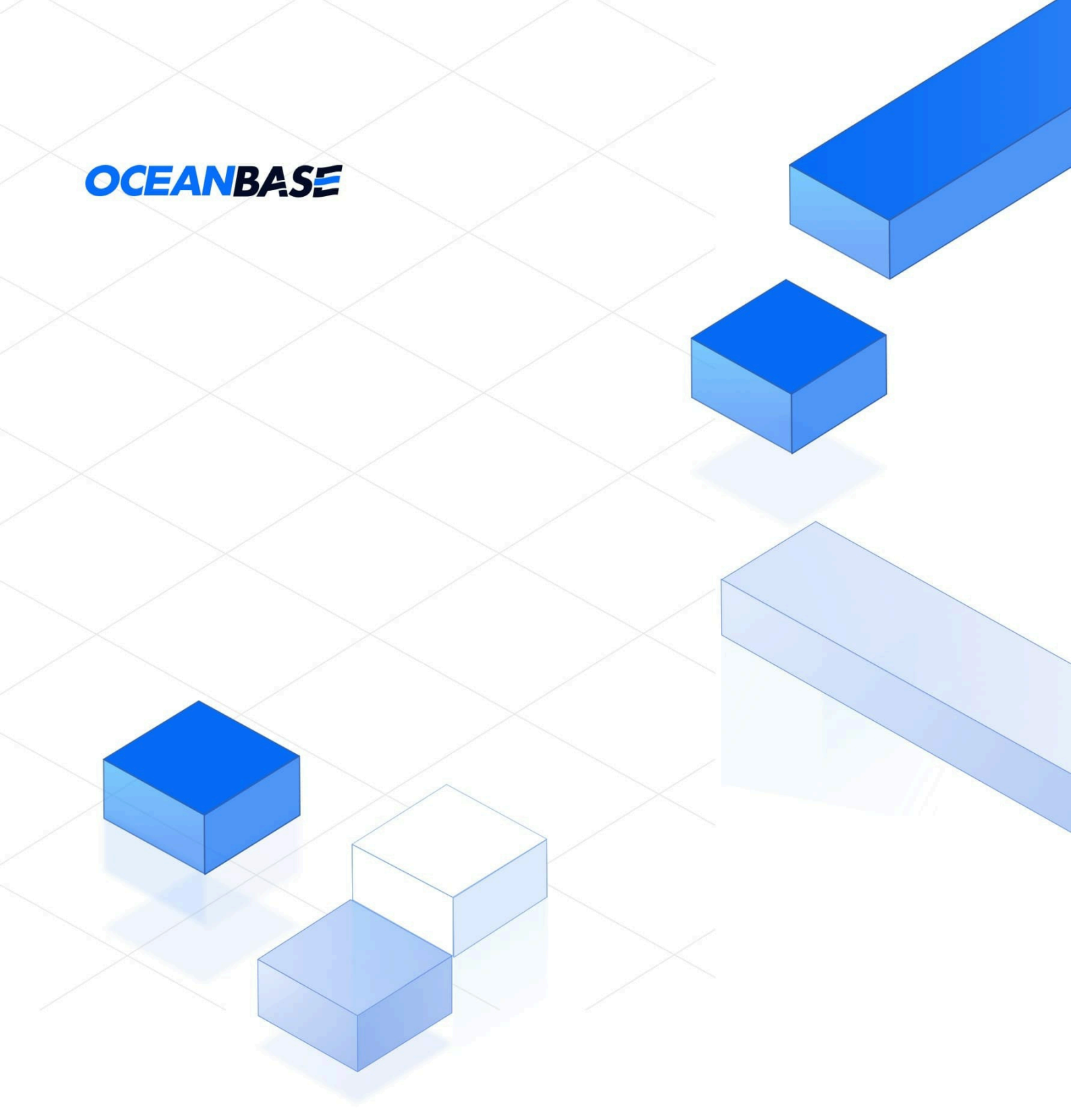


**OCEANBASE**



# OceanBase 数据库

## 常见问题

| 产品版本：V4.5.0


| 文档版本：20251219

# 声明

**北京奥星贝斯科技有限公司版权所有©2024，并保留一切权利。**

未经北京奥星贝斯科技有限公司事先书面许可，任何单位、公司或个人不得擅自摘抄、翻译、复制本文档内容的部分或全部，不得以任何方式或途径进行传播和宣传。

## 商标声明

 **OCEANBASE** 及其他 OceanBase 相关的商标均为北京奥星贝斯科技有限公司所有。本文档涉及的第三方的注册商标，依法由权利人所有。

## 免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。北京奥星贝斯科技有限公司保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在北京奥星贝斯科技有限公司授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过北京奥星贝斯科技有限公司授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

