FUNCTION(t1.c1, t1.c2, t2.c1, t2.c2)]), filter(nil), rows
et=16 |
|       dop=
1                                                                           |
|   2 - output([t1.c1], [t2.c1], [t1.c2], [t2.c2]), filter(nil), rowset=1
6               |
|       equal_conds([t1.c1 = t2.c1]), other_conds(nil
)                                        |
|   3 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                                      |
|   4 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                                      |
|       (#keys=1, [t1.c1]), dop=
1                                                         |
|   5 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                                      |
```

```
|       force partition granul
e                                                          |
|   6 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                                 |
|       access([t1.c1], [t1.c2]), partitions(p[0-4]
)                                     |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                      |
|   7 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                                 |
|       affinitize, force partition granul
e                                                  |
|   8 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                                 |
|       access([t2.c1], [t2.c2]), partitions(p[0-3]
)                                     |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t2.c1]), range(MIN ; MAX)always tru
e                                  |
+-----------------------------------------------------------------------------------
------+
```

执行计划将 `t1` 表的数据按照 `t2` 表的分区方式进行重分区：

- 4 号算子 `EXCHANGE OUT DISTR (PKEY)` 负责根据 `t2` 表的数据分区方式，以及查询的联接条件，确定 `t1` 表中的每一行数据应该发送到哪个节点才能完成与 `t2` 表对应分区的连接操作，并将 `t1` 表的各行数据发送到相应的节点。
- 3 号算子 `EXCHANGE IN DISTR` 负责在相应节点完成 `t1` 表数据的接收。
- 2 号算子 HASH JOIN 负责在各个节点，对 7 号算子迭代出的 `t2` 表各分区的数据，和 3 号算子接收到的 `t1` 表以 `t2` 表分区规则重分区后的对应分区数据，进行 JOIN 计算。
- 1 号算子 `EXCHANGE OUT DISTR` 负责把各节点、各分区 JOIN 之后的结果发送到 0 号算子。
- 0 号算子 `PX COORDINATOR` 负责接收各个节点 JOIN 的结果，并对结果进行汇总。

除了上述的 EXCH-IN/OUT (PKEY) 以外，优化器还会根据适合不同 SQL 的不同重分区方式，生成 EXCH-IN/OUT (HASH)、EXCH-IN/OUT (BC2HOST) 等算子，这里不再一一赘述。不同的数据重分布方式，详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/并行执行计划/并行执行计划简介 中 **常见的并行执行中的数据分发方式** 部分。

通过 Hint 生成指定计划

Hint 机制可以使优化器生成指定的执行计划。一般情况下，优化器会为用户查询选择最佳的执行计划，不需要用户使用 Hint 指定。但在某些场景下，优化器生成的执行计划可能不满足用户的要求，这时就需要用户使用 Hint 来指定生成某种执行计划。

Hint 语法

```
{ DELETE | INSERT | SELECT | UPDATE | REPLACE } /*+ [hint_text][,hint_text]... */
```

例如下面这条查询语句，通过 `/*+ PARALLEL(3)*/` 指定 SQL 的执行并行度为 `3`，通过 explain 就可以看到计划的 `dop = 3`，通过 explain EXTENDED_NOADDR 还可以看到计划中会展示一个 `Note: Degree of Parallelism is 3 because of hint`。

```
create t1 (c1 int, c2 int) PARTITION BY HASH(c1) PARTITIONS 5;

explain EXTENDED_NOADDR select /*+ PARALLEL(3) */* from t1 where c1 > 10;
+------------------------------------------------------------------------------+
| Query Plan                                                                   |
+------------------------------------------------------------------------------+
| ============================================================                 |
| |ID|OPERATOR              |NAME    |EST.ROWS|EST.TIME(us)|                   |
| ------------------------------------------------------------                 |
| |0 |PX COORDINATOR        |        |1       |3           |                   |
| |1 | └─EXCHANGE OUT DISTR  |:EX10000|1       |3           |                   |
| |2 |   └─PX BLOCK ITERATOR |        |1       |3           |                   |
| |3 |     └─TABLE RANGE SCAN|t1(idx) |1       |3           |                   |
| ============================================================                 |
| Outputs & filters:                                                           |
| -------------------------------------                                        |
|   0 - output([INTERNAL_FUNCTION(t1.c1, t1.c2)]), filter(nil), rowset=16     |
|   1 - output([INTERNAL_FUNCTION(t1.c1, t1.c2)]), filter(nil), rowset=16     |
|       dop=3                                                                  |
|   2 - output([t1.c1], [t1.c2]), filter(nil), rowset=16                       |
|   3 - output([t1.c1], [t1.c2]), filter(nil), rowset=16                       |
|       access([t1.__pk_increment], [t1.c1], [t1.c2]), partitions(p0)         |
|       is_index_back=true, is_global_index=false,                             |
|       range_key([t1.c1], [t1.__pk_increment]), range(10,MAX ; MAX,MAX),     |
|       range_cond([t1.c1 > 10])                                               |
| Used Hint:                                                                   |
| -------------------------------------                                        |
|   /*+                                                                        |
|       PARALLEL(3)                                                            |
```

```
|   */                                                                     |
|                                                                          |
|   ......                                                                 |
|                                                                          |
|   Note:                                                                  |
|       Degree of Parallelism is 3 because of hint                         |
+--------------------------------------------------------------------------+
```

Hint 从语法上看是一种特殊的 SQL 注释，所不同的是在注释的左标记后（`/*` 符号）增加了一个 `+`。因为 Hint 是注释，所以如果因为语法出错的原因，导致 OBServer 服务端无法识别 SQL 语句中的 Hint，优化器会选择忽略用户指定的 Hint，去生成默认的执行计划。另外，Hint 只影响优化器生成的执行计划，不会影响 SQL 语句的语义。

说明

如果使用 MySQL 的 C 客户端执行带 Hint 的 SQL 语句，需要使用 `-c` 选项登录，否则 MySQL 客户端会将 Hint 作为注释从用户 SQL 语句中去除，导致系统无法收到用户 Hint。

Hint 参数

Hint 相关参数名称、语义和语法如下表所示。

名称	语法	语义
NO_REWRITE	NO_REWRITE	禁止 SQL 改写。
READ_CONSISTENCY	READ_CONSISTENCY(WEAK [STRONG])	读一致性设置（弱/强）。
INDEX_HINT	INDEX(table_name index_name)	设置表索引。
QUERY_TIMEOUT	QUERY_TIMEOUT(INTNUM)	设置语句超时时间。
LOG_LEVEL	LOG_LEVEL([']log_level['])	设置日志级别，当设置模块级别语句时候，以第一个单引号（`'`）作为开始，第二个单引号（`'`）作为结束；例如 `'DEBUG'`。
LEADING	LEADING([qb_name] TBL_NAME_LIST)	设置联接顺序。

ORDERED	ORDERED	设置按照 SQL 中的顺序进行联接。
FULL	FULL([qb_name] TBL_NAME)	设置表访问路径为主表等价于 `INDEX(TBL_NAME PRIMARY)`。
USE_PLAN_CACHE	USE_PLAN_CACHE(NONE[DEFAULT])	设置是否使用计划缓存： • `NONE`：表示不使用计划缓存。 • `DEFAULT`：表示按照服务器本身的设置来决定是否使用计划缓存。
USE_MERGE	USE_MERGE([qb_name] TBL_NAME_LIST)	设置指定表在作为右表时使用 Merge Join。
USE_HASH	USE_HASH([qb_name] TBL_NAME_LIST)	设置指定表在作为右表时使用 Hash Join。
NO_USE_HASH	NO_USE_HASH([qb_name] TBL_NAME_LIST)	设置指定表在作为右表时不使用 Hash Join。
USE_NL	USE_NL([qb_name] TBL_NAME_LIST)	设置指定表在作为右表时使用 Nested Loop Join。
USE_BNL	USE_BNL([qb_name] TBL_NAME_LIST)	设置指定表在作为右表时使用 Block Nested Loop Join
USE_HASH_AGGREGATION	USE_HASH_AGGREGATION([qb_name])	设置聚合算法为 Hash。例如 Hash Group By 或者 Hash Distinct。
NO_USE_HASH_AGGREGATION	NO_USE_HASH_AGGREGATION([qb_name])	设置 Aggregate 方法不使用 Hash Aggregate，使用 Merge Group By 或者 Merge Distinct。
USE_LATE_MATERIALIZATION	USE_LATE_MATERIALIZATION	设置使用晚期物化。
NO_USE_LATE_MATERIALIZATION	NO_USE_LATE_MATERIALIZATION	设置不使用晚期物化。
TRACE_LOG	TRACE_LOG	设置收集 Trace 记录用于 SHOW TRACE 展示。

QB_NAME	QB_NAME(NAME)	设置 Query Block 的名称。
PARALLEL	PARALLEL(INTNUM)	设置分布式执行并行度。
TOPK	TOPK(PRECISION MINIMUM_ROWS)	设置模糊查询的精度和最小行数。 其中 PRECSION 为整型，取值范围 [0, 100]，表示模糊查询的行数百分比；MINIMUM_ROWS 为最小返回行数。
MAX_CONCURRENT |MAX_CONCURRENT(n)	限制这个 SQL 文本的并发数。	

说明

- QB_NAME 语法是: @NAME
- TBL_NAME 语法是: [db_name.]relation_name [qb_name]

QB_NAME 参数

在 DML 语句中，每一个 Query Block 都会有一个 QB_NAME（Query Block Name），可以由用户指定，也可以由系统自动生成。每一个 Query Block 都是一个语义上完整的查询语句，简单来看就是把 SQL 按 select、delete 这些词提取出来并结构化之后，按从左到右的方式标号。Query Block 的介绍详见官网《OceanBase 数据库》知识库 性能诊断和优化/SQL 诊断和优化/Query Block 介绍。

在用户没有用 Hint 指定 QB_NAME 的时候，系统会按照 SEL$1、SEL$2、UPD$1、DEL$1 方式从左到右（实际也是 Resolver 的解析顺序）依次生成。

通过 QB_NAME 可以精确定位到每一个表，也可以在某处指定任意 Query Block 的行为。TBL_NAME 中的 QB_NAME 用于定位表，在 Hint 中最前面的 QB_NAME 用于定位 Hint 作用于哪一个 Query Block。

如下例所示，对于 `SELECT *FROM t1, (SELECT* FROM t2 WHERE c2 = 1 LIMIT 5) WHERE t1.c1 = 1;` 这条 SQL，按照默认规则：

- 第一个 Query Block SEL$1 是最外层的：`SELECT * FROM t1, VIEW1 WHERE t1.c1 = 1`。

- 第二个 Query Block SEL$2 是 VIEW1（即计划里的 ANONYMOUS_VIEW1）：`SELECT * FROM t2 WHERE c2 = 1 LIMIT 5`。

优化器为 SEL$1 中的 `t1` 表选择走索引 `t1_c1` 路径，而为 SEL$2 中的 `t2` 表选择主表（Primary）访问。

```
CREATE TABLE t1(c1 INT, c2 INT, KEY t1_c1(c1));

CREATE TABLE t2(c1 INT, c2 INT, KEY t2_c1(c1));

EXPLAIN EXTENDED_NOADDR SELECT * FROM t1, (SELECT * FROM t2 WHERE c2 = 1 LIMIT 5)
WHERE t1.c1 =
+------------------------------------------------------------------------------------
--------------
| Query Pla
n

+------------------------------------------------------------------------------------
--------------
| ======================================================================
=
| |ID|OPERATOR                   |NAME            |EST.ROWS|EST.TIME(us)
|
| ----------------------------------------------------------------------
-
| |0 |NESTED-LOOP JOIN CARTESIAN |                |1       |11
|
| |1 | ├─TABLE RANGE SCAN         |t1(t1_c1)       |1       |7
|
| |2 | └─MATERIAL                 |                |1       |4
|
| |3 |   └─SUBPLAN SCAN           |ANONYMOUS_VIEW1|1       |4
|
| |4 |     └─TABLE FULL SCAN      |t2              |1       |4
|
| ======================================================================
=
| Outputs & filters
:
| ------------------------------------
-
|   0 - output([t1.c1], [t1.c2], [.c1], [.c2]), filter(nil), rowset=1
6
|       conds(nil), nl_params_(nil), use_batch=fals
e
|   1 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6
|       access([t1.__pk_increment], [t1.c1], [t1.c2]), partitions(p0
)
```

```
|       is_index_back=true, is_global_index=false
,
|       range_key([t1.c1], [t1.__pk_increment]), range(1,MIN ; 1,MAX)
,
|       range_cond([t1.c1 = 1]
)
|   2 - output([.c1], [.c2]), filter(nil), rowset=1
6
|   3 - output([.c1], [.c2]), filter(nil), rowset=1
6
|       access([.c1], [.c2]
)
|   4 - output([t2.c1], [t2.c2]), filter([t2.c2 = 1]), rowset=1
6
|       access([t2.c2], [t2.c1]), partitions(p0
)
|       limit(5), offset(nil), is_index_back=false, is_global_index=false, filter_
before_indexba
|       range_key([t2.__pk_increment]), range(MIN ; MAX)always tru
e
| Used Hint
:

| ------------------------------------
-
|   /*
+


|

|   *
/

| Qb name trace
:
| ------------------------------------
-
|   stmt_id:0, stmt_type:T_EXPLAI
N
|   stmt_id:1, SEL$
1
|   stmt_id:2, SEL$
2
| Outline Data
:
| ------------------------------------
-
|   /*
+
```

```
|       BEGIN_OUTLINE_DAT
A
|       LEADING(@"SEL$1" ("test"."t1"@"SEL$1" "ANONYMOUS_VIEW1"@"SEL$1")
)
|       USE_NL(@"SEL$1" "ANONYMOUS_VIEW1"@"SEL$1"
)
|       USE_NL_MATERIALIZATION(@"SEL$1" "ANONYMOUS_VIEW1"@"SEL$1"
)
|       INDEX(@"SEL$1" "test"."t1"@"SEL$1" "t1_c1"
)
|       FULL(@"SEL$2" "test"."t2"@"SEL$2"
)
|       OPTIMIZER_FEATURES_ENABLE('4.0.0.0'
)
|       END_OUTLINE_DAT
A
|   *
/

|   .....
.

+----------------------------------------------------------------------------------
--------------
```

如果 SQL 通过 Hint 来指定 SEL$1 的 t1 表走主表，SEL$2 的 t2 表走索引，示例如下：

```
EXPLAIN EXTENDED_NOADDR
SELECT /*+ INDEX(@SEL$1 t1 PRIMARY) INDEX(@SEL$2 t2 t2_c1) */ *
FROM t1 , (SELECT * FROM t2 WHERE c2 = 1 LIMIT 5) WHERE t1.c1 = 1;
+----------------------------------------------------------------------------------
--------------
| Query Pla
n

+----------------------------------------------------------------------------------
--------------
| ===================================================================
=
| |ID|OPERATOR                  |NAME           |EST.ROWS|EST.TIME(us)
|
| -------------------------------------------------------------------
-
| |0 |NESTED-LOOP JOIN CARTESIAN |               |1       |11
|
| |1 |├─TABLE FULL SCAN          |t1             |1       |4
|
| |2 |└─MATERIAL                 |               |1       |7
|
| |3 |  └─SUBPLAN SCAN           |ANONYMOUS_VIEW1|1       |7
```

```
|
| |4 |     └─TABLE FULL SCAN      |t2(t2_c1)      |1       |7
|
| ====================================================================
=
| Outputs & filters
:
| ------------------------------------
-
|   0 - output([t1.c1], [t1.c2], [.c1], [.c2]), filter(nil), rowset=1
6
|       conds(nil), nl_params_(nil), use_batch=fals
e
|   1 - output([t1.c1], [t1.c2]), filter([t1.c1 = 1]), rowset=1
6
|       access([t1.c1], [t1.c2]), partitions(p0
)
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e
|   2 - output([.c1], [.c2]), filter(nil), rowset=1
6
|   3 - output([.c1], [.c2]), filter(nil), rowset=1
6
|       access([.c1], [.c2]
)
|   4 - output([t2.c1], [t2.c2]), filter([t2.c2 = 1]), rowset=1
6
|       access([t2.__pk_increment], [t2.c2], [t2.c1]), partitions(p0
)
|       limit(5), offset(nil), is_index_back=true, is_global_index=false, filter_b
efore_indexbac
|       range_key([t2.c1], [t2.__pk_increment]), range(MIN,MIN ; MAX,MAX)always tr
ue
| Used Hint
:

| ------------------------------------
-
|   /*
+


|

|       INDEX("t1" "primary"
)
|       INDEX(@"SEL$2" "t2" "t2_c1"
)
|   *
```

```
/

| Qb name trace
:
| ------------------------------------
-
|   stmt_id:0, stmt_type:T_EXPLAI
N
|   stmt_id:1, SEL$
1
|   stmt_id:2, SEL$
2
| Outline Data
:
| ------------------------------------
-
|   /*
+

|       BEGIN_OUTLINE_DAT
A
|       LEADING(@"SEL$1" ("test"."t1"@"SEL$1" "ANONYMOUS_VIEW1"@"SEL$1")
)
|       USE_NL(@"SEL$1" "ANONYMOUS_VIEW1"@"SEL$1"
)
|       USE_NL_MATERIALIZATION(@"SEL$1" "ANONYMOUS_VIEW1"@"SEL$1"
)
|       FULL(@"SEL$1" "test"."t1"@"SEL$1"
)
|       INDEX(@"SEL$2" "test"."t2"@"SEL$2" "t2_c1"
)
|       OPTIMIZER_FEATURES_ENABLE('4.0.0.0'
)
|       END_OUTLINE_DAT
A
|   *
/

|   .....
.

+----------------------------------------------------------------------------------
--------------
```

读者可以自行研究一下，在加这个 `Hint /*+ INDEX(t1 PRIMARY) INDEX(@SEL$2 t2 t2_c1)*/` 前后，计划 Query Plan 部分和 Outline Data 部分有什么变化，以及变化是否是符合预期的？

上述示例中的 Hint 也可以写成如下三种方式，这几种写法都是等价的：

- 方式一:

```
SELECT /*+INDEX(@SEL$1 t1 PRIMARY) INDEX(@SEL$2 t2 t2_c1)*/ * FROM t1 , (SELECT *
FROM t2 WHERE c2 = 1 LIMIT 5) WHERE t1.c1 = 1;
```

- 方式二:

```
SELECT /*+INDEX(t1 PRIMARY) INDEX(@SEL$2 t2@SEL$2 t2_c1)*/ * FROM t1 , (SELECT * F
ROM t2 WHERE c2 = 1 LIMIT 5) WHERE t1.c1 = 1;
```

- 方式三:

```
SELECT /*+INDEX(t1 PRIMARY)*/ * FROM t1 , (SELECT /*+INDEX(t2 t2_c1)*/ * FROM t2 W
HERE c2 = 1 LIMIT 5) WHERE t1.c1 = 1;
```

Hint 使用规则

Hint 的一般使用规则如下:

- 对于没有指定 Query Block 的 Hint，表明作用于本 Query Block。
  - 示例 1：由于 Hint 写在 Query Block 1，但 Hint 中的 `t2` 表只出现在 Query Block 2，且优化器无法通过改写 SQL 将 SEL$2 中的 `t2` 表提升到 SEL$1 中，则 Hint 无法生效。

    输出如下:

```
+---------------------------------------------------------------------------------------
-------+
| Query Pla
n
+---------------------------------------------------------------------------------------
-------+
| ========================================================================
=                                     |
| |ID|OPERATOR                     |NAME            |EST.ROWS|EST.TIME(us)
|                                     |
| ------------------------------------------------------------------------
-                                     |
| |0 |NESTED-LOOP JOIN CARTESIAN |                |1        |11
|                                     |
| |1 |├─TABLE RANGE SCAN           |t1(t1_c1)       |1        |7
|                                     |
| |2 |└─MATERIAL                   |                |1        |4
|                                     |
| |3 |  └─SUBPLAN SCAN             |ANONYMOUS_VIEW1|1        |4
```

```
|                                              |
| |4 |     └─TABLE FULL SCAN      |t2            |1       |4
|                                              |
| =====================================================================
=                                              |
| .....
.
|
+---------------------------------------------------------------------------------------
-------+
```

- 示例 2：如果 SQL 可以通过优化器改写将 `t2` 表提升到 SEL$1，则 Hint 生效。如下面这个计划所示，原本的 SQL 明显是被改写成了一条不存在匿名视图的 SQL：`SELECT /*+INDEX(t2 t2_c1)*/ * FROM t1, t2 WHERE t1.c1 = 1 and t2.c2 = 1;`，在这条被改写之后的 SQL 中，t2 表被提升到了最外层的 SEL$1 里，所以 Hint 会生效。输出如下：

```
+---------------------------------------------------------------------------
-----------+
| Query Pla
n
|
+---------------------------------------------------------------------------
-----------+
| ===============================================================
=                    |
| |ID|OPERATOR                    |NAME     |EST.ROWS|EST.TIME(us)
|                    |
| ---------------------------------------------------------------
-                    |
| |0 |NESTED-LOOP JOIN CARTESIAN |         |1       |13
|                    |
| |1 | ├─TABLE RANGE SCAN         |t1(t1_c1)|1       |7
|                    |
| |2 | └─MATERIAL                 |         |1       |7
|                    |
| |3 |   └─TABLE FULL SCAN        |t2(t2_c1)|1       |7
|                    |
| ===============================================================
=                    |
| Outputs & filters
:                                                                   |
| -----------------------------------
-                                                 |
|   0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil), rowset=1
6           |
|       conds(nil), nl_params_(nil), use_batch=fals
e                                   |
|   1 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
```

```
6                              |
|       access([t1.__pk_increment], [t1.c1], [t1.c2]), partitions(p0
)                |
|       is_index_back=true, is_global_index=false
,                                  |
|       range_key([t1.c1], [t1.__pk_increment]), range(1,MIN ; 1,MAX)
,               |
|       range_cond([t1.c1 = 1]
)                                                      |
|   2 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                              |
|   3 - output([t2.c2], [t2.c1]), filter([t2.c2 = 1]), rowset=1
6                      |
|       access([t2.__pk_increment], [t2.c2], [t2.c1]), partitions(p0
)                |
|       is_index_back=true, is_global_index=false, filter_before_indexback
[false],    |
|       range_key([t2.c1], [t2.__pk_increment]), range(MIN,MIN ; MAX,MAX)a
lways true |
+---------------------------------------------------------------------------
-----------+
```

- 示例 1：由于 Hint 写在 Query Block 1，但 Hint 中的 `t2` 表只出现在 Query Block 2，且优化器无法通过改写 SQL 将 SEL$2 中的 `t2` 表提升到 SEL$1 中，则 Hint 无法生效。

```
EXPLAIN SELECT /*+INDEX(t2 t2_c1)*/ *
        FROM t1 , (SELECT * FROM t2 WHERE c2 = 1 LIMIT 5)
        WHERE t1.c1 = 1;
```

输出如下：

```
+---------------------------------------------------------------------------------
-------------------------+
| Query Pla
n
               |
+---------------------------------------------------------------------------------
-------------------------+
| =================================================================
=                                     |
| |ID|OPERATOR                   |NAME          |EST.ROWS|EST.TIME(us)
|                                     |
| -----------------------------------------------------------------
-                                     |
| |0 |NESTED-LOOP JOIN CARTESIAN |              |1       |11
|                                     |
| |1 | ├─TABLE RANGE SCAN          |t1(t1_c1)      |1       |7
|                                     |
```

```
| |2 |└─MATERIAL                  |              |1        |4
|                                     |
| |3 |  └─SUBPLAN SCAN            |ANONYMOUS_VIEW1|1        |4
|                                     |
| |4 |    └─TABLE FULL SCAN       |t2            |1        |4
|                                     |
| =====================================================================
=                                     |
| .....
.
                   |
+----------------------------------------------------------------------------------
--------------------------+
```

• 示例 2：如果 SQL 可以通过优化器改写将 `t2` 表提升到 SEL$1，则 Hint 生效。如下面这个计划所示，原本的 SQL 明显是被改写成了一条不存在匿名视图的 SQL：`SELECT /*+INDEX(t2 t2_c1)*/ * FROM t1, t2 WHERE t1.c1 = 1 and t2.c2 = 1;`，在这条被改写之后的 SQL 中，t2 表被提升到了最外层的 SEL$1 里，所以 Hint 会生效。

```
EXPLAIN SELECT /*+ INDEX(t2 t2_c1) */ *
        FROM t1 , (SELECT * FROM t2 WHERE c2 = 1)
        WHERE t1.c1 = 1;
```

输出如下：

```
+----------------------------------------------------------------------------------
---+
| Query Pla
n                                                                          |
+----------------------------------------------------------------------------------
---+
| ===============================================================
=                   |
| |ID|OPERATOR                  |NAME     |EST.ROWS|EST.TIME(us)
|                   |
| ---------------------------------------------------------------
-                   |
| |0 |NESTED-LOOP JOIN CARTESIAN |         |1        |13
|                   |
| |1 |├─TABLE RANGE SCAN         |t1(t1_c1)|1        |7
|                   |
| |2 |└─MATERIAL                 |         |1        |7
|                   |
| |3 |  └─TABLE FULL SCAN        |t2(t2_c1)|1        |7
|                   |
| ===============================================================
```

```
=                    |
| Outputs & filters
:                                                                  |
| ------------------------------------
-                                                    |
|   0 - output([t1.c1], [t1.c2], [t2.c1], [t2.c2]), filter(nil), rowset=1
6            |
|       conds(nil), nl_params_(nil), use_batch=fals
e                                 |
|   1 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                             |
|       access([t1.__pk_increment], [t1.c1], [t1.c2]), partitions(p0
)                 |
|       is_index_back=true, is_global_index=false
,                                |
|       range_key([t1.c1], [t1.__pk_increment]), range(1,MIN ; 1,MAX)
,                |
|       range_cond([t1.c1 = 1])
)                                                       |
|   2 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                             |
|   3 - output([t2.c2], [t2.c1]), filter([t2.c2 = 1]), rowset=1
6                     |
|       access([t2.__pk_increment], [t2.c2], [t2.c1]), partitions(p0
)                 |
|       is_index_back=true, is_global_index=false, filter_before_indexback[false]
,    |
|       range_key([t2.c1], [t2.__pk_increment]), range(MIN,MIN ; MAX,MAX)always tr
ue |
+-----------------------------------------------------------------------------------
---+
```

- 如果指定表行为，但在本 Query Block 中没有找到该表，或者发生冲突，那么 Hint 都将无效。

常用 Hint

与其他数据库的行为相比，OceanBase 数据库优化器是动态规划的，已经考虑了所有可能的最优路径，Hint 主要作用是指定优化器的行为，并按照 Hint 执行 SQL 查询。下面会介绍用户最为常用的一些 Hint。

INDEX Hint

INDEX Hint 的语法如下：

```
SELECT/*+ INDEX(table_name index_name) */ * FROM table_name;
```

在 SQL 语句中，表名存在别名即 `table_name [AS] alias`，必须写表别名，才能使 INDEX 生效。示例如下：

```
create table t1(c1 int, c2 int, c3 int);

create index idx1 on t1(c1);

create index idx2 on t1(c2);

-- 插入 1000 行测试数据
insert into t1 with recursive cte(n) as (select 1 from dual union all select n +
1 from cte wher

-- 收集指定表 t1 的统计信息
analyze table t1 COMPUTE STATISTICS for all columns size 128;

-- 对于这 1000 行测试数据的数据特征，c1 = 1 过滤性优于 c2 = 1
-- 所以在没有 index hint 或者 hint 不生效时，优化器在生成计划的过程中，默认会优先选择使用 idx
1 这个索引
explain select * from t1 where c1 = 1 and c2 = 1;
+--------------------------------------------------------------------------------
--+
| Query Pla
n                                                                          |
+--------------------------------------------------------------------------------
--+
| ====================================================
=                              |
| |ID|OPERATOR        |NAME    |EST.ROWS|EST.TIME(us)
|                              |
| ----------------------------------------------------
-                              |
| |0 |TABLE RANGE SCAN|t1(idx1)|1       |7
|                              |
| ====================================================
=                              |
| Outputs & filters
:                                                                  |
| ------------------------------------
-                                            |
|   0 - output([t1.c1], [t1.c2], [t1.c3]), filter([t1.c2 = 1]), rowset=1
6           |
|       access([t1.__pk_increment], [t1.c1], [t1.c2], [t1.c3]), partitions(p0
)      |
|       is_index_back=true, is_global_index=false, filter_before_indexback[false]
,   |
|       range_key([t1.c1], [t1.__pk_increment]), range(1,MIN ; 1,MAX)
```

```
,              |
|       range_cond([t1.c1 = 1])
)                                                      |
+--------------------------------------------------------------------------------
--+

-- 生效的 index hint
explain select /*+index(t idx2)*/ * from t1 as t where c1 = 1 and c2 = 1;
+--------------------------------------------------------------------------------
--+
| Query Pla
n                                                                        |
+--------------------------------------------------------------------------------
--+
| ================================================
=                               |
| |ID|OPERATOR        |NAME   |EST.ROWS|EST.TIME(us)
|                               |
| ------------------------------------------------
-                               |
| |0 |TABLE RANGE SCAN|t(idx2)|1       |871
|                               |
| ================================================
=                               |
| Outputs & filters
:                                                                |
| ---------------------------------
-                                         |
|   0 - output([t.c1], [t.c2], [t.c3]), filter([t.c1 = 1]), rowset=1
6               |
|       access([t.__pk_increment], [t.c1], [t.c2], [t.c3]), partitions(p0
)          |
|       is_index_back=true, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t.c2], [t.__pk_increment]), range(1,MIN ; 1,MAX)
,                |
|       range_cond([t.c2 = 1])
)                                                       |
+--------------------------------------------------------------------------------
--+

-- 不生效的 index，因为 t1 已经被赋予了一个别名 t
explain select /*+index(t1 idx2)*/ * from t1 t where c1 = 1 and c2 = 1;
+--------------------------------------------------------------------------------
--+
| Query Pla
n                                                                        |
+--------------------------------------------------------------------------------
--+
| ================================================
=                               |
```

```
| |ID|OPERATOR        |NAME    |EST.ROWS|EST.TIME(us)
|                               |
| --------------------------------------------------
-                               |
| |0 |TABLE RANGE SCAN|t(idx1)|1       |7
|                               |
| ==================================================
=                               |
| Outputs & filters
:                                                                 |
| ------------------------------------
-                                             |
|   0 - output([t.c1], [t.c2], [t.c3]), filter([t.c2 = 1]), rowset=1
6                |
|       access([t.__pk_increment], [t.c1], [t.c2], [t.c3]), partitions(p0
)           |
|       is_index_back=true, is_global_index=false, filter_before_indexback[false]
,   |
|       range_key([t.c1], [t.__pk_increment]), range(1,MIN ; 1,MAX)
,                 |
|       range_cond([t.c1 = 1])
)                                                       |
+--------------------------------------------------------------------------------
--+
```

FULL Hint

FULL Hint 的语法是用于指定表使用主表扫描，等价于 `INDEX Hint /*+ INDEX(table_name PRIMARY)*/`。

示例如下：

```
create table t1(c1 int, c2 int, c3 int);

create index idx1 on t1(c1);

-- c1 上有一个索引 idx1，且过滤条件中的列和结果列都是 c1，所以优化器会默认选择使用 idx1
explain select c1 from t1 where c1 = 1;
+-----------------------------------------------------------------------+
| Query Plan                                                            |
+-----------------------------------------------------------------------+
| ===================================================                   |
| |ID|OPERATOR        |NAME    |EST.ROWS|EST.TIME(us)|                  |
| ---------------------------------------------------                   |
| |0 |TABLE RANGE SCAN|t1(idx1)|1       |4           |                  |
| ===================================================                   |
| Outputs & filters:                                                    |
```

```
| ------------------------------------                                  |
|   0 - output([t1.c1]), filter(nil), rowset=4                          |
|       access([t1.c1]), partitions(p0)                                 |
|       is_index_back=false, is_global_index=false,                     |
|       range_key([t1.c1], [t1.__pk_increment]), range(1,MIN ; 1,MAX),  |
|       range_cond([t1.c1 = 1])                                         |
+-----------------------------------------------------------------------+

-- 通过 hint 让计划不走索引，变为全表扫描
explain select /*+ FULL(t1) */ c1 from t1 where c1 = 1;
+-------------------------------------------------------------------------------
---+
| Query Pla
n                                                                         |
+-------------------------------------------------------------------------------
---+
| =============================================
=                                    |
| |ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)
|                                    |
| ---------------------------------------------
-                                    |
| |0 |TABLE FULL SCAN|t1  |1       |4
|                                    |
| =============================================
=                                    |
| Outputs & filters
:                                                                 |
| -----------------------------------
-                                              |
|   0 - output([t1.c1]), filter([t1.c1 = 1]), rowset=
4                               |
|       access([t1.c1]), partitions(p0
)                                              |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                        |
+-------------------------------------------------------------------------------
---+

-- 通过 hint 让计划不走索引，变为全表扫描，和上面那条 SQL 等价
explain select /*+ index(t1 PRIMARY) */ c1 from t1 where c1 = 1;
+-------------------------------------------------------------------------------
---+
| Query Pla
n                                                                         |
+-------------------------------------------------------------------------------
---+
| =============================================
=                                    |
```

```
| |ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)
|                                    |
| ----------------------------------------------
-                                    |
| |0 |TABLE FULL SCAN|t1  |1       |4
|                                    |
| ==============================================
=                                    |
| Outputs & filters
:                                                                  |
| ------------------------------------
-                                              |
|   0 - output([t1.c1]), filter([t1.c1 = 1]), rowset=
4                               |
|       access([t1.c1]), partitions(p0
)                                              |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                  |
+----------------------------------------------------------------------------------
---+
```

LEADING Hint

LEADING Hint 可以指定表的联接顺序。语法是：`/*+ LEADING(table_name_list)*/`。在 table_name_list 中可以使用 `()` 表示内部各表的联接优先级，指定复杂的联接顺序，比 ordered 有更大的灵活性。示例如下：

```
EXPLAIN BASIC SELECT /*+LEADING(d c b a)*/ * FROM t1 a, t1 b, t1 c, t1 d;
+----------------------------------------------------------------------------------
--------------
| Query Pla
n

+----------------------------------------------------------------------------------
--------------
| ========================================
=
| |ID|OPERATOR                     |NAME
|
| ----------------------------------------
-
| |0 |NESTED-LOOP JOIN CARTESIAN     |
|
| |1 |├─NESTED-LOOP JOIN CARTESIAN    |
|
```

```
| |2 || ├─NESTED-LOOP JOIN CARTESIAN |
|
| |3 || | ├─TABLE FULL SCAN |d
|
| |4 || | └─MATERIAL |
|
| |5 || | └─TABLE FULL SCAN |c
|
| |6 || └─MATERIAL |
|
| |7 || └─TABLE FULL SCAN |b
|
| |8 | └─MATERIAL |
|
| |9 | └─TABLE FULL SCAN |a
|
| ========================================
=
+---------------------------------------------------------------------------------
--------------

EXPLAIN BASIC SELECT /*+LEADING((d c) (b a))*/ * FROM  t1 a, t1 b, t1 c, t1 d;
+---------------------------------------------------------------------------------
--------------
| Query Pla
n

+---------------------------------------------------------------------------------
--------------
| ========================================
=
| |ID|OPERATOR |NAME
|
| ----------------------------------------
-
| |0 |NESTED-LOOP JOIN CARTESIAN |
|
| |1 |├─NESTED-LOOP JOIN CARTESIAN |
|
| |2 || ├─TABLE FULL SCAN |d
|
| |3 || └─MATERIAL |
|
| |4 || └─TABLE FULL SCAN |c
|
| |5 |└─MATERIAL |
|
| |6 | └─NESTED-LOOP JOIN CARTESIAN |
|
| |7 | ├─TABLE FULL SCAN |b
```

```
|
| |8 |      └─MATERIAL                |
|
| |9 |        └─TABLE FULL SCAN        |a
|
| =========================================
=
+-----------------------------------------------------------------------------------
--------------

EXPLAIN BASIC SELECT /*+LEADING((d c b) a))*/ * FROM t1 a, t1 b, t1 c, t1 d;
+-----------------------------------------------------------------------------------
--------------
| Query Pla
n

+-----------------------------------------------------------------------------------
--------------
| =========================================
=
| |ID|OPERATOR                      |NAME
|
| -----------------------------------------
-
| |0 |NESTED-LOOP JOIN CARTESIAN     |
|
| |1 |├─NESTED-LOOP JOIN CARTESIAN   |
|
| |2 ||  ├─NESTED-LOOP JOIN CARTESIAN |
|
| |3 ||  |  ├─TABLE FULL SCAN          |d
|
| |4 ||  |  └─MATERIAL                 |
|
| |5 ||  |    └─TABLE FULL SCAN        |c
|
| |6 ||  └─MATERIAL                    |
|
| |7 ||    └─TABLE FULL SCAN           |b
|
| |8 |└─MATERIAL                      |
|
| |9 |  └─TABLE FULL SCAN             |a
|
| =========================================
=
+-----------------------------------------------------------------------------------
--------------
```

为确保按照用户指定的顺序进行联接，LEADING Hint 的检查比较严格，如果发现 Hint 指定的 table_name 不存在，LEADING Hint 失效；如果发现 Hint 中存在重复表，LEADING Hint 失效。如果在 Optimizer 联接期间，按 table_id 无法在 From Items 中找到对应的表，则可能发生改写，那么该表及后面的表指定的 JOIN 顺序失效，该表前面的 JOIN 顺序依然有效。

> **说明**
>
> 如果同时使用 ordered 和 leading，则仅 ordered 生效。

USE_NL Hint

和 join 相关的 Hint 的基本结构如下：`join_hint_name ( @ qb_name table_name_list)`，基本语义为，当联接的右表匹配 `table_name_list` 时，按照 Hint 语义生成计划。一般需要使用 LEADING Hint 指定联接顺序，使 `table_name_list` 中的表为右表，否则 Hint 会随着联接顺序变化而失效。

其中 `table_name_list` 可有以下形式：

- 单表 `use_nl ( t1 )`：以 `t1` 为右表时使用 Nested Loop Join。
- 多个单表 `use_nl ( t1 t2 ... )`：以 `t1` 或 `t2` 等为右表时使用 Nested Loop Join。
- 多表 `use_nl ((t1 t2))`：以 `t1 join t2` 为右表时使用 Nested Loop Join，忽略 `t1`/`t2` 连接顺序和方法。
- 多组表 `use_nl (t1 (t2 t3) (t4 t5 t6) ... )`：以 `t1`/`t2 join t3`/`t4 join t5 join t6` 等表为右表时使用 Nested Loop Join。

在 USE_NL 指定的表是 NLJ 的右表，在联接的时候会使用 Nested Loop Join 算法，语法是：`/*+ USE_NL(table_name_list)*/`。示例如下：

```
CREATE TABLE t0(c1 INT, c2 INT, c3 INT);

CREATE TABLE t1(c1 INT, c2 INT, c3 INT);

CREATE TABLE t2(c1 INT, c2 INT, c3 INT);

-- 如果想让 join 的顺序是 t0 join t1, join 为 nest loop join, 则应该这样写 hint:
EXPLAIN BASIC SELECT /*+ LEADING(t0 t1) USE_NL(t1) */ * FROM t0, t1 WHERE t0.c1 =
t1.c1;
+------------------------------------------------------------------------------------
```

```
-----------+
| Query Pla
n
|
+-------------------------------------------------------------------------------------
-----------+
| ============================
=                                                                  |
| |ID|OPERATOR           |NAME
|                                                                  |
| ----------------------------
-                                                                  |
| |0 |NESTED-LOOP JOIN   |
|                                                                  |
| |1 |├─TABLE FULL SCAN  |t0
|                                                                  |
| |2 |└─MATERIAL         |
|                                                                  |
| |3 |  └─TABLE FULL SCAN|t1
|                                                                  |
| ============================
=                                                                  |
+-------------------------------------------------------------------------------------
-----------+

-- 如果想让 join 的顺序是 t0 join (t1 join t2)，且想让最外层的 join 为 nest loop join，则应该这样写 hint:
EXPLAIN BASIC SELECT /*+ LEADING(t0 (t1 t2)) USE_NL((t1 t2)) */ * FROM t0, t1, t2
WHERE t0.c1 =
+-------------------------------------------------------------------------------------
--------------
| Query Pla
n

+-------------------------------------------------------------------------------------
--------------
| ==============================
=
| |ID|OPERATOR             |NAME
|
| ------------------------------
-
| |0 |NESTED-LOOP JOIN     |
|
| |1 |├─TABLE FULL SCAN    |t0
|
| |2 |└─MATERIAL           |
|
| |3 |  └─HASH JOIN        |
|
| |4 |    ├─TABLE FULL SCAN|t1
```

```
|
| |5 |     └─TABLE FULL SCAN|t2
|
| ==============================
=
+-------------------------------------------------------------------------------
--------------
```

注意

USE_NL、USE_HASH、USE_MERGE 这三个 hint 往往会配合 LEADING 这个 hint 一起使用。因为当 join 的右表匹配 `table_name_list` 时，才会按照 hint 语义生成计划。

前面这句话理解起来可能不是那么直观，接下来，我们来举一个最简单的小例子。假设用户希望对一条 SQL：`SELECT * FROM t1, t2 WHERE t1.c1 = t2.c1;` 中 `t1 join t2` 对应计划中的 join 计算方式进行干预。

原本的计划空间有六种，分别是：

- t1 nest loop join t2
- t1 hash join t2
- t1 merge join t2
- t2 nest loop join t1
- t2 hash join t1
- t2 merge join t1

如果加了 `hint: /*+ USE_NL(t1)*/`，则计划空间减少为四种，分别是：

- t1 nest loop join t2
- t1 hash join t2
- t1 merge join t2
- t2 nest loop join t1

因为计划空间中，只有当 `t1` 为 join 的右表时，才会按照 hint 生成 `t2 nest loop join t1` 的计划；当 `t1` 为 join 的左表时，则不受这个 hint 的影响。

如果加了 hint：/*+ LEADING(t2 t1) USE_NL(t1)*/，计划空间就只有确定的一种：t2 nest loop join t1

USE_HASH Hint

和 USE_NL 类似，USE_HASH 指定表作为 HJ 的右表，在联接的时候使用 Hash Join 算法，语法是：`/*+ USE_HASH(table_name_list)*/`。示例如下：

```
CREATE TABLE t0(c1 INT, c2 INT, c3 INT);

CREATE TABLE t1(c1 INT, c2 INT, c3 INT);

EXPLAIN BASIC SELECT /*+LEADING(t0 t1) USE_HASH(t1)*/ * FROM t0, t1 WHERE t0.c1 =
t1.c1;
+-------------------------------------------------------------------------------
-----------+
| Query Pla
n
|
+-------------------------------------------------------------------------------
-----------+
| ==========================
=                                                               |
| |ID|OPERATOR         |NAME
|                                                               |
| --------------------------
-                                                               |
| |0 |HASH JOIN        |
|                                                               |
| |1 |├─TABLE FULL SCAN|t0
|                                                               |
| |2 |└─TABLE FULL SCAN|t1
|                                                               |
| ==========================
=                                                               |
+-------------------------------------------------------------------------------
-----------+
```

USE_MERGE Hint

和 USE_NL 类似，USE_MERGE 指定表作为 MJ 的右表，在联接的时候使用 Merge Join 算法，语法是：`/*+ USE_MERGE(table_name_list)*/`。示例如下：

```
CREATE TABLE t0(c1 INT, c2 INT, c3 INT);
```

```
CREATE TABLE t1(c1 INT, c2 INT, c3 INT);

EXPLAIN BASIC SELECT /*+LEADING(t0 t1) USE_MERGE(t1)*/ * FROM t0, t1 WHERE t0.c1
= t1.c1;
+------------------------------------------------------------------------------------
-----------+
| Query Pla
n
|
+------------------------------------------------------------------------------------
-----------+
| ============================
=                                                              |
| |ID|OPERATOR           |NAME
|                                                              |
| ----------------------------
-                                                              |
| |0 |MERGE JOIN         |
|                                                              |
| |1 |├─SORT              |
|                                                              |
| |2 ||  └─TABLE FULL SCAN|t0
|                                                              |
| |3 |└─SORT              |
|                                                              |
| |4 |   └─TABLE FULL SCAN|t1
|                                                              |
| ============================
=                                                              |
+------------------------------------------------------------------------------------
-----------+
```

说明

OceanBase 数据库中 Merge Join 必须有等值的 join 联接条件。如果无等值条件的两个表联接，USE_MERGE 会失效。

PARALLEL Hint

PARALLEL 指定语句级别的并行度。语法是：`/*+ PARALLEL(n)*/`。其中 `n` 为整数，表示 SQL 的全局并行度。示例如下：

```
CREATE TABLE tbl1 (col1 INT) PARTITION BY HASH(col1) ;
```

```
EXPLAIN BASIC SELECT /*+ PARALLEL(5) */ * FROM tbl1;
+-----------------------------------------------------------------------+
| Query Plan                                                            |
+-----------------------------------------------------------------------+
| ===================================                                   |
| |ID|OPERATOR             |NAME    |                                   |
| -----------------------------------                                   |
| |0 |PX COORDINATOR       |        |                                   |
| |1 | └─EXCHANGE OUT DISTR |:EX10000|                                   |
| |2 |   └─PX BLOCK ITERATOR|        |                                   |
| |3 |     └─TABLE FULL SCAN|tbl1    |                                   |
| ===================================                                   |
| Outputs & filters:                                                    |
| -----------------------------------                                   |
|   0 - output([INTERNAL_FUNCTION(tbl1.col1)]), filter(nil), rowset=16 |
|   1 - output([INTERNAL_FUNCTION(tbl1.col1)]), filter(nil), rowset=16 |
|       dop=5                                                           |
|   2 - output([tbl1.col1]), filter(nil), rowset=16                     |
|   3 - output([tbl1.col1]), filter(nil), rowset=16                     |
|       access([tbl1.col1]), partitions(p0)                             |
|       is_index_back=false, is_global_index=false,                     |
|       range_key([tbl1.__pk_increment]), range(MIN ; MAX)always true  |
+-----------------------------------------------------------------------+
```

计划中显示的 `dop=5` 表明 hint 已经生效。

OceanBase 数据库同时也支持表级别的 PARALLEL Hint，语法是：`/*+ PARALLEL(table_name n)*/`。示例如下：

```
CREATE TABLE t1 (c1 INT, c2 INT) PARTITION BY HASH(c1) PARTITIONS 5;

CREATE TABLE t2 (c1 INT PRIMARY KEY, c2 INT) PARTITION BY HASH(c1) PARTITIONS 4;

EXPLAIN SELECT /*+ PARALLEL(3) PARALLEL(t2 5)*/* FROM t1, t2 WHERE t1.c1 = t2.c1;
EXPLAIN SELECT /*+ PARALLEL(3) PARALLEL(t1 4) PARALLEL(t2 5)*/* FROM t1, t2 WHERE
t1.c1 = t2.c1;
+----------------------------------------------------------------------------------
------+
| Query Pla
n                                                                          |
+----------------------------------------------------------------------------------
------+
| =======================================================================
=               |
| |ID|OPERATOR                        |NAME    |EST.ROWS|EST.TIME(us)
|               |
| -----------------------------------------------------------------------
-               |
| |0 |PX COORDINATOR                  |        |1       |9
```

```
|                |
| |1 | └─EXCHANGE OUT DISTR                |:EX10001|1        |9
|                |
| |2 |   └─HASH JOIN                        |        |1        |9
|                |
| |3 |     ├─PART JOIN FILTER CREATE        |:RF0000 |1        |5
|                |
| |4 |     |  └─PX PARTITION ITERATOR        |        |1        |5
|                |
| |5 |     |    └─TABLE FULL SCAN           |t1      |1        |5
|                |
| |6 |     └─EXCHANGE IN DISTR              |        |1        |4
|                |
| |7 |       └─EXCHANGE OUT DISTR (PKEY)    |:EX10000|1        |4
|                |
| |8 |         └─PX BLOCK HASH JOIN-FILTER|:RF0000 |1        |4
|                |
| |9 |           └─TABLE FULL SCAN          |t2      |1        |4
|                |
| ==================================================================
=                |
| Outputs & filters
:                                                                      |
| ------------------------------------
-                                                          |
|   0 - output([INTERNAL_FUNCTION(t1.c1, t1.c2, t2.c1, t2.c2)]), filter(nil), rows
et=16 |
|   1 - output([INTERNAL_FUNCTION(t1.c1, t1.c2, t2.c1, t2.c2)]), filter(nil), rows
et=16 |
|       dop=
4                                                                             |
|   2 - output([t1.c1], [t2.c1], [t1.c2], [t2.c2]), filter(nil), rowset=1
6               |
|       equal_conds([t1.c1 = t2.c1]), other_conds(nil
)                                   |
|   3 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                                 |
|       RF_TYPE(bloom), RF_EXPR[calc_tablet_id(t1.c1)
]                                   |
|   4 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                                 |
|       affinitiz
e                                                                      |
|   5 - output([t1.c1], [t1.c2]), filter(nil), rowset=1
6                                 |
|       access([t1.c1], [t1.c2]), partitions(p[0-4]
)                                     |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                     |
```

```
|   6 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                                 |
|   7 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                                 |
|       (#keys=1, [t2.c1]), dop=
5                                                       |
|   8 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                                 |
|   9 - output([t2.c1], [t2.c2]), filter(nil), rowset=1
6                                 |
|       access([t2.c1], [t2.c2]), partitions(p[0-3]
)                                     |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t2.c1]), range(MIN ; MAX)always tru
e                                  |
+----------------------------------------------------------------------------------
------+
```

READ_CONSISTENCY(WEAK) Hint

READ_CONSISTENCY Hint 用于指定 SQL 的读一致性级别为弱一致性，语法是：`/*+ READ_CONSISTENCY(WEAK)*/`。示例如下：

```
-- 读取 t1 表备副本上的数据
SELECT /*+ READ_CONSISTENCY(WEAK) */ * FROM t1;
```

对于实时性要求不高的偏 AP 类请求，为了实现 AP 业务和 TP 业务的读写分离，减少 AP 请求对主副本的影响，可以设置弱一致性读。设置弱一致性读的 hint 之后，读操作会到备副本上执行。弱一致性读的概念详见官网《OceanBase 数据库》文档 参考指南/系统原理/事务管理/事务并发和一致性/弱一致性读。

QUERY_TIMEOUT Hint

QUERY_TIMEOUT Hint 用于设置查询语句的超时时间，语法是：`/*+ query_timeout(n)*/`，其中的 `n` 是整数，单位是微秒（us）。示例如下：

```
-- 指定这条 SQL 执行的超时时间为 100000000 微秒，即 100 秒。
SELECT /*+ query_timeout(100000000) */ * FROM t1;
```

说明

- 用户可以通过执行 `SHOW VARIABLES LIKE 'ob_query_timeout';` 查看默认的 SQL 超时时间。
- OceanBase 数据库除了支持通过 hint 修改 SQL 级别的 SQL 超时时间，还支持通过执行 `SET SESSION ob_query_timeout = 100000000;` 修改会话（session）级别的SQL 超时时间，支持通过执行 `SET GLOBAL ob_query_timeout = 100000000;` 修改租户级别的 SQL 超时时间。

通过 Outline 进行计划绑定

通过对某条 SQL 创建 Outline 可实现计划绑定。

在生产系统上线前，可以直接在 SQL 语句中添加 Hint，控制优化器按 Hint 指定的行为进行计划生成。

但对于已上线的业务，如果出现优化器选择的计划不够优时，则需要在线进行计划绑定，无需业务对 SQL 进行更改。而是通过 DDL 操作将一组 Hint 加入到特定的 SQL 中，从而使优化器根据指定的一组 Hint，对该 SQL 生成更优计划。该组 Hint 称为 Outline。

Outline 相关的字典视图

Outline 视图为 `DBA_OB_OUTLINES`，其字段说明如下表所示。

字段名称	类型（MySQL 模式）	类型（Oracle 模式）	描述
CREATE_TIME	TIMESTAMP(6)	TIMESTAMP(6)	创建时间戳
MODIFY_TIME	TIMESTAMP(6)	TIMESTAMP(6)	修改时间戳
TENANT_ID	BIGINT(20)	NUMBER(38)	租户 ID
DATABASE_ID	BIGINT(20)	NUMBER(38)	数据库 ID
OUTLINE_ID	BIGINT(20)	NUMBER(38)	Outline ID
DATABASE_NAME	VARCHAR2(128)	VARCHAR2(128)	数据库名称
OUTLINE_NAME	VARCHAR2(128)	VARCHAR2(128)	Outline 名称
VISIBLE_SIGN	LONGTEXT	CLOB	Signature 的反序列化结果，为了便于查看 Sig

ATURE			nature 的信息。
SQL_TEXT	LONGTEXT	CLOB	创建 Outline 时，在 ON 子句中指定的 SQL。
OUTLINE_TARGET	LONGTEXT	CLOB	创建 Outline 时，在 TO 子句中指定的 SQL。
OUTLINE_SQL	LONGTEXT	CLOB	具有完整 Outline 信息的 SQL
SQL_ID	VARCHAR2(32)	VARCHAR2(32)	SQL 标识符
OUTLINE_CONTENT	LONGTEXT	CLOB	完整的执行计划 Outline 信息

创建 Outline

OceanBase 数据库支持通过两种方式创建 Outline，一种是通过 SQL_TEXT (用户执行的带参数的原始语句)，另一种是通过 SQL_ID。

注意

创建 Outline 需要进入对应的数据库下执行。

使用 SQL_TEXT 创建 Outline

使用 SQL_TEXT 创建 Outline 后，会生成一个 Key-Value 对存储在 Map 中，其中 Key 为绑定的 SQL 参数化后的文本，Value 为绑定的 Hint。具体参数化原则，详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 执行计划/快速参数化。

使用 SQL_TEXT 创建 Outline 的语法如下：

```
CREATE [OR REPLACE] OUTLINE <outline_name> ON <stmt>;
```

- 指定 `OR REPLACE` 后，可以对已经存在执行计划进行替换。
- 其中 `stmt` 一般为一个带有 Hint 和原始参数的 DML 语句。示例如下：

```
CREATE OUTLINE outline1 ON
SELECT /*+NO_REWRITE*/ *
FROM tbl1
WHERE col1 = 4 AND col2 = 6 ORDER BY 2 TO SELECT  * FROM tbl1 WHERE col1 = 4 AND c
ol2 = 6 ORDER BY 2;
```

注意

在使用 target_stmt 时，严格要求 stmt 与 target_stmt 在去掉 Hint 后完全匹配。

如下示例中，优化器可能默认会选择走 idx_c2 索引。索引上只有索引列 c2 和主键列 c1（索引上的主键列在索引中是隐藏列，仅用于索引回表），因为结果中还需要返回 c3 列的值，所以需要索引回表查出 c3 列。

索引回表一行的代价远远大于全表扫描一行的代价（大概是十倍左右的代价）。如果用户已知 c2 列的过滤性很差，所以把该 SQL 改为强制走全表扫描，性能反倒会更优。此时可以通过创建 Outline 将该 SQL 绑定全表扫描的计划。

```
CREATE TABLE t1 (c1 INT PRIMARY KEY, c2 INT, c3 INT, INDEX idx_c2(c2));

INSERT INTO t1 VALUES(1, 1, 1), (2, 1, 2), (3, 1, 3);

EXPLAIN SELECT * FROM t1 WHERE c2 = 1;
+-------------------------------------------------------------------+
| Query Plan                                                        |
+-------------------------------------------------------------------+
| ======================================================            |
| |ID|OPERATOR        |NAME      |EST.ROWS|EST.TIME(us)|            |
| ------------------------------------------------------            |
| |0 |TABLE RANGE SCAN|t1(idx_c2)|1       |7           |            |
| ======================================================            |
| Outputs & filters:                                                |
| -------------------------------------                             |
|   0 - output([t1.c1], [t1.c2], [t1.c3]), filter(nil), rowset=16 |
|       access([t1.c1], [t1.c2], [t1.c3]), partitions(p0)          |
|       is_index_back=true, is_global_index=false,                  |
|       range_key([t1.c2], [t1.c1]), range(1,MIN ; 1,MAX),          |
|       range_cond([t1.c2 = 1])                                     |
+-------------------------------------------------------------------+
```

根据如下 SQL 语句创建 Outline：

```
CREATE OR REPLACE OUTLINE otl_t1_full ON SELECT /*+ full(t1) */ * FROM t1 WHERE c
2 = 1;
```

为了确认某一条 SQL 在执行时的真实计划，不应该直接执行 explain 语句，而是要通过查询上

述的两个计划监控视图 `GV$OB_PLAN_CACHE_PLAN_STAT` 和 `GV$OB_PLAN_CACHE_PLAN_EXPLAIN`。详见下文 确定 Outline 创建生效 部分。

使用 SQL_ID 创建 Outline

使用 SQL_ID 创建 Outline 的语法如下:

```
CREATE OUTLINE outline_name ON sql_id USING HINT hint_text;
```

`SQL_ID` 为需要绑定的 SQL 对应的 `SQL_ID`，可以查询 `GV$OB_PLAN_CACHE_PLAN_STAT` 获取。

```
select
  TENANT_ID,
  SVR_IP,
  SVR_PORT,
  PLAN_ID,
  LAST_ACTIVE_TIME,
  QUERY_SQL,
  SQL_ID
from
  oceanbase.GV$OB_PLAN_CACHE_PLAN_STAT
where
  QUERY_SQL = 'SELECT * FROM t1 WHERE c2 = 1';
```

输出如下:

```
+-----------+-------------+----------+---------+----------------------------+----
---------------------------+----------------------------------+
| TENANT_ID | SVR_IP      | SVR_PORT | PLAN_ID | LAST_ACTIVE_TIME           | QUE
RY_SQL                     | SQL_ID                           |
+-----------+-------------+----------+---------+----------------------------+----
---------------------------+----------------------------------+
|      1002 | 10.10.10.1  |    22602 |   49820 | 2024-03-13 18:49:17.375906 | SEL
ECT * FROM t1 WHERE c2 = 1 | ED570339F2C856BA96008A29EDF04C74 |
+-----------+-------------+----------+---------+----------------------------+----
---------------------------+----------------------------------+
```

使用 `SQL_ID` 绑定 Outline，如下例所示:

```
DROP OUTLINE otl_t1_full;

CREATE OUTLINE otl_t1_idx_c2 ON "ED570339F2C856BA96008A29EDF04C74" USING HINT /*+
INDEX(t1 idx_c2)*/ ;
```

注意

- Hint 格式为 /*+ xxx*/。
- 使用 SQL_TEXT 方式创建的 Outline 会覆盖 SQL_ID 方式创建的 Outline。SQL_TEXT 方式创建的优先级更高。
- 如果 SQL_ID 对应的 SQL 语句已经有 Hint，则创建 Outline 指定的 Hint 会覆盖原始语句中所有 Hint。

Outline Data 是优化器为了完全复现某一计划而生成的一组 Hint 信息，以 `BEGIN_OUTLINE_DATA` 开始，并以 `END_OUTLINE_DATA` 结束。Outline Data 信息可以通过 EXPLAIN outline 命令获得，如下例所示：

```
EXPLAIN outline SELECT/*+ index(t1 idx_c2)*/ * FROM t1 WHERE c2 = 1;
```

输出如下：

```
-----------------------------------------------------------------+
| Query Plan                                                      |
+-----------------------------------------------------------------+
| =====================================================           |
| |ID|OPERATOR        |NAME      |EST.ROWS|EST.TIME(us)|          |
| -----------------------------------------------------           |
| |0 |TABLE RANGE SCAN|t1(idx_c2)|3       |12          |          |
| =====================================================           |
| Outputs & filters:                                              |
| -------------------------------------                           |
|   0 - output([t1.c1], [t1.c2], [t1.c3]), filter(nil), rowset=16 |
|       access([t1.c1], [t1.c2], [t1.c3]), partitions(p0)         |
|       is_index_back=true, is_global_index=false,                |
|       range_key([t1.c2], [t1.c1]), range(1,MIN ; 1,MAX),        |
|       range_cond([t1.c2 = 1])                                   |
| Outline Data:                                                   |
| -------------------------------------                           |
|   /*+                                                           |
|       BEGIN_OUTLINE_DATA                                        |
|       INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx_c2")              |
|       OPTIMIZER_FEATURES_ENABLE('4.0.0.0')                      |
|       END_OUTLINE_DATA                                          |
|   */                                                            |
+-----------------------------------------------------------------+
```

其中 Outline Data 信息如下所示：

```
Outline Data:
-------------------------------------
/*+
    BEGIN_OUTLINE_DATA
    INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx_c2")
    OPTIMIZER_FEATURES_ENABLE('4.0.0.0')
    END_OUTLINE_DATA
*/
```

Outline Data 也属于 Hint，因此可以用在计划绑定的过程中，如下例所示：

```
DROP OUTLINE otl_t1_idx_c2;

CREATE OUTLINE otl_t1_idx_c2
     ON "ED570339F2C856BA96008A29EDF04C74" USING HINT
/*+
    BEGIN_OUTLINE_DATA
    INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx_c2")
    OPTIMIZER_FEATURES_ENABLE('4.0.0.0')
    END_OUTLINE_DATA
*/;
```

确定 Outline 创建生效

确定创建的 Outline 是否成功且符合预期，需要进行如下三步的验证：

1. 确定 Outline 创建成功。通过查看 `DBA_OB_OUTLINES` 视图，确认是否成功创建对应名称的 Outline。

```
SELECT * FROM oceanbase.DBA_OB_OUTLINES WHERE OUTLINE_NAME = 'otl_t1_full'\G
```

输出如下：

```
*************************** 1. row ***************************
      CREATE_TIME: 2024-03-13 18:38:18.807692
      MODIFY_TIME: 2024-03-13 18:39:57.210761
        TENANT_ID: 1002
      DATABASE_ID: 500001
       OUTLINE_ID: 500133
    DATABASE_NAME: test
     OUTLINE_NAME: otl_t1_full
VISIBLE_SIGNATURE: SELECT * FROM t1 WHERE c2 = ?
         SQL_TEXT: SELECT/*+ full(t1) */ * FROM t1 WHERE c2 = 1
   OUTLINE_TARGET:
```

```
      OUTLINE_SQL: SELECT /*+BEGIN_OUTLINE_DATA FULL(@"SEL$1" "test"."t1"@"SEL$1"
) OPTIMIZER_FEATURES_ENABLE('4.0.0.0') END_OUTLINE_DATA*/* FROM t1 WHERE c2 = 1
           SQL_ID:
  OUTLINE_CONTENT: /*+BEGIN_OUTLINE_DATA FULL(@"SEL$1" "test"."t1"@"SEL$1") OPTIMI
ZER_FEATURES_ENABLE('4.0.0.0') END_OUTLINE_DATA*/
```

2. 确定新的 SQL 是否通过绑定的 Outline 生成了新执行计划。当绑定 Outline 的 SQL 执行新的查询后，查询 `GV$OB_PLAN_CACHE_PLAN_STAT` 表中该 SQL 对应的计划信息中的 `outline_id`。如果 `outline_id` 与在 `DBA_OB_OUTLINES` 中查到的 `outline_id` 相同，则表示是按绑定的 Outline 生成的执行计划，否则不是。

```
SELECT SQL_ID, PLAN_ID, STATEMENT, OUTLINE_ID, OUTLINE_DATA
FROM oceanbase.GV$OB_PLAN_CACHE_PLAN_STAT
WHERE STATEMENT LIKE '%SELECT * FROM t1 WHERE c2 =%'\G
```

输出如下：

```
*************************** 1. row ***************************
      SQL_ID: ED570339F2C856BA96008A29EDF04C74
     PLAN_ID: 49820
   STATEMENT: SELECT * FROM t1 WHERE c2 = ?
  OUTLINE_ID: 500133
OUTLINE_DATA: /*+BEGIN_OUTLINE_DATA FULL(@"SEL$1" "test"."t1"@"SEL$1") OPTIMIZER_F
EATURES_ENABLE('4.0.0.0') END_OUTLINE_DATA*/
```

3. 确定生成的执行计划是否符合预期。确定是通过绑定的 Outline 生成的计划后，需要确定生成的计划是否符合预期，可以通过查询 `GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 表查看 plan_cache 中缓存的执行计划形状，具体查看方式可参见官网《OceanBase 数据库》文档参考指南/性能调优/SQL 调优指南/SQL 执行计划/实时执行计划展示。

   1. 通过 `GV$OB_PLAN_CACHE_PLAN_STAT` 查询上述 SQL 对应的 `TENANT_ID`、`SVR_IP`、`SVR_PORT`、`PLAN_ID` 字段。输出如下：

```
+-----------+--------------+----------+---------+------------------------
----+--------------------------------+
| TENANT_ID | SVR_IP       | SVR_PORT | PLAN_ID | LAST_ACTIVE_TIM
E           | QUERY_SQL                        |
+-----------+--------------+----------+---------+------------------------
----+--------------------------------+
|      1002 | 10.10.10.1   |    22602 |   49820 | 2024-03-13 18:49:17.375
906 | SELECT * FROM t1 WHERE c2 = 1 |
+-----------+--------------+----------+---------+------------------------
```

```
----+------------------------------+
```

2. 根据输出，通过 `oceanbase.GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 来看 SQL 对应的真实计划，`GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 视图各字段含义详见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/性能视图/GV$OB_PLAN_CACHE_PLAN_EXPLAIN。输出如下：您也可以通过 `DBMS_XPLAN.DISPLAY_CURSOR` 来看 SQL 对应的真实计划，展示出来的信息比 `oceanbase.GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 更像 explain 展示出来的信息，详见官网《OceanBase 数据库》文档 参考指南/SQL 参考/PL 参考/PL 参考（MySQL 模式）/PL 系统包/DBMS_XPLAN/DISPLAY_CURSOR。输出如下：

```
==========================================================================
=======================
|ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)|REAL.ROWS|REAL.TIME(us)|I
O TIME(us)|CPU TIME(us)|
--------------------------------------------------------------------------
-----------------------
|0 |TABLE FULL SCAN|t1  |1       |4            |3         |0             |
0          |0            |
==========================================================================
=======================
Outputs & filters:
-------------------------------------
  0 - output([t1.c1], [t1.c2], [t1.c3]), filter([t1.c2 = :0]), rowset=16
      access([t1.c1], [t1.c2], [t1.c3]), partitions(p0)
      is_index_back=false, is_global_index=false, filter_before_indexback
[false],
      range_key([t1.c1]), range(MIN ; MAX)always true
```

4. 通过 `GV$OB_PLAN_CACHE_PLAN_STAT` 查询上述 SQL 对应的 `TENANT_ID`、`SVR_IP`、`SVR_PORT`、`PLAN_ID` 字段。

```
select
  TENANT_ID,
  SVR_IP,
  SVR_PORT,
  PLAN_ID,
  LAST_ACTIVE_TIME,
  QUERY_SQL
from
  oceanbase.GV$OB_PLAN_CACHE_PLAN_STAT
where
  QUERY_SQL = 'SELECT * FROM t1 WHERE c2 = 1';
```

```

```

输出如下：

```
+-----------+-------------+---------+--------+----------------------------+----
---------------------------+
| TENANT_ID | SVR_IP      | SVR_PORT | PLAN_ID | LAST_ACTIVE_TIME           | QUE
RY_SQL                     |
+-----------+-------------+---------+--------+----------------------------+----
---------------------------+
|      1002 | 10.10.10.1  |    22602 |   49820 | 2024-03-13 18:49:17.375906 | SEL
ECT * FROM t1 WHERE c2 = 1 |
+-----------+-------------+---------+--------+----------------------------+----
---------------------------+
```

5. 根据输出，通过 `oceanbase.GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 来看 SQL 对应的真实计划，`GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 视图各字段含义详见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/性能视图/GV$OB_PLAN_CACHE_PLAN_EXPLAIN。

```
SELECT
  *
FROM
  oceanbase.GV$OB_PLAN_CACHE_PLAN_EXPLAIN
WHERE
  tenant_id = 1002
  AND SVR_IP = '10.10.10.1'
  AND SVR_PORT = 22602
  AND PLAN_ID = 49820;
```

输出如下：

```
+-----------+-------------+---------+--------+------------+-------------+-----
----------+------+------+------+----------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
-------------------------------------+
| TENANT_ID | SVR_IP      | SVR_PORT | PLAN_ID | PLAN_DEPTH | PLAN_LINE_ID | OPER
ATOR          | NAME | ROWS | COST | PROPERT
Y

                                                                             |
+-----------+-------------+---------+--------+------------+-------------+-----
----------+------+------+------+----------------------------------------------
--------------------------------------------------------------------------------
--------------------------------------------------------------------------------
```

```
-----------------------------------+
|      1002 | 10.10.10.1   |    22602 |   49820 |          0 |            0 | PHY_
TABLE_SCAN | t1    |    1 |     3 | table_rows:3, physical_range_rows:3, logical_ran
ge_rows:3, index_back_rows:0, output_rows:1, avaiable_index_name[idx_c2,t1], prune
d_index_name[idx_c2], estimation info[table_id:500131, (table_type:0, version:-1--
1--1, logical_rc:3, physical_rc:3)] |
+-----------+--------------+----------+---------+------------+--------------+-----
-----------+------+------+------+-------------------------------------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-----------------------------------+
```

您也可以通过 `DBMS_XPLAN.DISPLAY_CURSOR` 来看 SQL 对应的真实计划，展示出来的信息比 `oceanbase.GV$OB_PLAN_CACHE_PLAN_EXPLAIN` 更像 explain 展示出来的信息，详见官网《OceanBase 数据库》文档 参考指南/SQL 参考/PL 参考/PL 参考（MySQL 模式）/PL 系统包/DBMS_XPLAN/DISPLAY_CURSOR。

```
SELECT DBMS_XPLAN.DISPLAY_CURSOR(
       49820,
        'typical',
        '10.10.10.1',
        22602,
        1002
      ) FROM DUAL;
```

输出如下：

```
==================================================================================
==============
|ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)|REAL.ROWS|REAL.TIME(us)|IO TIME(us)
|CPU TIME(us)|
----------------------------------------------------------------------------------
--------------
|0 |TABLE FULL SCAN|t1  |1       |4           |3         |0             |0
|0           |
==================================================================================
==============
Outputs & filters:
-------------------------------------
  0 - output([t1.c1], [t1.c2], [t1.c3]), filter([t1.c2 = :0]), rowset=16
      access([t1.c1], [t1.c2], [t1.c3]), partitions(p0)
      is_index_back=false, is_global_index=false, filter_before_indexback[false],
      range_key([t1.c1]), range(MIN ; MAX)always true
```

删除 Outline

删除 Outline 后，对应 SQL 将不再依据所绑定的 Outline 重新生成执行计划。删除 Outline 的语法如下：

```
DROP OUTLINE outline_name;
```

计划绑定与执行计划缓存关系

- 使用 SQL_TEXT 创建 Outline 后，SQL 请求生成新计划查找 Outline 使用的 Key 与计划缓存使用的 Key 是相同的，即均是 SQL 参数化后的文本串。
- 当创建和删除 Outline 后，对应 SQL 有新的请求时，会触发执行计划缓存中对应执行计划失效，更新为根据绑定的 Outline 所生成的执行计划。

计划绑定与限流规则的关系

从 OceanBase 数据库 4.2.2 版本开始，数据库的 `CREATE OUTLINE` 和 `ALTER OUTLINE` 命令扩展了功能。除了原有的绑定特定查询的执行计划外，还允许用户为查询设置最大并发执行次数的限制。限流功能能够帮助大家更有效地管理数据库负载，防止因高并发查询而导致的性能问题。

例如：

- 使用 `CREATE OUTLINE` 创建了一个 OUTLINE，其中包含了 `USE_NL(tbl2)` 提示，这告诉优化器对于表 `tbl2` 使用 nest loop join，同时 `MAX_CONCURRENT(1)` 这个 Hint 在下面这个示例中表示同一时刻，只允许一个线程执行查询：

```
CREATE OUTLINE otl2 ON SELECT /*+ USE_NL(tbl2) MAX_CONCURRENT(1) */ * FROM t WHER
E c1 = ?;
```

- 同样地，使用 `ALTER OUTLINE` 修改现有 OUTLINE，同样可以达到上述效果。

```
ALTER OUTLINE otl2 ON SELECT /*+ USE_NL(tbl2) MAX_CONCURRENT(1) */ * FROM t WHERE
c1 = ?;
```

7.6 常见的 SQL 调优方式

在上一节 阅读和管理 OceanBase 数据库 SQL 执行计划 中，大家已经学习如何通过 EXPLAIN 命令查看优化器针对 SQL 生成的逻辑执行计划，以及如何通过 Hint 和 Outline 来人为控制优化器的行为，使优化器生成指定的计划。

在本节中，会以上一节的内容为基础，继续为大家介绍用户需要了解的 OceanBase 数据库 SQL 性能调优中最基础的内容。本节的内容整体分为两个部分，第一部分是统计信息和计划缓存的介绍，第二部分是 OceanBase 数据库的使用者需要了解的几种性能调优手段。

统计信息

在数据库中，优化器针对每一个输入的 SQL 查询都会尝试生成最优的执行计划，而生成最优的执行计划往往需要准确的统计信息。统计信息指的是优化器统计信息（optimizer statistics），它是一个描述数据库中表和列信息的数据集合，是代价模型选取最优执行计划的非常关键的部分。

优化器代价模型（optimizer cost model）依赖于查询中涉及到的表、列、谓词等对象的统计信息来优化计划的选择。准确有效的统计信息能够帮助优化器选择出最优的执行计划。

说明

这里需要先介绍一下“直方图”和“桶”的概念，作为理解下面内容的基础。

- 直方图是一种特殊的列统计信息。默认情况下，优化器会认为列的数据是分布均匀的，会根据这一特征来估计行数，但是在真实的场景中，大多数表的数据分布都是不均匀的，这就会导致优化器错误的估计行数而选择不到最优的执行计划，这种场景下就需要有直方图。
- 直方图通过将数据保存到一系列有序的桶中来描述其列的整体数据分布特征，优化器可以依据直方图来估算出更准确的行数。直方图中桶的个数默认是 254 个，桶的数量越多，统计信息就会越准确，但是收集统计信息的代价也会越大。

本节只会介绍用户和 DBA 同学最需要了解的“如何收集和查询统计信息”，不会介绍如何管理统计信息，也不会介绍优化器使用统计信息进行估行的机制和原理。

OceanBase 数据库 V4.x 版本的统计信息收集依靠手动和自动这两种方式。

手动统计信息收集

为了让优化器生成的计划更优，用户可以通过手动执行命令让优化器收集相关的统计信息。

目前 OcenaBase 优化器针对手动统计信息收集提供了两种方式：DBMS_STATS（推荐）、ANALYZE 命令行；推荐使用 DBMS_STATS 系统包进行手动统计信息收集，因为 DBMS_STATS 系统包支持的功能会更加丰富。DBMS_STATS 系统包的介绍详见官网《OceanBase 数据库》文档 参考指南/SQL 参考/PL 参考/PL 参考（MySQL 模式）/PL 系统包/DBMS_STATS 章节。

DBMS_STATS 系统包（推荐）

DBMS_STATS 系统包里最常用的两个存储过程是:

- `GATHER_TABLE_STATS`，用于收集某张表的统计信息，语法和参数详见官网《OceanBase 数据库》文档 参考指南/SQL 参考/PL 参考/PL 参考（MySQL 模式）/PL 系统包/DBMS_STATS/GATHER_TABLE_STATS。
- `GATHER_SCHEMA_STATS`，用于收集某个库中所有表的统计信息，语法和参数详见官网《OceanBase 数据库》文档 参考指南/SQL 参考/PL 参考/PL 参考（MySQL 模式）/PL 系统包/DBMS_STATS/GATHER_SCHEMA_STATS。

以下是使用这两个存储过程的几个示例:

- 收集 `TEST` 库中表 `T1` 的全局级别的统计信息，所有列的桶个数设定为 `128`

```
call dbms_stats.gather_table_stats('TEST', 'T1', granularity=>'GLOBAL', method_opt
=>'FOR ALL COLUMNS SIZE 128');
```

- 收集 `TEST` 库中表 `T_PART1` 的分区级别的统计信息，并行度 `64`，只收集数据分布不均匀的列的直方图

```
call dbms_stats.gather_table_stats('TEST', 'T_PART1', degree=>64, granularity=>'PA
RTITION', method_opt=>'FOR ALL COLUMNS SIZE SKEWONLY');
```

- 收集 `TEST` 库中表 `T_SUBPART1` 所有的统计信息，只收集 50% 的数据

```
call dbms_stats.gather_table_stats('TEST', 'T_SUBPART1', estimate_percent=> '50',
```

```
granularity=>'ALL');
```

- 收集 `TEST` 库中所有表统计信息，并行度 `128`

```
call dbms_stats.gather_schema_stats('TEST', degree=>128);
```

> **注意**
>
> GATHER_TABLE_STATS 和 GATHER_SCHEMA_STATS 这两个存储过程的第一个参数均为 database_name 而非 user_name。OceanBase 官网上某些版本的内容可能有误。

除了上述两个最常用的存储过程以外，DMBS_STATS 系统包还提供了 `GATHER_INDEX_STATS` 用于收集索引统计信息、`GATHER_DATABASE_STATS_JOB_PROC` 用于收集整个租户所有库的表的统计信息。

ANALYZE 命令

除了使用 DBMS_STATS 系统包收集统计信息以外，在 OceanBase 数据库中还可以使用 ANALYZE 命令进行统计信息的收集。语法和参数详见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（MySQL 模式）/SQL 语句/ANALYZE。

在 MySQL 模式执行的时候，需要打开系统变量 `enable_sql_extension`。由于原生 MySQL 没有这种语法，因此需要在扩展模式执行。

```
ALTER SYSTEM SET enable_sql_extension = TRUE;
```

以下是使用这两个存储过程的几个简单示例：

- 收集表 `T1` 的统计信息，所有列的桶个数设定为 `128`

```
ANALYZE TABLE T1 COMPUTE STATISTICS FOR ALL COLUMNS SIZE 128;
```

- 收集表 `T_PART1` 的 GLOBAL 级别的统计信息，只收集数据分布不均匀的列的直方图

```
ANALYZE TABLE T_PART1 PARTITION('T_PART1') COMPUTE STATISTICS FOR ALL COLUMNS SIZ
E skewonly;
```

- 收集表 T_SUBPART1 的分区 p0sp0 和 p1ps2 的统计信息

```
ANALYZE TABLE T_SUBPART1 SUBPARTITION('p0sp0','p1ps2') COMPUTE STATISTICS FOR ALL
COLUMNS SIZE auto;
```

- 收集 T1 表 c1 和 c2 列的统计信息，列对应的桶个数为 30 个（这个是兼容的原生 MySQL 的 ANALYZE 语法）

```
ANALYZE TABLE T1 UPDATE HISTOGRAM ON c1, c2 with 30 buckets;
```

自动统计信息收集

除了手动收集统计信息以外，目前 OceanBase 数据库优化器通过 MAINTENANCE WINDOW（维护窗口）进行每日自动统计信息收集，从而保证统计信息能够自动地进行迭代更新。同原生 Oracle 类似，OceanBase 数据库优化器定义周一到周日 7 个自动统计信息的收集任务，周一到周五的任务默认开始时间为 22:00，最大收集时长 4 小时，周六周日的默认开始时间为 6:00，最大收集时长为 20 小时。

OceanBase 数据库提供了 OCEANBASE.DBA_SCHEDULER_WINDOWS 和 OCEANBASE.DBA_SCHEDULER_JOBS 两张系统视图，用于查询自动统计收集的执行情况。以下为简单的使用示例：

- 查询维护窗口执行信息

```
select WINDOW_NAME, LAST_START_DATE, NEXT_RUN_DATE
  from OCEANBASE.DBA_SCHEDULER_WINDOWS
  where LAST_START_DATE is not null order by LAST_START_DATE;
```

输出如下，OCEANBASE.DBA_SCHEDULER_WINDOWS 各字段的含义详见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/字典视图/oceanbase.DBA_SCHEDULER_WINDOWS。

```
+------------------+----------------------------+----------------------------+
| WINDOW_NAME      | LAST_START_DATE            | NEXT_RUN_DATE              |
+------------------+----------------------------+----------------------------+
| TUESDAY_WINDOW   | 2024-03-12 22:00:00.084516 | 2024-03-19 22:00:00.000000 |
| WEDNESDAY_WINDOW | 2024-03-13 22:00:00.090113 | 2024-03-20 22:00:00.000000 |
| THURSDAY_WINDOW  | 2024-03-14 22:00:00.105114 | 2024-03-21 22:00:00.000000 |
```

```
| FRIDAY_WINDOW    | 2024-03-15 22:00:00.080400 | 2024-03-22 22:00:00.000000 |
| SATURDAY_WINDOW  | 2024-03-16 06:00:00.104678 | 2024-03-23 06:00:00.000000 |
| SUNDAY_WINDOW    | 2024-03-17 06:00:00.089326 | 2024-03-24 06:00:00.000000 |
| MONDAY_WINDOW    | 2024-03-18 22:00:00.083798 | 2024-03-25 22:00:00.000000 |
+------------------+----------------------------+----------------------------+
7 rows in set
```

- 查询所有调度作业的信息

```
select JOB_NAME, REPEAT_INTERVAL, LAST_START_DATE, NEXT_RUN_DATE, MAX_RUN_DURATION
  from OCEANBASE.DBA_SCHEDULER_JOBS
  where LAST_START_DATE is not null order by LAST_START_DATE;
```

输出如下，`OCEANBASE.DBA_SCHEDULER_JOBS` 各字段的含义详见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/字典视图/oceanbase.DBA_SCHEDULER_JOBS。

```
+----------------------------+--------------------------+--------------------------
-+----------------------------+------------------+
| JOB_NAME                   | REPEAT_INTERVAL          | LAST_START_DAT
E             | NEXT_RUN_DATE              | MAX_RUN_DURATION |
+----------------------------+--------------------------+--------------------------
-+----------------------------+------------------+
| TUESDAY_WINDOW             | FREQ=WEEKLY; INTERVAL=1 | 2024-03-12 22:00:00.08451
6 | 2024-03-19 22:00:00.000000 |            14400 |
| WEDNESDAY_WINDOW           | FREQ=WEEKLY; INTERVAL=1 | 2024-03-13 22:00:00.09011
3 | 2024-03-20 22:00:00.000000 |            14400 |
| THURSDAY_WINDOW            | FREQ=WEEKLY; INTERVAL=1 | 2024-03-14 22:00:00.10511
4 | 2024-03-21 22:00:00.000000 |            14400 |
| FRIDAY_WINDOW              | FREQ=WEEKLY; INTERVAL=1 | 2024-03-15 22:00:00.08040
0 | 2024-03-22 22:00:00.000000 |            14400 |
| SATURDAY_WINDOW            | FREQ=WEEKLY; INTERVAL=1 | 2024-03-16 06:00:00.10467
8 | 2024-03-23 06:00:00.000000 |            72000 |
| SUNDAY_WINDOW              | FREQ=WEEKLY; INTERVAL=1 | 2024-03-17 06:00:00.08932
6 | 2024-03-24 06:00:00.000000 |            72000 |
| MONDAY_WINDOW              | FREQ=WEEKLY; INTERVAL=1 | 2024-03-18 22:00:00.08379
8 | 2024-03-25 22:00:00.000000 |            14400 |
| OPT_STATS_HISTORY_MANAGER | FREQ=DAYLY; INTERVAL=1  | 2024-03-19 11:10:46.01451
0 | 2024-03-20 11:10:46.000000 |             NULL |
+----------------------------+--------------------------+--------------------------
-+----------------------------+------------------+
8 rows in set
```

考虑到用户需要根据自身的业务特点去修改自动统计信息收集开始的时间及收集时长等属性，OceanBase 数据库优化器提供了如下方式去修改自动统计信息收集的属性：

- 禁止/启用自动统计信息任务收集：

  DBMS_SCHEDULER.DISABLE($window_name)

  DBMS_SCHEDULER.ENABLE($window_name);

- 设置自动统计信息任务下次开始的时间：

  DBMS_SCHEDULER.SET_ATTRIBUTE($window_name, 'NEXT_DATE', $next_time);

- 设置自动统计信息任务收集的时长：

  DBMS_SCHEDULER.SET_ATTRIBUTE($window_name, 'DURATION', $duation_time);

以下为简单的使用示例：

- 禁用周一自动收集统计信息