# 通用约定

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

# 目录

1 产品 FAQ	9
1.1 产品架构和特点 FAQ	9
1.1.1 我需要一个实例还是多个实例？	9
1.1.2 用户如何使用 OceanBase 数据库？	9
1.1.3 OceanBase 数据库与 JPA 冲突吗？	9
1.1.4 数据文件对应哪个级别的数据库管理？	9
1.1.5 OceanBase 数据库是如何支持 HTAP 的？	9
1.1.6 什么是实例，什么是租户，它们的关系是什么？	9
1.1.7 OceanBase 数据库的性能和机器数量的关系是什么？	10
1.1.8 使用 OceanBase 数据库在开发中要特别注意什么？	10
1.1.9 OceanBase 数据库是如何做到比传统数据库更省空间的？	10
1.1.10 OceanBase 数据库做 AP 能力处理所需的数据量大概多少？	10
1.1.11 OceanBase 数据库最新版本中对标准 SQL 的支持度是多少？	10
1.1.12 以前运行在 MySQL 上的业务，迁移到 OceanBase 数据库上的成本如何？	
1.1.13 OceanBase 数据库作为一个以 TP 见长的数据库，AP 计算能力怎么样呢？	11
1.1.14 OceanBase 数据库作为一款分布式数据库，业务 APP 连接数据库时和传统数据库是否有不同？	11
1.1.15 OceanBase 数据库的技术架构有哪些技术特点？	11
1.2 内存 FAQ	13
1.2.1 OceanBase 数据库有哪些内存区域？	13
1.2.2 OceanBase 数据库的资源在租户中哪些是共享的，哪些是不共享的？	14
1.2.3 OceanBase 数据库的内存使用有什么特征？	14
1.2.4 运行一段时间后，OceanBase 数据库使用内存接近 memory_limit 是否正常？	14
1.2.5 OBCore 节点的内存上限是否可以动态调整？	14
1.2.6 在使用 OceanBase 数据库定义资源单元以及资源池时是否允许内存超卖？	
1.2.7 OceanBase 数据库进行一次内存分配会检查哪些限制？	15
1.2.8 OceanBase 数据库的 KVCache 有哪些，分别有什么作用？	16
1.2.9 KV Cache 如何做到的动态伸缩，以及有什么样的淘汰规则？	17
1.2.10 OceanBase 数据库内存有哪些常见问题，可能导致的原因是什么？	17
1.3 多租户线程 FAQ	18
1.3.1 OceanBase 数据库在运行过程中是单进程还是多进程？	18
1.3.2 OBCore 节点有哪些后台线程，主要是做什么用的？	19
1.3.3 OBCore 节点的多租户架构中如何做到 CPU 资源的有效隔离？	21
1.3.4 如何读取 OBCore worker 线程的数量呢？OBCore 节点是否会在负载高的时候动态启动新的线程来执行负载呢？	21
1.3.5 大查询 CPU 资源分配原则是什么？OLAP 和 OLTP 同时存在的情况下会出现抢占 CPU 资源的情况么？	21