```
call dbms_scheduler.disable('MONDAY_WINDOW');
```

- 启用周一自动收集统计信息

```
call dbms_scheduler.enable('MONDAY_WINDOW');
```

- 设置周一自动收集统计信息开始的时间在晚上 20 点

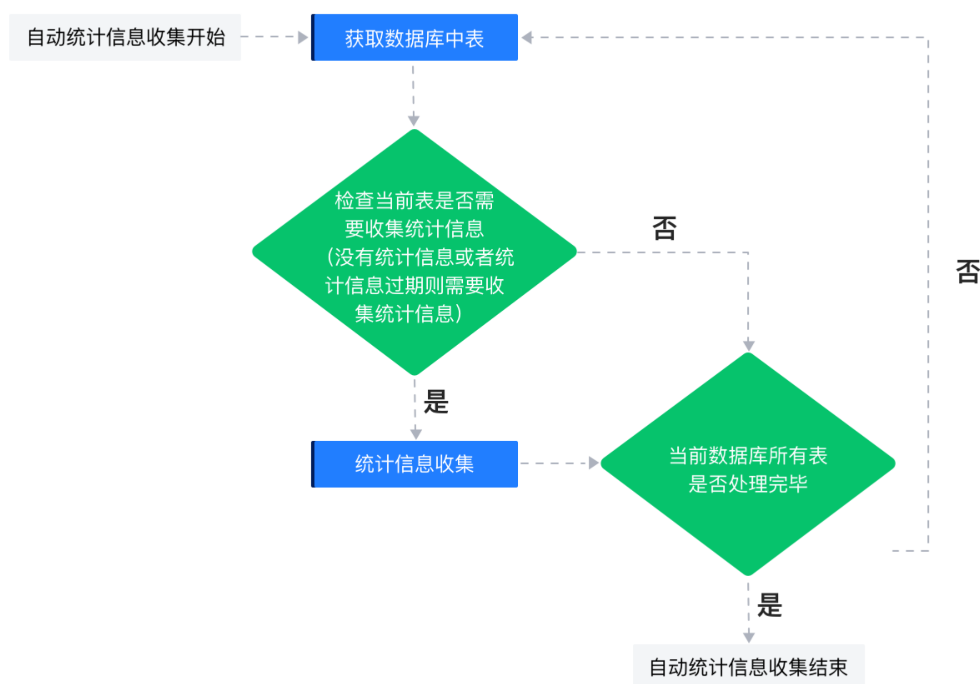
```
call dbms_scheduler.set_attribute('MONDAY_WINDOW', 'NEXT_DATE', '2022-09-12 20:00:
00');
```

- 设置周一自动收集统计信息的持续时长为 6 小时

```
call dbms_scheduler.set_attribute('MONDAY_WINDOW', 'DURATION', INTERVAL '6' HOUR);
```

自动统计信息收集的工作机制

上面介绍了自动统计信息收集相关的 MAINTENANCE WINDOW 的内容，下图展示了自动统计信息的工作机制：



判断一个表的统计信息是否过期，主要依据上一次统计信息收集时间到本次收集期间该表的做增/删/改的比例，默认值是 `10%`，即一张表的改动比例超过 10%，则认为表的统计信息过期，需要重新收集统计信息。需要注意的是，如果对应的表是分区表，则这个变化比例是分区级别的，比如一张分区表的某些分区的增/删/改的比例超过了 10%，那么也会重新收集这些分区的统计信息。

说明

默认的变化比例是可以配置的，业务可以根据实际情况通过设置 perfs 进行调整，具体方式参见官网《OceanBase 数据库》文档 管理数据库/性能调优/SQL 调优/执行计划优化/统计信息和估行机制/统计信息/管理统计信息/配置统计信息的收集策略 一节，这里不再详述。

除此之外，OceanBase 数据库优化器也提供了视图 `OCEANBASE.DBA_TAB_MODIFICATIONS` 用于查询一张表的增/删/改次数，`OCEANBASE.DBA_TAB_MODIFICATIONS` 中各字段含义请参见官网《OceanBase 数据库》文档 参考指南/系统视图/MySQL 租户系统视图/字典视图/oceanbase.DBA_TAB_MODIFICATIONS。以下为 `OCEANBASE.DBA_TAB_MODIFICATIONS` 的简单的使用示例：

```
select TABLE_OWNER as DB_NAME, TABLE_NAME, INSERTS, UPDATES, DELETES, TIMESTAMP
  from OCEANBASE.DBA_TAB_MODIFICATIONS
```

```
where TABLE_OWNER = 'test' and TABLE_NAME = 't2';
```

输出如下：

```
+---------+------------+---------+---------+---------+------------+
| DB_NAME | TABLE_NAME | INSERTS | UPDATES | DELETES | TIMESTAMP  |
+---------+------------+---------+---------+---------+------------+
| test    | t2         |    1000 |       0 |       0 | 2024-03-15 |
+---------+------------+---------+---------+---------+------------+
```

注意

系统视图 OCEANBASE.DBA_TAB_MODIFICATIONS 中的第一个列 TABLE_OWNER 的含义为 database_name 而非 user_name。OceanBase 官网上某些版本的内容可能有误。

查询统计信息

当收集完统计信息或者想确认某一个表有没有统计信息时，OceanBase 数据库优化器提供丰富的视图用于查询各类统计信息。

视图名称	描述
OCEANBASE.DBA_TAB_STATISTICS	用于查询表级的统计信息
OCEANBASE.DBA_IND_STATISTICS	用于查询索引的统计信息
OCEANBASE.DBA_TAB_COL_STATISTICS	用于查询 GLOBAL 列级的统计信息
OCEANBASE.DBA_PART_COL_STATISTICS	用于查询 PARTITON 列级的统计信息
OCEANBASE.DBA_SUBPART_COL_STATISTICS	用于查询 SUBPARTITON 列级的统计信息
OCEANBASE.DBA_TAB_HISTOGRAMS	用于查询 GLOBAL 列级直方图的信息
OCEANBASE.DBA_PART_HISTOGRAMS	用于查询 PARTITON 列级直方图的信息
OCEANBASE.DBA_SUBPART_HISTOGRAMS	用于查询 SUBPARTITON 列级直方图的信息

以下为简单的使用示例：

1. 创建一张分区表 `t_part`

```
create table t_part(c1 int) partition by hash(c1) partitions 4;
```

2. 向分区表 `t_part` 插入 1000 行数据，数据的值均为 `1`

```
insert into t_part with recursive cte(n) as (select 1 from dual union all select
n + 1 from cte where n < 1000) select 1 from cte;
```

3. 查询 `t_part` 的表级统计信息

```
select TABLE_NAME, PARTITION_NAME, PARTITION_POSITION, OBJECT_TYPE, NUM_ROWS, AVG_
ROW_LEN, LAST_ANALYZED
  from OCEANBASE.DBA_TAB_STATISTICS
  where OWNER = 'test' and TABLE_NAME = 't_part';
```

输出如下，因为还没有收集过，所以内容为空。

```
+------------+----------------+--------------------+-------------+----------+-----
--------+---------------+
| TABLE_NAME | PARTITION_NAME | PARTITION_POSITION | OBJECT_TYPE | NUM_ROWS | AVG_
ROW_LEN | LAST_ANALYZED |
+------------+----------------+--------------------+-------------+----------+-----
--------+---------------+
| t_part     | p3             |                  4 | PARTITION   |     NULL
|        NULL | NULL          |
| t_part     | p2             |                  3 | PARTITION   |     NULL
|        NULL | NULL          |
| t_part     | p1             |                  2 | PARTITION   |     NULL
|        NULL | NULL          |
| t_part     | p0             |                  1 | PARTITION   |     NULL
|        NULL | NULL          |
| t_part     | NULL           |               NULL | TABLE       |     NULL
|        NULL | NULL          |
+------------+----------------+--------------------+-------------+----------+-----
--------+---------------+
5 rows in set
```

4. 查看 SQL 执行计划

```
explain select * from t_part where c1 > 2;
```

5. 手动收集统计信息

```
analyze table t_part COMPUTE STATISTICS for all columns size 128;
```

6. 再次查询 `t_part` 的表级统计信息

```
select TABLE_NAME, PARTITION_NAME, PARTITION_POSITION, OBJECT_TYPE, NUM_ROWS, AVG_
ROW_LEN, LAST_ANALYZED
  from OCEANBASE.DBA_TAB_STATISTICS
  where OWNER = 'test' and TABLE_NAME = 't_part';
```

输出如下：

```
+------------+----------------+--------------------+-------------+----------+-----
--------+----------------------------+
| TABLE_NAME | PARTITION_NAME | PARTITION_POSITION | OBJECT_TYPE | NUM_ROWS | AVG_
ROW_LEN | LAST_ANALYZED              |
+------------+----------------+--------------------+-------------+----------+-----
--------+----------------------------+
| t_part     | NULL           |               NULL | TABLE       |     1000
|          20 | 2024-03-22 14:11:03.188965 |
| t_part     | p0             |                  1 | PARTITION   |        0
|           0 | 2024-03-22 14:11:03.188965 |
| t_part     | p1             |                  2 | PARTITION   |     1000
|          20 | 2024-03-22 14:11:03.188965 |
| t_part     | p2             |                  3 | PARTITION   |        0
|           0 | 2024-03-22 14:11:03.188965 |
| t_part     | p3             |                  4 | PARTITION   |        0
|           0 | 2024-03-22 14:11:03.188965 |
+------------+----------------+--------------------+-------------+----------+-----
--------+----------------------------+
5 rows in set
```

7. 查看 SQL 的执行计划

```
explain select * from t_part where c1 > 1;
```

输出如下，因为表中的 `c1` 列只有值为 `1` 的数据，所以对于 `c1 > 1` 计划的估行很小，符合预期。

```
+---------------------------------------------------------------------------------
---+
| Query Pla
n                                                                          |
+---------------------------------------------------------------------------------
---+
| ============================================================
=                      |
| |ID|OPERATOR                   |NAME     |EST.ROWS|EST.TIME(us)
|                      |
| ------------------------------------------------------------
-                      |
| |0 |PX COORDINATOR             |         |1       |64
```

```
|                       |
| |1 | └─EXCHANGE OUT DISTR     |:EX10000|1       |64
|                       |
| |2 |   └─PX PARTITION ITERATOR|        |1       |64
|                       |
| |3 |     └─TABLE FULL SCAN    |t_part  |1       |64
|                       |
| ============================================================
=                       |
| Outputs & filters
:                                                                  |
| ------------------------------------
-                                               |
|   0 - output([INTERNAL_FUNCTION(t_part.c1)]), filter(nil), rowset=1
6                |
|   1 - output([INTERNAL_FUNCTION(t_part.c1)]), filter(nil), rowset=1
6                |
|       dop=
1                                                                         |
|   2 - output([t_part.c1]), filter(nil), rowset=1
6                                  |
|       force partition granul
e                                                      |
|   3 - output([t_part.c1]), filter([t_part.c1 > 1]), rowset=1
6                       |
|       access([t_part.c1]), partitions(p[0-3]
)                                       |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t_part.__pk_increment]), range(MIN ; MAX)always tru
e               |
+-----------------------------------------------------------------------------------
---+
```

8. 再向分区表 `t_part` 插入 1000 行数据，数据的值均为 `99`

```
insert into t_part with recursive cte(n)
  as (select 1 from dual union all select n + 1 from cte where n < 1000) select 9
9 from cte;
```

9. 再次查看执行计划

```
explain select * from t_part where c1 > 1;
```

输出如下：

```
+-----------------------------------------------------------------------------------
```

```
---+
| Query Pla
n                                                                            |
+-----------------------------------------------------------------------------------
---+
| =============================================================
=                       |
| |ID|OPERATOR                |NAME    |EST.ROWS|EST.TIME(us)
|                       |
| -------------------------------------------------------------
-                       |
| |0 |PX COORDINATOR          |        |1       |64
|                       |
| |1 | └─EXCHANGE OUT DISTR    |:EX10000|1       |64
|                       |
| |2 |   └─PX PARTITION ITERATOR|        |1       |64
|                       |
| |3 |     └─TABLE FULL SCAN    |t_part  |1       |64
|                       |
| =============================================================
=                       |
| Outputs & filters
:                                                                    |
| ------------------------------------
-                                                 |
|   0 - output([INTERNAL_FUNCTION(t_part.c1)]), filter(nil), rowset=1
6                |
|   1 - output([INTERNAL_FUNCTION(t_part.c1)]), filter(nil), rowset=1
6                |
|       dop=
1                                                                           |
|   2 - output([t_part.c1]), filter(nil), rowset=1
6                                  |
|       force partition granul
e                                                         |
|   3 - output([t_part.c1]), filter([t_part.c1 > 1]), rowset=1
6                       |
|       access([t_part.c1]), partitions(p[0-3]
)                                        |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,  |
|       range_key([t_part.__pk_increment]), range(MIN ; MAX)always tru
e               |
+-----------------------------------------------------------------------------------
---+
```

3 号算子在 `filter([t_part.c1 > 1])` 过后的 output 行数应该有 1000 行，显然不止 `EST.ROWS` 显示的 1 行。这是因为表中的 `c1` 列出现 1000 行值为 99 的数据之后，因为优化

器还没有及时收集，所以对于 `c1 > 1` 计划的估行依然很小。这是不符合预期的输出，这时候就可以手动收集下统计信息。

10. 手动收集统计信息

```
analyze table t_part COMPUTE STATISTICS for all columns size 128;
```

11. 手动收集统计信息之后再次查看执行计划

```
explain select * from t_part where c1 > 1;
```

输出如下，可以看到符合预期的估行行数。

```
+-----------------------------------------------------------------------------------
---+
| Query Pla
n                                                                                  |
+-----------------------------------------------------------------------------------
---+
| ============================================================
=                       |
| |ID|OPERATOR                |NAME    |EST.ROWS|EST.TIME(us)
|                       |
| ------------------------------------------------------------
-                       |
| |0 |PX COORDINATOR          |        |1000    |707
|                       |
| |1 | └─EXCHANGE OUT DISTR     |:EX10000|1000    |515
|                       |
| |2 |   └─PX PARTITION ITERATOR|        |1000    |87
|                       |
| |3 |     └─TABLE FULL SCAN    |t_part  |1000    |87
|                       |
| ============================================================
=                       |
| Outputs & filters
:                                                                  |
| ------------------------------------
-                                               |
|   0 - output([INTERNAL_FUNCTION(t_part.c1)]), filter(nil), rowset=25
6               |
|   1 - output([INTERNAL_FUNCTION(t_part.c1)]), filter(nil), rowset=25
6               |
|       dop=
1                                                                                  |
|   2 - output([t_part.c1]), filter(nil), rowset=25
6                                  |
```

```
|       force partition granul
e                                                       |
|   3 - output([t_part.c1]), filter([t_part.c1 > 1]), rowset=25
6                     |
|       access([t_part.c1]), partitions(p[0-3]
)                                       |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,   |
|       range_key([t_part.__pk_increment]), range(MIN ; MAX)always tru
e              |
+-------------------------------------------------------------------------------
---+
```

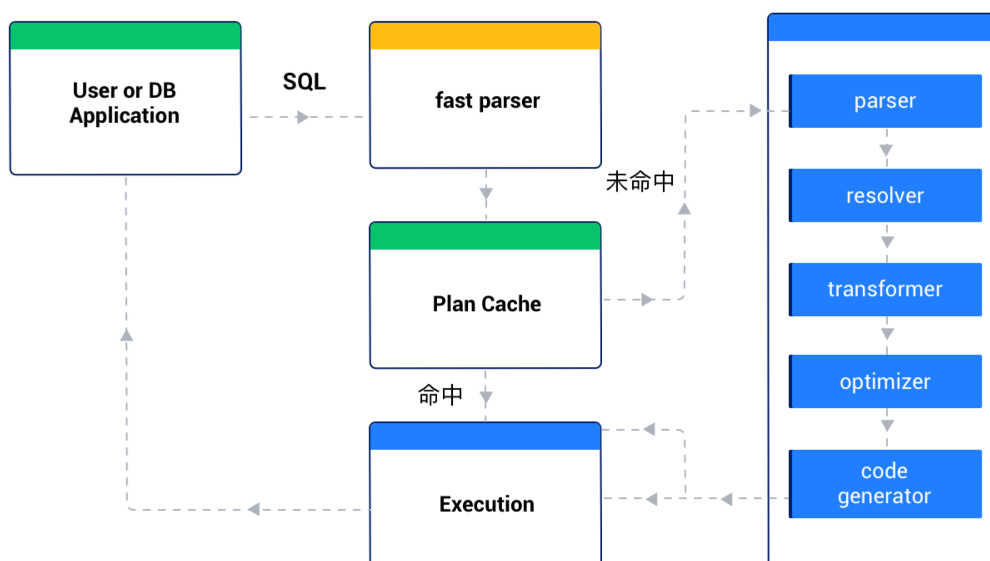
计划缓存

计划缓存概述

OceanBase 数据库中存在大量的改写规则和复杂的计划生成算法，这些都为 OceanBase 数据库带来了强大的优化能力。然而任何事情都具有两面性，更多改写的尝试、更复杂的计划生成算法必然会使得计划的生成时间变得更久。在极端的 TP 场景下，一个主键单点查询的 SQL 可能生成计划花了 1 ms，但是计划的执行只需要 0.5 ms。在这种场景下，如果每条 SQL 执行时都要重新生成计划，那么 SQL 的主要耗时就会花在生成计划上了。因此 OceanBase 数据库引入了计划缓存（Plan Cache）机制，让“相似的”SQL 可以共享执行计划。

当 OceanBase 数据库收到一条 SQL 请求后，首先会通过 fast parser 模块对 SQL 文本做一次快速参数化。快速参数化的作用是把 SQL 文本中的部分常量参数替换成通配符 ?，例如 `SELECT *FROM t1 WHERE c1 = 1` 会被替换为 `SELECT* FROM t1 WHERE c1 =?`。接着会从计划缓存中查看，对于这条参数化后的 SQL，有没有已经生成好的计划。如果找到了已经生成好的计划，就会直接执行这个计划。如果没有找到可用的计划，会重新为这条 SQL 生成执行计划，并把生成好的计划保存到计划缓存中以备后续的 SQL 使用。

通常情况下从计划缓存中直接获取执行计划相比于重新生成执行计划，耗时通常会低至少一个数量级。因此使用计划缓存可以大大降低获取执行计划的时间，从而减少 SQL 的响应时间。



计划缓存的 bad case

我们刚刚提到了对于一条参数化后的 SQL，生成执行计划后会将该计划保存到计划缓存中，后续的 SQL 执行都会命中这个执行计划。这种复用计划的方式虽然能够节省生成计划的时间，但在数据倾斜的场景下隐含了一个 bad case，这里我们举一个例子来说明：

1. 创建一张表 `t1`

```
create table t1 (c1 int, c2 int, key idx1(c1), key idx2(c2));
```

2. 插入 1000 行数据，`c1` 列只有两种值，其中值 `0` 占了全部数据的 0.1%，值 `1` 占了全部数据的 99.9%；`c2` 列有 `1` 到 `10` 共 10 种值，每种值各占了 10%

```
insert into t1 values(0, 0);

insert into t1 with recursive cte(n)
  as (select 1 from dual union all select n + 1 from cte where n < 999) select 1,
mod(n, 10) + 1 from cte;
```

3. 手动收集统计信息

```
analyze table t1 COMPUTE STATISTICS for all columns size 128;
```

4. 执行查询 SQL

```
select* from t1 where c1 = 0 and c2 = 0;

这条 SQL 参数化后的结果是: `select *from t1 where c1 = ? and c2 = ?`。
```

5. 查询 SQL 在 plan cache 里缓存的计划

```
SELECT SQL_ID, PLAN_ID, STATEMENT, OUTLINE_DATA, PLAN_ID, LAST_ACTIVE_TIME, QUERY_
SQL, HIT_COUNT
  FROM oceanbase.GV$OB_PLAN_CACHE_PLAN_STAT
  WHERE QUERY_SQL LIKE '%select * from t1 where c1 = %'\G
```

输出如下，因为 SQL 的过滤条件是 `c1 = 0`，过滤性极好，所以可以看到 `OUTLINE_DATA` 里可以看到这个计划使用了 `idx1` 这个索引。

```
*************************** 1. row ***************************
          SQL_ID: F296DCC7D661BF78D15FD5E4A753B53B
         PLAN_ID: 21485
       STATEMENT: select * from t1 where c1 = ? and c2 = ?
    OUTLINE_DATA: /*+BEGIN_OUTLINE_DATA INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx1"
) OPTIMIZER_FEATURES_ENABLE('') END_OUTLINE_DATA*/
         PLAN_ID: 21485
LAST_ACTIVE_TIME: 2024-06-03 11:23:30.914958
       QUERY_SQL: select * from t1 where c1 = 0 and c2 = 0
       HIT_COUNT: 0
```

6. 执行如下 SQL，将过滤条件设为 `c1=1`

```
select * from t1 where c1 = 1 and c2 = 1;
```

过滤条件是 `c1 = 1`，过滤性极差，这条 SQL 参数化后的结果也是 `select * from t1 where c1 =?and c2 =?`，所以会复用刚刚已经生成好的计划。

7. 查询上条 SQL 在 plan cache 里缓存的计划

```
SELECT SQL_ID, PLAN_ID, STATEMENT, OUTLINE_DATA, PLAN_ID, LAST_ACTIVE_TIME, QUERY_
SQL, HIT_COUNT
  FROM oceanbase.GV$OB_PLAN_CACHE_PLAN_STAT
  WHERE QUERY_SQL LIKE '%select * from t1 where c1 = %'\G
```

输出如下，`LAST_ACTIVE_TIME` 和 `HIT_COUNT` 的值均发生了变化，说明这个 plan 被重复利用了，即过滤条件是 `c1 = 1 and c2 = 1` 的 SQL，在 plan cahce 中复用了过滤条件为 `c1 = 0`

`and c2 = 0` 那条 SQL 的计划。

```
*************************** 1. row ***************************
          SQL_ID: F296DCC7D661BF78D15FD5E4A753B53B
         PLAN_ID: 21485
       STATEMENT: select * from t1 where c1 = ? and c2 = ?
    OUTLINE_DATA: /*+BEGIN_OUTLINE_DATA INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx1"
) OPTIMIZER_FEATURES_ENABLE('') END_OUTLINE_DATA*/
         PLAN_ID: 21485
LAST_ACTIVE_TIME: 2024-06-03 11:24:48.867697
       QUERY_SQL: select * from t1 where c1 = 0 and c2 = 0
       HIT_COUNT: 1
```

上述示例中，当过滤条件是 `c1 = 1 and c2 = 1` 时，应该使用 `idx2` 这个索引，因为此时 `idx2` 的过滤性远远比 `idx1` 更好，但是由于 plan cahce，导致上面真实执行时的计划选错了一个不优的索引。对于这个问题，可以选择通过上一节 阅读和管理 OceanBase 数据库 SQL 执行计划 中介绍的 Hint 和 Outline 对计划进行控制，这里不再赘述。

您也可通过让参数化后的结果发生变化，让其无法复用 plan cache 里的计划；或者直接清空特定的计划缓存。

- 改变参数化后的结果

> **注意**
>
> 这里给出的方法不是标准做法，仅供参考。

示例如下，在 SQL 中间多加一个空格，让 `c2 = 1` 变成 `c2 =  1`，这样参数化后的东西就变了（详见示例中的 `STATEMENT` 字段）。

> **注意**
>
> 空格需要加载 SQL 的中间，不能加在开头（第一个关键字之前）和结尾（分号之前且紧挨分号）。

修改后的 SQL 为 `select * from t1 where c1 = 1 and c2 =  1;`，查看该 SQL 在 plan cache 里缓存的计划。

```
SELECT SQL_ID, PLAN_ID, STATEMENT, OUTLINE_DATA, PLAN_ID, LAST_ACTIVE_TIME, QUERY_
SQL
  FROM oceanbase.GV$OB_PLAN_CACHE_PLAN_STAT
  WHERE QUERY_SQL LIKE '%select * from t1 where c1 = %'\G
```

```

```

输出如下，生成了新的计划，且从 `OUTLINE_DATA` 中可以看到，计划中的索引变成了更优的 `idx2`。

```
*************************** 1. row ***************************
          SQL_ID: F296DCC7D661BF78D15FD5E4A753B53B
         PLAN_ID: 52941
       STATEMENT: select * from t1 where c1 = ? and c2 = ?
    OUTLINE_DATA: /*+BEGIN_OUTLINE_DATA INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx1"
) OPTIMIZER_FEATURES_ENABLE('4.0.0.0') END_OUTLINE_DATA*/
         PLAN_ID: 52941
LAST_ACTIVE_TIME: 2024-03-19 17:13:25.877066
       QUERY_SQL: select * from t1 where c1 = 0 and c2 = 0
*************************** 2. row ***************************
          SQL_ID: 3DC824F8228724AE4B1435111273F59C
         PLAN_ID: 52949
       STATEMENT: select * from t1 where c1 = ? and c2 =  ?
    OUTLINE_DATA: /*+BEGIN_OUTLINE_DATA INDEX(@"SEL$1" "test"."t1"@"SEL$1" "idx2"
) OPTIMIZER_FEATURES_ENABLE('4.0.0.0') END_OUTLINE_DATA*/
         PLAN_ID: 52949
LAST_ACTIVE_TIME: 2024-03-19 17:14:44.234663
       QUERY_SQL: select * from t1 where c1 = 1 and c2 =  1
```

- 直接清空特定的计划缓存

  详细介绍可参见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/系统租户/ALTER SYSTEM/FLUSH PLAN CACHE。示例如下：

```
ALTER SYSTEM FLUSH PLAN CACHE
  sql_id='F296DCC7D661BF78D15FD5E4A753B53B' databases='test' GLOBAL;
```

索引调优

SQL 性能调优相关的内容中，需要用户掌握的主要分为三个部分：索引调优、连接调优、排序和 limit 调优。

当我们发现某一条 SQL 存在性能问题时，我们可以通过很多方式对这条 SQL 进行优化，其中最常见的是索引调优。索引调优通过为数据表创建合适的索引来达到减少数据扫描量，消除排序等目的。索引调优是一种比较简单的调优方式，也是 SQL 出现性能问题时通常在第一时间考虑的优化方式，创建一个合适的索引往往可以极大地提高 SQL 的执行性能。

在建索引前，我们需要考虑是否有必要建索引、应该在哪些列上建索引、索引列的顺序应该怎样安排。接下来这部分内容就会介绍下 OceanBase 数据库中索引的一些基础知识，以及创建合适索引的方法。

OceanBase 数据库索引的基础知识

OceanBase 数据库的索引中除了有索引键，还会包含主表的主键。因为在使用索引的时候，需要通过主表的主键去关联索引中的某一行与主表中的某一行。也就意味着索引表中需要包含主表的主键才能去反向查找定位主表中的具体某一行（OceanBase 数据库中常把这个操作叫做索引回表），因此需要把主表的主键加到索引表里面。

我们可以简单做一个实验看下，创建一个索引叫 `idx_b`。

```
create table test(a int primary key, b int, c int, key idx_b(b));
```

再去 `oceanbase.__all_column` 中查询一下这个索引有哪些列，可以看到虽然这个索引创建在 `b` 列上，但是这个索引中还包含了主表的主键列 `a` 列。

```
select
  column_id,
  column_name,
  rowkey_position,
  index_position
from
  oceanbase.__all_column
where
  table_id = (
    select
      table_id
    from
      oceanbase.__all_table
    where
      data_table_id = (
        select
          table_id
        from
          oceanbase.__all_table
        where
          table_name = 'test'
      )
  );
```

输出如下：

```
+-----------+-------------+----------------+----------------+
| column_id | column_name | rowkey_position | index_position |
+-----------+-------------+----------------+----------------+
|        16 | a           |              2 |              0 |
|        17 | b           |              1 |              1 |
+-----------+-------------+----------------+----------------+
2 rows in set
```

说明

OceanBase 数据库目前所有的表都有主键，表在磁盘上的数据都按照主键有序。如果是无主键表，数据库会自动为其创建一个隐藏的自增列，并在这个自增列上创建隐藏一个主键。所以在无主键表上创建索引的时候，索引实际会包含这个隐藏的主键列。大家可以自行通过 `oceanbase.__all_column` 和 `oceanbase.__all_table` 这两张内部表验证一下上面这个说法。

索引的几个作用

查询时走索引相对于走主表有三个优势：

- 减少读行：可以根据索引列的条件去快速定位数据，来减少数据的扫描量。
- 消除排序：索引列是有序的，可以利用此特性消掉一些排序操作。
- 节省 I/O：索引上的列一般比主表上的列少，如果过滤条件的过滤性好或者查询的列数较少，可以少扫描一些不需要查询的列数据，节省系统的 I/O 资源。

快速定位数据

索引的第一个优势是快速定位数据。可以将索引列上的过滤条件转化成索引扫描的开始位置和结束位置。在实际扫描的时候，只需要从开始位置一直扫描到结束位置，两个位置之间的数据就是满足索引列上的过滤条件的数据。扫描的开始位置到结束位置称为 query range。这里需要记住的一个重要规则就是：索引可以从开头匹配多个等值谓词，直到匹配到第一个范围谓词为止。

举个例子，在 `test` 表上有个建在 `b`、`c` 两列上的索引 `idx_b_c`，按照前文提到的内容，其实它是建在 `b`、`c`、`a` 三个列的索引，因为 `a` 是主键。且索引上的数据就按照 `b`、`c` 有序。

1. 创建 `test` 表

```
create table test(a int primary key, b int, c int, d int, key idx_b_c(b, c));
```

2. 插入一些数据，验证一下索引 `idx_b_c` 上的数据是否按照 `b`、`c`、`a` 有序

```
insert into test values(1, 2, 3, 4);

insert into test values(2, 3, 4, 5);

insert into test values(5, 1, 2, 3);

insert into test values(4, 1, 3, 3);

insert into test values(3, 1, 3, 3);
```

3. 查询全表

```
select * from test;
```

输出如下，全表扫默认会走主表，看到的行序是按主表主键列 `a` 有序的。

```
+---+------+------+------+
| a | b    | c    | d    |
+---+------+------+------+
| 1 |    2 |    3 |    4 |
| 2 |    3 |    4 |    5 |
| 3 |    1 |    2 |    3 |
| 4 |    1 |    3 |    3 |
| 5 |    1 |    2 |    3 |
+---+------+------+------+
5 rows in set
```

4. 通过 hint 强制走索引 `idx_b_c`

```
select /*+ index(test idx_b_c) */ * from test;
```

输出如下，行序就变成按索引列 `b`、`c` 有序了，这个就是索引上真实的数据顺序，了解这个会对理解后续的内容有所帮助。

```
+---+------+------+------+
| a | b    | c    | d    |
+---+------+------+------+
| 3 |    1 |    2 |    3 |
| 5 |    1 |    2 |    3 |
| 4 |    1 |    3 |    3 |
```

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

接下来执行几条 SQL，看看这些 SQL 能否充分利用这个索引。

- 示例一：下面这条 SQL 有一个 `b = 1` 的过滤条件，对应的 query range 是 `(1,MIN,MIN ; 1,MAX,MAX)`。即要从 `b = 1`, `c = min`, `a = min` 向量点开始，一直扫到 `b = 1`, `c = max`, `a = max` 向量点。因为这两个向量点之间所有数据都满足 `b = 1` 条件，所以不需要再进行额外的过滤操作。

```
explain select /*+ index(test idx_b_c) */ * from test where b = 1;
```

输出如下：

```
+--------------------------------------------------------------------------------+
| Query Plan                                                                     |
+--------------------------------------------------------------------------------+
| ==========================================================                     |
| |ID|OPERATOR        |NAME          |EST.ROWS|EST.TIME(us)|                     |
| ----------------------------------------------------------                     |
| |0 |TABLE RANGE SCAN|test(idx_b_c)|1       |5           |                     |
| ==========================================================                     |
| Outputs & filters:                                                             |
| -------------------------------------                                          |
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter(nil), rowset=256 |
|       access([test.a], [test.b], [test.c], [test.d]), partitions(p0)          |
|       is_index_back=true, is_global_index=false,                               |
|       range_key([test.b], [test.c], [test.a]), range(1,MIN,MIN ; 1,MAX,MAX),  |
|       range_cond([test.b = 1])                                                 |
+--------------------------------------------------------------------------------+
```

- 示例二：下面这条 SQL 有一个 `b > 1` 的过滤条件，和上面那条 SQL 类似，不赘述。

```
explain select /*+index(test idx_b_c)*/ * from test where b > 1;
```

输出如下：

```
+----------------------------------------------------------------------------------
+
| Query Plan
|
```

```
+--------------------------------------------------------------------------------
+
| ==========================================================
|
| |ID|OPERATOR        |NAME         |EST.ROWS|EST.TIME(us)|
|
| ----------------------------------------------------------
|
| |0 |TABLE RANGE SCAN|test(idx_b_c)|1       |5           |
|
| ==========================================================
|
| Outputs & filters:
|
| -------------------------------------
|
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter(nil), rowset=256
|
|       access([test.a], [test.b], [test.c], [test.d]), partitions(p0)
|
|       is_index_back=true, is_global_index=false,
|
|       range_key([test.b], [test.c], [test.a]), range(1,MAX,MAX ; MAX,MAX,MAX),
|
|       range_cond([test.b > 1])
|
+--------------------------------------------------------------------------------
+
```

- 示例三：下面这条 SQL 的过滤条件是 `b = 1, c > 1`，对应的 query range 是 `(1,1,MAX ; 1,MAX,MAX)`，range_cond 是 `([test.b = 1], [test.c > 1])`，因为第一个谓词是等值条件 `b = 1`，所以索引还会继续向后匹配，直到出现第一个范围条件 `c > 1`。

```
explain select/*+index(test idx_b_c)*/ * from test where b = 1 and c > 1;
```

输出如下：

```
+--------------------------------------------------------------------------------+
| Query Plan                                                                     |
+--------------------------------------------------------------------------------+
| ==========================================================                     |
| |ID|OPERATOR        |NAME         |EST.ROWS|EST.TIME(us)|                     |
| ----------------------------------------------------------                     |
| |0 |TABLE RANGE SCAN|test(idx_b_c)|1       |5           |                     |
| ==========================================================                     |
| Outputs & filters:                                                             |
| -------------------------------------                                          |
```

```
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter(nil), rowset=256 |
|       access([test.a], [test.b], [test.c], [test.d]), partitions(p0)          |
|       is_index_back=true, is_global_index=false,                              |
|       range_key([test.b], [test.c], [test.a]), range(1,1,MAX ; 1,MAX,MAX),    |
|       range_cond([test.b = 1], [test.c > 1])                                  |
+-------------------------------------------------------------------------------+
12 rows in set
```

- 示例四：下面这条 SQL 的过滤条件是 `b > 1, c > 1`。query range 在索引上抽 range 的时候，只能抽到第一个范围谓词为止。比如说这里 `b > 1 , c > 1`，发现索引的第一列就被用来当范围谓词了，那么往后再出现任何的等值条件或范围条件，都不能再抽取 range。因此，此 SQL 对应的 query range 是 `(1,MAX,MAX ; MAX,MAX,MAX)`，因为这里是用两个向量点去描述起始和结束位置，然而两个向量点是无法精确地描述出多个范围条件的。看下面计划中 range_cond 是 `([test.b > 1])`，表明这条 SQL 在索引上也只完成了 `b > 1` 这个条件的过滤，索引回表（`is_index_back=true`）之后，还需要再对 `c > 1` 进行一次过滤（`filter([test.c > 1])`）。

```
explain select /*+ index(test idx_b_c) */ * from test where b > 1 and c > 1;
```

输出如下：

```
+-----------------------------------------------------------------------------------
-------+
| Query Pla
n                                                                               |
+-----------------------------------------------------------------------------------
-------+
| ========================================================
=                               |
| |ID|OPERATOR        |NAME        |EST.ROWS|EST.TIME(us)
|                               |
| --------------------------------------------------------
-                               |
| |0 |TABLE RANGE SCAN|test(idx_b_c)|1       |3
|                               |
| ========================================================
=                               |
| Outputs & filters
:                                                                       |
| ------------------------------------
-                                                   |
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter([test.c > 1]), rows
et=256 |
```

```
|       access([test.a], [test.b], [test.c], [test.d]), partitions(p0
)                    |
|       is_index_back=true, is_global_index=false, filter_before_indexback[true]
,        |
|       range_key([test.b], [test.c], [test.a]), range(1,MAX,MAX ; MAX,MAX,MAX)
,         |
|       range_cond([test.b > 1]
)                                                          |
+-----------------------------------------------------------------------------------
-------+
```

消除排序的开销

索引本身是有序的，可以利用此特性来消除排序的开销。

下面举几个简单的例子：

先创建一张表。

```
create table test(a int primary key, b int, c int, d int, key idx_b_c_d(b, c, d));
```

下面这条 SQL 是 `b = 1 order by c`。这条 SQL 用到了索引 `idx_b_c_d`，在计划中可以看到只有 table scan 算子而没有 sort 算子，说明索引回表后不需要再对 `c` 列进行排序。因为索引是按照 `b`、`c`、`d`、`a` 有序，但在扫描结果中，b 是一个常量 1，那么返回数据本身就是按照 `c`、`d`、`a` 有序的，`order by c` 自然也就不需要通过 sort 算子进行排序了。注意这里 `is_index_back=false` 说明索引扫描完成之后不需要回表，直接就可以输出结果，因为索引里已经包含了所查询的所有列。

```
explain select /*+ index(test idx_b_c_d) */ * from test where b = 1 order by c;
```

输出如下：

```
+-----------------------------------------------------------------------------------
----------------+
| Query Pla
n
     |
+-----------------------------------------------------------------------------------
----------------+
| ==========================================================
=                                      |
| |ID|OPERATOR         |NAME            |EST.ROWS|EST.TIME(us)
```

```
|                                    |
| ---------------------------------------------------------
-                                    |
| |0 |TABLE RANGE SCAN|test(idx_b_c_d)|1        |2
|                                    |
| =========================================================
=                                    |
| Outputs & filters
:                                                                        |
| -----------------------------------
-                                                      |
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter(nil), rowset=25
6                   |
|       access([test.a], [test.b], [test.c], [test.d]), partitions(p0
)                            |
|       is_index_back=false, is_global_index=false
,                                          |
|       range_key([test.b], [test.c], [test.d], [test.a]), range(1,MIN,MIN,MIN ; 1
,MAX,MAX,MAX),   |
|       range_cond([test.b = 1])
)                                                                 |
+--------------------------------------------------------------------------------
----------------+
```

下面这条 SQL 就需要排序了。这里多了一个 `or`，看计划会通过索引去扫描出两批数据（因为 range 有两个 `(1,MIN,MIN,MIN ; 1,MAX,MAX,MAX)` 和 `(2,MIN,MIN,MIN ; 2,MAX,MAX,MAX)`），虽然两批数据内部都是有序的，但是两批数据之间却是无序的，所以在 table scan 上层还会再分配一个 sort 算子用于对 `c` 列进行排序。

```
explain select /*+ index(test idx_b_c_d) */ * from test where b = 1 or b = 2 orde
r by c;
```

输出如下：

```
+--------------------------------------------------------------------------------
-------------------------------------------------+
| Query Pla
n
                                     |
+--------------------------------------------------------------------------------
-------------------------------------------------+
| ===========================================================
=                                                                  |
| |ID|OPERATOR          |NAME          |EST.ROWS|EST.TIME(us)
|                                                                  |
| -----------------------------------------------------------
```

```
-                                                                         |
| |0 |SORT              |               |1       |2
|                                                                         |
| |1 |└─TABLE RANGE SCAN|test(idx_b_c_d)|1       |2
|                                                                         |
| ============================================================
=                                                                         |
| Outputs & filters
:
                              |
| ------------------------------------
-
           |
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter(nil), rowset=25
6                                                       |
|       sort_keys([test.c, ASC]
)
                  |
|   1 - output([test.a], [test.b], [test.c], [test.d]), filter(nil), rowset=25
6                                                       |
|       access([test.a], [test.b], [test.c], [test.d]), partitions(p0
)                                                                |
|       is_index_back=false, is_global_index=false
,                                                                                  |
|       range_key([test.b], [test.c], [test.d], [test.a]), range(1,MIN,MIN,MIN ; 1
,MAX,MAX,MAX), (2,MIN,MIN,MIN ; 2,MAX,MAX,MAX),  |
|       range_cond([test.b = 1 OR test.b = 2]
)
    |
+----------------------------------------------------------------------------------
--------------------------------------------------+
```

这条 SQL 与第一条类似，同理，也不需要排序。

```
explain select /*+ index(test idx_b_c_d) */ * from test where b = 1 and c = 2 orde
r by c;
```

输出如下：

```
+----------------------------------------------------------------------------------
------------+
| Query Pla
n
 |
+----------------------------------------------------------------------------------
------------+
| ==========================================================
=                                  |
| |ID|OPERATOR        |NAME          |EST.ROWS|EST.TIME(us)
```

```
|                                  |
| ------------------------------------------------------------
-                                  |
| |0 |TABLE RANGE SCAN|test(idx_b_c_d)|1       |2
|                                  |
| ============================================================
=                                  |
| Outputs & filters
:                                                                          |
| ------------------------------------
-                                                          |
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter(nil), rowset=25
6                |
|       access([test.a], [test.b], [test.c], [test.d]), partitions(p0
)                        |
|       is_index_back=false, is_global_index=false
,                                                    |
|       range_key([test.b], [test.c], [test.d], [test.a]), range(1,2,MIN,MIN ; 1,2
,MAX,MAX),   |
|       range_cond([test.b = 1], [test.c = 2]
)                                                        |
+-----------------------------------------------------------------------------------
------------+
```

下面这条 SQL 里的 c 是一个常量，索引是按照 b、c、d、a 有序的。因此，如果要求按照 b、d 去排序，直接在索引表上利用 c = 1 这个过滤条件（filter([test.c = 1])）查询就好了。

例如索引中 b 列有两个不同的值 1 和 2，那么利用 c = 1 这个过滤条件过滤之后，会返回索引上两批离散的数据，一批数据是 b = 1, c = 1, d、a，另一批数据是 b = 2, c = 1, d、a，这两批数据虽然在索引上可能是离散的，但是各批数据内，以及各批数据间，都是有序的，所以就不需要通过再分配 sort 算子去进行排序了。

```
explain select /*+ index(test idx_b_c_d) */ * from test where c = 1 order by b, d;
```

输出如下:

```
+-----------------------------------------------------------------------------------
------------------------------+
| Query Pla
n
                  |
+-----------------------------------------------------------------------------------
------------------------------+
| ===========================================================
=                                                     |
| |ID|OPERATOR       |NAME           |EST.ROWS|EST.TIME(us)
```

```
|                                                    |
| ---------------------------------------------------------
-                                                    |
| |0 |TABLE FULL SCAN|test(idx_b_c_d)|1       |2
|                                                    |
| =========================================================
=                                                    |
| Outputs & filters
:
          |
| ------------------------------------
-                                                                      |
|   0 - output([test.a], [test.b], [test.c], [test.d]), filter([test.c = 1]), rows
et=256                        |
|       access([test.a], [test.c], [test.b], [test.d]), partitions(p0
)                                         |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,                             |
|       range_key([test.b], [test.c], [test.d], [test.a]), range(MIN,MIN,MIN,MIN
; MAX,MAX,MAX,MAX)always true |
+----------------------------------------------------------------------------------
-----------------------------+
```

查询指定列时，相比主表可以扫描更少的数据

这点比较好理解，例如一张大宽表，有 100 个列，创建的是一张行存表。如果经常要查询其中一个列，最好在这个列上创建索引，索引一般只会包含少数的几个列，可以有效避免每次都进行全表扫描（对于 OceanBase 数据库的列存表，这种大宽表查询少数列的场景就可以不用创建索引了）。

例如下面这条 SQL，计划中的 NAME 中显示 `test(idx_b)`，说明查询用到了索引 `idx_b`，避免了扫描多余列 `a`、`c`、`d` 的数据。