2 SQL FAQ	23
2.1 SQL 操作执行 FAQ	23
2.1.1 如何定位创建 PL 的错误?	23
2.1.2 如何分析 PL 的错误日志?	23
2.1.3 如何查询已经创建的 PL 对象的源码?	23
2.1.4 MySQL 模式是否支持 INSERT ALL INTO 这种语法?	23
2.1.5 SQL 引擎、事务引擎等对于租户而言, 资源是如何分配和隔离的?	23
2.1.6 OceanBase 数据库的 Batch 执行是什么?	23
2.1.7 为什么要使用 Batch 执行?	24
2.1.8 JDBC 中哪些 class/object 可以实现 Batch 执行?	24
2.1.9 使用 Batch 执行需要设置什么 JDBC 配置属性?	24
2.1.10 哪些 OceanBase 数据库的配置项和 Batch 执行有关?	27
2.1.11 Statement 和 PreparedStatement 在行为和使用方法上有什么不同?	
2.1.12 OceanBase 数据库的 Batch 执行有哪些类型, 分别对请求的优化处理有哪些?	28
2.1.12.1 说明	30
2.1.13 如何在不同的场景下选择不同的配置?	32
2.1.13.1 说明	33
2.1.14 如何查看 OceanBase 数据库的 Batch 执行是否生效?	35
2.1.14.1 注意	36
2.1.15 Batch 执行时, executeBatch 方法返回的值是多少?	36
2.1.16 如何解决 SQL 查询“大小账号”的问题	36
2.2 SQL 调优 FAQ	39
2.2.1 数据库物理设计降低查询性能	39
2.2.2 系统负载影响单条 SQL 的响应时间	39
2.2.3 代价模型缺陷导致的执行计划选择错误	39
2.2.4 客户端路由与服务器之间出现路由反馈逻辑错误	39
2.3 索引监控 FAQ	40
2.3.1 索引监控默认启动, 会不会影响性能?	40
2.3.2 采样模式下的数据是否准确? 例如查询了 1 次, 是否一定会被记录下来?	
2.3.3 备库的索引监控是只统计自己还是同步主库的?	40 40
2.4 分区交换 FAQ	40
2.4.1 Oceanbase 数据库 Oracle 模式下, 分区表和非分区表都创建了同样的全局索引, 为什么分区交换报错, 是索引不匹配?	40
2.4.2 为什么不能交换二级分区表的一级分区?	40

2.5 复制表 FAQ .....	40
2.5.1 什么是复制表? .....	40
2.5.2 复制表适用场景? .....	41
2.5.2.1 适用场景一: 写入频率较低、读操作延迟和负载均衡要求高的场景	
2.5.2.2 适用场景二: 业务上不能设置为分区表、同时频繁参与跟分区表 JOIN 的场景	41
2.5.3 复制表误用场景? .....	41
2.5.3.1 误用场景一: 为了避免两个分区表的跨节点连接, 将其中一个分区表设成复制表	41
2.5.3.2 误用场景二: 事务中含有复制表的写和读 .....	42
2.5.3.3 误用场景三: 复制表 JOIN 普通表 .....	42
2.5.3.4 误用场景四: 复制表设置为分区表 .....	42
2.6 自增列 FAQ .....	42
2.6.1 什么情况下需要将自增列数据类型设置为 BIGINT? .....	42
2.6.2 什么情况下允许自增列数据类型保持为 INT? .....	43
2.6.3 什么情况下可以将自增列改为 NOORDER? .....	43
2.6.4 什么情况下可以改小自增列 auto_increment_cache_size? .....	43
3 部署 FAQ .....	44
3.1 安装 OceanBase 服务器对系统有什么要求吗? .....	44
3.1.0.1 说明 .....	44
3.1.0.2 说明 .....	44
3.1.0.3 说明 .....	45
3.2 在生产环境中, OceanBase 数据库会如何部署? .....	45
3.3 什么是 LSE? .....	46
3.3.1 OCP 什么版本开始适配带 nolse 的包? .....	46
3.3.2 OBD 部署时注意事项 .....	47
3.3.3 OCP 部署时注意事项 .....	47

4 集群管理 FAQ .....	48
4.1 选举 FAQ .....	48
4.1.1 OceanBase 集群的选举有哪几种，分别是由谁发起的？ .....	48
4.1.2 什么样的副本可以参与选举？ .....	48
4.1.3 OceanBase 集群的选举模块的对基础设施的要求有哪些？ .....	48
4.1.4 OceanBase 集群的选举模块是如何保证 RTO 不大于 8s 的？ .....	48
4.1.5 如果多数派副本宕机，OceanBase 集群的选举模块是否还可以选出 Leader 副本保证高可用？ .....	49
4.1.6 OceanBase 集群的选举模块是否需要做任何运维操作？ .....	49
4.1.7 在选举中哪个副本会被选为主，如何保证选举到的 Leader 副本是更好的选择？ .....	49
4.1.8 在什么情况下会有自动切主？ .....	49
4.1.9 如何查看触发切主的原因？ .....	50
4.1.10 如何查看无主选举？ .....	50
4.1.11 OceanBase 集群报错 -4038 是什么含义？ .....	50
4.1.12 CLOG 模块会影响副本选举吗，是如何影响的？ .....	50
4.1.13 CLOG 在发生主动卸主后会重新选主么？ .....	50
4.2 CLOG FAQ .....	51
4.2.1 OceanBase CLOG 同步压缩功能中不同的压缩算法有什么区别，应该怎么选择？ .....	51
4.2.2 事务执行到提交的过程中，数据更改是如何提交到 MemStore，持久层并保存在 clog 中的？ .....	51
4.2.3 如果 OBCServer 节点的滑动窗口卡住或者满掉，会出现什么报错，会造成什么后果，应当如何调优？ .....	52
4.2.4 CLOG reconfirm 失败的场景以及失败后有什么影响？如何分析 CLOG reconfirm 失败的情景 .....	53
4.2.5 CLOG 满盘应该如何进行问题分析和应急操作？ .....	58
4.2.5.1 应急手段 .....	59
4.2.5.2 可能原因 .....	59
4.2.6 如果 OBCServer 节点重启时在初始化 CLOG 失败会报什么错，如果 OBCServer 节点启动时初始化失败报错应当如何操作恢复系统？ .....	60
4.3 存储 FAQ .....	61
4.3.1 OceanBase 数据库的存储引擎和传统数据库有什么不一样的地方？ ..	61
4.3.2 数据在 OceanBase 数据库中是如何存储的？ .....	62
4.3.3 应该使用分区表吗？ .....	62
4.3.4 在 OceanBase 数据库中，局部索引与全局索引在实现上的区别是什么？ .....	62