```
create table test(a int, b int, c int, d int, key idx_b(b));

explain select b from test;
+----------------------------------------------------------------------------------
------+
| Query Pla
n                                                                              |
+----------------------------------------------------------------------------------
------+
| =====================================================
=                                      |
| |ID|OPERATOR       |NAME       |EST.ROWS|EST.TIME(us)
```

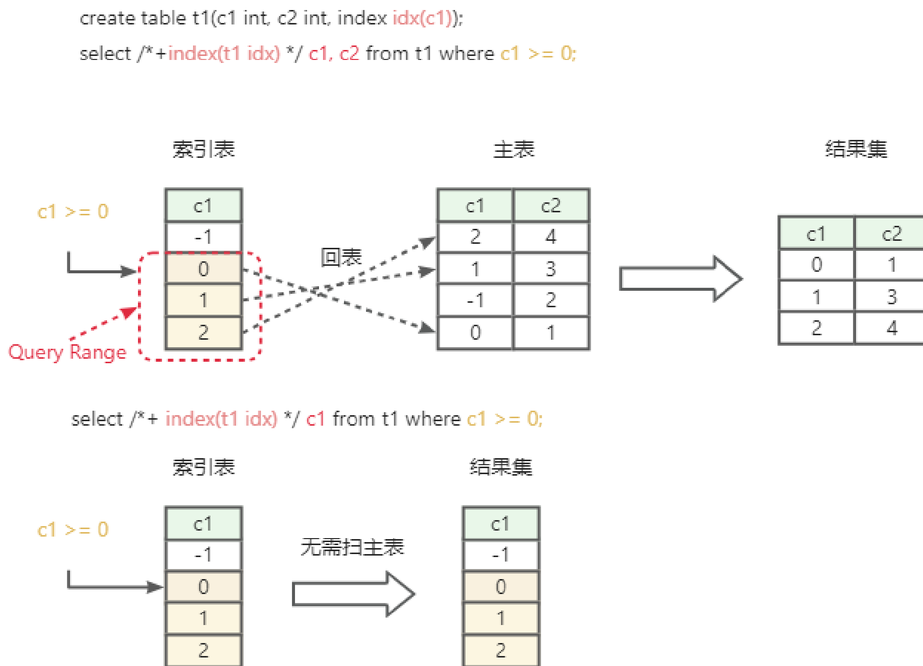
```
|                                |
| -----------------------------------------------------
-                                |
| |0 |TABLE FULL SCAN|test(idx_b)|1        |2
|                                |
| =====================================================
=                                |
| Outputs & filters
:                                                                    |
| ------------------------------------
-                                                 |
|   0 - output([test.b]), filter(nil), rowset=25
6                                        |
|       access([test.b]), partitions(p0
)                                                |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([test.b], [test.__pk_increment]), range(MIN,MIN ; MAX,MAX)alway
s true |
+-------------------------------------------------------------------------------------
------+
11 rows in set
```

又例如下面这条 SQL，优化器会选择列数最少的索引 `idx_b`（或者说数据量最小的索引）进行扫描。

```
create table test(a int, b int, c int, d int, key idx_b(b));

explain select count(*) from test;
+-------------------------------------------------------------------------------------
--------+
| Query Pla
n                                                                               |
+-------------------------------------------------------------------------------------
--------+
| =======================================================
=                                |
| |ID|OPERATOR          |NAME        |EST.ROWS|EST.TIME(us)
|                                |
| -------------------------------------------------------
-                                |
| |0 |SCALAR GROUP BY  |            |1        |2
|                                |
| |1 | └─TABLE FULL SCAN|test(idx_b)|1        |2
|                                |
| =======================================================
=                                |
| Outputs & filters
:                                                                      |
```

```
| ------------------------------------
-                                                         |
|   0 - output([T_FUN_COUNT_SUM(T_FUN_COUNT(*))]), filter(nil), rowset=25
6                |
|       group(nil), agg_func([T_FUN_COUNT_SUM(T_FUN_COUNT(*))]
)                           |
|   1 - output([T_FUN_COUNT(*)]), filter(nil), rowset=25
6                                 |
|       access(nil), partitions(p0
)                                                              |
|       is_index_back=false, is_global_index=false
,                                            |
|       range_key([test.b], [test.__pk_increment]), range(MIN,MIN ; MAX,MAX)alway
s true,   |
|       pushdown_aggregation([T_FUN_COUNT(*)]
)                                                 |
+-----------------------------------------------------------------------------------
--------+
```

这里也有一个劣势，就是如果查询的列比较多时，如果走了索引，就需要拿着从索引上得到的主表的主键列回到主表查询其余列（索引回表）。索引回表的代价是很高的，一般索引回表的性能只有直接全表扫的十分之一，如果过滤条件的过滤很差但是依然走了索引，索引回表的代价就无法被忽略了。



例如下面几条 SQL，索引建在 b 上，查询 a、b 的所有行，如果不走索引直接全表扫，优化器估

计的代价 `EST.TIME` 是 2 us。

```
create table test(a int, b int, c int, d int, key idx_b(b));

explain select a, b from test;
+-----------------------------------------------------------------------+
| Query Plan                                                            |
+-----------------------------------------------------------------------+
| ===============================================                       |
| |ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)|                       |
| -----------------------------------------------                       |
| |0 |TABLE FULL SCAN|test|1       |2           |                       |
| ===============================================                       |
| Outputs & filters:                                                    |
| -------------------------------------                                 |
|   0 - output([test.a], [test.b]), filter(nil), rowset=256             |
|       access([test.a], [test.b]), partitions(p0)                      |
|       is_index_back=false, is_global_index=false,                     |
|       range_key([test.__pk_increment]), range(MIN ; MAX)always true |
+-----------------------------------------------------------------------+
```

如果通过指定 hint 强制走索引，优化器估计的代价 `EST.TIME` 是 5 us，反倒比不走索引更慢了，这就是索引回表（`is_index_back=true`）带来的额外开销。

```
explain select /*+ index(test idx_b) */ a, b from test;
```

输出如下：

```
+-----------------------------------------------------------------------------------
------+
| Query Pla
n                                                                            |
+-----------------------------------------------------------------------------------
------+
| =====================================================
=                                |
| |ID|OPERATOR       |NAME         |EST.ROWS|EST.TIME(us)
|                                |
| -----------------------------------------------------
-                                |
| |0 |TABLE FULL SCAN|test(idx_b)|1       |5
|                                |
| =====================================================
=                                |
| Outputs & filters
:                                                                    |
| -----------------------------------
```

```
-                                                  |
|   0 - output([test.a], [test.b]), filter(nil), rowset=25
6                            |
|       access([test.__pk_increment], [test.a], [test.b]), partitions(p0
)              |
|       is_index_back=true, is_global_index=false
,                                       |
|       range_key([test.b], [test.__pk_increment]), range(MIN,MIN ; MAX,MAX)alway
s true |
+-------------------------------------------------------------------------------------
------+
```

如何衡量走索引的耗时时间

走索引的耗时时间由两部分构成：

1. 扫描索引的时间（由扫描的数据行数决定）。
2. 索引回表的时间（由需要回表的数据行数决定）。

假设这张表有 10000 行数据，扫描索引的时间是 1 ms 1000 行，索引回表的时间是 1 ms 100 行（大概是十倍的关系）。

```
create table test(a int primary key, b int, c int, d int, e int, key idx_b_e_c(b,
e, c));
```

用 `b = 1 and c = 1` 这个过滤条件进行过滤，会返回 1000 行数据。

如果我们执行 `select a, b, c from test where b = 1 and c = 1;` 这条 SQL，走索引的话，开销就是：在索引上扫描 1000 行的数据，大概是 1 ms。

```
explain select /*+index(test idx_b_c_d)*/ a, b, c from test where b = 1 and c = 1;
```

不走索引的话，开销就是：在主表上扫描 10000 行的数据，大概是 10 ms。

```
explain select /*+index(test primary)*/ a, b, c from test where b = 1 and c = 1;
```

所以在这种场景下，还是走索引更快，如果生成的计划没有走索引，就可以自己指定个 `hint /*+index(test idx_b_c_d)*/` 强制让它走索引。

如果我们执行 `select * from test where b = 1 and c = 1;` 这条 SQL，走 `idx_b_c_d` 这个索

引的开销就是：在索引上扫描 1000 行的数据 + 索引回表 1000 行的数据，大概是 1 ms + 10 ms = 11 ms。

```
explain select /*+index(test idx_b_c_d)*/ * from test where b = 1 and c = 1;
```

这条 SQL 不走索引的开销就是：在主表上扫描 10000 行的数据，耗时大概是 10 ms。

```
explain select /*+index(test primary)*/ * from test where b = 1 and c = 1;
```

所以在这种场景下，不走索引而走主表反倒更快，如果生成的计划走了索引，就可以自己指定个 `hint /*+ index(test primary)*/` 强制让它走主表。

如何获取类似于“用 `b = 1 and c = 1` 这个过滤条件进行过滤，会返回 100 行数据”这种信息？执行一条 SQL 看下 count 就好了。

```
select count(*) from test where b = 1 and c = 1;
```

输出如下：

```
+----------+
| count(*) |
+----------+
|      1000|
+----------+
```

索引调优总结

创建索引的策略，大体可以用下面三句话总结：

- 将常用查询中等值条件的列放在索引的前面，将存在范围条件的列放在索引的后面。
- 如果常用查询中，在多个列上均存在范围条件，则将索引建在过滤性更强的列上。
- 通过创建覆盖索引（建在多个列上的索引），可以有效避免回表。

例如一条 SQL 中存在三个过滤条件，分别是 `a = 1`、`b > 0`、`c between 1 and 12`。其中 `b > 0` 可以过滤掉 30% 的数据，`c between 1 and 12` 可以过滤掉 90% 的数据，那么按照我们的基础策略，对于这条 SQL 可以在 `(a, c, b)` 上建一个索引进行优化。大家可以思考下为什么要这么创建索引？

索引中前两列是 `a` 和 `c` 很好理解，最后在索引中加上 `b` 列的原因是为了在 `select b` 的时候，可以消除回表的开销。

避免回表这里多说两句。在一些极端的场景中，可能会遇到一个查询的读行量并不多，但是这个查询的 QPS 非常高，导致整体上这个查询产生非常大量的数据读取。如果机器的 I/O 能力有限，这时候磁盘 I/O 可能会被击穿。以下面场景为例：

```
-- 创建一张 4000 个列的大宽表
CREATE TABLE T1 (C1 INT, C2 INT, C3 INT, C4 INT, ... , C4000 INT);

CREATE INDEX IDX_C1 ON T1 (C1);

EXPLAIN SELECT C3, C4 FROM T1 WHERE C1 = 1;
+-------------------------------------------------------------------------+
| Query Plan                                                              |
+-------------------------------------------------------------------------+
| ======================================================                  |
| |ID|OPERATOR        |NAME      |EST.ROWS|EST.TIME(us)|                  |
| ------------------------------------------------------                  |
| |0 |TABLE RANGE SCAN|t1(IDX_C1)|1       |7           |                  |
| ======================================================                  |
| Outputs & filters:                                                      |
| -------------------------------------                                   |
|   0 - output([t1.C3], [t1.C4]), filter(nil), rowset=16                  |
|       access([t1.__pk_increment], [t1.C3], [t1.C4]), partitions(p0)     |
|       is_index_back=true, is_global_index=false,                        |
|       range_key([t1.C1], [t1.__pk_increment]), range(1,MIN ; 1,MAX),    |
|       range_cond([t1.C1 = 1])                                           |
+-------------------------------------------------------------------------+
```

上面这个查询使用 `C1` 列上的索引后，已经得到了最优的 Query Range。该查询每次在 `IDX_C1` 上扫描的数据量很少。

但是，该查询需要得到 `C3` 和 `C4` 列的结果，而 `IDX_C1` 上没有保存这两列的值，所以它还要利用索引上记录的主表主键信息回查主表上的 `C3` 和 `C4` 列的值（`is_index_back=true`）。每次查询执行，会在 `IDX_C1` 上产生少量的顺序读取；还会在主表上产生少量的随机读取。当查询的 QPS 很高时，可能对主表会产生大量的随机读取，这会极大地消耗系统的 I/O 资源。

针对这种场景，我们需要考虑额外一个优化方向：创建覆盖索引避免回查主表。以上面的查询为例，我们可以创建一个覆盖索引来优化，例如：

```
CREATE INDEX IDX_C1_C3_C4 ON T1 (C1, C3, C4);
```

```
EXPLAIN SELECT C3, C4 FROM T1 WHERE C1 = 1;
+-------------------------------------------------------------------------------------
--------------
| Query Pla
n

+-------------------------------------------------------------------------------------
--------------
| ============================================================
=
| |ID|OPERATOR        |NAME            |EST.ROWS|EST.TIME(us)
|
| ------------------------------------------------------------
-
| |0 |TABLE RANGE SCAN|t1(IDX_C1_C3_C4)|1       |4
|
| ============================================================
=
| Outputs & filters
:
| -------------------------------------
-
|   0 - output([t1.C3], [t1.C4]), filter(nil), rowset=1
6
|       access([t1.C3], [t1.C4]), partitions(p0
)
|       is_index_back=false, is_global_index=false
,
|       range_key([t1.C1], [t1.C3], [t1.C4], [t1.__pk_increment]), range(1,MIN,MIN
,MIN ; 1,MAX,M
|       range_cond([t1.C1 = 1]
)
+-------------------------------------------------------------------------------------
--------------
```

这个查询访问 `IDX_C1_C3_C4` 时既有最优的 Query Range，也不需要回查主表（`is_index_back=false`）。实际查询执行时，只会对索引产生少量的顺序扫描，这能极大减少查询产生的 I/O 资源消耗。

一般来说，需要创建覆盖索引的场景需要满足几个特征：

- 数据库的 I/O 压力非常大。
- 大量的 I/O 是若干高频查询回查主表产生的随机 I/O 导致的。
- 高频查询需要读取的列并不多。

在这种情况下，我们可以对这些高频查询针对性地创建覆盖索引进行优化。

说明

上面这三个创建索引基础策略并不是万能的，在实际优化时往往需要结合实际场景，具体问题具体分析。

连接调优

刚刚介绍的索引调优，是 SQL 调优中必知必会的内容。

接下来要介绍的连接调优，相比索引调优，理解起来可能会困难一些，不过这里只是简单介绍几种 JOIN，不会有深度内容。

在 OceanBase 数据库中，有三种基础的连接算法：Nested-Loop Join、Merge Join 以及 Hash Join。

- Nested-Loop Join：首先把 join 左侧的数据扫描出来，然后用左侧的每一行去遍历一次右表的数据，从里面找到所有能连接上的数据行做连接。它的代价 = 左表扫描的代价 + 左表的行数 * 左表每一行遍历右表的代价，即：`cost(NLJ) = cost(left) + N(left) * cost(right)`，时间复杂度是 `O(m* n)`。
- Merge Join（这个应该也可以叫做 sort merge join）：先对左表和右表的连接键分别排序，然后用类似移动指针的方式不断地调整指针，找到匹配行做连接。它的代价 = 左右表排序的代价 + 左右表扫描的代价，即：`cost(MJ) = sort(left) + sort(right) + cost(left) + cost(right)`，时间复杂度就是排序的时间复杂度 `O(n * logn)`。
- Hash Join：扫描左表并对每一行建哈希表，扫描右表并哈希表中做探测，匹配并连接。它的代价 = 扫描左表的代价 + 左表的行数 * 每一行建哈希表的代价 + 扫描右表的代价 + 右表的行数 * 每一行探测哈希表的代价，即：`cost(HJ) = cost(left) + N(left) * create_hash_cost + cost(right) + N(right) * probe_hash_cost`。

Nested-Loop Join

OceanBase 数据库里的 Nested-Loop Join 有两种执行方式，分别为非条件下压的 Nested-Loop Join 和条件下压的 Nested-Loop Join。

我们接下来会看一下非条件下压的 NLJ 和条件下压的 NLJ 的开销有什么不同。开始前，我们做一些准备工作，先创建两张表 `t1` 和 `t2`，通过 recursive cte 分别插入 1000 行数据，然后通过系统

包函数 `dbms_stats.gather_table_stats` 收集一下统计信息。

```
drop table t1;

drop table t2;

CREATE TABLE t1
WITH RECURSIVE my_cte(a, b, c) AS
(
  SELECT 1, 0, 0
  UNION ALL
  SELECT a + 1, round((a + 1) / 2, 0), round((a + 1) / 3, 0) FROM my_cte WHERE a
< 1000
)
SELECT * FROM my_cte;

alter table t1 add primary key(a);

CREATE TABLE t2
WITH RECURSIVE my_cte(a, b, c) AS
(
  SELECT 1, 0, 0
  UNION ALL
  SELECT a + 1, round((a + 1) / 2, 0), round((a + 1) / 3, 0) FROM my_cte WHERE a
< 1000
)
SELECT * FROM my_cte;

alter table t2 add primary key(a);

call dbms_stats.gather_table_stats('TEST', 'T1', method_opt=>'for all columns siz
e auto', est

call dbms_stats.gather_table_stats('TEST', 'T2', method_opt=>'for all columns siz
e auto', est
```

非条件下压的 Nested-Loop Join

我们通过指定 hint `/*+ leading(t1 t2) use_nl(t1, t2)*/` 的方式强制让下面这条 SQL 生成 NESTED-LOOP JOIN 的计划，`t2` 上没有合适的索引可用，主键中也没有包含 `b` 列，就需要先扫描 `t2` 的全部数据，然后通过 material 算子将它物化到内存里。意味着接下来在处理 `t1` 的每一行时，都要完整地遍历 `t2` 的所有行，相当于做了笛卡尔积，时间复杂度是 `O(m * n)`，所以性能非常差。

OceanBase 数据库中只会出现有条件下压的 NLJ，在没有通过 hint 和 outline 对计划进行控制的时候，理论上不应该出现这种非条件下压的 NLJ：

```
explain select /*+ leading(t1 t2) use_nl(t1, t2)*/ * from t1, t2 where t1.b = t2.b
;
```

输出如下：

```
+-----------------------------------------------------------------------------------
------+
| Query Pla
n                                                                              |
+-----------------------------------------------------------------------------------
------+
| ==================================================
=                                    |
| |ID|OPERATOR           |NAME|EST.ROWS|EST.TIME(us)
|                                    |
| --------------------------------------------------
-                                    |
| |0 |NESTED-LOOP JOIN   |    |1877    |11578
|                                    |
| |1 |├─TABLE FULL SCAN  |t1  |1000    |84
|                                    |
| |2 |└─MATERIAL         |    |1000    |179
|                                    |
| |3 |  └─TABLE FULL SCAN|t2  |1000    |84
|                                    |
| ==================================================
=                                    |
| Outputs & filters
:                                                                      |
| ------------------------------------
-                                                 |
|   0 - output([t1.a], [t1.b], [t1.c], [t2.a], [t2.b], [t2.c]), filter(nil), rowse
t=256 |
|       conds([t1.b = t2.b]), nl_params_(nil), use_batch=fals
e                            |
|   1 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                           |
|       access([t1.a], [t1.b], [t1.c]), partitions(p0
)                                    |
|       is_index_back=false, is_global_index=false
,                                       |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                    |
|   2 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
6                           |
|   3 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
```

```
6                           |
|       access([t2.a], [t2.b], [t2.c]), partitions(p0
)                                     |
|       is_index_back=false, is_global_index=false
,                                        |
|       range_key([t2.a]), range(MIN ; MAX)always tru
e                                     |
+--------------------------------------------------------------------------------
------+
21 rows in set
```

条件下压的 Nested-Loop Join

我们改变连接条件为 `t1.a = t2.a`，并通过指定 hint `/*+ leading(t1 t2) use_nl(t1, t2)*/` 的方式强制让下面这条 SQL 生成 NESTED-LOOP JOIN 的计划。

可以看到在 `nl_params` 里面有 `t1.a`，意味着执行过程中会首先扫描 join 的左支（`t1` 表），然后把获取到的 `t1` 每一行的 `a` 值当做过滤条件，到右支上利用 `t1.a = t2.a` 作为 range_cond 去进行的 table get（主键查询）。因为右支 `t2` 表在 `a` 列上有主键，所以可以直接通过 table get 快速获取到任何一个具体的值，时间复杂度只有 `O(m)`。

```
explain select /*+ leading(t1 t2) use_nl(t1, t2)*/ * from t1, t2 where t1.a = t2.a
;
```

输出如下:

```
+--------------------------------------------------------------------------------
------+
| Query Pla
n                                                                              |
+--------------------------------------------------------------------------------
------+
| =======================================================
=                                |
| |ID|OPERATOR               |NAME|EST.ROWS|EST.TIME(us)
|                                |
| -------------------------------------------------------
-                                |
| |0 |NESTED-LOOP JOIN       |    |1000    |16274
|                                |
| |1 |├─TABLE FULL SCAN      |t1  |1000    |84
|                                |
| |2 |└─DISTRIBUTED TABLE GET|t2  |1       |16
|                                |
```

```
| ======================================================
=                                |
| Outputs & filters
:                                                                      |
| ------------------------------------
-                                                  |
|   0 - output([t1.a], [t1.b], [t1.c], [t2.a], [t2.b], [t2.c]), filter(nil), rowse
t=256 |
|       conds(nil), nl_params_([t1.a(:0)]), use_batch=tru
e                                      |
|   1 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                                 |
|       access([t1.a], [t1.b], [t1.c]), partitions(p0
)                                          |
|       is_index_back=false, is_global_index=false
,                                             |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                          |
|   2 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
6                                 |
|       access([GROUP_ID], [t2.a], [t2.b], [t2.c]), partitions(p0
)                              |
|       is_index_back=false, is_global_index=false
,                                             |
|       range_key([t2.a]), range(MIN ; MAX)
,                                                    |
|       range_cond([:0 = t2.a]
)                                                                |
+----------------------------------------------------------------------------------
------+
```

在 OceanBase 数据库中，一般情况下都只会选择条件下压的 Nested-Loop Join。除非没有等值连接条件，并且 Nested-Loop Join 也没有合适的索引可用，才有可能会考虑生成非条件下压的 Nested-Loop Join，生成这种非条件下压的 NLJ 的概率非常小，一般都会用 HJ 或 MJ 代替，如果出现，就要仔细分析下是否合理了。

Subplan Filter

这里需要多提一句和子查询相关的 subplan filter 算子，这个算子的执行方式跟 Nested Loop Join 类似，和 NLJ 一样，也需要创建合适的索引或者主键，让条件能够下压。

我们还继续用之前创建的两张表 `t1` 和 `t2`，主键都建在两张表的 `a` 列上。下面这条 SQL 是 subplan filter 没有合适的索引或主键的情况，计划和没有条件下压的 NLJ 几乎一模一样，这里不再赘述了。

```
explain select /*+no_rewrite*/ a from t1 where b > (select b from t2 where t1.b =
t2.b);
```

输出如下：

```
+-----------------------------------------------------------------------------------
-----------+
| Query Pla
n
|
+-----------------------------------------------------------------------------------
-----------+
| ================================================
=                                           |
| |ID|OPERATOR         |NAME|EST.ROWS|EST.TIME(us)
|                                           |
| ------------------------------------------------
-                                           |
| |0 |SUBPLAN FILTER   |    |334     |45415
|                                           |
| |1 |├─TABLE FULL SCAN|t1  |1000    |60
|                                           |
| |2 |└─TABLE FULL SCAN|t2  |2       |46
|                                           |
| ================================================
=                                           |
| Outputs & filters
:                                                                          |
| ------------------------------------
-                                                     |
|   0 - output([t1.a]), filter([t1.b > subquery(1)]), rowset=25
6                             |
|       exec_params_([t1.b(:0)]), onetime_exprs_(nil), init_plan_idxs_(nil), use_b
atch=false |
|   1 - output([t1.a], [t1.b]), filter(nil), rowset=25
6                                      |
|       access([t1.a], [t1.b]), partitions(p0
)                                               |
|       is_index_back=false, is_global_index=false
,                                           |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                        |
|   2 - output([t2.b]), filter([:0 = t2.b]), rowset=25
6                                      |
|       access([t2.b]), partitions(p0
)                                                       |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false]
,          |
|       range_key([t2.a]), range(MIN ; MAX)always tru
```

```
e                                         |
+----------------------------------------------------------------------------------
-----------+
```

下面这条 SQL 是 subplan filter 有合适主键的情况，计划和有条件下压的 NLJ 几乎一模一样，这里不再赘述了。

```
explain select /*+no_rewrite*/ a from t1 where b > (select b from t2 where t1.a =
t2.a);
```

输出如下：

```
+----------------------------------------------------------------------------------
----------+
| Query Pla
n                                                                                  |
+----------------------------------------------------------------------------------
----------+
| =======================================================
=                                    |
| |ID|OPERATOR               |NAME|EST.ROWS|EST.TIME(us)
|                                    |
| -------------------------------------------------------
-                                    |
| |0 |SUBPLAN FILTER         |    |334     |18043
|                                    |
| |1 |├─TABLE FULL SCAN      |t1  |1000    |60
|                                    |
| |2 |└─DISTRIBUTED TABLE GET|t2  |1       |18
|                                    |
| =======================================================
=                                    |
| Outputs & filters
:                                                                          |
| -------------------------------------
-                                                      |
|   0 - output([t1.a]), filter([t1.b > subquery(1)]), rowset=25
6                              |
|       exec_params_([t1.a(:0)]), onetime_exprs_(nil), init_plan_idxs_(nil), use_b
atch=true |
|   1 - output([t1.a], [t1.b]), filter(nil), rowset=25
6                                       |
|       access([t1.a], [t1.b]), partitions(p0
)                                                |
|       is_index_back=false, is_global_index=false
,                                         |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                        |
```

```
|   2 - output([t2.b]), filter(nil), rowset=25
6                                              |
|       access([GROUP_ID], [t2.b]), partitions(p0
)                                           |
|       is_index_back=false, is_global_index=false
,                                          |
|       range_key([t2.a]), range(MIN ; MAX)always true
,                                      |
|       range_cond([:0 = t2.a]
)                                                              |
+-----------------------------------------------------------------------------------
----------+
```

在 OceanBase 数据库中，并不是所有的子查询都能被 unnest，有时候根据 SQL 的语义，只能用 subplan filter 算子进行计算。subplan filter 的执行方式跟 Nested Loop Join 类似，所以也需要创建合适的索引避免出现非条件下压的 subplan filter。

Hash Join

- cost(NLJ) = cost(left) + N(left) * cost(right)
- cost(HJ) = cost(left) + N(left) create_hash_cost + cost(right) + N(right) probe_hash_cost

上面列出了 NLJ 和 HJ 的代价计算公式，OceanBase 数据库的优化器如果要在 NLJ 和 HJ 中进行选择，在满足下面两个条件时，才会选择 NLJ：

1. 右表有合适的索引或者主键。
2. 右表的行数/左表的行数超过一定的阈值，在 OceanBase 数据库中，大概是 20。

我们来验证一下上面的结论，先创建两张表：第一张无主键表 `t1` 有 10 行；第二张有主键表 `t2` 主键是 `a` 列，有 1000 行。

```
drop table t1;

drop table t2;

CREATE TABLE t1
WITH RECURSIVE my_cte(a, b, c) AS
(
  SELECT 1, 0, 0
  UNION ALL
  SELECT a + 1, round((a + 1) / 2, 0), round((a + 1) / 3, 0) FROM my_cte WHERE a
```

```
< 10
)
SELECT * FROM my_cte;

CREATE TABLE t2
WITH RECURSIVE my_cte(a, b, c) AS
(
  SELECT 1, 0, 0
  UNION ALL
  SELECT a + 1, round((a + 1) / 2, 0), round((a + 1) / 3, 0) FROM my_cte WHERE a
< 1000
)
SELECT * FROM my_cte;

alter table t2 add primary key(a);

call dbms_stats.gather_table_stats('TEST', 'T1', method_opt=>'for all columns siz
e auto', est

call dbms_stats.gather_table_stats('TEST', 'T2', method_opt=>'for all columns siz
e auto', est
```

当用不上 `t2` 表的主键时，如果要生成 NLJ，则会生成非条件下压的 NLJ，显然代价会很大，所以这里会生成一个 HJ 的计划：

```
explain select * from t1, t2 where t1.b = t2.b;
```

输出如下：

```
+---------------------------------------------------------------------------------
------+
| Query Pla
n                                                                            |
+---------------------------------------------------------------------------------
------+
| ================================================
=                                    |
| |ID|OPERATOR         |NAME|EST.ROWS|EST.TIME(us)
|                                    |
| ------------------------------------------------
-                                    |
| |0 |HASH JOIN        |    |1       |4
|                                    |
| |1 | ├─TABLE FULL SCAN|t1  |1       |2
|                                    |
| |2 | └─TABLE FULL SCAN|t2  |1       |2
|                                    |
```

```
| ================================================
=                                     |
| Outputs & filters
:                                                                              |
| ------------------------------------
-                                                  |
|   0 - output([t1.a], [t1.b], [t1.c], [t2.a], [t2.b], [t2.c]), filter(nil), rowse
t=256 |
|       equal_conds([t1.b = t2.b]), other_conds(nil
)                                    |
|   1 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                          |
|       access([t1.a], [t1.b], [t1.c]), partitions(p0
)                                  |
|       is_index_back=false, is_global_index=false
,                                     |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                  |
|   2 - output([t2.b], [t2.a], [t2.c]), filter(nil), rowset=25
6                          |
|       access([t2.b], [t2.a], [t2.c]), partitions(p0
)                                  |
|       is_index_back=false, is_global_index=false
,                                     |
|       range_key([t2.__pk_increment]), range(MIN ; MAX)always tru
e                      |
+---------------------------------------------------------------------------------
------+
```

当能用上 `t2` 表的主键列 `t2.a` 去进行 table get，而且右表和左表有明显的大小表关系时（右表 `t2` 有 1000 行，左表 `t1` 只有 10 行），这里就会生成一个 NLJ 的计划：

```
explain select * from t1, t2 where t1.a = t2.a;
```

输出如下：

```
+---------------------------------------------------------------------------------
------+
| Query Pla
n                                                                            |
+---------------------------------------------------------------------------------
------+
| ====================================================
=                               |
| |ID|OPERATOR                   |NAME|EST.ROWS|EST.TIME(us)
|                               |
| ----------------------------------------------------
-                               |
```

```
| |0 |NESTED-LOOP JOIN       |     |10      |165
|                                |
| |1 |├─TABLE FULL SCAN       |t1  |10      |3
|                                |
| |2 |└─DISTRIBUTED TABLE GET|t2  |1       |16
|                                |
| ======================================================
=                                |
| Outputs & filters
:                                                                     |
| ------------------------------------
-                                                    |
|   0 - output([t1.a], [t1.b], [t1.c], [t2.a], [t2.b], [t2.c]), filter(nil), rowse
t=256 |
|       conds(nil), nl_params_([t1.a(:0)]), use_batch=tru
e                               |
|   1 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                          |
|       access([t1.a], [t1.b], [t1.c]), partitions(p0
)                                  |
|       is_index_back=false, is_global_index=false
,                                     |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always tru
e                      |
|   2 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
6                          |
|       access([GROUP_ID], [t2.a], [t2.b], [t2.c]), partitions(p0
)                      |
|       is_index_back=false, is_global_index=false
,                                     |
|       range_key([t2.a]), range(MIN ; MAX)
,                                            |
|       range_cond([:0 = t2.a]
)                                                        |
+----------------------------------------------------------------------------------
------+
```

Merge Join

- cost(MJ) = cost(left) + cost(right) + sort(left) + sort(right)
- cost(HJ) = cost(left) + N(left) * hash_cost + cost(right) + N(right) * probe_cost

上面列出了 NLJ 和 HJ 的代价计算公式，它们都需要完整地扫描左表和右表，区别在于 Merge Join 要分别对两侧在连接键上进行排序，而哈希则是对左侧建哈希表、对右侧做哈希探测。相比于构建哈希表和哈希探测（`O(n)`）来说，做排序的代价会更高（`O(nlogn)`）。因此，在一般情况

下，一定是 Hash Join 优于 Merge Join。

只有在一些非常特殊的场景下，才会选择 Merge Join 。比如两侧都有序时，就可以省去排序的代价，直接做一次归并就好了。

还是拿一开始的两张表 `t1` 和`t2` 做实验，`t1` 和 `t2` 都有建在 `a` 列上的主键。

```
drop table t1;

drop table t2;

CREATE TABLE t1
WITH RECURSIVE my_cte(a, b, c) AS
(
  SELECT 1, 0, 0
  UNION ALL
  SELECT a + 1, round((a + 1) / 2, 0), round((a + 1) / 3, 0) FROM my_cte WHERE a
< 1000
)
SELECT * FROM my_cte;

alter table t1 add primary key(a);

CREATE TABLE t2
WITH RECURSIVE my_cte(a, b, c) AS
(
  SELECT 1, 0, 0
  UNION ALL
  SELECT a + 1, round((a + 1) / 2, 0), round((a + 1) / 3, 0) FROM my_cte WHERE a
< 1000
)
SELECT * FROM my_cte;

alter table t2 add primary key(a);

call dbms_stats.gather_table_stats('TEST', 'T1', method_opt=>'for all columns siz
e auto', est

call dbms_stats.gather_table_stats('TEST', 'T2', method_opt=>'for all columns siz
e auto', est
```

如果连接条件都是本来就有序的主键 `a` 列，则会生成 merge join。

```
explain select * from t1, t2 where t1.a = t2.a;
+------------------------------------------------------------------------------
------+
| Query Pla
n                                                                              |
```

```
+------------------------------------------------------------------------------------
------+
| =================================================
=                                      |
| |ID|OPERATOR         |NAME|EST.ROWS|EST.TIME(us)
|                                      |
| -------------------------------------------------
-                                      |
| |0 |MERGE JOIN       |    |1000    |301
|                                      |
| |1 | ├─TABLE FULL SCAN|t1  |1000    |84
|                                      |
| |2 | └─TABLE FULL SCAN|t2  |1000    |84
|                                      |
| =================================================
=                                      |
| Outputs & filters
:                                                                               |
| -------------------------------------
-                                                  |
|   0 - output([t1.a], [t1.b], [t1.c], [t2.a], [t2.b], [t2.c]), filter(nil), rowse
t=256 |
|       equal_conds([t1.a = t2.a]), other_conds(nil
)                                     |
|       merge_directions([ASC]
)                                                          |
|   1 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                          |
|       access([t1.a], [t1.b], [t1.c]), partitions(p0
)                                   |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                   |
|   2 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
6                          |
|       access([t2.a], [t2.b], [t2.c]), partitions(p0
)                                   |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t2.a]), range(MIN ; MAX)always tru
e                                   |
+------------------------------------------------------------------------------------
------+
```

如果连接条件都是无序的 `b` 列，则会生成 hash join。

```
explain select * from t1, t2 where t1.b = t2.b;
```

输出如下：

```
+------------------------------------------------------------------------------------
------+
| Query Pla
n                                                                              |
+------------------------------------------------------------------------------------
------+
| ================================================
=                                      |
| |ID|OPERATOR          |NAME|EST.ROWS|EST.TIME(us)
|                                      |
| ------------------------------------------------
-                                      |
| |0 |HASH JOIN         |    |1877    |481
|                                      |
| |1 |├─TABLE FULL SCAN|t1  |1000    |84
|                                      |
| |2 |└─TABLE FULL SCAN|t2  |1000    |84
|                                      |
| ================================================
=                                      |
| Outputs & filters
:                                                                      |
| ------------------------------------
-                                                  |
|   0 - output([t1.a], [t1.b], [t1.c], [t2.a], [t2.b], [t2.c]), filter(nil), rowse
t=256 |
|       equal_conds([t1.b = t2.b]), other_conds(nil
)                                    |
|   1 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                          |
|       access([t1.a], [t1.b], [t1.c]), partitions(p0
)                                 |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                 |
|   2 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
6                          |
|       access([t2.a], [t2.b], [t2.c]), partitions(p0
)                                 |
|       is_index_back=false, is_global_index=false
,                                      |
|       range_key([t2.a]), range(MIN ; MAX)always tru
e                                 |
+------------------------------------------------------------------------------------
------+
```

如果连接条件都是无序的 b 列，并通过指定 hint 强制要求生成 merge join 的计划，那么执行计划中一定会被先分配 sort 算子，通过 sort 算子进行排序后再进行 merge join，这种计划的代价往往会比 hash join 高。

```
explain select /*+ USE_MERGE(t1 t2) */ * from t1, t2 where t1.b = t2.b;
```

输出如下：

```
+-------------------------------------------------------------------------------------
------+
| Query Pla
n                                                                                |
+-------------------------------------------------------------------------------------
------+
| ==================================================
=                                    |
| |ID|OPERATOR           |NAME|EST.ROWS|EST.TIME(us)
|                                    |
| --------------------------------------------------
-                                    |
| |0 |MERGE JOIN          |    |1877    |750
|                                    |
| |1 |├─SORT              |    |1000    |294
|                                    |
| |2 ||  └─TABLE FULL SCAN|t1  |1000    |84
|                                    |
| |3 |└─SORT              |    |1000    |294
|                                    |
| |4 |  └─TABLE FULL SCAN|t2  |1000    |84
|                                    |
| ==================================================
=                                    |
| Outputs & filters
:                                                                       |
| ------------------------------------
-                                                  |
|   0 - output([t1.a], [t1.b], [t1.c], [t2.a], [t2.b], [t2.c]), filter(nil), rowse
t=256 |
|       equal_conds([t1.b = t2.b]), other_conds(nil
)                                     |
|       merge_directions([ASC]
)                                                          |
|   1 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                           |
|       sort_keys([t1.b, ASC]
)                                                           |
|   2 - output([t1.a], [t1.b], [t1.c]), filter(nil), rowset=25
6                           |
```

```
|       access([t1.a], [t1.b], [t1.c]), partitions(p0
)                                  |
|       is_index_back=false, is_global_index=false
,                                     |
|       range_key([t1.a]), range(MIN ; MAX)always tru
e                                  |
|   3 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
6                         |
|       sort_keys([t2.b, ASC]
)                                                           |
|   4 - output([t2.a], [t2.b], [t2.c]), filter(nil), rowset=25
6                         |
|       access([t2.a], [t2.b], [t2.c]), partitions(p0
)                                  |
|       is_index_back=false, is_global_index=false
,                                     |
|       range_key([t2.a]), range(MIN ; MAX)always tru
e                                  |
+--------------------------------------------------------------------------------
------+
```

连接调优总结

这三个 join 方式是数据库最基础的知识点，最后简单地总结一下需要大家记住的几个点：

- 计划里绝大多数的情况都只会选择有条件下压的 Nested-Loop Join，如果选择了非条件下压的 NLJ，需要创建合适的索引让计划变成有条件下压的 NLJ，或者指定 hint 变更 join 的方式。与联接操作相关的 Hint 详见官网《OceanBase 数据库》文档 参考指南/SQL 参考/SQL 语法/普通租户（Oracle 模式）/基本元素/注释/Hint/Hint 清单/与联接操作相关的 Hint。
- subplan filter 的执行方式跟 Nested Loop Join 类似，所以也需要创建合适的索引避免出现非条件下压的 subplan filter。
- 计划里如果有 merge join，往往是可以利用下层算子的有序性。如果下层算子都是无序的，计划中在 merge 前还专门分配了一些 sort 进行排序，需要分析下是否需要通过 hint 改成使用 hash join 进行连接。
- 如果没有可用的索引和主键，也没有有序性，那么一般 hash join 的代价是最低的。

排序和 limit 调优

order by 和 limit 用于获取排序后的一定行数的结果集，在业务查询中十分常见。为获取这些结果集，如果优化器生成的是一个执行效率高的计划，执行时可能只需要从相关表上读取很少的数据；反之，可能需要读取相关表的全部数据。在这种场景的调优中，需要理解产生这两种计划的原因并通过一些优化方法获得效率较高的计划。

排序算子的生成与优化

查询中包含 limit 子句时，生成的执行计划中是否包含阻塞性算子（需要获取孩子节点的全部行后才能开始向父亲节点吐行）对执行性能影响很大。排序作为常见的阻塞性算子，在查询中包含 ORDER BY 子句或生成的计划需要分配基于排序算法的算子时(如 MERGE DISTINCT/MERGE GROUP BY/MERGE JOIN 等)，就需要尝试分配排序算子，以获取按指定 sort key 有序的（中间）结果集。为了避免产生不必要的排序算子，同时充分利用已经有序的中间结果集，优化器维护了一些序相关的信息，并在尝试分配排序算子时进行了一些优化。

计划树中的序

在执行计划中，算子的输入序可能是有序的，对于基表扫描这个序来自于表结构索引产生的序，对于其它算子这个序可能来自孩子算子的输出序。不同于 order by 明确要求的输出序，但大部分使用排序算法的算子需要输入的序是可以调整的，并且这个序可以保留继承到算子输出序。通过优化选择排序型算子使用的序，可以避免进行多次排序。

优化器在确定排序型算子使用的序时，会通过已有输入序和之后可能需要的序得到需要最终使用的序，以下方查询为例，group by 使用 merge group by 算子时，是可以任意调整算子输入序中 `c1`、`c2` 的前后位置及排序方向的，例如：

先创建表和索引：

```
create table t1(c1 int, c2 int, c3 int, c4 int, pk int primary key);

create index idx on t1(c1, c2, c3);
```

- 查看执行计划

```
explain select /*+ no_use_hash_aggregation */ c1, c2, sum(c3)
  from t1 t
  group by c2, c1
  order by c1, c2;
```

输出如下，选择索引 `idx` 产生了 `c1, c2` 的序，避免产生排序算子，并避免了 `order by c1, c2` 需要分配的排序算子。

```
+------------------------------------------------------------------------------------
---------------------+
| Query Pla
n
          |
+------------------------------------------------------------------------------------
---------------------+
| ==================================================
=                                                  |
| |ID|OPERATOR          |NAME   |EST.ROWS|EST.TIME(us)
|                                                  |
| --------------------------------------------------
-                                                  |
| |0 |MERGE GROUP BY    |       |1        |4
|                                                  |
| |1 | └─TABLE FULL SCAN|t(idx)|1        |4
|                                                  |
| ==================================================
=                                                  |
| Outputs & filters
:
  |
| ------------------------------------
-                                                                |
|   0 - output([t.c1], [t.c2], [T_FUN_SUM(t.c3)]), filter(nil), rowset=1
6                               |
|       group([t.c1], [t.c2]), agg_func([T_FUN_SUM(t.c3)]
)                                              |
|   1 - output([t.c1], [t.c2], [t.c3]), filter(nil), rowset=1
6                                          |
|       access([t.c1], [t.c2], [t.c3]), partitions(p0
)                                              |
|       is_index_back=false, is_global_index=false
,                                                 |
|       range_key([t.c1], [t.c2], [t.c3], [t.pk]), range(MIN,MIN,MIN,MIN ; MAX,MAX
,MAX,MAX)always true |
+------------------------------------------------------------------------------------
---------------------+
```

- 调整 order by，并通过 hint 指定使用主键（`full(t)`）