5	高可用 FAQ .....	63
5.1	什么是系统的高可用? .....	63
5.2	OceanBase 数据库有哪些产品能力保证了数据服务的高可用? .....	63
5.3	OceanBase 数据库如果少数派宕机会发生什么? 多久可以恢复? .....	66
5.4	OceanBase 数据库选举如何避免脑裂问题? .....	66
5.5	OceanBase 集群当多数派失败的时候是否依旧可以服务? .....	67
5.6	OceanBase 集群的副本自动补齐功能是如何工作的, 副本自动补齐是否能够保证即使在节点宕机的情况下, 相应的分区副本依旧齐全? .....	67
5.7	OceanBase 集群是否支持多机房多地区的部署方式, 在部署的时候有哪些基础设施的要求, 应该如何选择方案? .....	68
6	列存 FAQ .....	70
6.1	列存允许增删列吗? .....	70
6.2	列存表的查询有何特点? .....	70
6.3	列存支持事务吗, 对事务大小有什么限制? .....	70
6.4	列存表的日志同步、备份恢复等有什么特别之处吗? .....	70
6.5	是否支持将行存表用 DDL 变成列存表? .....	70
6.5.0.1	说明 .....	71
6.6	列存里支持多个列集合在一起吗? .....	71
6.7	列存支持更新吗, 以及 MemTable 里的结构是怎样的? .....	71
6.7.0.1	注意 .....	71
6.8	支持对列存的某一列建索引吗? .....	72
6.9	“列存索引”是什么? .....	72
6.10	列存表与列存副本的区别是什么? .....	73



# 1 产品 FAQ

## 1.1 产品架构和特点 FAQ

### 1.1.1 我需要一个实例还是多个实例？

如果有多个子系统，每个子系统在数据库层面没有交互，建议每个子系统使用不同的实例。

### 1.1.2 用户如何使用 OceanBase 数据库？

和传统数据库一样，OceanBase 数据库提供了 SQL 接口，用户可以通过 SQL 语言访问、操作数据库。

### 1.1.3 OceanBase 数据库与 JPA 冲突吗？

JPA (Java Persistence API) 是 Java 标准中的一套 ORM 规范，借助 JPA 技术可以通过注解或者 XML 来描述对象与关系表之间的映射关系，并将实体对象持久化到数据库中（即完成 Object Model 与 Data Model 间的映射）。而 OceanBase 数据库是阿里巴巴和蚂蚁集团不基于任何开源产品，完全自研的原生分布式关系型数据库软件。两者不冲突。

### 1.1.4 数据文件对应哪个级别的数据库管理？

OceanBase 数据库中目前有两类数据文件，且两类数据文件均属于集群级别：

- Data 文件：保存各个分区的数据，包括各个分区的 Checkpoint 点。
- clog 相关：包含 clog（也称为 Redo Log 或 WAL 日志）和它的索引文件。

### 1.1.5 OceanBase 数据库是如何支持 HTAP 的？

OceanBase 数据库自创的分布式计算引擎，能够让系统中多个计算节点同时运行 OLTP 类型的应用和复杂的 OLAP 类型的应用，真正实现了用一套计算引擎同时支持混合负载的能力，让用户通过一套系统解决 80% 的问题，充分利用用户的计算资源，节省用户购买额外的硬件资源、软件授权带来的成本。

### 1.1.6 什么是实例，什么是租户，它们的关系是什么？

OceanBase 数据库是一个多租户系统，一个实例即 OceanBase 集群中的一个租户。租户之间的数据不能互相访问。

## 1.1.7 OceanBase 数据库的性能和机器数量的关系是什么？

从 OceanBase 数据库的 TPC-C 报告中显示，相同场景下的基本都是线性扩展的。

## 1.1.8 使用 OceanBase 数据库在开发中要特别注意什么？

以下列出开发过程中的注意事项，仅供参考：

- 大数据量导入需要特别关注内存的使用情况。
- 索引生效时间较长，建议在建表时将索引语句一并纳入。
- mysql-connector-java 的版本建议使用 5.1.30 及以上。
- 列类型修改有较大限制。Varchar 长度只能由短变长，不能由长变短。
- 如果一个连接超过 15 分钟空闲，服务端会主动断开，在使用连接池的时候需要设置一个连接最大的空闲时间。例如，Druid 的 `minEvictableIdleTimeMillis` 小于 15 分钟。

## 1.1.9 OceanBase 数据库是如何做到比传统数据库更省空间的？

OceanBase 数据库通过数据编码压缩技术实现高压缩。数据编码是基于数据库关系表中不同字段的值域和类型信息，所产生的一系列的编码方式，它比通用的压缩算法更懂数据，从而能够实现更高的压缩效率。

## 1.1.10 OceanBase 数据库做 AP 能力处理所需的数据量大概多少？

具体根据业务需求 100 G 到 PB 级不设限。AP 是 OceanBase 数据库提供的一个能力。数据量较小时，把并行设置小一些，用的计算资源就小；数据规模较大时，计算资源设置多一些。本质上并没有一个量的限制。AP 的能力，主要体现在把机器硬件资源充分调动起来的能力，以及生成高效的并行计划等方面的能力。

## 1.1.11 OceanBase 数据库最新版本中对标准 SQL 的支持度是多少？

在实际应用场景中，MySQL 模式的大部分业务均能够做到平滑迁移；Oracle 模式下基本的 Oracle 功能也都支持，只需要很少的改动就可以平滑地由 Oracle 数据库迁移到 OceanBase 数据库。

### 1.1.12 以前运行在 MySQL 上的业务，迁移到 OceanBase 数据库上的成本如何？

OceanBase 数据库兼容常用 MySQL 功能及 MySQL 前后台协议，业务零修改或少量修改即可从 MySQL 迁移至 OceanBase 数据库。

### 1.1.13 OceanBase 数据库作为一个以 TP 见长的数据库，AP 计算能力怎么样呢？

AP 主要采用行列混合存储、编译执行、向量引擎、基于代价的查询改写和优化等技术，再加上 OceanBase 数据库良好的可扩展性，OceanBase 数据库在 AP 领域里面的实时分析能力都是业界领先的。此外，对于离线大数据处理，Spark 等大数据方案也是更合适的选择。

### 1.1.14 OceanBase 数据库作为一款分布式数据库，业务 APP 连接数据库时和传统数据库是否有不同？

OceanBase 数据库作为分布式数据库，副本可能分布在不同的机器上，为了尽可能地减少跨机数据访问，OceanBase 数据库提供了 obproxy。obproxy 作为 OceanBase 分布式关系数据库专用反向代理服务器，为前端用户请求提供了高性能、高准确率的路由转发服务，为后端 Server 服务提供了高可用易扩展的容灾保障。相对于其他数据库的代理服务器，obproxy 根据实际单机环境和 OceanBase 多集群部署的特点，采用异步框架和流式转发的设计，采用 FastParse 和 LockFree 的内存方案，具备有限资源占用下百万 QPS 的能力和海量部署下丰富便捷的运维支持能力。

### 1.1.15 OceanBase 数据库的技术架构有哪些技术特点？

OceanBase 数据库作为一款原生的分布式数据库，有以下技术特点：

- 弹性扩展

OceanBase 数据库支持在线弹性扩展，当集群存储容量或是处理能力不足时，可以随时加入新的 OBServer，系统自动进行数据迁移，根据机器的处理能力，将合适的数据分区迁移到新加入的机器上；同样在系统容量充足和处理能力富余时，也可以将机器下线，降低成本。比如在双 11 大促之类的活动中，可以提供良好的弹性伸缩能力。