```
explain select /*+ no_use_hash_aggregation full(t) */ c1, c2, sum(c3)
  from t1 t
  group by c1, c2
```

```
  order by c2, c1;
```

输出如下，merge group by 直接使用 `order by c2, c1` 的序作为算子聚合的输入序，并在 group by 下方分配排序算子，`sort_keys` 是 `c2, c1`。

```
+--------------------------------------------------------------------------+
| Query Plan                                                               |
+--------------------------------------------------------------------------+
| ===================================================                      |
| |ID|OPERATOR           |NAME|EST.ROWS|EST.TIME(us)|                      |
| ---------------------------------------------------                      |
| |0 |MERGE GROUP BY     |    |1       |5           |                      |
| |1 | └─SORT            |    |1       |4           |                      |
| |2 |   └─TABLE FULL SCAN|t   |1       |4           |                      |
| ===================================================                      |
| Outputs & filters:                                                       |
| -------------------------------------                                    |
|   0 - output([t.c1], [t.c2], [T_FUN_SUM(t.c3)]), filter(nil), rowset=16 |
|       group([t.c2], [t.c1]), agg_func([T_FUN_SUM(t.c3)])                 |
|   1 - output([t.c2], [t.c1], [t.c3]), filter(nil), rowset=16             |
|       sort_keys([t.c2, ASC], [t.c1, ASC])                                |
|   2 - output([t.c1], [t.c2], [t.c3]), filter(nil), rowset=16             |
|       access([t.c1], [t.c2], [t.c3]), partitions(p0)                     |
|       is_index_back=false, is_global_index=false,                        |
|       range_key([t.pk]), range(MIN ; MAX)always true                     |
+--------------------------------------------------------------------------+
```

- 通过 hint 指定使用索引 idx（`index(t idx)`）

```
explain select /*+no_use_hash_aggregation index(t idx)*/ c1, c2, sum(c3)
  from t1 t
  group by c1, c2
  order by c2, c1;
```

输出如下，按照代价生成了 merge group by，直接使用 `idx` 产生的 `c1, c2` 的序，并在聚合后为 `order by c2, c1` 分配排序算子。

```
+------------------------------------------------------------------------------------
---------------------+
| Query Pla
n
          |
+------------------------------------------------------------------------------------
---------------------+
| ===================================================
```

```
=                                                   |
| |ID|OPERATOR           |NAME   |EST.ROWS|EST.TIME(us)
|                                                   |
| ---------------------------------------------------
-                                                   |
| |0 |SORT               |       |1       |4
|                                                   |
| |1 |└─MERGE GROUP BY   |       |1       |4
|                                                   |
| |2 |  └─TABLE FULL SCAN|t(idx)|1       |4
|                                                   |
| ===================================================
=                                                   |
| Outputs & filters
:
  |
| ------------------------------------
-                                                                |
|   0 - output([t.c1], [t.c2], [T_FUN_SUM(t.c3)]), filter(nil), rowset=1
6                           |
|       sort_keys([t.c2, ASC], [t.c1, ASC]
)                                                           |
|   1 - output([t.c2], [t.c1], [T_FUN_SUM(t.c3)]), filter(nil), rowset=1
6                           |
|       group([t.c1], [t.c2]), agg_func([T_FUN_SUM(t.c3)]
)                                              |
|   2 - output([t.c1], [t.c2], [t.c3]), filter(nil), rowset=1
6                                         |
|       access([t.c1], [t.c2], [t.c3]), partitions(p0
)                                                  |
|       is_index_back=false, is_global_index=false
,                                                     |
|       range_key([t.c1], [t.c2], [t.c3], [t.pk]), range(MIN,MIN,MIN,MIN ; MAX,MAX
,MAX,MAX)always true |
+----------------------------------------------------------------------------------
---------------------+
```

排序算子的分配与优化

尝试分配排序算子以获得期望排序时，孩子算子的输出可能是有序的，首先根据输入序和其它信息进行检查，判断输入序能否满足期望的输出序。

对于下方的三个查询，包含 order by 子句需要最终结果有序，由于匹配了索引或主键提供的序，最终不需要分配排序算子：

```
create table t1(c1 int, c2 int, c3 int, c4 int, pk int primary key);
```

```
create index idx on t1(c1, c2, c3);

-- idx 索引可以提供 (c1, c2, c3) 的序，输出序匹配 order by c1, c2
explain select /*+index(t idx)*/ * from t1 t order by c1, c2;
+-----------------------------------------------------------------------------------
--------------
| Query Pla
n

+-----------------------------------------------------------------------------------
--------------
| =================================================
=
| |ID|OPERATOR       |NAME   |EST.ROWS|EST.TIME(us)
|
| -------------------------------------------------
-
| |0 |TABLE FULL SCAN|t(idx)|1       |7
|
| =================================================
=
| Outputs & filters
:
| -----------------------------------
-
|   0 - output([t.c1], [t.c2], [t.c3], [t.c4], [t.pk]), filter(nil), rowset=1
6
|       access([t.pk], [t.c1], [t.c2], [t.c3], [t.c4]), partitions(p0
)
|       is_index_back=true, is_global_index=false
,
|       range_key([t.c1], [t.c2], [t.c3], [t.pk]), range(MIN,MIN,MIN,MIN ; MAX,MAX
,MAX,MAX)alway
+-----------------------------------------------------------------------------------
--------------

-- idx 索引可以提供 (c1, c2, c3) 的序， 由于包含 c1 = 4 谓词，输出序匹配 order by c2
explain select /*+index(t idx)*/ * from t1 t where c1 = 4 order by c2;
+-----------------------------------------------------------------------------------
--------+
| Query Pla
n                                                                                |
+-----------------------------------------------------------------------------------
--------+
| ==================================================
=                                        |
| |ID|OPERATOR        |NAME   |EST.ROWS|EST.TIME(us)
|                                        |
| --------------------------------------------------
-                                        |
| |0 |TABLE RANGE SCAN|t(idx)|1       |7
```

```
|                                       |
| =================================================
=                                       |
| Outputs & filters
:                                                                  |
| ------------------------------------
-                                                    |
|   0 - output([t.c1], [t.c2], [t.c3], [t.c4], [t.pk]), filter(nil), rowset=1
6             |
|       access([t.pk], [t.c1], [t.c2], [t.c3], [t.c4]), partitions(p0
)                     |
|       is_index_back=true, is_global_index=false
,                                         |
|       range_key([t.c1], [t.c2], [t.c3], [t.pk]), range(4,MIN,MIN,MIN ; 4,MAX,MAX
,MAX),  |
|       range_cond([t.c1 = 4]
)                                                           |
+----------------------------------------------------------------------------------
--------+

-- 主键提供 (pk) 的序，由于主键唯一性，输出序匹配 order by pk, c3, c2, c1
explain select /*+index(t primary)*/ * from t1 t order by pk, c3, c2, c1;
+-------------------------------------------------------------------------------+
| Query Plan                                                                    |
+-------------------------------------------------------------------------------+
| ===============================================                               |
| |ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)|                               |
| -----------------------------------------------                               |
| |0 |TABLE FULL SCAN|t   |1       |4           |                               |
| ===============================================                               |
| Outputs & filters:                                                            |
| -------------------------------------                                         |
|   0 - output([t.c1], [t.c2], [t.c3], [t.c4], [t.pk]), filter(nil), rowset=16 |
|       access([t.pk], [t.c1], [t.c2], [t.c3], [t.c4]), partitions(p0)         |
|       is_index_back=false, is_global_index=false,                             |
|       range_key([t.pk]), range(MIN ; MAX)always true                          |
+-------------------------------------------------------------------------------+
```

对于无法消除的排序，优化器分配排序算子前会对需要排序的列进行一定的优化，方法分别是前缀排序和简化排序列。

前缀排序

需要分配排序算子时，如果算子输入序能够匹配部分期望的序，使用前缀排序利用部分输入序进行优化。

下方查询使用 idx 索引后 table scan 算子输出序为 `(c1, c2, c3)`，使用前缀排序后对每组 `c1` 仅对 pk 进行排序（`prefix_pos(1)`）。

```
explain select /*+ index(t idx) */ c1 from t1 t order by c1, pk;
```

输出如下：

```
+------------------------------------------------------------------------------------
---------------------+
| Query Pla
n
          |
+------------------------------------------------------------------------------------
---------------------+
| ===================================================
=                                                   |
| |ID|OPERATOR         |NAME  |EST.ROWS|EST.TIME(us)
|                                                   |
| ---------------------------------------------------
-                                                   |
| |0 |SORT             |      |1       |4
|                                                   |
| |1 |└─TABLE FULL SCAN|t(idx)|1       |4
|                                                   |
| ===================================================
=                                                   |
| Outputs & filters
:
  |
| ------------------------------------
-                                                                 |
|   0 - output([t.c1]), filter(nil), rowset=1
6                                                          |
|       sort_keys([t.c1, ASC], [t.pk, ASC]), prefix_pos(1
)                                                |
|   1 - output([t.pk], [t.c1]), filter(nil), rowset=1
6                                                      |
|       access([t.pk], [t.c1]), partitions(p0
)                                                          |
|       is_index_back=false, is_global_index=false
,                                                     |
|       range_key([t.c1], [t.c2], [t.c3], [t.pk]), range(MIN,MIN,MIN,MIN ; MAX,MAX
,MAX,MAX)always true |
+------------------------------------------------------------------------------------
---------------------+
```

简化排序列

对需要排序的列进行一定的简化优化，减少需要排序列数量。

化简排序列过程中主要使用以下方式:

- 利用等值条件关系进行化简排序列

  示例如下，使用 `c3 = 1` 及 `c2 = c1` 进行化简。

```
explain select c1 from t1 where c3 = 1 and c2 = c1 order by c3, c2, c1;
```

  输出如下，此处可阅读一下这个计划，并思考一下这个计划中为什么不需要分配 sort 算子。

```
+----------------------------------------------------------------------------------
---------------------+
| Query Pla
n
          |
+----------------------------------------------------------------------------------
---------------------+
| ================================================
=                                                   |
| |ID|OPERATOR       |NAME   |EST.ROWS|EST.TIME(us)
|                                                   |
| ------------------------------------------------
-                                                   |
| |0 |TABLE FULL SCAN|t(idx)|1       |4
|                                                   |
| ================================================
=                                                   |
| Outputs & filters
:
  |
| -------------------------------------
-                                                               |
|   0 - output([t.c1]), filter([t.c3 = 1], [t.c2 = t.c1]), rowset=1
6                                |
|       access([t.c3], [t.c2], [t.c1]), partitions(p0
)                                            |
|       is_index_back=false, is_global_index=false, filter_before_indexback[false,
false],                |
|       range_key([t.c1], [t.c2], [t.c3], [t.pk]), range(MIN,MIN,MIN,MIN ; MAX,MAX
,MAX,MAX)always true |
+----------------------------------------------------------------------------------
---------------------+
```

- 利用主键/唯一索引化简排序列

```
explain select * from t1 order by pk, c3, c2;
```

输出如下，因为 pk 为主键，pk 列满足唯一、非空的特性，且主表按照 pk 有序，因此无需再继续对 c3, c2 进行排序。

```
+-------------------------------------------------------------------+
| Query Plan                                                        |
+-------------------------------------------------------------------+
| ================================================                  |
| |ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)|                   |
| ------------------------------------------------                  |
| |0 |TABLE FULL SCAN|t1  |1       |4           |                   |
| ================================================                  |
| Outputs & filters:                                                |
| -------------------------------------                             |
|   0 - output([t1.c1], [t1.c3], [t1.c2]), filter(nil), rowset=16  |
|       access([t1.c1], [t1.c3], [t1.c2]), partitions(p0)           |
|       is_index_back=false, is_global_index=false,                 |
|       range_key([t1.pk]), range(MIN ; MAX)always true             |
+-------------------------------------------------------------------+
```

排序 + limit 场景性能优化

消除排序

对包含 order by + limit 的单表查询，可以通过创建联合索引，在使用索引进行快速扫描的同时消除排序，实现一个流式计划（即没有 sort 这类阻塞算子的计划）。

```
create table t1(c1 int, c2 int, c3 int, c4 int, pk int primary key);

create index idx on t1(c1, c2, c3);

-- 索引上的数据按照 c1、c2 有序，只需对 c3 > 0 进行扫描，找到前十个符合条件的行即可（filter([t
1.c3 > 0]) + limit(10))
explain select * from t1 where c1 = 1 and c3 > 0 order by c2 limit 10;
+----------------------------------------------------------------------------------
--------------
| Query Pla
n

+----------------------------------------------------------------------------------
--------------
| ==================================================
```

```
=
| |ID|OPERATOR         |NAME    |EST.ROWS|EST.TIME(us)
|
| ---------------------------------------------------
-
| |0 |TABLE RANGE SCAN|t1(idx)|1       |5
|
| ===================================================
=
| Outputs & filters
:
| ------------------------------------
-
|   0 - output([t1.c1], [t1.c2], [t1.c3], [t1.c4], [t1.pk]), filter([t1.c3 > 0]),
rowset=16
|       access([t1.pk], [t1.c1], [t1.c3], [t1.c2], [t1.c4]), partitions(p0
)
|       limit(10), offset(nil), is_index_back=true, is_global_index=false, filter_
before_indexba
|       range_key([t1.c1], [t1.c2], [t1.c3], [t1.pk]), range(1,MIN,MIN,MIN ; 1,MAX
,MAX,MAX),
|       range_cond([t1.c1 = 1]
)
+----------------------------------------------------------------------------------
--------------
```

上面这条 SQL 能够利用 `c1 = 1` 这个等值条件抽取 query range 进行索引快速扫描，由于 `c1` 上的等值条件，使用 `idx` 索引扫描也直接获得了按 `c2` 有序的结果集，在扫描得到 10 行数据后能够提前结束计划的执行。

> **注意**
>
> SQL 中有一个 `c3 > 0` 谓词，如果以 (c1, c3, c2) 为顺序创建索引，可以直接使用索引抽取 query range 实现 `c3 > 0` 的过滤，但是此时无法获得按 c2 有序的结果集。当 `c3 > 0` 有很强的过滤性时，消除排序获得流式计划减少数据扫描基本不会带来优化，使用 idx(c1, c2, c3) 反而使得计划无法充分利用 `c3 > 0` 的过滤性。所以，SQL 调优的方法没有哪个是万金油，还是要“具体问题具体分析”。

减少扫描 / 计算开销

对于大宽表或包含 longtext 等大对象表的扫描，当查询中包含 order by 和 limit 时，优化器将使用晚期物化的优化策略，能够减少大量数据的扫描。

对于下方查询 SQL，过滤条件及排序列均在索引 `idx` 上，先通过索引 `idx` 扫描排序列、过滤条件

列及主键，通过排序后得到最终需要返回的行，再通过主键进行主表扫描返回所有列。对于包含 order by 和 limit 的复杂查询，也可以按照相同的策略对查询进行改写。

先创建表和索引：

```
create table t1(c1 int, c2 int, c3 int, c4 longtext, pk int primary key);

create index idx on t1(c1, c2, c3);
```

- 通过 hint 指定使用索引 `idx`（`index(t idx)`）

```
explain select /*+ index(t idx) */ * from t1 where c1 > 0 order by c2 limit 10;
```

输出如下，过滤条件及排序列均在索引 `idx` 上。

```
+-----------------------------------------------------------------------------------
--------------+
| Query Pla
n
   |
+-----------------------------------------------------------------------------------
--------------+
| ========================================================
=                                       |
| |ID|OPERATOR            |NAME    |EST.ROWS|EST.TIME(us)
|                                       |
| --------------------------------------------------------
-                                       |
| |0 |NESTED-LOOP JOIN    |        |1       |6
|                                       |
| |1 |├─TOP-N SORT         |        |1       |4
|                                       |
| |2 ||  └─TABLE RANGE SCAN|t1(idx) |1       |4
|                                       |
| |3 |└─TABLE GET          |t1_alias|1       |2
|                                       |
| ========================================================
=                                       |
| Outputs & filters
:                                                                               |
| ------------------------------------
-                                                           |
|   0 - output([t1.c1], [t1.c2], [t1_alias.c3], [t1_alias.c4], [t1.pk]), filter(ni
l)            |
|       conds(nil), nl_params_([t1.pk(:0)]), use_batch=fals
e                                     |
|   1 - output([t1.pk], [t1.c1], [t1.c2]), filter(nil
```

```
)                                          |
|       sort_keys([t1.c2, ASC]), topn(10
)                                                       |
|   2 - output([t1.pk], [t1.c1], [t1.c2]), filter(nil
)                                          |
|       access([t1.pk], [t1.c1], [t1.c2]), partitions(p0
)                                       |
|       is_index_back=false, is_global_index=false
,                                             |
|       range_key([t1.c1], [t1.c2], [t1.c3], [t1.pk]), range(0,MAX,MAX,MAX ; MAX,M
AX,MAX,MAX),  |
|       range_cond([t1.c1 > 0]
)                                                                 |
|   3 - output([t1_alias.c3], [t1_alias.c4]), filter(nil
)                                       |
|       access([t1_alias.c3], [t1_alias.c4]), partitions(p0
)                                    |
|       is_index_back=false, is_global_index=false
,                                             |
|       range_key([t1_alias.pk]), range(MIN ; MAX)
,                                             |
|       range_cond([t1_alias.pk = :0]
)                                                          |
+-------------------------------------------------------------------------------------
---------------+
```

从输出结果中可以发现执行的流程为：

1. 2 号算子 `TABLE RANGE SCAN` 利用索引的有序性，吐出 `c1 > 0` 的行（此时行中只包含 `pk` 和索引列 `c1`、`c2`）。

2. 1 号算子 `TOP-N SORT` 接着对 2 号算子的输出，进行 `c2` 列的排序，吐出 `c1 > 0 order by c2 limit 10` 的行（此时行中只包含 `pk` 和索引列 `c1`、`c2`）。

3. 3 号算子 `TABLE GET` 利用主键 `pk` 对 1 号算子吐出的 10 行里的 10 个 `pk` 值进行随机访问，补齐需要返回给用户的列（`c3`、`c4`），这里只需要对 10 个 longtext 值进行 I/O 即可。

4. 0 号算子返回 `pk`、`c1`、`c2`、`c3`、`c4` 这几个列的值。

> **说明**
>
> 对于包含 order by 和 limit 的复杂查询，也可以按照相同的策略对查询进行改写。

- 2 号算子 `TABLE RANGE SCAN` 利用索引的有序性，吐出 `c1 > 0` 的行（此时行中只包含 `pk` 和索引列 `c1`、`c2`）。
- 1 号算子 `TOP-N SORT` 接着对 2 号算子的输出，进行 `c2` 列的排序，吐出 `c1 > 0 order by c2 limit 10` 的行（此时行中只包含 `pk` 和索引列 `c1`、`c2`）。
- 3 号算子 `TABLE GET` 利用主键 `pk` 对 1 号算子吐出的 10 行里的 10 个 `pk` 值进行随机访问，补齐需要返回给用户的列（`c3`、`c4`），这里只需要对 10 个 longtext 值进行 I/O 即可。
- 0 号算子返回 `pk`、`c1`、`c2`、`c3`、`c4` 这几个列的值。
- 通过 `no_use_late_materialization` 禁止晚期物化优化

此时需要通过索引回表（`is_index_back=true`）读取表中所有满足过滤条件的匹配行，这个计划可能会对大量的大对象 `c4` 列进行不必要的 I/O，和上面晚期物化的计划相比，一般都属于较差的计划。

```
explain select /*+ index(t idx) no_use_late_materialization */ * from t1 where c1
> 0 order by c2 limit 10;
```

输出如下：

```
+------------------------------------------------------------------------------
--------------+
| Query Pla
n
   |
+------------------------------------------------------------------------------
--------------+
| ====================================================
=                                         |
| |ID|OPERATOR          |NAME    |EST.ROWS|EST.TIME(us)
|                                         |
| ----------------------------------------------------
-                                         |
| |0 |TOP-N SORT        |        |1        |7
|                                         |
| |1 | └─TABLE RANGE SCAN|t1(idx)|1        |7
|                                         |
| ====================================================
=                                         |
```

```
| Outputs & filters
:                                                                              |
| ------------------------------------
-                                                           |
|   0 - output([t1.c1], [t1.c2], [t1.c3], [t1.c4], [t1.pk]), filter(nil), rowset=1
6             |
|       sort_keys([t1.c2, ASC]), topn(10
)                                                         |
|   1 - output([t1.pk], [t1.c1], [t1.c2], [t1.c3], [t1.c4]), filter(nil), rowset=1
6             |
|       access([t1.pk], [t1.c1], [t1.c2], [t1.c3], [t1.c4]), partitions(p0
)                     |
|       is_index_back=true, is_global_index=false
,                                                |
|       range_key([t1.c1], [t1.c2], [t1.c3], [t1.pk]), range(0,MAX,MAX,MAX ; MAX,M
AX,MAX,MAX),  |
|       range_cond([t1.c1 > 0]
)                                                                  |
+---------------------------------------------------------------------------------
--------------+
```

7.7 SQL 性能问题的典

大家通过学习前两个节，已经了解了如何阅读和管理 SQL 的执行计划，以及常见的几种 SQL 调优方式，获得了学习这一节的基础知识。

当用户遇到由于 SQL 原因导致的性能问题时，一般可以通过以下几个步骤进行排查：

1. 通过全链路追踪确认各阶段耗时占比，确认耗时长的阶段是什么？
2. 如果上一步显示慢在 observer 模块，则可以通过 `oceanbase.gv$ob_sql_audit` 分析具体是 observer 内的什么阶段耗时长了？
3. 如果上一步耗时长的阶段在执行阶段，则先根据上文的内容判断是否存在 buffer 表、大小账号、硬解析等问题？
4. 如果上述问题均不存在，则需要通过 explain extended 展示的执行计划来分析优化器的估行和真实行数是否有巨大差距，如果有明显差距，则需要手动收集统计信息。否则就进一步考虑是需要创建更合适的索引、通过 hint 调整计划形态、通过 hint 调整并行度（设置并行执行的并行度详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/并行执行计划/设置并发执行/设置并行执行并行度）等等。这里附上一个 SQL Explain 进行优化的实践，详见官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/SQL 执行计划/SQL Explain 优化实践（理解起来可能稍有难度）。

在本节中，首先会为大家展示上面排查步骤中提到的几个常被用于进行 SQL 性能问题分析的工具，然后介绍如何通过这几个工具找到 SQL 性能优化的方向，最后会对 SQL 调优的最典型的场景和常见问题进行一个汇总。

SQL 性能问题分析工具

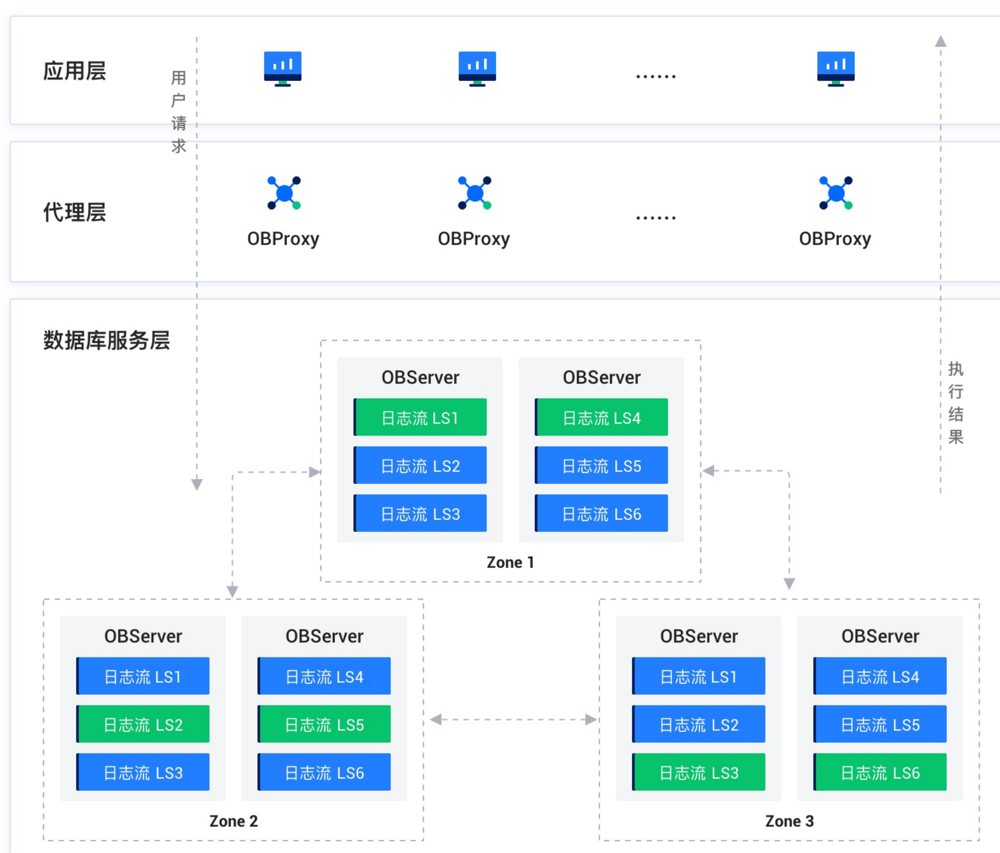
在介绍 SQL 性能问题的典型场景之前，先简单介绍几个用于分析性能问题的常用工具，分别是 show trace、sql audit 和 sql plan monitor。

全链路追踪（show trace）

OceanBase 数据库的数据链路为 APPServer <-> OBProxy <-> OBServer。APPServer 通过数

据库驱动连接 ODP 发送请求，由于 OceanBase 数据库的分布式架构，用户数据以多分区多副本的方式分布于多个 OBServer 节点上，ODP 将用户请求转发到最合适的 OBServer 节点执行，并将执行结果返回用户。每个 OBServer 节点也有路由转发的功能，如果发现请求不能在当前节点执行，则会转发请求到正确的 OBServer 节点。

当出现端到端的性能问题时（在数据库场景下，端到端表示在应用服务器上观察到 SQL 请求的 RT 很高），此时首先需要定位是数据库访问链路上哪个组件的问题，再排查组件内的具体问题。



全链路追踪覆盖了两条主要的数据流路径：

- 一条是请求从应用出发，通过客户端（比如 JDBC 或 OCI）传递至 ODP（代理服务器），然后由 ODP 转发至 OBServer 节点，最终结果返回给应用程序。
- 另一条则是请求直接从应用程序通过客户端发送至 OBServer 节点，然后结果直接返回。

接下来用这两条路径，分别举个一简单的例子。

1. 通过 ODP 连接 OceanBase 数据库，创建表并插入数据。

```
create table t1(c1 int);

insert into t1 values(123);
```

2. 在当前 session 上开启全链路诊断的 show trace 功能。

```
SET ob_enable_show_trace = ON;
```

3. 执行一条简单的查询语句。

```
SELECT c1 FROM t1 LIMIT 2;
```

输出如下：

```
+-------+
| c1    |
+-------+
|   123 |
+-------+
1 row in set
```

4. 执行 show trace 语句。

```
SHOW TRACE;
```

输出如下：

```
+-----------------------------------------------------------+---------------------
-------+------------+
| Operation                                                 | StartTim
e                   | ElapseTime |
+-----------------------------------------------------------+---------------------
-------+------------+
| ob_proxy                                                  | 2024-03-20 15:07:46.
419433 | 191.999 ms |
| ├── ob_proxy_partition_location_lookup                     | 2024-03-20 15:07:4
6.419494 | 181.839 ms |
| ├── ob_proxy_server_process_req                            | 2024-03-20 15:07:4
6.601697 | 9.138 ms   |
| └── com_query_process                                      | 2024-03-20 15:07:4
6.601920 | 8.824 ms   |
|     └── mpquery_single_stmt                                | 2024-03-20 15:07:4
6.601940 | 8.765 ms   |
|         ├── sql_compile                                    | 2024-03-20 15:07:4
```

```
6.601984 | 7.666 ms   |
|          |    ├─── pc_get_plan                                | 2024-03-20 15:07:
46.602051 | 0.029 ms   |
|          |    └─── hard_parse                                 | 2024-03-20 15:07:
46.602195 | 7.423 ms   |
|          |        ├─── parse                                  | 2024-03-20 15:07:
46.602201 | 0.137 ms   |
|          |        ├─── resolve                                | 2024-03-20 15:07:
46.602393 | 0.555 ms   |
|          |        ├─── rewrite                                | 2024-03-20 15:07:
46.603104 | 1.055 ms   |
|          |        ├─── optimize                               | 2024-03-20 15:07:
46.604194 | 4.298 ms   |
|          |        |    ├─── inner_execute_read                  | 2024-03-20 15:07:
46.605959 | 0.825 ms   |
|          |        |    |    ├─── sql_compile                      | 2024-03-20 15:07
:46.606078 | 0.321 ms   |
|          |        |    |    |    └─── pc_get_plan                   | 2024-03-20 15:0
7:46.606124 | 0.147 ms   |
|          |        |    |    ├─── open                             | 2024-03-20 15:07
:46.606418 | 0.129 ms   |
|          |        |    |    └─── do_local_das_task                | 2024-03-20 15:07
:46.606606 | 0.095 ms   |
|          |        |    └─── close                               | 2024-03-20 15:07:
46.606813 | 0.240 ms   |
|          |        |        ├─── close_das_task                  | 2024-03-20 15:07:
46.606879 | 0.022 ms   |
|          |        |        └─── end_transaction                 | 2024-03-20 15:07:
46.607009 | 0.023 ms   |
|          |        ├─── code_generate                          | 2024-03-20 15:07:
46.608527 | 0.374 ms   |
|          |        └─── pc_add_plan                            | 2024-03-20 15:07:
46.609375 | 0.207 ms   |
|          └─── sql_execute                                    | 2024-03-20 15:07:4
6.609677 | 0.832 ms   |
|              ├─── open                                       | 2024-03-20 15:07:4
6.609684 | 0.156 ms   |
|              ├─── response_result                            | 2024-03-20 15:07:4
6.609875 | 0.327 ms   |
|              |    └─── do_local_das_task                      | 2024-03-20 15:07:
46.609905 | 0.136 ms   |
|              └─── close                                      | 2024-03-20 15:07:4
6.610221 | 0.225 ms   |
|                  ├─── close_das_task                         | 2024-03-20 15:07:4
6.610229 | 0.029 ms   |
|                  └─── end_transaction                        | 2024-03-20 15:07:4
6.610410 | 0.019 ms   |
+------------------------------------------------------------+--------------------
-------+------------+
29 rows in set
```

```

```

从上面这条 show trace 的结果里，大家可以分析出一些信息：

- 这条 SQL 一共花了 191.999 ms。

- SQL 的整体的耗时主要在 `ob_proxy_partition_location_lookup` 上，占了 181.839 ms，顾名思义，这一部分耗时是 ODP（即 proxy） 在寻找表 `t1` 的主副本的位置信息。但是因为这张表刚刚创建，ODP 里暂时还没有对应的位置信息的缓存（也就是 location cache），所以第一次花的时间会比较久，不过之后 ODP 里就会缓存下这张表的 location cache 信息。

- SQL 转发到合适的 OBServer 节点之后，在 `com_query_process` 的耗时 8.824 ms。

5. 这条 SQL 一共花了 191.999 ms。
6. SQL 的整体的耗时主要在 `ob_proxy_partition_location_lookup` 上，占了 181.839 ms，顾名思义，这一部分耗时是 ODP（即 proxy） 在寻找表 `t1` 的主副本的位置信息。但是因为这张表刚刚创建，ODP 里暂时还没有对应的位置信息的缓存（也就是 location cache），所以第一次花的时间会比较久，不过之后 ODP 里就会缓存下这张表的 location cache 信息。
7. SQL 转发到合适的 OBServer 节点之后，在 `com_query_process` 的耗时 8.824 ms。
8. 在同一个 session 中再重新执行一遍 `SELECT c1 FROM t1 LIMIT 2;` 这条 SQL，执行 show trace 语句查看有什么变化。

```
SHOW TRACE;
```

输出如下：

```
+-----------------------------------------------+-----------------------------+----
--------+
| Operation                                     | StartTime                   | Ela
pseTime |
+-----------------------------------------------+-----------------------------+----
--------+
| ob_proxy                                      | 2024-03-20 15:34:14.879559 | 7.3
90 ms    |
```

```
| ├─── ob_proxy_partition_location_lookup        | 2024-03-20 15:34:14.879652 | 4
.691 ms   |
| ├─── ob_proxy_server_process_req               | 2024-03-20 15:34:14.884785 | 1
.514 ms   |
| └─── com_query_process                         | 2024-03-20 15:34:14.884943 | 1
.237 ms   |
|     └─── mpquery_single_stmt                   | 2024-03-20 15:34:14.884959 | 1
.207 ms   |
|         ├─── sql_compile                       | 2024-03-20 15:34:14.884997 | 0
.279 ms   |
|         |    └─── pc_get_plan                   | 2024-03-20 15:34:14.885042 |
0.071 ms   |
|         └─── sql_execute                       | 2024-03-20 15:34:14.885300 | 0
.809 ms   |
|             ├─── open                          | 2024-03-20 15:34:14.885310 | 0
.139 ms   |
|             ├─── response_result               | 2024-03-20 15:34:14.885513 | 0
.314 ms   |
|             |    └─── do_local_das_task         | 2024-03-20 15:34:14.885548 |
0.114 ms   |
|             └─── close                         | 2024-03-20 15:34:14.885847 | 0
.190 ms   |
|                 ├─── close_das_task            | 2024-03-20 15:34:14.885856 | 0
.030 ms   |
|                 └─── end_transaction           | 2024-03-20 15:34:14.885997 | 0
.019 ms   |
+-------------------------------------------------+----------------------------+----
--------+
14 rows in set
```

从上面这条 show trace 的结果里，大家可以分析出一些信息：

- 第二次执行这条 SQL，时间变快了很多，从 191.999 ms 缩短到了 7.390 ms。

- ODP 有了 location cache 信息，所以寻找路由信息和转发都变快了，从之前的 181.839 ms 缩减到了 4.691 ms。

- SQL 转发到合适的 OBServer 节点之后，在 `com_query_process` 的耗时从 8.824 ms 缩减到了 1.237 ms。这里多啰嗦两句，大家仔细看一下就会发现，SQL 分为编译阶段和执行阶段，编译阶段是优化器生成执行计划，执行阶段是执行引擎根据执行计划计算结果。第二次执行时，编译阶段的流程变短了，速度也变快了，大概是因为编译阶段在最开始的时候，先会去计划缓存里查询下有没有之前已经生成好并且缓存下来的计划（即

上面的 `pc_get_plan`），如果有的话，就不需要重新生成计划了，所以省了一个完整解析 SQL 并生成计划（即上面的 `hard_parse`）的开销。

9. 第二次执行这条 SQL，时间变快了很多，从 191.999 ms 缩短到了 7.390 ms。
10. ODP 有了 location cache 信息，所以寻找路由信息和转发都变快了，从之前的 181.839 ms 缩减到了 4.691 ms。
11. SQL 转发到合适的 OBServer 节点之后，在 `com_query_process` 的耗时从 8.824 ms 缩减到了 1.237 ms。这里多啰嗦两句，大家仔细看一下就会发现，SQL 分为编译阶段和执行阶段，编译阶段是优化器生成执行计划，执行阶段是执行引擎根据执行计划计算结果。第二次执行时，编译阶段的流程变短了，速度也变快了，大概是因为编译阶段在最开始的时候，先会去计划缓存里查询下有没有之前已经生成好并且缓存下来的计划（即上面的 `pc_get_plan`），如果有的话，就不需要重新生成计划了，所以省了一个完整解析 SQL 并生成计划（即上面的 `hard_parse`）的开销。
12. 再通过直连 OBServer 节点登录 OceanBase 数据库，再重新执行一遍 `SELECT c1 FROM t1 LIMIT 2;` 这条 SQL，执行 show trace 语句查看有什么变化。

```
SHOW TRACE;
```

输出如下：

```
+-------------------------------------------+----------------------------+--------
----+
| Operation                                 | StartTime                  | ElapseT
ime |
+-------------------------------------------+----------------------------+--------
----+
| com_query_process                         | 2024-03-20 15:54:38.772699 | 1.746 m
s   |
| └──── mpquery_single_stmt                   | 2024-03-20 15:54:38.772771 | 1.64
7 ms    |
|       ├──── sql_compile                     | 2024-03-20 15:54:38.772835 | 0.35
6 ms    |
|       |     └──── pc_get_plan                | 2024-03-20 15:54:38.772900 | 0.14
3 ms    |
|       └──── sql_execute                     | 2024-03-20 15:54:38.773209 | 1.05
2 ms    |
|             ├──── open                      | 2024-03-20 15:54:38.773232 | 0.15
0 ms    |
```

```
|          ├─── response_result               | 2024-03-20 15:54:38.773413 | 0.42
1 ms    |
|          |    └─── do_local_das_task        | 2024-03-20 15:54:38.773479 | 0.19
2 ms    |
|          └─── close                         | 2024-03-20 15:54:38.773857 | 0.37
9 ms    |
|               ├─── close_das_task           | 2024-03-20 15:54:38.773913 | 0.06
9 ms    |
|               └─── end_transaction          | 2024-03-20 15:54:38.774139 | 0.05
8 ms    |
+--------------------------------------------+----------------------------+--------
----+
11 rows in set
```

从上面这条 show trace 的结果里，大家可以分析出一些信息：

- 直连 OBServer 节点，没了 ODP，所以也就没了和 proxy 相关的开销。
- 这条 SQL 的计划，之前已经进了这台 OBServer 节点的计划缓存里，所以在多次执行时，编译阶段都不需要再走一遍完整的硬解析流程。

13. 直连 OBServer 节点，没了 ODP，所以也就没了和 proxy 相关的开销。
14. 这条 SQL 的计划，之前已经进了这台 OBServer 节点的计划缓存里，所以在多次执行时，编译阶段都不需要再走一遍完整的硬解析流程。

最后简单总结一下：对于性能不符合预期的一条 SQL，通过全链路追踪，可以看到耗时在什么阶段：是耗时在 ODP（proxy）转发阶段？还是耗时在计划生成阶段？亦或是耗时在执行阶段？在各个阶段中，我们还可以看到更加细节的耗时数据（上面那条 SQL 实在太简单，又利用到了计划缓存，所以看上去细节不多），然后我们就可以针对具体慢的地方做进一步的分析。

如果慢在 ODP 转发阶段，可以怀疑是不是 ODP 和 OBServer 节点之间的网络出现故障了，或者是不是 ODP 还没缓存位置信息等问题。ODP 的性能问题分析详见官网《OceanBase 数据库代理》文档 运维/性能分析，大家按需了解即可，这里不再赘述。

如果慢在 SQL 执行阶段，就可以怀疑是不是创建的索引不够优等问题，需要通过在前两节学习到的内容来分析执行计划，并进行相应的 SQL 调优。

sql audit

`GV$OB_SQL_AUDIT` 是最常用的 SQL 监控视图，能够记录每一次 SQL 请求的来源、执行状态、资源消耗及等待事件，除此之外还记录了 SQL 文本、执行计划等关键信息。该视图是诊断 SQL 问题的利器，视图介绍详见官网《OceanBase 数据库》文档 管理数据库/监控指标/监控指标/SQL 监控/SQL Audit。

在线上如果出现 RT 抖动，但 RT 并不是持续很高的情况，可以考虑在抖动出现后，通过执行 `alter system set ob_enable_sql_audit = 0;` 将 sql audit 关闭，从而确保该抖动的 SQL 请求的信息在 sql audit 中不被淘汰。

然后可以通过 sql audit 查看某段时间内执行时间 TOP N 的请求，分析异常的 SQL。

```
-- 查看某段时间内执行时间 TOP N 的请求
select /*+ parallel(15) */ sql_id, elapsed_time, trace_id, substr(query_sql, 1, 6
) -- 这里为了展示方便，对 query_sql 做了截断
from oceanbase.gv$ob_sql_audit
where tenant_id = 1
      and IS_EXECUTOR_RPC = 0
      and request_time > (time_to_usec(now()) - 10000000)
      and request_time < time_to_usec(now())
order by elapsed_time desc
limit 10;
```

输出如下：

```
+----------------------------------+--------------+------------------------------
-----+-------------------------+
| sql_id                           | elapsed_time | trace_i
d                              | substr(query_sql, 1, 6) |
+----------------------------------+--------------+------------------------------
-----+-------------------------+
| 3BD6E04969DEE524A788E629AFD1D202 |       124223 | Y584A0BA1CC5A-0006141213C0C8FD
-0-0 | select                  |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        19367 | Y584E0BA1CC5A-00061412076765AD
-0-0 | select                  |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        18789 | Y58480BA1CC5B-00061412075A8FD9
-0-0 | select                  |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        18307 | Y584E0BA1CC5A-00061412076765AE
-0-0 | select                  |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        17801 | Y584E0BA1CC5A-00061412076765AF
-0-0 | select                  |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        17305 | Y58480BA1CC5B-00061412075A8FD8
-0-0 | select                  |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        16928 | Y58480BA1CC5B-00061412075A8FD7
-0-0 | select                  |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        12575 | Y584A0BA1CC5A-00061412076125CC
-0-0 | select                  |
```

```
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        11960 | Y584A0BA1CC5A-00061412076125CB
-0-0 | select                    |
| B6F2A13C4C81145FFAA2F2A5CF9587A9 |        11869 | Y584A0BA1CC5A-00061412076125CA
-0-0 | select                    |
+-----------------------------------+---------------+-------------------------------
-----+--------------------------+
10 rows in set
```

在 sql audit 记录中分析异常 SQL 的思路一般是：

- 通过 `GET_PLAN_TIME` 查看获取执行计划的时间，如果很长，一般会伴随 `IS_HIT_PLAN = 0`，表示没有命中计划缓存（plan cache），导致走了完整的硬解析流程。
- 通过 `SSSTORE_READ_ROW_COUNT` 和 `MEMSTORE_READ_ROW_COUNT` 这两个字段，分析是否可能由于 Buffer 表问题导致的 SQL 性能问题。Buffer 表问题的分析方法详见本节的 Buffer 表问题 部分。
- 通过 `QUEUE_TIME` 查看 queue time 是否很长，如果 `QUEUE_TIME` 较大，说明租户的工作线程出现了等待问题。等待问题的分析方法详见本节的 等待问题 部分。
- 通过 `RETRY_CNT` 字段查看 retry 次数是否很多，如果次数很多，则可以进一步对是否有锁冲突或切主等情况进行排查。
- 通过 `EXECUTE_TIME` 字段的值，可以结合 GV$OB_PLAN_CACHE_PLAN_EXPLAIN 字典视图分析执行时的计划是否合理。如果不合理，则需要进一步考虑是否需要创建合适的索引、是否需要通过 hint 调整连接顺序和连接算法等等，具体的 SQL 调优方式详见 常见的 SQL 调优方式 的内容。

```
select c1 from t1;
+-------+
| c1    |
+-------+
|   123 |
+-------+
1 row in set

select last_trace_id();
+-------------------------------------+
| last_trace_id()                     |
+-------------------------------------+
| Y584A0B9E1F14-0006104BC96894F3-0-0 |
+-------------------------------------+
1 row in set
```

```
select QUERY_SQL, PLAN_ID, RETURN_ROWS, NET_WAIT_TIME, QUEUE_TIME, GET_PLAN_TIME,
EXECUTE_TIME
  from  oceanbase.GV$OB_SQL_AUDIT
  where trace_id = 'Y584A0B9E1F14-0006104BC96894F3-0-0';
+-------------------+---------+-------------+---------------+------------+--------
-------+------
| QUERY_SQL         | PLAN_ID | RETURN_ROWS | NET_WAIT_TIME | QUEUE_TIME | GET_PLA
N_TIME | EXECU
+-------------------+---------+-------------+---------------+------------+--------
-------+------
| select c1 from t1 |   53777 |           1 |            10 |         46
|          234 |
+-------------------+---------+-------------+---------------+------------+--------
-------+------
1 row in set
```

说明

如果在 sql audit 中 RT 抖动的请求已被淘汰，则需要查看抖动时间点是否有慢查询的 trace 日志，如果有，则可以转为分析 trace 日志。

sql plan monitor（高阶）

sql plan monitor 是分析并行度高且计划复杂时 SQL 性能问题的工具，集成于 obdiag 工具中，用于监控 SQL 执行算子实时执行情况，以下三种场景会将 SQL 纳入实时监控：

- 当 SQL 加上了`/*+ monitor*/` 的 hint 时。
- 当 SQL 是分布式执行且 `parallel > 1` 时。
- 当 SQL 的执行时间超过 1 s 时。

这个工具相比 show trace 和 sql audit 的上手难度会稍高一些，需要使用者能够初步阅读 SQL 的执行计划。通过 obdiag 收集 sql plan monitor 信息之后，可以直观地看出具体慢在哪个算子上，还可以直观地看出这个算子（在开并行时）的各个线程的任务划分是否均匀，或者说是否因为数据全都倾斜到了其中某几个线程而导致的性能问题等。

下面举一个简单的例子，先进行一个并行度为三的聚合计算（注意：如果没有并行，需要加一个 `/*+monitor*/` 的 hint 才能收集 sql plan monitor 信息）。

```
select /*+parallel(3) */ count(c1) from t1;
+------------+
```

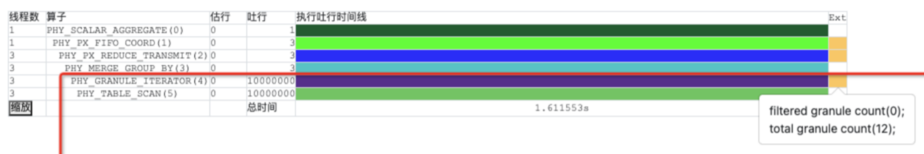
```
| COUNT(C1) |
+-----------+
| 110000000 |
+-----------+
1 row in set

select last_trace_id() from dual;
+-------------------------------------+
| LAST_TRACE_ID()                     |
+-------------------------------------+
| Y58480B7C052B-0005EB5FF9C060FC-0-0 |
+-------------------------------------+
1 row in set
```

拿着 trace id 收集下 sql plan monitor 信息。

```
[admin@test001 .obdiag]$ obdiag gather plan_monitor --trace_id Y58480B7C052B-0005E
B5FF9C060FC-0-0
```

收集的信息是一个 html 文件，直接点开就能看到详细结果。

因为上面那条 SQL 简单，所以计划也简单，就是两阶段算 count，第一阶段三号算子用三个线程各算各的 count，第二阶段把各个线程的 count 结果汇总到零号算子再算一次 count 的 sum。下面这张 sql plan monitor 的图上，每个算子用了几个线程，吐了多少行，都看的很清楚，不再细说。



要额外多说一句的是，看 sql plan monitor 里面 `PHY_GRANULE_ITERATOR` 后面的黄色块的 `granule` 数量。有多个 granule 时，先是三个线程一人一个，然后线程做完了自己手头的这个，就会去抢占池子里剩下的。这里把鼠标放在黄色块上就能看到，一共被分成了 12 个 granule。

如果暂时还不了解 granule 的概念，那就请到官网《OceanBase 数据库》文档 参考指南/性能调优/SQL 调优指南/并行执行计划/并行执行简介 里全局搜索一下 granule 关键字。

通过下图里 rescan 列的信息，可以看到三个线程分别处理了 1 + 2、1 + 3、1 + 4 个 granule 的数据。1 就是各线程先分到的一个，2、3、4 就它们后来又在池子里抢占到的。

SQL_PLAN_MONITOR 详情

算子优先视图

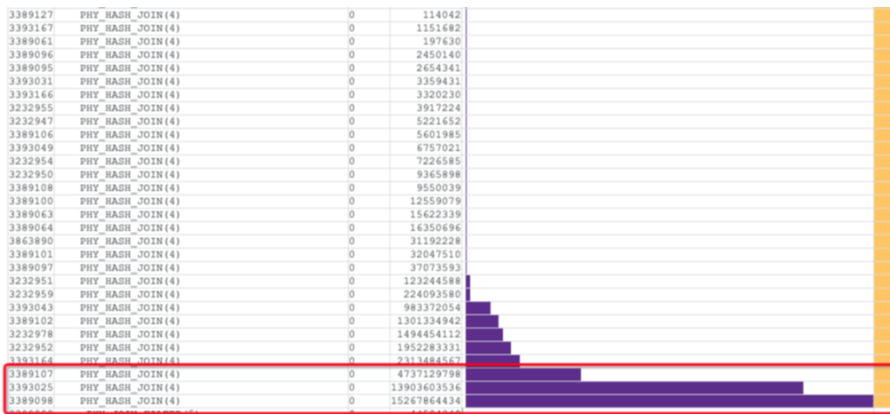
线程ID	算子	RESCAN	吐行	执行吐行时间线	Ext
78412	PHY_SCALAR_AGGREGATE(0)	0	1		
78412	PHY_PX_FIFO_COORD(1)	0	3		
80639	PHY_PX_REDUCE_TRANSMIT(2)	0	1		
80640	PHY_PX_REDUCE_TRANSMIT(2)	0	1		
80641	PHY_PX_REDUCE_TRANSMIT(2)	0	1		
80639	PHY_MERGE_GROUP_BY(3)	0	1		
80640	PHY_MERGE_GROUP_BY(3)	0	1		
80641	PHY_MERGE_GROUP_BY(3)	0	1		
80639	PHY_GRANULE_ITERATOR(4)	0	2980383		
80640	PHY_GRANULE_ITERATOR(4)	0	3308157		
80641	PHY_GRANULE_ITERATOR(4)	0	3711460		
80639	PHY_TABLE_SCAN(5)	2	2980383		
80640	PHY_TABLE_SCAN(5)	3	3308157		
80641	PHY_TABLE_SCAN(5)	4	3711460		
缩放			总时间	1.607305s	

线程优先视图

线程ID	算子	RESCAN	吐行	执行吐行时间线	Ext
78412	PHY_SCALAR_AGGREGATE(0)	0	1		
78412	PHY_PX_FIFO_COORD(1)	0	3		
80639	PHY_PX_REDUCE_TRANSMIT(2)	0	1		
80639	PHY_MERGE_GROUP_BY(3)	0	1		
80639	PHY_GRANULE_ITERATOR(4)	0	2980383		
80639	PHY_TABLE_SCAN(5)	2	2980383		
80640	PHY_PX_REDUCE_TRANSMIT(2)	0	1		
80640	PHY_MERGE_GROUP_BY(3)	0	1		
80640	PHY_GRANULE_ITERATOR(4)	0	3308157		
80640	PHY_TABLE_SCAN(5)	3	3308157		
80641	PHY_PX_REDUCE_TRANSMIT(2)	0	1		
80641	PHY_MERGE_GROUP_BY(3)	0	1		
80641	PHY_GRANULE_ITERATOR(4)	0	3711460		
80641	PHY_TABLE_SCAN(5)	4	3711460		
缩放			总时间	1.607305s	

这个例子里因为数据量实在太小，所以 granule 的分配勉强还算是均衡的。

如果你在 sql plan monitor 里看到数据量超大，分配了很多线程，但是只有一小部分线程在工作，大部分线程都在偷懒，例如下图的 4 号算子 hash join。不用犹豫，可以直接拿着这个 sql plan monitor 的结果去社区论坛的问答区发帖，找技术支持同学分析为什么会产生严重的数据倾斜问题了。



3389127	PHY_HASH_JOIN(4)	0	114042
3393167	PHY_HASH_JOIN(4)	0	1151682
3389061	PHY_HASH_JOIN(4)	0	197630
3389096	PHY_HASH_JOIN(4)	0	2450140
3389095	PHY_HASH_JOIN(4)	0	2654341
3393031	PHY_HASH_JOIN(4)	0	3359431
3393166	PHY_HASH_JOIN(4)	0	3320230
3232955	PHY_HASH_JOIN(4)	0	3917224
3232947	PHY_HASH_JOIN(4)	0	5221652
3389106	PHY_HASH_JOIN(4)	0	5601985
3393049	PHY_HASH_JOIN(4)	0	6757021
3232954	PHY_HASH_JOIN(4)	0	7226585
3232950	PHY_HASH_JOIN(4)	0	9365898
3389108	PHY_HASH_JOIN(4)	0	9550039
3389100	PHY_HASH_JOIN(4)	0	12559079
3389063	PHY_HASH_JOIN(4)	0	15622339
3389064	PHY_HASH_JOIN(4)	0	16350696
3863890	PHY_HASH_JOIN(4)	0	31192228
3389101	PHY_HASH_JOIN(4)	0	32047510
3389097	PHY_HASH_JOIN(4)	0	37073593
3232951	PHY_HASH_JOIN(4)	0	123244588
3232959	PHY_HASH_JOIN(4)	0	224093580
3393043	PHY_HASH_JOIN(4)	0	983372054
3389102	PHY_HASH_JOIN(4)	0	1301334942
3232978	PHY_HASH_JOIN(4)	0	1494454112
3232952	PHY_HASH_JOIN(4)	0	1952283331
3393164	PHY_HASH_JOIN(4)	0	2313484567
3389107	PHY_HASH_JOIN(4)	0	4737129798
3393025	PHY_HASH_JOIN(4)	0	13903603536
3389098	PHY_HASH_JOIN(4)	0	15267864434
[illegible]	PHY_JOIN_FILTER(5)	0	[illegible]

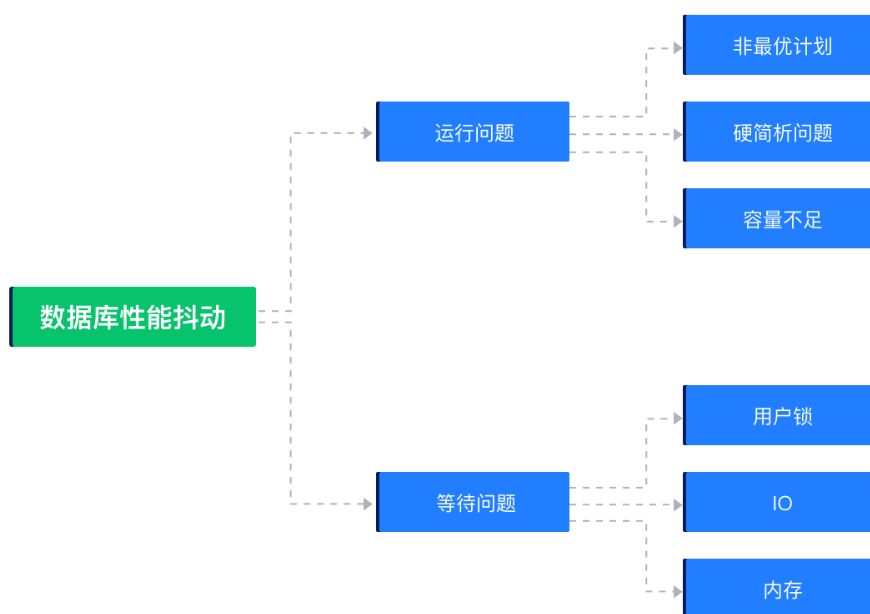
OBServer SQL 性能问题的典型场景

当 OBServer 端出现性能瓶颈的时候，我们首先需要去识别瓶颈点在哪里，也就是工作线程当前处于阻塞态，还是长时间运行态。前者表示正在等待某个系统对象，后者表示正在扫描大量数据，或者是资源不足。不同的瓶颈点意味着不同的根因和解决方案。

OceanBase 数据库提供了丰富的内部表和系统日志用来定位性能瓶颈，主要通过 sql_audit 分析 OBServer 行为。

- 运行态问题：一般表示与执行计划相关，可能是优化器生成了非最优的执行计划导致 SQL 请求需要扫描大量数据，可能是计划缓存命中率不高导致 SQL 执行时花费额外的编译时间。除了执行计划问题，也可能是运行期的资源不足。运行态问题往往可以通过调整执行计划或者增加资源来解决。
- 阻塞态问题：数据库中的工作线程，要么在进行 CPU 计算，要么就在进行等待，比如等待 I/O、网络、临界区等，我们统称为阻塞态。阻塞态意味着系统的并发瓶颈，无法通过增加资源的手段去提升吞吐量。阻塞态问题首先要识别阻塞点，并针对性优化。

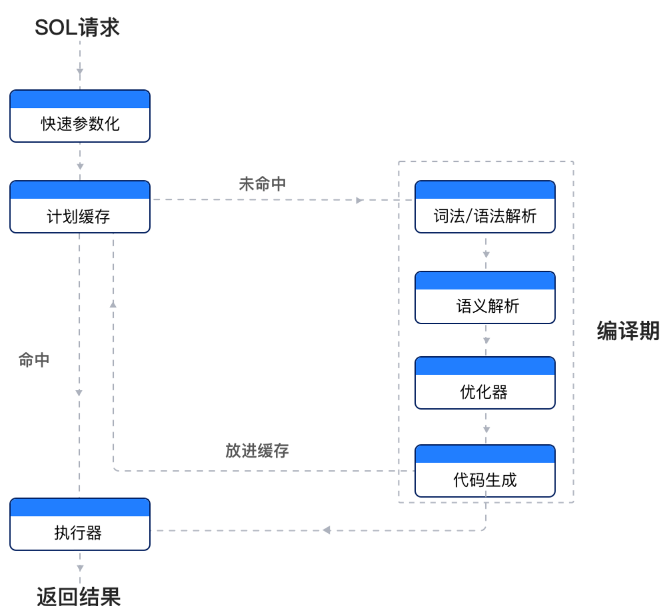
通过以上分析，OBServer 端的常见问题主要分为以下几类：



执行问题

执行时计划问题

SQL 请求的执行过程分为编译期和执行期。编译期包括词法解析、语法解析、语法优化、代码生成等子阶段，最终生成执行计划，提交执行器执行并返回结果。解析 SQL 请求并生成执行计划的过程，称为硬解析。硬解析需要消耗一定的资源并影响 SQL 耗时，为避免每次请求时的硬解析消耗，执行计划会被放进计划缓存，后续同样的 SQL 请求到来时优先判断是否命中计划缓存，命中则直接提取缓存中的计划提交执行器执行并返回结果，称为软解析。因此 SQL 请求是否命中计划缓存，以及计划是否最优，是性能优化的关键。



优化器可能由于各种情况生成非最优执行计划，如索引缺失、统计信息落后、Buffer 表、大小账号等原因。非最优执行计划执行时会产生较多的逻辑读，扫描较多的数据量从而导致耗时增加和吞吐量下降。

Buffer 表问题

Buffer 表表示频繁插入删除的表，这里的“表”也包括索引表（主表上更新索引列，体现在索引表上就是删除和插入动作）。

Buffer 表问题是由于 OceanBase 数据库基于 LSM-Tree 机制实现的存储引擎而引入的：LSM-Tree 架构下被删除的数据是标记删除，在合并前不会物理生效。当增量数据中积累了大量标记删除的数据时，从上层应用视角实际存在的行很少，但范围查询时可能需要处理较多的标记删除的数据，从而导致 SQL 耗时不够理想。同时 Buffer 表场景下也容易导致优化器生成非最优执行计

划。

Buffer 表问题是一种特殊业务场景下才会触发的一种 SQL 异常：执行 INSERT 后，大部分数据很快就会被 DELETE，即表存量数据很小。一般情况下，因为 OceanBase 数据库会对表的数据块“空洞”做回收，所以问题不大。但是如果短时间内 INSERT 和 DELETE 的数据量级非常大，且表的数据块因为各种原因没有能够及时回收、或者 INSERT 量级大于 DELETE 量级导致数据积压，导致真实扫描的表数据块数据较多，SQL 性能下降。

下面举一个简单的例子：插入 1000 行数据，隔行删除，共删除五百行，再去扫描这张表，通过计划里的 `physical_range_rows` 和 `logical_range_rows` 来看实际扫描的物理行数和表中需要扫描的逻辑行数。

> **说明**
>
> 在 SQL 执行完成之后，还可以通过 sql audit 中的 SSSTORE_READ_ROW_COUNT 和 MEMSTORE_READ_ROW_COUNT 字段来看对应 SQL 的物理行数和逻辑行数。

1. 创建一张表，用于进行 Buffer 表的测试

   ```
   create table t1(c1 int);
   ```

2. 插入 1000 行数据

   ```
   insert into t1 with recursive cte(n) as (select 1 from dual union all select n +
   1 from cte where n < 1000) select n from cte;
   ```

3. 隔行删除，共删除五百行

   ```
   delete from t1 where c1 % 2 = 0;
   ```

4. 查看执行计划

   ```
   explain extended_noaddr select * from t1;
   ```

   输出如下，关注 `physical_range_rows` 和 `logical_range_rows` 的信息。

   ```
   +--------------------------------------------------------------------+
   | Query Plan                                                         |
   +--------------------------------------------------------------------+
   ```

```
| ===============================================                   |
| |ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)|                   |
| -----------------------------------------------                   |
| |0 |TABLE FULL SCAN|t1  |501     |38          |                   |
| ===============================================                   |
| Outputs & filters:                                                |
| -------------------------------------                             |
|   0 - output([t1.c1]), filter(nil), rowset=256                    |
|       access([t1.c1]), partitions(p0)                             |
|       is_index_back=false, is_global_index=false,                 |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always true |
| ......                                                            |
| Optimization Info:                                                |
| -------------------------------------                             |
|   t1:                                                             |
|       table_rows:501                                              |
|       physical_range_rows:1000                                    |
|       logical_range_rows:500                                      |
|       index_back_rows:0                                           |
|       output_rows:501                                             |
|       table_dop:1                                                 |
|       dop_method:Table DOP                                        |
|       avaiable_index_name:[t1]                                    |
|       stats version:0                                             |
|       dynamic sampling level:0                                    |
|   Plan Type:                                                      |
|       LOCAL                                                       |
|   Note:                                                           |
|       Degree of Parallelisim is 1 because of table property       |
+-------------------------------------------------------------------+
45 rows in set
```

本教程 阅读和管理 OceanBase 数据库 SQL 执行计划 中介绍过 `physical_range_rows` 和 `logical_range_rows` 这两个指标的含义，需要扫描的逻辑行 `logical_range_rows` 只有 500 行，因为 LSM tree 在内存里是标记删除，且没有经过合并，所以真实扫描的物理行 `physical_range_rows` 依然会是 1000 行。

所以我们可以手动触发一次合并，再来看合并之后的 `physical_range_rows` 和 `logical_range_rows` 信息。

5. 手动触发合并

```
ALTER SYSTEM MAJOR FREEZE;
```

触发合并的详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统管理/存储管理/

合并管理/手动触发合并。

6. 查看合并进度

```
SELECT START_TIME, LAST_FINISH_TIME, STATUS FROM oceanbase.DBA_OB_MAJOR_COMPACTION
;
```

输出如下，详细介绍可参见官网《OceanBase 数据库》文档 参考指南/系统管理/存储管理/合并管理/查看合并过程。

```
+----------------------------+----------------------------+--------+
| START_TIME                 | LAST_FINISH_TIME           | STATUS |
+----------------------------+----------------------------+--------+
| 2024-03-20 17:54:18.610008 | 2024-03-20 17:54:50.738156 | IDLE   |
+----------------------------+----------------------------+--------+
1 row in set
```

7. 再次查看执行计划

```
explain extended_noaddr select * from t1;
```

输出如下，关注合并之后的 `physical_range_rows` 和 `logical_range_rows` 的信息。

```
+--------------------------------------------------------------------+
| Query Plan                                                         |
+--------------------------------------------------------------------+
| ===============================================                    |
| |ID|OPERATOR       |NAME|EST.ROWS|EST.TIME(us)|                    |
| -----------------------------------------------                    |
| |0 |TABLE FULL SCAN|t1  |501     |21          |                    |
| ===============================================                    |
| Outputs & filters:                                                 |
| -------------------------------------                              |
|   0 - output([t1.c1]), filter(nil), rowset=256                     |
|       access([t1.c1]), partitions(p0)                              |
|       is_index_back=false, is_global_index=false,                  |
|       range_key([t1.__pk_increment]), range(MIN ; MAX)always true |
| ......                                                             |
| Optimization Info:                                                 |
| -------------------------------------                              |
|   t1:                                                              |
|       table_rows:501                                               |
|       physical_range_rows:500                                      |
|       logical_range_rows:500                                       |
|       index_back_rows:0                                            |
|       output_rows:501                                              |
```

```
|        table_dop:1                                                   |
|        dop_method:Table DOP                                          |
|        avaiable_index_name:[t1]                                      |
|        stats version:0                                               |
|        dynamic sampling level:0                                      |
|    Plan Type:                                                        |
|        LOCAL                                                         |
|    Note:                                                             |
|        Degree of Parallelisim is 1 because of table property         |
+----------------------------------------------------------------------+
45 rows in set
```

因为刚刚手动触发的合并，会将标记删除的数据进行物理意义上的删除，所以没有了空洞，`physical_range_rows` 和 `logical_range_rows` 的行数也相等了。

最后总结一下：大家可以通过分析计划中 `physical_range_rows` 和 `logical_range_rows` 的数值是否差距巨大来确定是否存在 Buffer 表问题，可以通过手动触发合并规避这个问题。当然，如果担心合并时间过长，也可以分析下是否能够通过 Hint 和 Outline 来改出更优的计划。

计划缓存的 bad case（大小账号问题）

计划缓存的 bad case 是优化器生成的执行计划对于当前的数据集是最优的，该计划被放进了计划缓存中。但对于随后的数据集并不是最优，进而导致随后命中相同计划缓存的 SQL 请求耗时异常。

这个问题的示例和解决方法请详见本教程 常见的 SQL 调优方式 中 **计划缓存的 bad case** 部分。

硬解析问题

频繁对 SQL 进行硬解析，往往由于计划缓存的命中率不高，每次 SQL 请求时都需要经历一次完整的 SQL 编译过程，导致 SQL 请求耗时增加。可以通过 `GV$OB_SQL_AUDIT` 视图中的 `GET_PLAN_TIME` 字段判断获取执行计划的耗时是否异常。正常情况下该阶段的耗时小于 0.1 ms，异常请求下该字段的耗时可超过 100 ms。

这个问题一般是由于计划缓存空间设置过小，导致计划被频繁淘汰。可以通过调大计划缓存的大小系统变量 `ob_plan_cache_percentage` 用于设置计划缓存可使用的内存所占租户内存的百分比。计划缓存最多可使用内存（内存上限绝对值）= 租户内存上限 * ob_plan_cache_percentage / 100，默认值为 `5`。

```
show variables like 'ob_plan_cache_percentage';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| ob_plan_cache_percentage | 5     |
+--------------------------+-------+
1 row in set (0.010 sec)

-- 修改租户系统变量 ob_plan_cache_percentage
set global ob_plan_cache_percentage = 10;

show variables like 'ob_plan_cache_percentage';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| ob_plan_cache_percentage | 10    |
+--------------------------+-------+
```

修改租户系统变量的方法详见官网《OceanBase 数据库》文档 管理数据库/租户管理/租户常见操作/修改租户系统变量。

执行时容量问题

当出现性能问题时，如果能够排除执行计划不优的原因，那么大概率就是容量问题。

非最优计划问题，往往会在 `GV$OB_SQL_AUDIT` 视图中看到明显的逻辑读和 `EXECUTE_TIME` 耗时增加的 SQL，且 `EXECUTE_TIME` 部分主要为 CPU 时间的增加。

通过 `GV$OB_SQL_AUDIT` 视图判断是否为容量问题的标志一般是：`QUEUE_TIME` 明显增加，且不存在明显的逻辑读和 `EXECUTE_TIME` 耗时增加的 SQL。

容量问题可能发生于以下场景：

- 应用流量增加。如果 TopSQL 的执行计划未发生明显变化，但是租户 CPU 使用率增加，伴随 SQL 执行次数增加，那么大概率是应用流量增加引起的性能问题。
- 应用负载发生变化。区别于应用流量增加，应用的工作负载发生变化，例如大请求的比例增加。
- 基础设施层发生计算资源争抢。

对于该问题，可以通过以下方式解决：

- 对特定 SQL 进行限流，限流方法详见官网《OceanBase 数据库》文档 参考指南/告警参考/

附录/限制 OceanBase 集群的流量。

- 调大租户 CPU 规格。
- 调整集群参数。集群参数 `cpu_quota_concurrency` 用于控制租户 CPU 提供的工作线程数，租户最小工作线程数 = cpu_quota_concurrency * MIN_CPU。当租户出现容量问题时，如果物理机维度的 CPU 负载不高，说明多数工作线程并没有实际使用 CPU 资源，可以调整该参数以进一步压榨物理机的计算资源（该参数值不能过大，过大会导致频繁上下文切换，及线程频繁创建销毁引发系统问题）。`cpu_quota_concurrency` 的介绍详见官网《OceanBase 数据库》文档 参考指南/配置项和系统变量/配置项/租户级别配置项/cpu_quota_concurrency。

1. 查看 `cpu_quota_concurrency` 参数的值输出如下：

```
+-------+----------+-------------+---------+-----------------------+---
--------+-------+----------------------------------------------------------
+---------+-------+---------+------------------+
| zone  | svr_type | svr_ip      | svr_port | name                  | da
ta_type | value | info
| section | scope  | source  | edit_level       |
+-------+----------+-------------+---------+-----------------------+---
--------+-------+----------------------------------------------------------
+---------+-------+---------+------------------+
| zone1 | observer | 1.2.3.4     |    12345 | cpu_quota_concurrency | NU
LL      | 2     | max allowed concurrency for 1 CPU quota. Range: [1,20]
| TENANT  | TENANT | DEFAULT | DYNAMIC_EFFECTIVE |
+-------+----------+-------------+---------+-----------------------+---
--------+-------+----------------------------------------------------------
+---------+-------+---------+------------------+
1 row in set
```

2. 修改参数建议配置为 `4`，ARM 系统建议配置为 `2`。调小可能会影响性能，甚至可能导致稳定性风险；调大可能会增加内存占用，也可能导致 CPU 负载过高。谨慎调整。

```
ALTER SYSTEM SET cpu_quota_concurrency = '4';
```

3. 再次查看参数值输出如下：

```
+-------+----------+-------------+---------+-----------------------+---
--------+-------+----------------------------------------------------------
+---------+-------+---------+------------------+
| zone  | svr_type | svr_ip      | svr_port | name                  | da
ta_type | value | info
| section | scope  | source  | edit_level       |
```

```
+--------+----------+--------------+----------+-----------------------+---
--------+-------+----------------------------------------------------------
+---------+-------+---------+-------------------+
| zone1 | observer | 1.2.3.4      |    12345 | cpu_quota_concurrency | NU
LL       | 4      | max allowed concurrency for 1 CPU quota. Range: [1,20]
| TENANT   | TENANT | DEFAULT | DYNAMIC_EFFECTIVE |
+--------+----------+--------------+----------+-----------------------+---
--------+-------+----------------------------------------------------------
+---------+-------+---------+-------------------+
1 row in set
```

- 查看 `cpu_quota_concurrency` 参数的值

```
show parameters like 'cpu_quota_concurrency';
```

输出如下：

```
+-------+----------+--------------+----------+-----------------------+-----------+
-------+----------------------------------------------------------+---------+-------
-+---------+-------------------+
| zone  | svr_type | svr_ip       | svr_port | name                  | data_type
| value | info                                                     | section | scop
e  | source  | edit_level        |
+-------+----------+--------------+----------+-----------------------+-----------+
-------+----------------------------------------------------------+---------+-------
-+---------+-------------------+
| zone1 | observer | 1.2.3.4      |    12345 | cpu_quota_concurrency | NULL
| 2     | max allowed concurrency for 1 CPU quota. Range: [1,20] | TENANT  | TENAN
T | DEFAULT | DYNAMIC_EFFECTIVE |
+-------+----------+--------------+----------+-----------------------+-----------+
-------+----------------------------------------------------------+---------+-------
-+---------+-------------------+
1 row in set
```

- 修改参数

建议配置为 `4`，ARM 系统建议配置为 `2`。调小可能会影响性能，甚至可能导致稳定性风险；调大可能会增加内存占用，也可能导致 CPU 负载过高。谨慎调整。

```
ALTER SYSTEM SET cpu_quota_concurrency = '4';
```

- 再次查看参数值

```
show parameters like 'cpu_quota_concurrency';
```

输出如下：

```
+-------+----------+-------------+----------+----------------------+-----------
-------+---------------------------------------------------+--------+-------
-+---------+------------------+
| zone  | svr_type | svr_ip      | svr_port | name                 | data_type
| value | info                                              | section | scop
e  | source  | edit_level        |
+-------+----------+-------------+----------+----------------------+-----------
-------+---------------------------------------------------+--------+-------
-+---------+------------------+
| zone1 | observer | 1.2.3.4     |    12345 | cpu_quota_concurrency | NULL
| 4     | max allowed concurrency for 1 CPU quota. Range: [1,20] | TENANT  | TENAN
T | DEFAULT | DYNAMIC_EFFECTIVE |
+-------+----------+-------------+----------+----------------------+-----------
-------+---------------------------------------------------+--------+-------
-+---------+------------------+
1 row in set
```

说明

当出现容量问题时，一般采用扩容的方式解决（调大租户的 CPU 规格，或者是调大 cpu_quota_concurrency），工作线程增加后需要使用的内存资源也会增加，所以一般还需要同步调大租户的内存规格。

等待问题

当出现性能问题时，如果能够排除是执行时的问题（执行计划问题和执行期容量问题），那么大概率是内部等待问题。

内部等待问题表示工作线程正在等待某些资源，例如：Lock、Latch、I/O、内存等。

可以通过 GV$OB_SQL_AUDIT 视图判断是否为内部等待问题：例如存在明显 EXECUTE_TIME 部分耗时增加的 SQL，且 EXECUTE_TIME 部分主要为 TOTAL_WAIT_TIME_MICRO 的增加。TOTAL_WAIT_TIME_MICRO 由 APPLICATION_WAIT_TIME、CONCURRENCY_WAIT_TIME、USER_IO_WAIT_TIME、SCHEDULE_TIME 等几个部分组成，可用于进一步判断工作线程正在等待的资源类型。

在 GV$OB_SQL_AUDIT 视图中，记录了等待事件如下相关信息：

- 记录了 4 大类等待事件分别的耗时（APPLICATION_WAIT_TIME、CONCURRENCY_WAIT_TIME、

USER_IO_WAIT_TIME、SCHEDULE_TIME），每类等待事件都涉及很多种具体的等待事件。

- 记录了耗时最多的等待事件名称（EVENT）及该等待事件耗时（WAIT_TIME_MICRO）。
- 记录了所有等待事件的发生的次数（TOTAL_WAITS）及所有等待事件总耗时（TOTAL_WAIT_TIME_MICRO）。

一般情况下，如果等待事件总耗时较多，我们可以通过查看耗时最多的等待事件名称（EVENT）来初步定位具体在什么类型的等待上消耗了时间。如下面这条特定 trace_id 的 SQL，主要耗时就在 I/O 等待上，等待的是读索引上的数据。

```
select sql_id, elapsed_time, queue_time, get_plan_time, execute_time,
       application_wait_time, concurrency_wait_time, user_io_wait_time,
       schedule_time, event, wait_class, wait_time_micro, total_wait_time_micro
from oceanbase.gv$ob_sql_audit
where trace_id = 'YB420B84FE35-0005648A67211DC9'\G
```

输出如下：

```
*************************** 1. row ***************************
               sql_id: 5316DBF96556040831142D61BBD9014F
         elapsed_time: 953
           queue_time: 18
        get_plan_time: 58
         execute_time: 867
application_wait_time: 0
concurrency_wait_time: 0
    user_io_wait_time: 550
        schedule_time: 0
                event: db file data index read
           wait_class: USER_IO
      wait_time_micro: 352
total_wait_time_micro: 550
```

然后就可以再通过 sql audit 进一步分析具体是什么 SQL 产生了大量的磁盘 I/O，导致租户里的其他工作线程出现 I/O 等待，整体过程可以参见官网《OceanBase 数据库》文档 管理数据库/监控指标/案例。

写在最后

第七章中的 7.4 ~ 7.7 这几个节只是介绍了一些常用的分析工具，阅读完之后不能保证大家就地成

为 SQL 性能分析专家。成长为专家的路上，需要循序渐进，在大量的实践中逐步积累经验，还是那句老话：“无他，唯手熟尔”。

学习完第七章的内容之后，大家如果能够开始尝试通过上面这几个步骤进行 SQL 性能问题排查，笔者就已经非常欣慰了。如果以上方法均不可行，可以去社区论坛的 问答区 发帖咨询，会有技术支持同学来负责协助大家分析和排查各类问题。

7.8 通过 SQL Diagnoser 工具进行 SQL 性能诊断和分析

推荐通过 SQL Diagnoser 工具进行 SQL 性能诊断和分析。

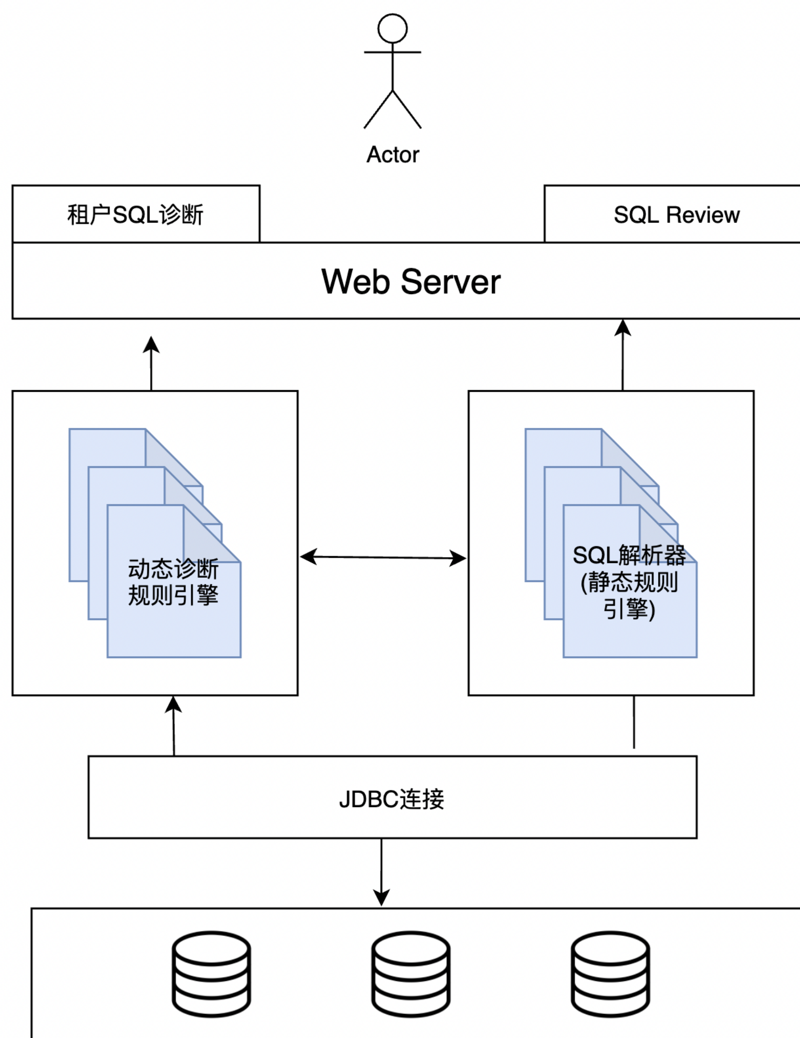
SQL Diagnoser 工具介绍

OceanBase SQL Diagnoser 是 OceanBase 的敏捷版 SQL 诊断工具。包括两大功能:

- SQL Advisor：通过简洁的白屏操作，直接分析业务集群的运行过的 SQL，找出常见的可疑 SQL，从而帮助用户找到隐藏性能问题, 给出优化建议。
- SQL Review：通过简洁的白屏操作，用户在白屏输入想要 Review 的 SQL 文本，工具根据常见的规则项帮助用户分析 SQL 语句是否合理，并给出改进建议。

SQL Diagnoser 工具的架构图如下所示，核心部分有两个诊断规则引擎负责处理。其中:

- 静态规则引擎是基于 SQL 语法解析来实现的，解析 SQL 语法树，根据语法树的节点进行静态规则的实现。比如谓词计算、隐式转换等。
- 动态规则引擎则主要是查询 OceanBase 数据库的内部性能视图，`gv$(ob_)sql_audit` 和 `gv$(ob_)plan_cache_plan_stat` 以及表和索引等信息来分析语句存在的问题。



SQL Diagnoser 使用

SQL Diagnoser 启动方式

1. 解压软件包
2. 进入软件包
3. 执行下面命令，`-Dserver.port=9090` 指定端口号，可选，若不选，默认为 `8080`

```
java -Dserver.port=9090 -jar sql-diagnoser-4.2.0.0.jar &
```

说明

若在生产环境的机器部署，请使用 JVM 参数 限制该进程使用的内存大小，避免该进程消耗过多内存。

SQL Diagnoser 关闭方式

1. 执行下面命令获取进程号