- 负载均衡能力

OceanBase 数据库管理着许多台 OBCServer 节点形成一个 OBCServer 集群为多个租户提供数据服务。OceanBase 集群管控的所有 OBCServer 节点可以被视为一个超级大的“资源蛋糕”，在分配资源时，按需分配给创建租户时申请的资源。OceanBase 数据库的负载均衡能力能够保证多个租户在整个 OBCServer 集群中申请的资源占用相对均衡，并且在动态场景下（比如添加删除 OBCServer、添加删除租户、增删过程中分区数据量发生倾斜等场景），负载均衡算法仍然能在已有的节点上平衡资源。

OceanBase 数据库通过 Root Service 管理各个节点间的负载均衡。不同类型的副本需求的资源各不相同，Root Service 在执行分区管理操作时需要考虑的因素包括每台 OBCServer 节点上的 CPU、磁盘使用量、内存使用量、IOPS 使用情况；避免同一张表格的分区全部落到少数几台 OBCServer；让耗内存多的副本和耗内存少的副本位于同一台机器上；让占磁盘空间多的副本和占磁盘空间少的副本位于同一台机器上。经过负载均衡，最终会使得所有机器的各类型资源占用都处于一种比较均衡的状态，充分利用每台机器的所有资源。

- 分布式事务 ACID 能力

OceanBase 数据库架构下事务的 ACID 的实现方式是：

- Atomicity：使用两阶段提交保证快照事务原子性。
- Consistency：保证事务的一致性。
- Isolation：使用多版本机制进行并发控制。
- Durability：事务日志使用 Paxos 协议做多副本同步。

- 高可用

OceanBase 数据库中的每个分区都维护了多个副本，这些分区的多个副本之间通过 Paxos 协议进行日志同步。每个分区和它的副本构成一个独立的 Paxos 组，其中一个副本为主（Leader），其它副本为备（Follower）。每台 OBCServer 服务的一部分分区为 Leader，一部分分区为 Follower。当 OBCServer 节点出现故障时，Follower 分区不受影响，Leader 分区的写服务短时间内会受到影响，直到通过 Paxos 协议将该分区的某个 Follower 选为新的 Leader 为止，整个过程不超过 30s。通过引入 Paxos 协议，可以保证在数据强一致的情况下，具有极高的可用性及性能。

同时 OceanBase 数据库也支持主备库架构。OceanBase 集群的多副本机制可以提供丰富的容灾能力，在机器级、机房级、城市级故障情况下，可以实现自动切换，并且不丢数据（即 RPO = 0）。当主集群出现计划内或计划外（多数派副本故障）的不可用情况时，备集群可以接管服务，并且提供无损切换（RPO = 0）和有损切换（RPO > 0）两种容灾能力，最大限度降低服务停机时间。

OceanBase 数据库支持创建、维护、管理和监控一个或多个备集群。备集群是生产库数据的热备份。管理员可以选择将资源密集型的报表操作分配到备集群，以便提高系统的性能和资源利用率。

- 高效的存储引擎

OceanBase 数据库的存储引擎基于 LSM-Tree 架构，数据被划分为 MemTable（也称作 MemStore）和 SSTable 两部分。其中 MemTable 提供读写，而 SSTable 是只读的。用户插入/删除/更新的数据先写入 MemTable，通过 Redo Log 来保证事务性，Redo Log 会在三副本间使用 Paxos 协议进行同步，当单台 Server 宕机时，通过 Paxos 协议保证数据的完整性，并通过较短的恢复时间来保证数据的高可用。

当 MemTable 的大小超过一定的阈值时，需要将 MemTable 中的数据转存到 Mini SSTable 中以释放内存，该过程称为 Mini Compaction。随着用户数据的写入，Mini SSTable 的数量越来越多，当 Mini SSTable 的数量超过一定的阈值时，后台会自动触发 Minor Compaction。转储（Mini Compaction）会生成新的 Mini SSTable。当转储（Mini Compaction）的次数超过一定阈值时，或者在每天的业务低峰期，系统会将基线 SSTable（Major SSTable）与之后转储的增量 SSTable（Mini/Minor SSTable）合并为一个新的 Major SSTable，该过程称为合并。OceanBase 数据库通过转储和合并的过程，优化了数据存储空间的基础，提供了高效的读写服务，保证事务性和数据的完整性。

- 多租户

OceanBase 数据库是一个支持多租户的分布式数据库，一个集群支持多个业务系统，也就是通常所说的多租户特性。

多租户的架构优势在于可以充分利用系统资源，使得同样的资源可以服务更多的业务。通过将波峰、波谷期不同的业务系统部署到一个集群，以实现对系统资源最大限度的使用。在租户的应用方面，保证了租户之间的隔离性；在数据安全方面，不允许跨租户的数据访问，确保用户的数据资产没有泄露的风险；在资源使用方面，租户“独占”其资源配额，该租户对应的前端应用，无论是响应时间还是 TPS/QPS 都比较平稳，不会受到其他租户负载轻重的影响。