```
ps -ef  |  grep sql-diagnoser
```

2. 执行下面命令杀掉进程

```
kill -9 <进程号>
```

SQL Diagnoser 使用说明

租户 SQL 诊断

租户 SQL 诊断直接分析业务集群，找出常见的可疑 SQL，从而帮助用户找到隐藏性能问题。并且会为 SQL 提供索引优化建议和 SQL 改写建议。

全表扫描，有索引未走	全表扫描，SQL 涉及的表均有索引
全表扫描，无可用索引	全表扫描，SQL 涉及的表未建索引
走索引但性能很差	SQL 走了索引，但性能依然很差（响应时间或 CPU 时间较差）
Hint 未生效	SQL 执行时，没有走 Hint 里指定的索引
参与分区数过多	SQL 执行时，过多无关分区参与了计算，浪费系统资源

下面几个诊断项是根据一些开发规范来的，既影响 SQL 响应时间，又对系统造成较大压力。

影响行数过多	SQL 影响的行数超过设定值
返回行数过多	SQL 返回的行数超过设定值
SQL 涉及的表过多	SQL 涉及的表数目超过设定值

租户 SQL 诊断使用说明

1. 获取用户登录连接信息，使用该连接信息登录，验证该用户可以访问 `oceanbase` 和

`information_schema` 下的表输入这些信息到下面输入框。

2. 点击一键诊断即可。

2881
root@ocp_monitor
响应时间（选填，默认值：20_000）
采样数（选填，默认值：10000）
开始时间：2022-09-15 17:35:14
结束时间：2022-09-15 18:05:14
一键诊断

编号	Server	数据库	SQL ID	响应时间	诊断类型	诊断描述	SQL	调优建议	改写建议
1	[illegible]	monitordb	AB92D67EA22E83A33FB935ED9C86B3C6	1962	走索引且性能较差	cpu时间为：2200.0，逻辑读：0.0，执行计划走的索引：[o.PRIMARY]	SELECT * FROM sql_aggregate_data o WHERE o.id>65000782 AND timestamp>=1663235231 and o.tags like '%\"sql_alarm_type\":\"trans_stat\"%' order by id desc limit 2000	在monitordb.sql_aggregate_data上创建索引，列顺序为：(id,timestamp,tags)	
2	[illegible]	monitordb	BDC739A7C999D42E90435757F3228C80	4388	走索引且性能较差	cpu时间为：4674.0，逻辑读：0.0，执行计划走的索引：[obhistpack0_.PRIMARY]	select obhistpack0_.`cluster_name` as cluster_1_6_, obhistpack0_.`end_interval_time` as end_inte2_6_, obhistpack0_.`ip` as ip3_6_, obhistpack0_.`ob_cluster_id` as ob_clust4_6_, obhistpack0_.`ob_tenant_id` as ob_…		SQL 的查询条件未包含 表: ob_hist_packet 的分区键: [end_interval_time], 请带上分区键，降低资源消耗
3	[illegible]	monitordb	A498DE4ED56C36718FECD2D66F5513AA	3752	走索引且性能较差	cpu时间为：4115.0，逻辑读：0.0，执行计划走的索引：[ob_hist_snapshot.PRIMARY]	UPDATE ob_hist_snapshot SET error_count = 0, error_info = concat(error_info, '') WHERE cluster_id = 4 AND snapshot_id = 1031		SQL 的查询条件未包含 表: ob_hist_snapshot 的分区键: [end_interval_time], 请带上分区键，降低资源消耗
4	[illegible]	monitordb	4B7418AFE0A6770E440500D92C8AA1E9	2051	走索引且性能较差	cpu时间为：2398.0，逻辑读：2.0，执行计划走的索引：[ob_hist_sql_audit_stat_1.idx_ob_hist_sql_audit_stat_1_cluster_server_ti	select /*+ query_timeout(30000000) read_consistency(weak) */ min(begin_interval_time) from ob_hist_sql_audit_stat_1 where ob_cluster_id = 100004 and cluster_name = 'obocp' and ob_server_id = 1 and ob_tenan…	在monitordb.ob_hist_sql_audit_stat_1上创建索引，列顺序为：(ob_cluster_id,cluster_name,ob_server_id,begin_interval_time,ob_te	

输入框说明：

- **响应时间**：超过该值的SQL执行记录会被查询出来进行分析。

- **采用数**：每次诊断默认会采集 10000 条 执行记录分析，若想分析更多，可以调整该参数。

- **登录用户**：若是直连 OBServer 节点，格式为：`user@tenant`，如 `root@ocp_meta`；若为 OBProxy 连接，格式为：`user@tenant#cluster`，如：`root@ocp_meta#obcluster`；若是普通模式，直接填用户名。

3. **响应时间**：超过该值的SQL执行记录会被查询出来进行分析。
4. **采用数**：每次诊断默认会采集 10000 条 执行记录分析，若想分析更多，可以调整该参数。
5. **登录用户**：若是直连 OBServer 节点，格式为：`user@tenant`，如 `root@ocp_meta`；若为 OBProxy 连接，格式为：`user@tenant#cluster`，如：`root@ocp_meta#obcluster`；若是普通模式，直接填用户名。

SQL Review

SQL Review 功能，主要通过分析表结构和 SQL 结构，发现 SQL 写法存在的问题。分析过程不会执行 SQL。建议再新业务上线前对 SQL 进行检查。

SQL Review 检测规则

- 0001 索引相关规则 1：查询条件中索引列上做计算导致索引失效，无法抽取 queryrange，如 `ID+1=10`。
- 0002 索引相关规则 2：查询条件中索引列使用模糊匹配或者左模糊匹配。
- 0003 索引相关规则 3：查询条件中索引列存在隐式类型或者精度转换。
- 1001 执行计划相关规则 1：IN 数目太多，建议不超过 200 个， 如 `in(?,?,?..)`。
- 1002 执行计划相关规则 2：表连接数太多，建议不超过 10 个，否则剪枝后无最优计划。
- 1003 执行计划相关规则 3：Not in 子句需要添加 `not null` 标识，避免 nested loop join。
- 2001 高危SQL相关规则 1：update/delete 不带 where 条件或者 where 条件恒真。
- 2002 高危SQL相关规则 2：少用 Not in，存在空值处理的特殊逻辑，很难把控语义。
- 2003 高危SQL相关规则 3：不建议使用不含字段列表的 insert/replace 语句。
- 3001 性能相关的规则 1：Update/delete/select 语句不带索引键，导致全表扫描。
- 3002 性能相关的规则 2：分区表操作不带分区键，导致无法进行分区裁剪。
- 4001 DDL 相关的规则 1：索引不宜过多，避免冗余重复。
- 4002 DDL 相关的规则 2：索引列是否有数据倾斜问题，导致效果不好。
- 4003 DDL 相关的规则 3：创建索引需要手动指定 `global`/`local` 关键字，推荐局部索引。
- 4004 DDL 相关的规则 4：表字段数目不宜过多。

SQL Review 使用说明

1. 在输入区 1 中，输入租户的连接信息，需要能访问业务库（只读即可）。SQL Review 不会真正执行该 SQL。
2. 在文本输入框中输入 SQL 文本（注意，除字符串外，不要包含中文字符）。

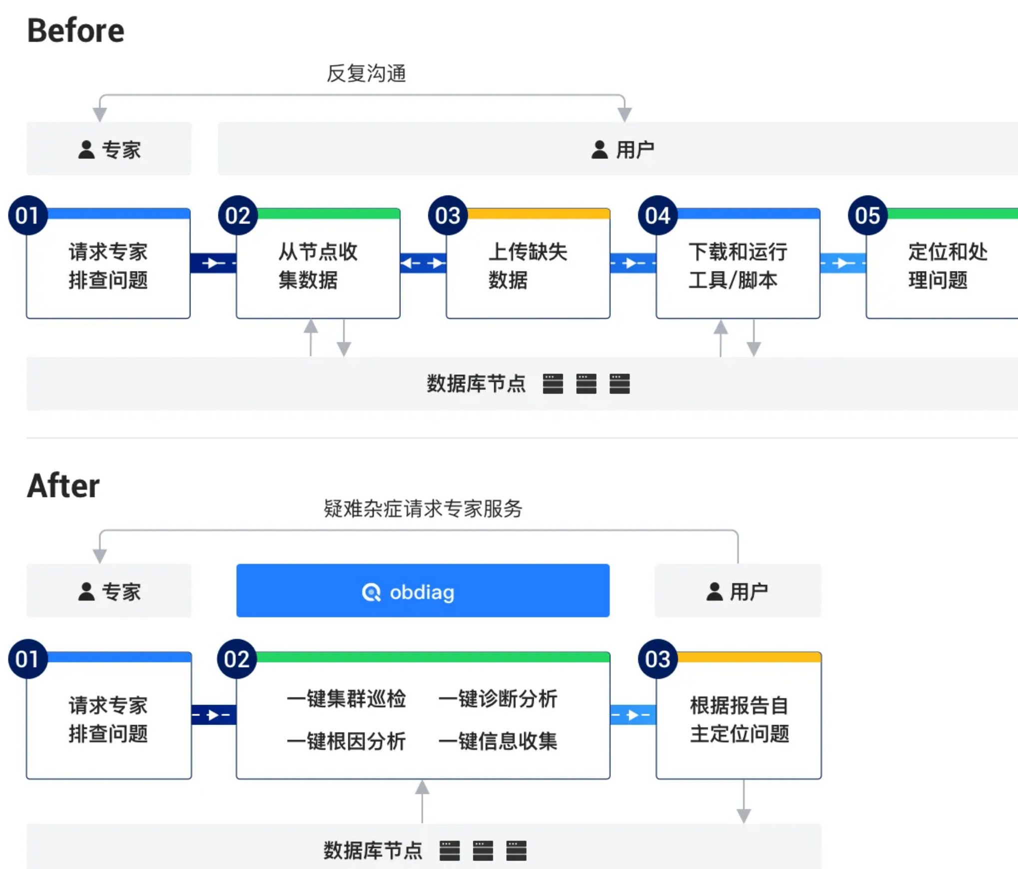
3. 点击一键检查，在最下方会给出改写建议。

7.9 通过 obdiag 工具进行诊断和分析

推荐通过 obdiag 工具进行诊断和分析。

obdiag 介绍

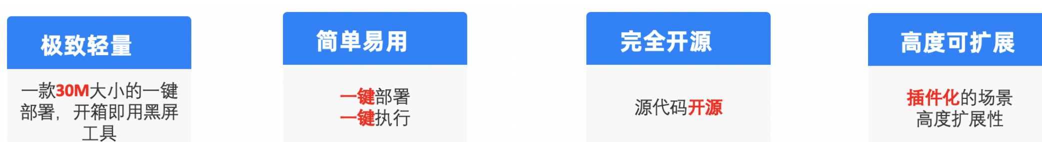
OceanBase 数据库是原生分布式数据库系统，故障根因分析通常是比较繁琐的，因为涉及的因素可能有很多，如机器环境、配置参数、运行负载等等。专家在排查问题的时候需要获取大量的信息来分析故障，如何高效的获取故障场景下分散在各个节点的信息，挖掘出其中的关联性便是 OceanBase 诊断工具（OceanBase Diagnostic Tool），简称 obdiag 需要解决的问题。



obdiag 特性简介

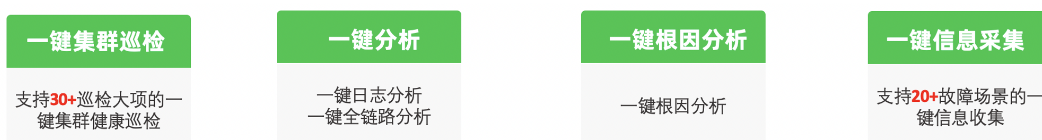
obdiag 定位为 OceanBase 敏捷诊断工具。整体使用上具备以下的特点：

- **极致轻量**：提供 RPM 包和 obd 上部署的模式，均可一键部署安装，RPM 包才不到 30 MB 大小。可以选择部署到任意一台能连接到集群各个节点的上。部署 obdiag 的机器可以在跳板机上、管控机器上，不一定要部署到 OBServer 所在的节点。
- **简单易用**：一条命令搞定安装，一键集群巡检、一键信息收集、一键诊断分析、一键根因分析等功能全部可以通过一条命令搞定，简单易用。
- **完全开源**：obdiag 是 python 代码开发的，源代码 100% 开源，github 地址仓库：https://github.com/oceanbase/obdiag。
- **高度可扩展**：obdiag 的一键巡检功能、一键场景化信息收集功能、一键根因分析功能都是插件化的，用户可自行低成本的添加场景来定制化诊断的场景。



obdiag 功能简介

obdiag 现有功能包含了对于 OceanBase 数据库日志、SQL Audit 以及 OceanBase 数据库进程堆栈等信息进行扫描、收集、分析、诊断，可以在 OceanBase 集群不同的部署模式下（OCP，obd 或用户根据文档手工部署）实现一键诊断执行。obdiag 的功能如下：



- **一键集群巡检**：使用 obdiag check 命令可对 OceanBase 数据库集群相关状态进行巡检，目前支持从系统内核参数、内部表等方式对 OceanBase 集群进行分析，发现可能会导致集群出现异常问题的点并给出运维建议。
- **一键诊断分析**：使用 obdiag analyze 命令可对 OceanBase 数据库相关的诊断信息进行分析，目前支持对 OceanBase 数据库的日志进行一键分析，找出发生过的错误信息；一键全链路诊断分析，展示全链路诊断树，定位链路慢在何处。
- **一键信息收集**：使用 obdiag gather 命令可对 OceanBase 数据库相关的诊断信息进行收集。目前支持基础诊断信息收集和基于场景的诊断信息一键收集。

- **一键根因分析**：使用 obdiag rca 命令可对 OceanBase 数据库相关的诊断信息进行分析，目前支持对 OceanBase 数据库的异常场景进行分析，找出可能导致问题的原因。

安装部署

安装

obdiag 工具可以独立部署使用也可以通过 obd 来使用，以下方式二选一。

方式一：obdiag 独立部署使用

如果您的待诊断集群是通过其他方式部署（非 obd 部署），可以下面的方式进行安装部署 obdiag。

- 在线部署（可访问外网的情况下可选择）

```
sudo yum install -y yum-utils
sudo yum-config-manager --add-repo https://mirrors.aliyun.com/oceanbase/OceanBase.
repo
sudo yum install -y oceanbase-diagnostic-tool
source /usr/local/oceanbase-diagnostic-tool/init.sh
```

- 离线部署（不可访问外网的情况下可选择）

  在 OceanBase 软件下载中心 下载新版的 OceanBase 数据库定制的敏捷诊断工具（obdiag）。

```
yum install -y oceanbase-diagnostic-tool*.rpm
source /usr/local/oceanbase-diagnostic-tool/init.sh
```

- debian系（如 Ubuntu）部署

  在 OceanBase 软件下载中心 下载新版的 OceanBase 数据库定制的敏捷诊断工具（obdiag）。

```
apt-get update
apt-get install alien -y
alien --scripts --to-deb  oceanbase-diagnostic-tool*.rpm #转为 deb 包
dpkg -i oceanbase-diagnostic-tool*.deb
```

```
source /usr/local/oceanbase-diagnostic-tool/init.sh
```

方式二：obd 部署方式

如果你的待诊断集群是 obd 部署的，建议升级 obd 到 V2.5.0 及以上版本，可通过 obd 的命令直接使用 obdiag 工具，详细命令可参考官网《OceanBase 安装部署工具》文档 obd 命令/诊断工具命令组。

- 在线部署（可访问外网的情况下可选择）

```
开启 obd 远程镜像拉取模式
obd mirror enable remote
通过 obd 部署 obdiag
obd obdiag deploy
```

- 离线部署（不可访问外网的情况下可选择）

  在 OceanBase 软件下载中心 下载新版的 OceanBase 数据库定制的敏捷诊断工具（obdiag）。

```
将离线的 obdiag 包拷贝到 obd 的镜像仓库中
obd mirror clone oceanbase-diagnostic-tool-xxxxxxxx.rpm
通过 obd 部署 obdiag
obd obdiag deploy
```

配置

obdiag 配置文件的路径有两个，一个是用户侧的配置文件，支持自定义路径。一个是 obdiag 自用的系统配置文件，一般情况下无需修改。下面分别介绍这两个配置文件。

用户侧配置文件

用户侧配置文件可通过 `obdiag config <option>` 命令快速生成或者直接编辑配置文件，文件的默认路径是 `~/.obdiag/config.yml`，其样板文件位于 `~/.obdiag/example`。

```
obdiag config -h <db_host> -u <sys_user> [-p password] [-P port]
```

配置项说明如下：

参数名称	是否必须	说明
db_host	是	OceanBase 集群 sys 租户的连接地址。
sys_user	是	OceanBase 集群 sys 租户的连接用户，为避免权限问题建议使用root@sys。当通过 OBProxy 进行连接时，需要附带加上集群名（如root@sys#obtest）。
-p password	否	OceanBase 集群 sys 租户的连接密码，默认为空。
-P port	否	OceanBase 集群 sys 租户的端口，默认为2881。

例子：

- 密码非空

```
obdiag config -hxx.xx.xx.xx -uroot@sys -p***** -P2881
```

- 密码为空

```
obdiag config -hxx.xx.xx.xx -uroot@sys -P2881
```

- 通过 OBProxy 连接

```
obdiag config -hxx.xx.xx.xx -uroot@sys#obtest  -p***** -P2883
```

执行完成后在 `~/.obdiag/config.yml` 中会生成一份新的配置，如果原来 `~/.obdiag/config.yml` 存在内容，将会将老配置备份到 `~/.obdiag/backup_conf` 目录下。

Tips:

- 如何管理多个集群的配置文件?

  生成多个集群的配置，使用 obdiag 的时候直接指定对应的集群配置即可，示例如下：

```
obdiag gather log -c cluster_1.yaml
obdiag gather log -c cluster_2.yaml
```

- obd 部署的集群，在使用 obdiag 的时候无需额外生成配置文件，直接使用功能即可。obd 上使用 obdiag 需要在 obdiag 的命令前面增加 obd 命令，并且加上具体的集群名，例如：

`obdiag gather log` 即变成 `obd obdiag gather log <cluster_name>`，obd 会负责生成 obdiag 的配置。

系统配置文件

位于 `/usr/local/oceanbase-diagnostic-tool/conf/inner_config.yml`

```
obdiag:
  basic:
    config_path: ~/.obdiag/config.yml # 用户侧的配置文件路径
    config_backup_dir: ~/.obdiag/backup_conf # 通过 obdiag config 命令执行时，老配置文件会进行备份，备份路径
    file_number_limit: 20 # 对单台远程主机执行一次采集命令回传的文件数量上限
    file_size_limit: 2G # 对单台远程主机执行一次采集命令回传的文件大小上限
  logger:
    log_dir: ~/.obdiag/log # obdiag 自身的执行日志存储路径
    log_filename: obdiag.log # obdiag 自身的执行日志存储文件名
    file_handler_log_level: DEBUG # obdiag 自身的执行日志输出的最低级别
    log_level: INFO # obdiag 自身的执行日志级别
    mode: obdiag
    stdout_handler_log_level: INFO # obdiag 打印到屏幕上的最低日志级别
check: # 巡检所需配置，一般场景下不需要变更
  ignore_version: false # 忽略 OceanBase 数据库的版本
  report:
    report_path: "./check_report/" # 巡检报告输出路径
    export_type: table # 巡检报告输出类型
  package_file: "~/.obdiag/check_package.yaml" # 巡检套餐文件路径
  tasks_base_path: "~/.obdiag/tasks/" # 巡检任务的基础目录
gather:
  scenes_base_path: "~/.obdiag/gather/tasks" # gather 场景的目录
rca:
  result_path: "./rca/" # rca 结果存储路径
```

说明

经常修改的配置：`obdiag.basic.file_number_limit` 和 `obdiag.basic.file_size_limit`，在日志收集功能使用时遇到提示远程主机的日志超过一定数量或者压缩文件超过一定大小，可以通过调整这两个参数来完成收集日志。

使用说明

一键集群巡检

使用 obdiag check 命令可帮助 OceanBase 数据库集群相关状态巡检，目前支持从系统内核参数、内部表等方式对 OceanBase 集群进行分析，发现已存在或可能会导致集群出现异常问题的原因分析并提供运维建议。

巡检场景清单

```
obdiag check list
```

输出如下:

```

[check cases about observer]:
------------------------------------------------------------------------------------
--------------------------------
command                                     info_e
n                                                             info_cn
------------------------------------------------------------------------------------
--------------------------------
obdiag check                                default check all task without filte
r                       默认执行除filter组里的所有巡检项
obdiag check --cases=ad                     Test and inspection task
s                                         测试巡检任务
obdiag check --cases=build_before           Deployment environment chec
k                                      部署环境检查
obdiag check --cases=sysbench_run           Collection of inspection tasks when executin
g sysbench       执行sysbench时的巡检任务集合
obdiag check --cases=sysbench_free          Collection of inspection tasks before executi
ng sysbench     执行sysbench前的巡检任务集合
------------------------------------------------------------------------------------
--------------------------------

[check cases about obproxy]:
------------------------------------------------------------------------------------
-------------
command                                     info_en                                 info_
cn
------------------------------------------------------------------------------------
-------------
obdiag check                                default check all task without filter   默认执
行除filter组里的所有巡检项
obdiag check --obproxy-cases=proxy          obproxy version check                   obpro
xy 版本检查
------------------------------------------------------------------------------------
-------------
```

```
#
```

一键巡检

```
# 默认执行除 filter 组里的所有巡检项
obdiag check

# 部署环境检查
obdiag check --cases=build_before

# 执行 sysbench 时的巡检任务集合
obdiag check --cases=sysbench_run

# 执行 sysbench 前的巡检任务集合
obdiag check --cases=sysbench_free

# obproxy 版本检查
obdiag check --obproxy-cases=proxy
```

一键分析功能

使用方法

```
obdiag analyze <analyze type> [options]
```

analyze type 包含如下：

- log：一键分析 OceanBase 数据库的日志。
- flt_trace：一键全链路诊断。

一键日志分析

使用该命令可以一键在线分析 OceanBase 集群的日志，或者通过 `--files` 开启离线分析模式。

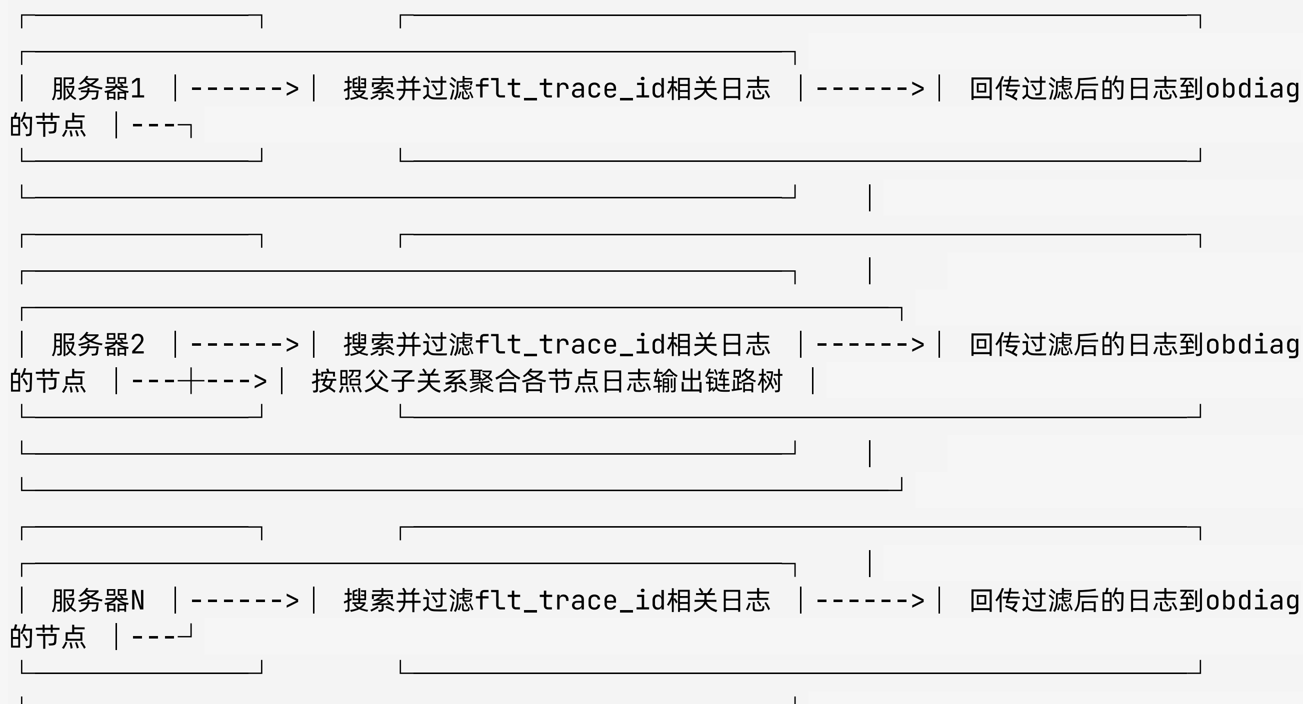
```
# 在线分析最近一小时的日志，该指令执行的时候会从远程主机上拉取最近一小时的日志进行分析，诊断出出现过的错误
obdiag analyze log --since 1h


# 在线分析最近 30 分钟的日志，该指令执行的时候会从远程主机上拉取最近 30 分钟的日志进行分析，诊断出
```

```
出现过的错误
obdiag analyze log --since 30m


# 离线分析指定的日志文件
obdiag analyze log --files observer.log.20230831142211247

# 离线分析日志文件夹
obdiag analyze log --files ./test/
```

一键全链路诊断

全链路诊断是什么：OceanBase 数据库是分布式数据库，因此调用链路复杂，当出现超时问题时，往往无法快速定位是 OceanBase 数据库内部组件或是网络的问题，运维人员只能根据经验和 observer 日志进行分析。OceanBase 数据库在 4.0 版本新增了 `trace.log` 日志，可以用于分析全链路诊断。全链路有两条路径，一条是从应用通过客户端（JDBC 或 OCI 等）下发请求给 ODP（代理服务器）访问 OBServer，访问结果返回给应用；另一条是从应用通过客户端（JDBC 或 OCI 等）直接访问 OBServer，访问结果返回给应用。全链路诊断是对全链路所有组件进行问题定位的诊断。

obdiag全链路诊断做了什么：



obdiag 全链路诊断用法

```
obdiag analyze flt_trace [options]
```

步骤一: 查找疑似慢的 SQL

在 `gv$ob_sql_audit` 中，如果有明确的 SQL 语句可以通过 `query_sql` 查到疑似慢 SQL 的 `flt_trace_id`，例如:

```
OceanBase(root@test)>select query_sql, flt_trace_id from oceanbase.gv$ob_sql_audi
t where query_sql like 'select @@version_comment limit 1';
```

输出如下:

```
+----------------------------------+-------------------------------------+
| query_sql                        | flt_trace_id                        |
+----------------------------------+-------------------------------------+
| select @@version_comment limit 1 | 00060aa3-d607-f5f2-328b-388e17f687cb |
+----------------------------------+-------------------------------------+
1 row in set
```

其中 `flt_trace_id` 为 `00060aa3-d607-f5f2-328b-388e17f687cb`。

或者你也可从 obproxy、oceanbase 的 `trace.log` 日志中找到 `flt_trace_id`。

```
head trace.log

[2023-12-07 22:20:07.242229] [489640][T1_L0_G0][T1][YF2A0BA2DA7E-00060BEC28627BEF-
0-0] {"trace_id":"00060bec-275e-9832-e730-7c129f2182ac","name":"close_das_task","i
d":"00060bec-2a20-bf9e-56c9-724cb467f859","start_ts":1701958807240606,"end_ts":170
1958807240607,"parent_id":"00060bec-2a20-bb5f-e03a-5da01aa3308b","is_follow":false
}
```

其中 `00060bec-275e-9832-e730-7c129f2182ac` 就是其 `flt_trace_id`。

步骤二: 执行全链路诊断命令

```
obdiag analyze flt_trace --flt_trace_id 000605b1-28bb-c15f-8ba0-1206bcc08aa3

root node id: 000605b1-28bb-c15f-8ba0-1206bcc08aa3

TOP time-consuming leaf span:
```

```
+---+-----------------------------------+-------------+---------------------+
| ID| Span Name                         | Elapsed Time|       NODE          |
+---+-----------------------------------+-------------+---------------------+
| 18| px_task                           | 2.758 ms    | OBSERVER(xx.xx.xx.1)|
| 5 | pc_get_plan                       | 52 μs       | OBSERVER(xx.xx.xx.1)|
| 16| do_local_das_task                 | 45 μs       | OBSERVER(xx.xx.xx.1)|
| 10| do_local_das_task                 | 17 μs       | OBSERVER(xx.xx.xx.1)|
| 17| close_das_task                    | 14 μs       | OBSERVER(xx.xx.xx.1)|
+---+-----------------------------------+-------------+---------------------+
Tags & Logs:
-------------------------------------
18 - px_task  Elapsed: 2.758 ms
     NODE:OBSERVER(xx.xx.xx.1)
     tags: [{'group_id': 0}, {'qc_id': 1}, {'sqc_id': 0}, {'dfo_id': 1}, {'task_id': 1}
]
5 - pc_get_plan  Elapsed: 52 μs
    NODE:OBSERVER(xx.xx.xx.1)
16 - do_local_das_task  Elapsed: 45 μs
     NODE:OBSERVER(xx.xx.xx.3)
10 - do_local_das_task  Elapsed: 17 μs
     NODE:OBSERVER(xx.xx.xx.1)
17 - close_das_task  Elapsed: 14 μs
     NODE:OBSERVER(xx.xx.xx.3)

Details:
+---+-----------------------------------+-------------+---------------------+
| ID| Span Name                         | Elapsed Time|   NODE              |
+---+-----------------------------------+-------------+---------------------+
| 1 | TRACE                             | -           | -                   |
| 2 | └─com_query_process               | 5.351 ms    | OBPROXY(xx.xx.xx.1) |
| 3 |   └─mpquery_single_stmt           | 5.333 ms    | OBSERVER(xx.xx.xx.1)|
| 4 |     ├─sql_compile                 | 107 μs      | OBSERVER(xx.xx.xx.1)|
| 5 |     │ └─pc_get_plan               | 52 μs       | OBSERVER(xx.xx.xx.1)|
| 6 |     └─sql_execute                 | 5.147 ms    | OBSERVER(xx.xx.xx.1)|
| 7 |       ├─open                      | 87 μs       | OBSERVER(xx.xx.xx.1)|
| 8 |       ├─response_result           | 4.945 ms    | OBSERVER(xx.xx.xx.1)|
| 9 |       │ ├─px_schedule             | 2.465 ms    | OBSERVER(xx.xx.xx.1)|
| 10|       │ │ ├─do_local_das_task     | 17 μs       | OBSERVER(xx.xx.xx.1)|
| 11|       │ │ ├─px_task               | 2.339 ms    | OBSERVER(xx.xx.xx.2)|
| 12|       │ │ │ ├─do_local_das_task   | 54 μs       | OBSERVER(xx.xx.xx.2)|
| 13|       │ │ │ └─close_das_task      | 22 μs       | OBSERVER(xx.xx.xx.2)|
| 14|       │ │ ├─do_local_das_task     | 11 μs       | OBSERVER(xx.xx.xx.1)|
| 15|       │ │ ├─px_task               | 2.834 ms    | OBSERVER(xx.xx.xx.3)|
| 16|       │ │ │ ├─do_local_das_task   | 45 μs       | OBSERVER(xx.xx.xx.3)|
| 17|       │ │ │ └─close_das_task      | 14 μs       | OBSERVER(xx.xx.xx.3)|
| 18|       │ │ └─px_task               | 2.758 ms    | OBSERVER(xx.xx.xx.1)|
| 19|       │ ├─px_schedule             | 1 μs        | OBSERVER(xx.xx.xx.1)|
| 20|       │ └─px_schedule             | 1 μs        | OBSERVER(xx.xx.xx.1)|
| ..|      ......                       | ...         | ......              |
+---+-----------------------------------+-------------+---------------------+
```

```
For more details, please run cmd ' cat analyze_flt_result/000605b1-28bb-c15f-8ba0-
1206bcc08aa3.txt '
```

一键信息采集功能

obdiag 的信息采集功能包括一键基础信息的采集和一键场景化信息采集。

一键常规信息的采集

使用方式:

```
obdiag gather <gather type> [options]
```

gather type 包含如下:

- log：一键收集所属 OceanBase 集群的日志。
- sysstat：一键收集所属 OceanBase 集群主机信息。
- clog：一键收集所属 OceanBase 集群的 clog 日志。
- slog：一键收集所属 OceanBase 集群的 slog 日志。
- plan_monitor：一键收集所属 OceanBase 集群指定 `trace_id` 的并行 SQL 的执行详情信息。
- stack：一键收集所属 OceanBase 集群的堆栈信息。
- perf：一键收集所属 OceanBase 集群的 perf 信息。
- obproxy_log：一键收集所属 OceanBase 集群所依赖的 ODP 的日志。
- all：一键统一收集所属 OceanBase 集群的诊断信息，包括收集 OceanBase 集群日志、主机信息、OceanBase 集群堆栈信息、OceanBase 集群 perf 信息。

一键场景化信息采集

使用 `obdiag gather scenes` 命令可以一键执行将某些问题场景所需要的排查信息统一捞回，解决分布式节点信息捞取难的痛点。

支持场景

使用 `obdiag gather scene list` 命令。

```
obdiag gather scene list

[Other Problem Gather Scenes]:
---------------------------------------------------------------------------------------
-----
command                                                          info_en                in
fo_cn
---------------------------------------------------------------------------------------
-----
obdiag gather scene run --scene=other.application_error   [application error]   [
应用报错问题]
---------------------------------------------------------------------------------------
-----

[Obproxy Problem Gather Scenes]:
---------------------------------------------------------------------------------------
command                                                  info_en             info_cn
---------------------------------------------------------------------------------------
obdiag gather scene run --scene=obproxy.restart   [obproxy restart]   [obproxy无故
重启]
---------------------------------------------------------------------------------------

[Observer Problem Gather Scenes]:
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------
comman
d
                                                   info_e
n                                        info_cn
---------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------
---------------------------------------
obdiag gather scene run --scene=observer.backu
p
          [backup problem]                              [数据备份问题]
obdiag gather scene run --scene=observer.backup_clea
n
    [backup clean]                                [备份清理问题]
obdiag gather scene run --scene=observer.clog_disk_ful
l
  [clog disk full]                               [clog盘满]
obdiag gather scene run --scene=observer.cluster_dow
n
    [cluster down]                                [集群无法连接]
obdiag gather scene run --scene=observer.compactio
n
```

```
      [compaction]                                  [合并问题]
obdiag gather scene run --scene=observer.cpu_hig
h
        [High CPU]                                    [CPU高]
obdiag gather scene run --scene=observer.delay_of_primary_and_backu
p                                                                    [delay of p
rimary and backup]                [主备库延迟]
obdiag gather scene run --scene=observer.i
o
              [io problem]                                  [io问题]
obdiag gather scene run --scene=observer.log_archiv
e
     [log archive]                                  [日志归档问题]
obdiag gather scene run --scene=observer.long_transactio
n
[long transaction]                             [长事务]
obdiag gather scene run --scene=observer.memor
y
          [memory problem]                             [内存问题]
obdiag gather scene run --scene=observer.perf_sql --env "{db_connect='-h127.0.0.1
-P2881 -utest@test -p****** -Dtest', trace_id='Yxx'}"   [SQL performance problem
]                      [SQL性能问题]
obdiag gather scene run --scene=observer.px_collect_log --env "{trace_id='Yxx', es
timated_time='2024-04-19 14:46:17'}"                    [Collect error source nod
e logs for SQL PX]   [SQL PX 收集报错源节点日志]
obdiag gather scene run --scene=observer.recover
y
        [recovery]                                    [数据恢复问题]
obdiag gather scene run --scene=observer.restar
t
         [restart]                                     [observer无故重启]
obdiag gather scene run --scene=observer.rootservice_switc
h                                                                            [r
ootservice switch]                           [有主改选或者无主选举的切主]
obdiag gather scene run --scene=observer.sql_err --env "{db_connect='-h127.0.0.1 -
P2881 -utest@test -p****** -Dtest', trace_id='Yxx'}"    [SQL execution error
]                          [SQL 执行出错]
obdiag gather scene run --scene=observer.suspend_transactio
n                                                                           [su
spend transaction]                          [悬挂事务]
obdiag gather scene run --scene=observer.unit_data_imbalanc
e                                                                           [un
it data imbalance]                          [unit迁移/缩小 副本不均衡问题]
obdiag gather scene run --scene=observer.unknow
n
         [unknown problem]                             [未能明确问题的场景]
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
--------------------------------------
```

一键场景信息采集

使用方法：

```
obdiag gather scene run --scene={SceneName}
```

```
# 应用报错问题
obdiag gather scene run --scene=other.application_error

# obproxy无故重启
obdiag gather scene run --scene=obproxy.restart

# 数据备份问题
obdiag gather scene run --scene=observer.backup

# 备份清理问题
obdiag gather scene run --scene=observer.backup_clean

# clog盘满
obdiag gather scene run --scene=observer.clog_disk_full

# 合并问题
obdiag gather scene run --scene=observer.compaction

# CPU高
obdiag gather scene run --scene=observer.cpu_high

# 主备库延迟
obdiag gather scene run --scene=observer.delay_of_primary_and_backup

# 日志归档问题
obdiag gather scene run --scene=observer.log_archive

# 长事务
obdiag gather scene run --scene=observer.long_transaction

# 内存问题
obdiag gather scene run --scene=observer.memory

# SQL性能问题, 此处env中的trace_id对应gv$ob_sql_audit的trace_id
obdiag gather scene run --scene=observer.perf_sql --env "{db_connect='-hxx -Pxx -u
xx -pxx -Dxx', trace_id='xx'}"

# 数据恢复问题
obdiag gather scene run --scene=observer.recovery

# observer无故重启
```

```
obdiag gather scene run --scene=observer.restart

# 有主改选或者无主选举的切主
obdiag gather scene run --scene=observer.rootservice_switch

# SQL 执行出错， 此处env中的trace_id对应gv$ob_sql_audit的trace_id
obdiag gather scene run --scene=observer.sql_err --env "{db_connect='-hxx -Pxx -uxx -pxx -Dxx', trace_id='xx'}"

# 悬挂事务
obdiag gather scene run --scene=observer.suspend_transaction

# unit迁移/缩小 副本不均衡问题
obdiag gather scene run --scene=observer.unit_data_imbalance

# 未能明确问题的场景
obdiag gather scene run --scene=observer.unknown

# SQL PX 收集报错源节点日志，两个参数：trace_id：必填，会根据该trace_id去搜集px的日志， estimated_time：选填，默认为当前时间，会搜 这个时间点以前一周的日志
obdiag gather scene run --scene=observer.px_collect_log --env "{trace_id='Yxx', estimated_time='2024-04-19 14:46:17'}"
```

一键根因分析功能

使用 `obdiag rca` 命令可帮助 OceanBase 数据库相关的诊断信息分析，目前支持对 OceanBase 数据库的异常场景进行分析，找出可能导致问题的原因。

支持场景清单

使用该命令可以获取根因分析目前支持的场景。

```
obdiag rca list
```

一键根因分析

使用如下命令可一键分析根因。

```
obdiag rca run --scene=<scene_name>
```

例子：

```
obdiag rca run --scene=disconnection
```

输出如下:

```
+---------------------------------------------------------------------------------
--------------------------+
|                                                    recor
d                                                  |
+------+--------------------------------------------------------------------------
--------------------------+
| step | inf
o
              |
+------+--------------------------------------------------------------------------
--------------------------+
|  1   | node:xxx.xxx.xxx obproxy_diagnosis_log:[2024-01-18 17:48:37.667014] [2317
3][Y0-00007FAA5183E710] |
|      | [CONNECTION](trace_type="CLIENT_VC_TRACE", connection_diagnosis={cs_id:10
65, ss_id:4559,            |
|      | proxy_session_id:837192278409543969, server_session_id:3221810838
,                                  |
|      | client_addr:"xxx.xxx.xxx.xxx:xxxx", server_addr:"xxx.xxx.xxx.xxx:2883", c
luster_name:"obcluster",      |
|      | tenant_name:"sys", user_name:"root", error_code:-10010, error_msg:"An une
xpected connection event  |
|      | received from client while obproxy reading request", request_cmd:"COM_SLE
EP", sql_cmd:"COM_END",    |
|      | req_total_time(us):5315316}{vc_event:"VC_EVENT_EOS", user_sql:""}
)                                  |
|  2   | cs_id:1065, server_session_id:322181083
8                                                          |
|  3   | trace_type:CLIENT_VC_TRAC
E                                                                        |
|  4   | error_code:-1001
0
 |
+------+--------------------------------------------------------------------------
--------------------------+
The suggest: Need client cooperation for diagnosis
```

诊断场景添加

本章介绍如何自定义扩充诊断场景，用以快速适应自身集群的诊断需求。

一键巡检场景添加

巡检场景所在目录 `~/.obdiag/check`；展开后可看到已有的巡检场景目录如下，其中 `obproxy_check_package.yaml` 和 `observer_check_package.yaml` 分别存的是 obproxy 和 observer 的巡检项集合。

```
#tree
.
├── obproxy_check_package.yaml
├── observer_check_package.yaml
└── tasks
    ├── obproxy
    │   └── version
    │       └── bad_version.yaml
    └── observer
        ├── cluster
        │   ├── core_file_find.yaml
        │   ├── data_path_settings.yaml
        │   ├── deadlocks.yaml
        │   ├── ...
        ├── cpu
        │   └── oversold.yaml
        ├── disk
        │   ├── clog_abnormal_file.yaml
        │   ├── disk_full.yaml
        │   ├── disk_hole.yaml
        │   ├── ...
        ├── err_code
        │   ├── find_err_4000.yaml
        │   ├── ...
        ├── sysbench
        │   ├── sysbench_free_test_cpu_count.yaml
        │   ├── sysbench_free_test_memory_limit.yaml
        │   ├── sysbench_free_test_network_speed.yaml
        │   ├── ...
        ├── system
        │   ├── aio.yaml
        │   ├── dependent_software_swapon.yaml
        │   ├── dependent_software.yaml
        │   ├── ...
        └── version
            ├── bad_version.yaml
            └── old_version.yaml
```

其中 tasks 文件夹存储了已经有的巡检项，一个大的巡检项目就是一个 yaml 文件。我们所说的自定义巡检项，其实就是按照一定的规则去编写一个 yaml 文件的 task。

task 文件的示例如下：

```
info: testinfo
task:
  - version: "[3.1.0,3.2.4]"
    steps:
      {steps_object}
  - version: "[4.2.0.0,4.3.0.0]"
    steps:
      {steps_object}
```

说明：

参数名	是否必填	描述
info	是	声明这个 yaml 使用的场景，方便维护。
version	否	表示适用的版本，用 Str 的形式表示范围，需要完整的数字的版本号。 OceanBase 数据库 V3.x 版本为三位，例如 [3.1.1,3.2.0]。 OceanBase 数据库 V4.x 版本为四位，例如：[4.1.0.0,4.2.0.0]。
steps	是	所执行步骤，为 list 结构。

具体编写教程参见官网《OceanBase 敏捷诊断工具（obdiag）》文档：一键集群巡检/一键集群巡检。

一键信息采集场景添加

场景所在目录 `~/.obdiag/gather/tasks`，展开后可看到已有的巡检场景目录如下，其中 `obproxy` 目录下为 OBProxy 的采集场景，`observer` 目录下为针对 OceanBase 数据库的采集场景，`other` 目录下一般是指其他组件的诊断采集场景。

```
#tree
.
├── obproxy
│   └── restart.yaml
├── observer
│   ├── backup_clean.yaml
│   ├── backup.yaml
│   ├── clog_disk_full.yaml
│   ├── cluster_down.yaml
│   ├── compaction.yaml
│   ├── delay_of_primary_and_backup.yaml
│   ├── io.yaml
│   ├── log_archive.yaml
│   ├── long_transaction.yaml
│   ├── memory.yaml
```

```
│    ├── recovery.yaml
│    ├── restart.yaml
│    ├── rootservice_switch.yaml
│    ├── suspend_transaction.yaml
│    ├── unit_data_imbalance.yaml
│    └── unknown.yaml
└── other
    └── application_error.yaml
```

诊断信息收集的场景 yaml 增加和本教程 一键巡检场景添加 的巡检场景类似，本节不做赘述，下面举个例子。

```
#cat backup.yaml
info_en: "[backup problem]"
info_cn: "[数据备份问题]"
command: obdiag gather scene run --scene=observer.backup
task:
  - version: "[2.0.0.0, 4.0.0.0]"
    steps:
      - type: sql
        sql: "show variables like 'version_comment';"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.v$ob_cluster"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.__all_zone WHERE name='idc';"
        global: true
      - type: sql
        sql: "select svr_ip,zone,with_rootserver,status,block_migrate_in_time,star
t_service_time,stop_time,build_version from oceanbase.__all_server order by zone;"
        global: true
      - type: sql
        sql: "SELECT zone, concat(svr_ip, ':', svr_port) observer, cpu_capacity, c
pu_total, cpu_assigned, cpu_assigned_percent, mem_capacity, mem_total, mem_assigne
d, mem_assigned_percent, unit_Num, round(`load`, 2) `load`, round(cpu_weight, 2) c
pu_weight, round(memory_weight, 2) mem_weight, leader_count FROM oceanbase.__all_v
irtual_server_stat ORDER BY zone,svr_ip;"
        global: true
      - type: sql
        sql: "select tenant_id,tenant_name,primary_zone,compatibility_mode from oc
eanbase.__all_tenant;"
        global: true
      - type: sql
        sql: "show parameters like '%syslog_level%';"
        global: true
      - type: sql
        sql: "show parameters like '%syslog_io_bandwidth_limit%';"
```

```
        global: true
      - type: sql
        sql: "select count(*),tenant_id,zone_list,unit_count from oceanbase.__all_
resource_pool group by tenant_id,zone_list,unit_count;"
        global: true
      - type: sql
        sql: "show parameters like '%auto_delete_expired_backup%';"
        global: true
      - type: sql
        sql: "select * from oceanbase.__all_virtual_backup_task;"
        global: true
      - type: sql
        sql: "select * from oceanbase.__all_virtual_backup_info;"
        global: true
      - type: sql
        sql: "select * from oceanbase.__all_virtual_sys_task_status where comment
like '%backup%';"
        global: true
      - type: sql
        sql: "select count(*),status from oceanbase.__all_virtual_pg_backup_task g
roup by status;"
        global: true
      - type: sql
        sql: "select svr_ip, log_archive_status, count(*) from oceanbase.__all_vir
tual_pg_backup_log_archive_status group by svr_ip, log_archive_status;"
        global: true
      - type: sql
        sql: "select * from oceanbase.__all_rootservice_event_history where gmt_cr
eate > ${from_time} and gmt_create < ${to_time} order by gmt_create desc;"
        global: true
      - type: sql
        sql: "select b.* from oceanbase.__all_virtual_pg_backup_log_archive_statu
s a,oceanbase.__all_virtual_pg_log_archive_stat b where a.table_id=b.table_id and
a.partition_id=b.partition_id  order by log_archive_cur_ts limit 5;"
        global: true
      - type: log
        global: false
        grep: ""
      - type: sysstat
        global: false
        sysstat: ""
  - version: "[4.0.0.0, *]"
    steps:
      - type: sql
        sql: "show variables like 'version_comment';"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.DBA_OB_ZONES;"
        global: true
      - type: sql
```

```
        sql: "SELECT * FROM oceanbase.DBA_OB_SERVERS;"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.GV$OB_SERVERS;"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.DBA_OB_UNIT_CONFIGS;"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.DBA_OB_RESOURCE_POOLS;"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.DBA_OB_TENANTS;"
        global: true
      - type: sql
        sql: "SELECT c.TENANT_ID, e.TENANT_NAME, concat(c.NAME, ': ', d.NAME) `poo
l:conf`,concat(c.UNIT_COUNT, ' unit: ', d.min_cpu, 'C/', ROUND(d.MEMORY_SIZE/1024/
1024/1024,0), 'G') unit_info FROM oceanbase.DBA_OB_RESOURCE_POOLS c, oceanbase.DBA
_OB_UNIT_CONFIGS d, oceanbase.DBA_OB_TENANTS e WHERE c.UNIT_CONFIG_ID=d.UNIT_CONFI
G_ID AND c.TENANT_ID=e.TENANT_ID AND c.TENANT_ID>1000 ORDER BY c.TENANT_ID;"
        global: true
      - type: sql
        sql: "SELECT a.TENANT_NAME,a.TENANT_ID,b.SVR_IP FROM oceanbase.DBA_OB_TENA
NTS a, oceanbase.GV$OB_UNITS b WHERE a.TENANT_ID=b.TENANT_ID;"
        global: true
      - type: sql
        sql: "show parameters like '%syslog_level%';"
        global: true
      - type: sql
        sql: "show parameters like '%syslog_io_bandwidth_limit%';"
        global: true
      - type: sql
        sql: "show parameters like '%backup%';"
        global: true
      - type: sql
        sql: "show parameters like '%ha_low_thread_score%';"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.CDB_OB_BACKUP_PARAMETER"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.CDB_OB_BACKUP_JOBS limit 20;"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.DBA_OB_ROOTSERVICE_EVENT_HISTORY WHERE modul
e='backup_data' AND event ='start_backup_data';"
        global: true
      - type: sql
        sql: "SELECT * FROM oceanbase.CDB_OB_BACKUP_TASKS limit 20;"
        global: true
```

```
      - type: sql
        sql: "SELECT * FROM oceanbase.__all_virtual_backup_schedule_task limit 20"
        global: true
      - type: sql
        sql: "SELECT * from oceanbase.CDB_OB_BACKUP_JOB_HISTORY where STATUS = 'FA
ILED' limit 20;"
        global: true
      - type: log
        global: false
        grep: ""
      - type: sysstat
        global: false
        sysstat: ""
```

说明：

- `info_en`：英文描述，在 `obdiag gather scene list` 的时候展示的英文描述就出自于此。
- `info_cn`：中文描述，在 `obdiag gather scene list` 的时候展示的中文描述就出自于此。
- `command`：执行命令。
- `task.version`：诊断组件的版本适用范围。
- `task.steps.type`：执行的类型，目前支持 ssh/sql/log/obproxy_log/sysstat 几种。
- `task.steps.type.global`：相对于节点来说是否是全局采集项，`global` 值为 `true` 的只会在第一个节点上执行一次，`global` 值为 `false` 的会在每个节点上都执行。

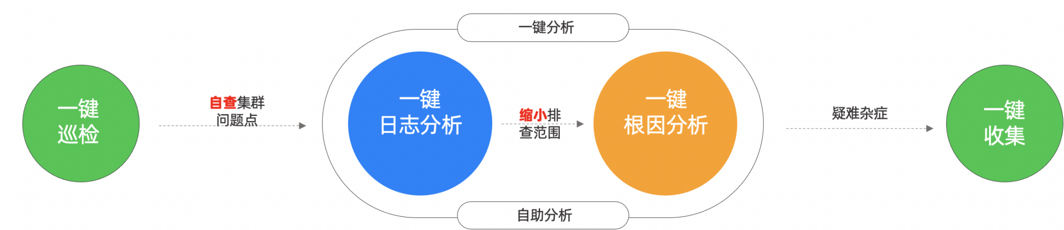
根因分析场景添加

根因分析场景的目录为 `~/.obdiag/rca`，展开后里边是 python 文件，根因分析场景的添加稍微复杂一点，需要在 python 代码中处理根因分析的具体链路逻辑，更多详情参见 obdiag 语雀文档 obdiag 项目开发手册。

```
#tree
.
├── ddl_disk_full_scene.py
├── disconnection_scene.py
├── lock_conflict_scene.py
├── ...
└── major_hold_scene.py
```

总结

数据库自助诊断三板斧，一键巡检自查集群问题点，按照诊断报告修复集群。在不明确问题场景的情况下先进行一键日志分析，去缩小排查范围，如果明确场景的情况下直接进行根因分析，通过根因分析报告查看问题根因，如果以上两个步骤都没能解决你的问题，那么请直接执行一键信息收集，将问题排查需要的故障诊断信息打包回传给社区问答区或者前线支持人员。



第八章 OceanBase 数据库故障排查和诊断

本章介绍在使用 OceanBase 产品遇到问题时如何自主排查和提问，以及一些常见的故障和恢复手段。

本章目录

8.1 遇到问题如何在官网上进行自主排查 731

8.2 遇到问题如何向技术支持同学提问 735

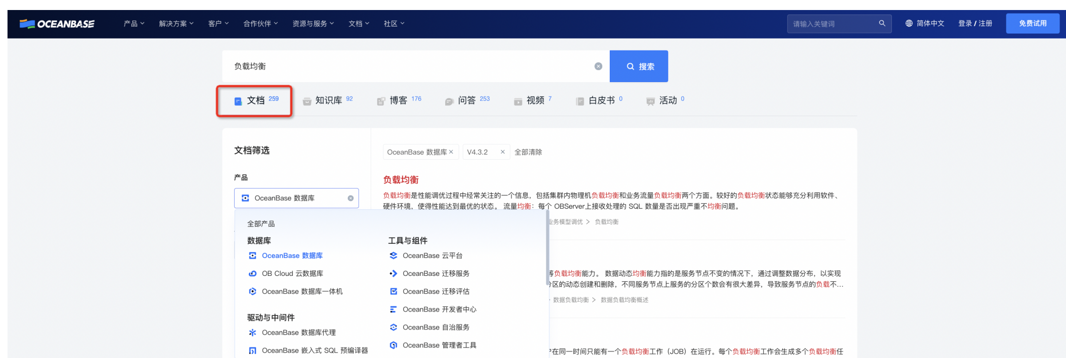
8.3 常见的故障及其恢复手段 738

8.1 遇到问题如何在官网上进行自主排查

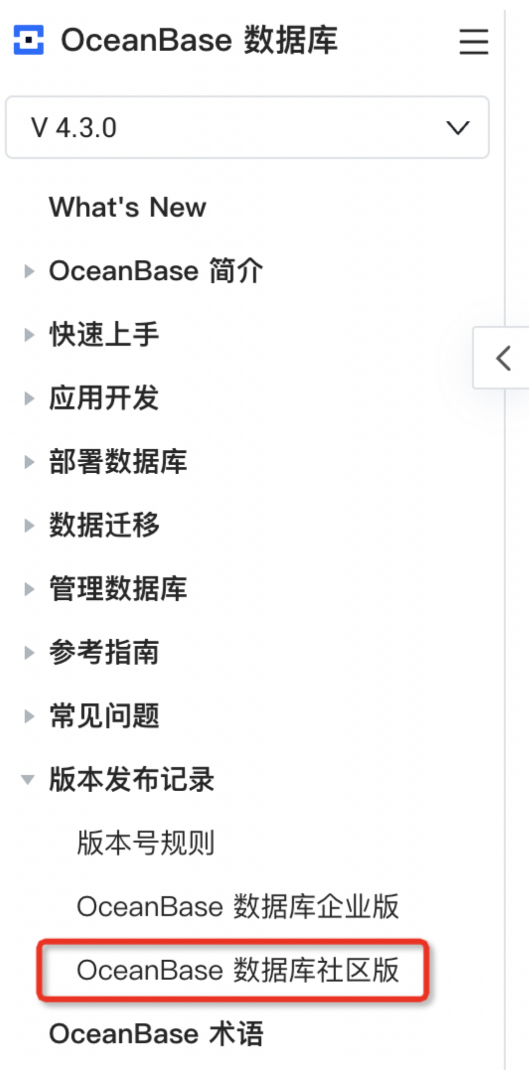
当遇到问题时，通过在官网首页（https://www.oceanbase.com）提供的搜索，会获得一个包括：文档、知识库、博客、问答、视频，白皮书 的搜索结果。可以先关注文档，知识库，博客的搜索结果，如果问题没解决，可以继续关注问答区的搜索结果。

在官网文档检索

选择所要查找产品的名称，选择合适的版本可以更加精准/快速地找到需要的信息。



同时可以在版本发布记录里查找一下对应版本的新功能和做的一些调整来初步确认是否符合预期。OceanBase 数据库社区版的版本发布记录可参见官网《OceanBase 数据库》文档 版本发布记录/OceanBase 数据库社区版。



在官网知识库检索

通过官网首页检索后进入到的知识库，可以根据产品的名称和关联的标签快速过滤出相关的知识库文档。

或者打开官网首页（https://www.oceanbase.com/），点击 **文档** 导航栏，找到完整的知识库。这里提供了一站式知识获取平台，可以获取：技术点和功能介绍、产品使用技巧、问题排查

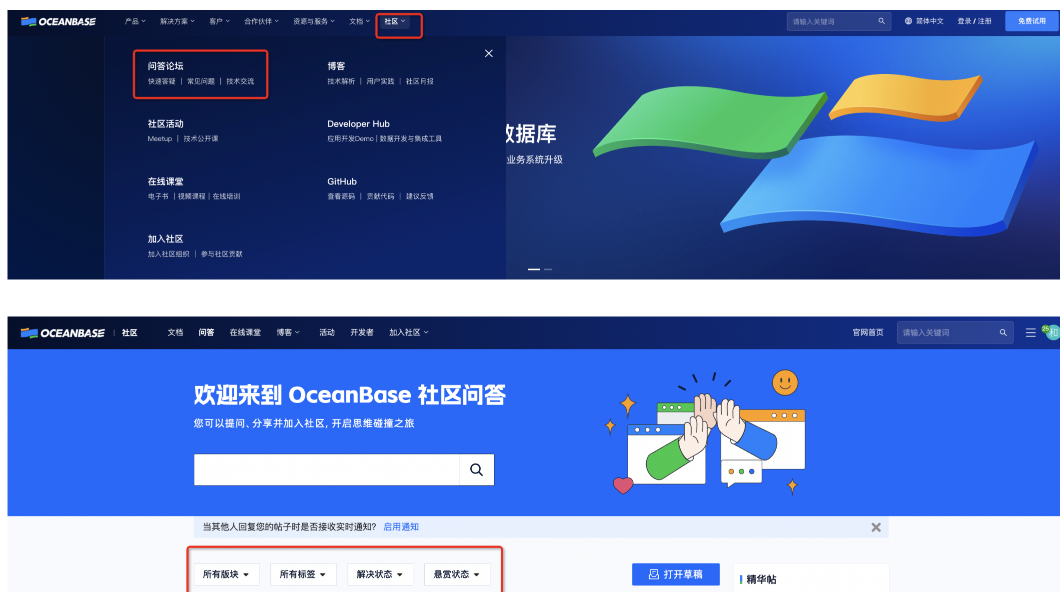
和处理方案，使用最佳实践等。

可以通过观看视频快速上手知识库的使用。

在问答区检索

对于常见的问题，推荐在问答区进行检索，被解决的问题一般都会有交流的记录，有出现问题时的日志进行参考，方便我们确认是否是同样的问题。可以按需选择合适的分类和问题标签来缩小检索数据的范围。

也可以直接登录问答区（https://ask.oceanbase.com/）根据解决问题的状态，标签等条件来查找感兴趣的问题。



在官网博客检索

打开官网首页（https://www.oceanbase.com/），点击 **社区** 导航栏，通过 **博客** 可以了解产品版本的特性解读、用户的运维实践、产品设计原理解读等，通过这些内容可以辅助我们排查和解决问题。

8.2 遇到问题如何向技术支持同学提问

遇到问题向技术支持同学提问时，需要详细描述所遇到的问题，并提供问题发生的详细步骤，以方便技术支持同学复现和处理。

问题描述

详细说明遇到的问题，包括问题的现象、错误信息（如果有）、问题场景、问题发生的时间点等。如果可能，请提供问题的具体描述和相关日志（如果是 `observer.log` 请提供 `observer.log` 日志文件（可压缩），不要提供 `.wf` 后缀的文件）。对于软件问题，可能需要提供操作系统版本、应用程序版本等信息。

【 使用环境 】生产环境 or 测试环境

【 OB or 其他组件 】

【 使用版本 】

【问题描述】清晰明确描述问题

【复现路径】问题出现前后相关操作

【附件及日志】推荐使用 OceanBase 敏捷诊断工具 obdiag 收集诊断信息，详情参见链接：https://ask.oceanbase.com/t/topic/35605619

OceanBase 数据库相关

- 使用 obdiag 收集 OceanBase 数据库基础信息：`obdiag gather scene run --scene=observer.base`。
- 可参考本教程第七章 通过 obdiag 工具进行诊断和分析 按需收集。

> **说明**
>
> 如果目前 obdiag 日志里没有覆盖到问题场景，建议使用一键常规信息的采集。另外，提供日志前请确认遇到问题时的日志级别，建议不高于 WDIAG，如果不满足调整日志级别后，在问题未恢复前，重新收集日志。日志级别的详细介绍可参见官网《OceanBase 数据库》文档 管理数据库/日志/日志级别。

obd 相关

- 版本：`obd --version`。
- 需要提供 obd 命令执行报错的截图和 `~/.obd/log/obd`。如果是升级操作，需要同时提供当前目录下的升级日志，升级相关的日志文件名 `upgrade_*.log`。

OCP 相关

- OCP 帮助中心 -> 关于 OCP -> 版本号。
- 如果是部署 OCP 报错：需要提供 `~/.obd/log/obd`、 `~/.obd/app.log` 和 OCP 部署报错的截图。
- 如果是访问的页面报错且可以稳定复现，可以参考博客：ocp多节点如何定位日志，找到对应的节点和 trace_id，然后将完整的 `ocp-server.log` 日志附件，具体的日志位置和文件名见：OCP 的系统管理 -> 系统参数对应的 `logging.file.name` 对应的，将对应报错时间段的日志打包和 trace_id 的说明一起提供给相关的技术支持同学。
- 如果 OCP 对应的任务报错，在 OCP **任务中心** 找到失败的任务，然后在右上方点击下载完整的任务日志和完整的任务截图（截图中需要包括完整的任务名称和报错的任务位置）。
- 如果是升级操作，需要同时提供升级日志，位于 `ocp/tmp` 下，文件格式：{版本信息}/upgrade_*_{时间}.log，例如：`/tmp/4.2.1.2-102000042023120514_upgrade_post_20240411101214.log`。

OMS 相关

- OMS 帮助中心 -> 关于 OMS -> 版本号
- 业务场景（迁移/同步）
- 源和目标数据源类型
- 需要提供 OMS 相关任务或者页面的报错截图，并将对应组件日志打包，具体操作可参见官网《OceanBase 迁移服务》文档 运维 OMS 社区版/OMS 社区版日志排查

OBProxy 相关

- OBProxy 版本：`./bin/obproxy -V`。
- OBProxy 日志，具体每个日志的作用可参见官网《OceanBase 数据库代理》文档 日志 章节。

  默认需要提供一下对应时间点的 `obproxy_error.log` 和 `obproxy.log`，另外根据具体的问题，例如：断连接、路由不准等按需提供 `obproxy_diagnosis.log` 相关的日志。

重现步骤

如果问题可以稳定复现，请提供问题发生的详细步骤，以便技术支持同学按照同样的路径复现问题，更好地理解问题并尽可能快速地定位问题。

8.3 常见的故障及其恢复手段

我们在官方文档、问答区和对外的 gitbook 文档中总结和沉淀了不同组件服务故障场景示例以及恢复手段，通过关键字报错信息、问题现象帮助使用者快速定位到类似问题，协助提供问题排查思路和解决方法。

说明

本教程中所贴官网文档链接版本均为编写教程时的最新版本，若和您正在使用的版本不匹配，请在文档所在网页左上角选择适当的版本。

- obd 常见问题可参见官网《OceanBase 安装部署工具》文档 常见问题/常见问题汇总。
- OCP 常见问题可参见官网《OceanBase 云平台》文档 常见问题 章节，具体包含如下内容：
    - 部署常见问题
    - 升级常见问题
    - 运维常见问题
    - 监控常见问题
    - 告警常见问题
    - OCP 系统常见问题
- 部署常见问题
- 升级常见问题
- 运维常见问题
- 监控常见问题
- 告警常见问题

- OCP 系统常见问题
- OMS 常见问题可参见官网《OceanBase 迁移服务》文档 常见问题 章节，具体包含如下内容：
    - 安装部署
    - 一般运维
    - 项目诊断
    - 运维监控
    - 数据同步
    - 数据迁移
- 安装部署
- 一般运维
- 项目诊断
- 运维监控
- 数据同步
- 数据迁移
- OceanBase 数据库常见问题可参见官网《OceanBase 数据库》文档 常见问题 章节，具体包含如下内容：
    - 产品 FAQ
    - SQL FAQ
    - 部署 FAQ

  - 集群管理 FAQ
  - 高可用 FAQ
  - 列存 FAQ
- 产品 FAQ
- SQL FAQ
- 部署 FAQ
- 集群管理 FAQ
- 高可用 FAQ
- 列存 FAQ
- OBProxy 数据库代理常见问题处理可参见官网《OceanBase 数据库代理》文档 运维 章节，具体包含如下内容：
  - 问题排查思路
  - 性能分析
  - show trace
  - 路由诊断
  - 内存诊断
- 问题排查思路
- 性能分析
- show trace

- 路由诊断
- 内存诊断
- 导数工具常见问题可参见官网《OceanBase 导数工具》文档中如下内容：
    - 导入数据/常见问题

    - 导出数据/常见问题

- 导入数据/常见问题
- 导出数据/常见问题
- ODC 常见问题处理可参见官网《OceanBase 开发者中心》文档 故障排查 章节，具体包含如下内容：
    - 故障排查

    - 信息收集

    - 常见问题

- 故障排查
- 信息收集
- 常见问题
- 问答区 SOP 系列汇总可参见官网问答区 SOP 汇总。
- 实战 FAQ 汇总可参见 生态组件 FAQ 大全。

第九章 OceanBase 集群运维管理之用户实操

概述

本章内容从运维实操角度，以不同的用户实践案例，讲述生产环境中的 OceanBase 集群运维技巧，希望结合前八章的理论为大家提供更具参考价值的内容。

本章内容均为 OceanBase 数据库用户投稿，且投稿长期开放，欢迎大家将自己的运维经验总结成文或录制为视频，分享至 OceanBase 社区博客，完整且优质的内容将被集合进本章节。

OceanBase 集群日常运维之备份恢复

视频简介：备份和恢复对 DBA 而言并不陌生，OceanBase 数据库在集群模式中默认三副本模式，其中任意一个副本不工作，都不影响服务的连续性。两个副本都不工作时，才会中断服务。该模式本身就具备高可用，为什么还需要备份策略呢？

数据库备份的意义在于恢复，也是灾难发生时最后一道数据安全屏障。本视频介绍 OceanBase 集群备份的主要对象、备份策略与方法。

视频作者：管元峥，360 商业化业务线数据库负责人，曾担任多个行业的数据库运维工作，对每个行业的业务特点有深入了解。专注于 OceanBase、MySQL 等主流关系型数据库以及 Redis、MongoDB、Aerospike、Pika 等 NoSQL 数据库的运维工作。对新技术有着浓厚的兴趣，热衷于与厂商进行产品完善、压力测试及分析等工作。在数据库管理、数据分析、分布式系统、性能优化、系统可靠性和灾难恢复等领域拥有深厚的知识和丰富的实践经验。

地址：https://www.oceanbase.com/video/9001574

OceanBase 数据库高可用架构之主备库

视频简介：在 OceanBase 数据库 V4.2.0 中，物理备库采用独立的主备库架构，主备关系存在于租户级别。不同于以前版本的集中式架构（集群级主备），独立主备库架构下，各个集群相互独立，用户可以更加灵活地管理集群。本视频介绍 OceanBase 数据库 V4.2.0 主备库方案的适用场景及管理方式。

视频作者：严军，阿里巴巴数据库架构师，常年负责淘宝、支付宝核心系统数据库架构演进、性能优化、单元化多活容灾、 “双 11”大促护航保障等工作。主导过蚂蚁交付、支付、会员、红包核心系统去 IOE，从 0 到 1 构建了阿里云 GTS 端到端的数据库交付与售后服务体系。专注于数据库、大数据、存储领域十余年，对数据库、大数据、分布式、高并发、高性能、高可用容灾有丰富的经验。精通主流数据库产品（Oracle、MySql、PolardDB、OceanBase），拥有阿里云-云计算架构师 ACE、OceanBase OBCE、红帽 RHCSA 等专业技术认证。

地址：https://www.oceanbase.com/video/9001484

在流量高峰如何做好 OceanBase 集群保障？

作者简介：田朋，任职于同程旅行，主要负责同程旅行分布式数据库推广和维护，分布式数据库爱好者。

流量高峰期间分为可预期流量和不可预期流量。可预期流量只是规律的流量，比如“618”、“双11”、“双12”。不可预期流量如：明星的热点事件，突发性重大新闻等情况。

针对上述情况，DBA 可以提前做准备，保障系统的稳定运行。如何提前准备？

知己知彼才能百战百胜

首先，做到知己，通过多次（至少 3 次）配合业务进行流量压测。尤其是核心服务的流量压测，以及公网流量压测，得出当前核心服务及整体链路能够承受多大的 QPS。同时，制定整体改进方案，比如数据库是否缺少索引、当前容量是否充足、响应耗时是否满足业务需求。

其次，做到知彼。通过往年的“618”、“双11”等活动得知流量峰值，以及运营同事对活动流量峰值的评估，假设预估峰值和流量相比去年增长 10%，那么就根据去年的峰值和流量增加 10%，如果去年是 100w/QPS，那么今年就是 110w/QPS 。

对于压测时间，以“618”大促为例，5 月 10 日左右需要完成至少 3 次全链路压测，1次公网流量压测。在 5 月 20 日左右进行封网操作，原则上线上业务禁止变更，准备迎接大促的到来。

阶段一：活动前

在活动开始前，配合业务进行压测的过程中会发现慢查询 SQL。部分需要业务进行调整还需要 DBA 进行索引的优化。

OceanBase 优化

涉及查找慢 SQL、索引优化、分区表和分区键选择、执行计划抖动优化、容量评估等。

- 查找慢 SQL：可以根据官网《OceanBase 云平台》文档 SQL 调优实践/常见的异常 SQL 定位/如何定位租户下 CPU 负载高的 SQL 在 OCP 页面进行查找。
- 索引优化：好的索引可以根据减少读行、避免回表、避免排序等角度进行优化。推荐使用联合索引替换单列索引。联合索引可以减少读行、减少回表的数据量、减少执行计划错误概率。

  根据实际情况选择全局索引或局部索引，推荐优先本地索引，这是为了减少分布式事务，不建议使用全局索引。如果有不带分区键的SQL且涉及的数据量较小时，可以使用全局索引会提升效果。
- 分区表和分区键的选择：当数据量超过 10 亿，推荐使用分区表。
  - 对于分区键，推荐选择80%业务查询的关键词，比如订单号，建议使用分区键 + 主键的形式。
  - 分区表 SQL 不带分区键在高并发场景下会导致租户线程繁忙、CPU 被“打爆”的问题，进行影响性能。
  - 当分区表扫描过多分区的 SQL 时，在高并发场景下会导致性能抖动，比如按日期做分区，查询条件为整个月时。

- 对于分区键，推荐选择80%业务查询的关键词，比如订单号，建议使用分区键 + 主键的形式。
- 分区表 SQL 不带分区键在高并发场景下会导致租户线程繁忙、CPU 被“打爆”的问题，进行影响性能。
- 当分区表扫描过多分区的 SQL 时，在高并发场景下会导致性能抖动，比如按日期做分区，查询条件为整个月时。

- 执行计划抖动优化：可以接收 OCP 的报警，发现执行计划异常，需要手动执行绑定。
- 容量评估：包含磁盘空间容量、CPU 容量、响应耗时等。
    - 除了慢查询的情况，还需对 OceanBase 数据库和 ODP 进行提前扩容。需要注意的是，为扩容 OBServer 时避免产生分布式事务，可以通过 table group 方式，聚合到某个OBServer节点上。
    - 针对容量比较大的 db，提前进行归档操作。
    - 针对分区表和自增主键进行检查，避免自增主键用满及分区表无法写入的情况。
- 除了慢查询的情况，还需对 OceanBase 数据库和 ODP 进行提前扩容。需要注意的是，为扩容 OBServer 时避免产生分布式事务，可以通过 table group 方式，聚合到某个OBServer节点上。
- 针对容量比较大的 db，提前进行归档操作。
- 针对分区表和自增主键进行检查，避免自增主键用满及分区表无法写入的情况。

此外，建议 DBA 每周或每天把慢查询 SQL 的情况发送给研发人员，还要排查是否存在大事务未提及的情况。

硬件调整

磁盘、raid 卡 、出口带宽、机柜超电、双电链路、温湿度等根据实际需求进行调整。

阶段二：活动中

在大促活动中，DBA 需要提前进行以下操作：

1. 关闭备份任务，活动结束后开启备份任务。
2. 调整转储和合并策略，尽量避免活动中触发大合并。
3. 流量高峰期前，推迟 2 小时 ETL 数据抽取、数据推送、业务跑批。
4. 停止数据归档操作，减少 I/O 消耗。
5. 做好应急预案，并测试预案的有效性。预案包括：主备集群切换、保护机制（自动限流大

SQL 或超时之后自动断开）、SQL 执行计划异常问题处理方案、等等。

阶段三：活动后

针对活动中出现的问题进行详细记录，并复盘、优化整体活动流程，避免下次活动遇到同样的问题。另外，对活动前扩容的 Zone 和 ODP 进行缩容。

由于流量的不确定性，或许突发的流量高峰对研发人员和运维人员而已是一次基本功的考验。对于运维人员来说：需要检查 OceanBase 集群中是否有慢查询 SQL，如果有慢查询 SQL，可以根据经验进行添加索引操作，针对流量大的租户，进行租户资源扩容。而对于研发人员，需要合理利用 cache，或进行降级操作，让 SQL 尽快执行。

经得起考验才能无惧风雨

无论可预期流量还是不可预期流量，都是针对 DB 的一次大考验，只有经历过大流量，大规模的 DB，基本功经得起考验，才是一款被打磨好的 DB，才能无惧风雨。

流量高峰期间的数据库支持不只是运维团队的事情，还需要研发、测试等团队的通力合作，才能交出一份满意的答卷。

SQL 优化：利用 batch join 优化分布式中的 rescan

作者简介：胡呈清，爱可生 DBA 团队成员，擅长故障分析、性能优化，

在分布式数据库中，关联查询不可避免地对分布在不同节点的表、分区进行跨节点的关联操作，数据在不同节点间的分发方式对关联查询的性能至关重要。在 NLJ（nested-loop join） 和 SPF（subplan filter）中，需要针对驱动表中的每一行重新扫描被驱动表中的数据，如果被驱动表是分区表且分布在不同的机器上，在分布式重新扫描（rescan）的过程中会包括资源释放、调度重启等过程，这些操作常常伴随着消息等待、同步及网络传输操作，影响执行效率。

- 分布式交互包括释放上一次子协调节点执行资源的控制消息、调度子协调节点的控制消息等，这些若干控制消息的 RPC 交互会带来额外的开销。
- 调度重启的代价高是由于 Nested Loop Join 被驱动表可能是分布式复杂子计划，需要重新建立子计划调度关系，这个重新建立的过程会带来调度的开销。

另外，大量的网络交互也会极大地增加响应时间。

下面通过举例说明在 OceanBase 集群中利用 batch join 优化分布式中的 rescan 性能问题。

问题描述

这个问题出现在一个测试环境的 OceanBase 数据库 V4.2.1 BP3 版本中，表 `a` 驱动表 `f` 做 nested-loop join，`a`、`f` 表不在同一个 OBServer 节点且都是非分区表。驱动表输出结果 5 万行左右，被驱动表 `f` 走高效索引，最终结果输出 11 万行左右，这个 SQL 执行耗时 60 秒：

```
select  a.ah,f.aj
from dc_sjjz.V_TYCX_AJJBXX a
join flws_new.flws f on a.ah=f.ah
where a.sxrq>=date'2020-01-01'
and aydm in (3029,6209,6210,6365,6366,6469,8179,8180,8271,8294,8789)
and ajxzdm='2'
and jafsmc like '%判决%';
```

在这个例子中，对被驱动表做了 5 万次 rescan，产生了 5 万次 rpc 访问，网络上两个 OBServer 节点之间一次 rtt 平均 0.8ms，非常慢。OceanBase 数据库优化器支持 batch join，会减少 rescan 次数提升效率，但此处似乎没有发挥作用，下文基于此进行解析。

分析过程

1. 分析 `GV$SQL_AUDIT`。

   主要目的是获取两个信息：trace_id、执行耗时 61秒。

```
+-----------------------------+----------------+-----------+-----------------------
--------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------
-------------------------------------------------+-----------------------------------
---+-------------+-------------+
| usec_to_time(request_time) | svr_ip         | plan_type | query_sq
l

                                                                | trace_i
d                           | elapsed_time | execute_time |
+-----------------------------+----------------+-----------+-----------------------
--------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------
-------------------------------------------------+-----------------------------------
---+-------------+-------------+
```

```
| 2024-03-26 16:04:53.124344 | xxx |         1 | select /*+ monitor */
a.ah
from dc_sjjz.V_TYCX_AJJBXX a
join flws_new.flws f on a.ah=f.ah
where a.sxrq>=date'2020-01-01'
and aydm in (3029,6209,6210,6365,6366,6469,8179,8180,8271,8294,8789)
and ajxzdm='2'
and jafsmc like '%判决%';   | YB4289116C6E-000613F0D09FF63D-0-0 |     61845207
|     61782771 |
+-----------------------------+----------------+-----------+----------------------
----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
-------------------------------------------------+--------------------------------
---+--------------+--------------+
1 row in set
```

2. 分析执行计划。

从执行计划来看，NESTED-LOOP JOIN 算子中被驱动表走了非常高效的索引，每次查询只扫 1 行，要挑毛病的话，只有驱动表 `A` 没走索引、没使用 batch join 了（use_batch=false）。

3. 分析 `GV$SQL_PLAN_MONITOR`。

执行计划看似没问题，但这个 SQL 不应该这么慢，因此，需要通过 `GV$SQL_PLAN_MONITOR` 了解真实的执行情况，查询命令如下：

```
SELECT op_id, op, output_rows, rescan, threads,
       close_time - open_time AS open_dt,
       last_row_eof_time - first_row_time AS row_dt,
       open_time, close_time, first_row_time, last_row_eof_time
FROM (SELECT plan_line_id AS op_id,concat(lpad(' ', max(plan_depth), ' '), plan_op
eration) AS op
               sum(output_rows) AS output_rows,  --算子输出行数
               sum(STARTS) AS rescan, --算子被 rescan 的次数
               min(first_refresh_time) AS open_time,  --算子开始(监控)时间
               max(last_refresh_time) AS close_time,  --算子结束(监控)时间
               min(first_change_time) AS first_row_time,  --算子吐出首行数据时间
               max(last_change_time) AS last_row_eof_time, --算子吐出最后一行数据时间
               count(1) AS threads  --线程数量
        FROM gv$sql_plan_monitorWHERE trace_id = '替换成trace_id'GROUP BY plan_line
_id, plan_operat
        ORDER BY plan_line_id
) a;
```

我们可以看出几个信息：

- 驱动表输出 49557 行，近 5 万行，等于被驱动表 rescan 次数，被驱动表输出 11 万行左右。

- 每个算子的耗时都是 61 秒，这是正常的，因为这里只做了 nested-loop join，过程是：
  - 驱动表取一行数据，开始计时，这是 1 号算子的开始吐行时间。

  - 从被驱动表进行查询开始计时，这是 2 号算子的开始吐行时间。

  - 直到循环结束，1、2 号算子的最后吐行时间是一样的。因此不能认为 1 号算子即驱动表耗时了 61 秒，需要从 5 万次 rescan 入手。

4. 驱动表输出 49557 行，近 5 万行，等于被驱动表 rescan 次数，被驱动表输出 11 万行左右。
5. 每个算子的耗时都是 61 秒，这是正常的，因为这里只做了 nested-loop join，过程是：
   - 驱动表取一行数据，开始计时，这是 1 号算子的开始吐行时间。

   - 从被驱动表进行查询开始计时，这是 2 号算子的开始吐行时间。

   - 直到循环结束，1、2 号算子的最后吐行时间是一样的。因此不能认为 1 号算子即驱动表耗时了 61 秒，需要从 5 万次 rescan 入手。

6. 驱动表取一行数据，开始计时，这是 1 号算子的开始吐行时间。
7. 从被驱动表进行查询开始计时，这是 2 号算子的开始吐行时间。
8. 直到循环结束，1、2 号算子的最后吐行时间是一样的。因此不能认为 1 号算子即驱动表耗时了 61 秒，需要从 5 万次 rescan 入手。
9. 确认 rpc_count 和表位置。

   通过 `GV$CDB_OB_TABLE_LOCATIONS` 确认了 `db1.t1`、`db2.t2` 两张表不在一个 Unit ，因此 join 时需要跨 OBServer 节点，通过 `GV$OB_SQL_AUDIT` 确认这个 SQL 执行时产生了 49557

次 RPC（rpc_count=49557）。

对应的执行计划中，join 算子 `use_batch=false`，同样说明被驱动表 `f` 进行了 49557 次 rescan。

10. 开启 batch join。

    要减少被驱动表的 rescan 次数，需要让 batch join 生效，这由租户的隐藏变量 `_NLJ_BATCHING_ENABLED` 来控制，通过 `CDB_OB_SYS_VARIABLES` 可以查询隐藏变量的设置情况：

```
select TENANT_ID,NAME,VALUE,INFO,DEFAULT_VALUE from CDB_OB_SYS_VARIABLES where name='_NLJ_BATCHING_ENABLED';
```

    结果如下，发现 batch join 默认关闭了。

```
obclient [oceanbase]> select TENANT_ID,NAME,VALUE,INFO,DEFAULT_VALUE from CDB_OB_SYS_VARIABLES where name='_NLJ_BATCHING_ENABLED';
+-----------+-----------------------+-------+-----------------------------------------+---------------+
| TENANT_ID | NAME                  | VALUE | INFO                                    | DEFAULT_VALUE |
+-----------+-----------------------+-------+-----------------------------------------+---------------+
|         1 | _nlj_batching_enabled | 0     | enable batching of the RHS IO in NLJ    | 0             |
|      1001 | _nlj_batching_enabled | 0     | enable batching of the RHS IO in NLJ    | 0             |
|      1002 | _nlj_batching_enabled | 0     | enable batching of the RHS IO in NLJ    | 0             |
|      1003 | _nlj_batching_enabled | 0     | enable batching of the RHS IO in NLJ    | 0             |
|      1004 | _nlj_batching_enabled | 0     | enable batching of the RHS IO in NLJ    | 0             |
+-----------+-----------------------+-------+-----------------------------------------+---------------+
5 rows in set (0.196 sec)
```

    将 batch join 开启：`SET global_NLJ_BATCHING_ENABLED=true;`，然后再执行 SQL，耗时从 61 秒下降到 2.8 秒，执行计划中 `use_batch=true`。

结论

在上述例子中，SQL 执行慢的原因是被驱动表与驱动表不在一个 OBServer 节点上，当前使用版本 OceanBase 数据库 V4.2.1 BP3，`_NLJ_BATCHING_ENABLED` 默认关闭，这就是没有使用 DAS Group rescan 优化即没有使用 batch join 的原因，以至于被驱动表 rescan 次数太多，且跨 OBServer 节点 rescan，导致执行非常慢。

在这个场景中，需要注意几点事项：

- `GV$SQL_PLAN_MONITOR` 中的 rescan 不能代表真实的 rescan 次数，应该恒等于驱动表的输出行数，即使优化器进行了 DAS Group rescan 优化。
- `GV$OB_SQL_AUDIT` 中 `rpc_count` 字段可以用来确认 NLJ 算子中被驱动表的 rescan 次数。
- `GV$SQL_PLAN_MONITOR` 中查询出来的算子输出行的总时间结果有可能多个算子一样，这要根

据具体算子的执行逻辑解读。

- batch join 并不总是默认开启的，我后来在 release note 中找到了原因：因为一些 bug 在 OceanBase 数据库 V4.2.1 BP2 Hotfix1 版本中，默认关闭了该优化功能，后面又在 OceanBase 数据库 V4.2.1 BP5 默认打开了该优化开关。

致谢

《数据库管理与运维：OceanBase从入门到实践》一书是 OceanBase 开源团队与社区成员共同编写、修订，这是一个相互协作的过程，很多人付出了时间和精力来共创本书，对此，OceanBase 社区深表感谢。

以下列出本书编撰人员，以及为本书慷慨贡献的人员（排名不分先后）。

编撰人员

李博洋：OceanBase 解决方案工程师。

李作群：OceanBase 技术专家。

李晓东：OceanBase技术专家。

汤庆：OceanBase 技术专家。

肖帆：OceanBase 技术专家。

郑晓锋：OceanBase 技术专家。曾就职于金融科技公司，主要负责云数据库产品化研发，涉及 MySQL、MyCat、MongoDB 等多种数据库，在内部做过数据库迁移上云、同城异地容灾、容器化等多个项目。目前专注于 OceanBase 社区版的开源布道，为用户提供分布式数据库解决方案。

张鑫：OceanBase 技术专家，精通 OceanBase、MySQL 等关系型数据库，负责过多家头部金融及互联网公司的数据库国产化迁移改造。

严军：阿里巴巴数据库架构师，常年负责淘宝、支付宝核心系统数据库架构演进、性能优化、单元化多活容灾、“双11”大促护航保障等工作。主导过蚂蚁交付、支付、会员、红包核心系统“去IOE”，从 0 到 1 构建了阿里云 GTS 端到端的数据库交付与售后服务体系。专注于数据库、大数据、存储领域十余年，对数据库、大数据、分布式、高并发、高性能、高可用容灾有丰富的经验。精通主流数据库产品（Oracle、MySql、PolardDB、OceanBase），拥有阿里云-云计算架构师 ACE、OceanBase OBCE、红帽 RHCSA 等专业技术认证。

管元峥：360 商业化业务线数据库负责人，曾担任多个行业的数据库运维工作，对每个行业的业务特点有深入了解。专注于 OceanBase、MySQL 等主流关系型数据库，以及 Redis、

MongoDB、Aerospike、Pika 等 NoSQL 数据库的运维工作。对新技术有着浓厚的兴趣，热衷于与厂商进行产品完善、压力测试及分析等工作。在数据库管理、数据分析、分布式系统、性能优化、系统可靠性和灾难恢复等领域拥有深厚的知识和丰富的实践经验。

田朋：任职于同程旅行，主要负责同程旅行分布式数据库推广和维护，分布式数据库爱好者。

胡呈清：爱可生 DBA 团队成员，擅长故障分析、性能优化。

修订人员

张瑞远：新炬网络 DBA，曾经从事银行、证券数仓设计、开发、优化类工作，现主要从事电信级IT系统及数据库的规划设计、架构设计、运维实施、运维服务、故障处理、性能优化等工作。OceanBase OCEC 委员会成员，持有 Orale OCM、MySQL OCP 及国产代表数据库认证。获得的专业技能与认证包括 OceanBase OBCE、Oracle OCP 11g、OracleOCM 11g、MySQL OCP 5.7、腾讯云 TBase、腾讯云 TDSQL、阿里云 ACP、KingBase KCP。

夏克：任职于大连商品交易所，大连飞创信息技术有限公司，负责核心交易系统设计研发，从业 14 年，熟悉达梦数据库、PostgreSQL、MogDB、OceanBase。

罗呈祥：某头部药企算法工程师、DBA，负责 AI for Drug Discovery 的相关工作，包括数仓建设，算法模型设计与落地。持有 HCNP、MySQL OCP 等证书，并在网络、数据库方向有大量的实践经验。

管长龙：爱可生开源社区运营经理。

洪云飞：任职于重庆市金色束合科技有限公司、重庆三十七度健康管理有限公司。负责所有内部研发管理工作和核心架构搭建工作。从业 11 年，熟悉各类数据库（MySQL，PostgreSQL、OceanBase），熟悉 Golang、Rust、Python、Java、Vue3 等开发工作。熟悉网络架构搭建、虚拟化数据中心搭建。

姚力：翼鸥教育高级 DBA。

审校人员

梅庆（庆涛）：拥有 19 年数据库相关从业经验，熟悉 Oracle、MySQL、SQLserver、DRDS、OceanBase，擅长数据库应用的架构方案，数据库性能诊断。

刘江：翼鸥教育运维负责人，目前负责翼鸥数据库、应用运维和运维开发 3 个团队，在服务治理和稳定性保障、自动化开发等方向拥有丰富的实践经验。

冀浩东：陌陌（现挚文集团）数据库负责人。目前负责集团数据库团队建设及数据库存储运营工作，在大规模数据源稳定性建设、团队建设、成本优化、机房迁移等技术领域积累了深厚的专业经验与实战心得。

薛世杰（雪辉）：拥有多年数据库运维经验，熟悉 MySQL、MongoDB、TiDB、OceanBase 等，持有 MySQL5.7、MySQL8.0、阿里云 ACP 证书，擅长数据库架构设计、故障处理、性能优化等。

蔡飞志：OceanBase 高级技术专家，开源研发负责人。

陈玉静：OceanBase 开源测试团队负责人，技术支持团队负责人。

傅容锋：OceanBase 开源管控生态负责人。

田玮靖：OceanBase 社区运营，社区内容负责人。

张素娟：OceanBase 文档开发工程师，OceanBase 初级 DBA。