- Oracle 兼容和 MySQL 兼容

OceanBase 数据库支持 Oracle 兼容模式和 MySQL 兼容模式，用户可以根据不同的需要选择不同的模式。

## 1.2 内存 FAQ

### 1.2.16 OceanBase 数据库有哪些内存区域？

OceanBase 数据库中主要有以下内存区域：

- kv cache：LSM-Tree 中 SSTable 及数据库表模式等的缓存。
- memory store：LSM-Tree 中的 MemStore 的内存。
- sql work area：租户执行 SQL 过程中各个 Operator 工作区占用的内存，超出部分通常会走落盘流程。
- system memory：预留给 Net IO、Disk IO、Election 与负载均衡等各种功能的内存。



## 1.2.17 OceanBase 数据库的资源在租户中哪些是共享的，哪些是不共享的？

对内存而言，sql work area、memory store 与 kv cache 等资源为租户独享；system memory 为多个租户共享。对线程而言，sql worker 为租户间隔离；Net IO、Disk IO 与 clog Writer 等资源为租户间不隔离。

## 1.2.18 OceanBase 数据库的内存使用有什么特征？

OceanBase 数据库在启动时会需要加载大约几个 GB 的内存，在运行过程中会逐渐按需进一步申请内存直至 memory\_limit。而一旦 OBServer 节点向 OS 进行了内存申请，通常是不会释放或者返回给 OS，而是会维护在内存管理的使用列表和 Free List 当中。这个是 OceanBase 数据库设计上所建立的内存管理机制。

## 1.2.19 运行一段时间后，OceanBase 数据库使用内存接近 memory\_limit 是否正常？

在 OceanBase 数据库的集群运行一段时间以后，内存消耗接近于 memory\_limit 水平位置，且不发生降低的现象是符合预期的。

有一个例外是若设置了参数 memory\_chunk\_cache\_size，OBServer 节点会尝试将 Free List 超过 memory\_chunk\_cache\_size 的内存块还给 OS，增大 Free List 在 OceanBase 数据库内部的复用率，减少 RPC 由于内存操作慢而导致超时的风险。通常情况是不需要配置 memory\_chunk\_cache\_size 的，在特定的场景下需要与 OceanBase 数据库 支持进行场景分析确定是否需要动态调整 memory\_chunk\_cache\_size。

## 1.2.20 OBServer 节点的内存上限是否可以动态调整？

OBServer 节点的内存上限可以通过调整 memory\_limit 或 memory\_limit\_percentage 来动态实现。需要注意的是调整前要检查内存资源是否够用，OBServer 节点内存上限目标是否已经低于了所有租户以及 500 租户的内存分配（租户建立时分配）的总和。memory\_limit 的单位是 MB，例如，如果需要通过调整 memory\_limit 参数来调整

OBSERVER 节点内存上限为64G，可以通过语句 `memory_limit = '64G'` 或者通过 `memory_limit = 65536` 来指定。

另一方面，在使用 OBSERVER 节点的过程中，可能通过 `memory_limit` 来限制 OBSERVER 节点内存的生效参数。若将 `memory_limit` 设置为 0，还可以通过 `memory_limit_percentage` 以比例的形式更灵活地约束 OBSERVER 节点内存的使用情况。

## 1.2.21 在使用 OceanBase 数据库定义资源单元以及资源池时是否允许内存超卖？

在 OceanBase 数据库 V4.x 版本中，不再支持内存超卖，引入内存超卖可能会导致租户工作不稳定。

## 1.2.22 OceanBase 数据库进行一次内存分配会检查哪些限制？



OceanBase 数据库内核限制了单次的内存申请不能超过 4G，这个限制并非是对用户可见的。而是为了内存的优化使用和避免不合理的内存分配在内核中添加的限制。除此之外 OceanBase 数据库内核中每次进行内存申请还会进行如下检查，在出现相应的报错时，请根据对应的报错信息来具体分析：

内存限制	observer.log 日志中报错关键字	问题排查思路
租户下某个上下文 (context) 上限	ctx memory has reached upper limit	当前租户下某个 context 达到上限，此时需要查看在这个上下文中有哪些 mod 占用异常。只有部分 context 有这种限制，如 WORK_AREA，其他的 context 模块与其他内存一起共同受到租户内存的 limit 约束。
租户的内存上限	tenant memory has reached the upper limit	当前租户的内存使用达到上限，需要查看当前租户的哪些上下文内存占用超出预期，找到后再继续细分。
OceanBase 数据库内存上限	server memory has reached the upper limit	OceanBase 数据库的内存总和达到上限，此时需要查看哪些租户内存占用超出预期，找到后再继续细分。
物理内存上限	physical memory exhausted	一般是从物理内存申请不足导致，通常与部署方式以及参数有关。此时需要查看物理内存的大小、observer memory_limit 的配置、物理机上运行的 OBCServer 节点个数以及其他消耗内存的进程的部署，通过这个的方式来拆解整个物理内存的消耗情况。

## 1.2.23 OceanBase 数据库的 KVCache 有哪些，分别有什么作用？

对于 OceanBase 数据库的 KVCache 可以通过 GV\$OB\_KVCACHE 视图进行查询，大体上有一下几种：

- BloomFilter Cache：OceanBase 数据库的 BloomFilter 是构建在宏块上的，按需自动构建，当一个宏块上的空查次数超过某个阈值时，就会自动构建 BloomFilter，并将 BloomFilter 放入 Cache。

- **Row Cache**: 缓存具体的数据行, 在进行 Get/MultiGet 查询时, 经常会将对应查到的数据行放入 **Row Cache**, 这样在进行热点行的查询时, 就能极大地提升查询性能。
- **Block Index Cache**: 缓存微块的索引, 类似于 Btree 的中间层, 由于中间层通常不大, **Block Index Cache** 的命中率通常都比较高。
- **Block Cache**: OceanBase 数据库的 **Buffer Cache**, 缓存具体的数据块, 实际上 **Block Cache** 中缓存是解压后的微块。
- **Partition Location Cache**: 用于缓存 **Partition** 的位置信息, 来帮助对一个查询进行路由。
- **Schema Cache**: 缓存数据表的元信息, 用于执行计划的生成以及后续的查询。
- **clog Cache**: 缓存 clog 数据, 用于加速某些情况下 Paxos 日志的拉取。

## 1.2.24 KV Cache 如何做到的动态伸缩, 以及有什么样的淘汰规则?

可动态伸缩的内存主要部分是 KV Cache, 而 OceanBase 数据库将绝大多数的 KV 格式的缓存统一在了 KV Cache 中进行管理。

KV Cache 一般不需要配置, 特殊场景下可以通过参数控制各种 KV 的优先级, 优先级高的 KV 类比优先级低的 KV 类更容易被保留在 Cache 中。若要调整默认的 KV Cache 相关优先级的默认配置, 需要联系 OceanBase 数据库技术支持工程师。

## 1.2.25 OceanBase 数据库内存有哪些常见问题, 可能导致的原因是什么?

OceanBase 数据库内存有如下常见问题:

- 工作区 (work area) 报内存不足

工作区内存  $\text{limit} = \text{租户内存} * \text{ob\_sql\_work\_area\_percentage}$  (默认5%), 如果请求并发量较大, 且每个请求占用的工作区内存比较多, 可能出现工作区内存不足的报错。经常出现的场景有 union、sort、group by 等。

可以通过适当调大工作区系统变量来规避, 比如

```
set global
```

```
ob_sql_work_area_percentage = 10。
```

- 模块的内存超限

客户端报错 "Over tenant memory limits"，错误码 4030。通常需要进一步分析当前内存消耗的情况。

可以通过 GV\$OB\_MEMORY 视图，也可通过 observer.log 中的日志查看是哪个 mod 消耗的内存绝对占比比较大。

- MemStore 内存用尽

写入的速度大于转储的速度，导致 MemStore 耗尽。比如：大量高并发的数据导入，MemStore 的写操作过快，系统未能及时转储，MemStore 就被用完，返回用户报错 4030。

可以通过分析转储速度缓慢瓶颈、尝试提高转储速度，或者是减缓写入速度来减缓、解决这类问题。

- OBCServer 节点整体内存使用超限

OBCServer 节点进程在启动过程中，会根据配置的参数来计算当前进程最大使用的物理内存上限。如果 memory\_limit 配置数值，那么 OBCServer 节点的内存会直接取 memory\_limit。如果 memory\_limit 为 0，则会通过 物理内存 \* memory\_limit\_percentage 来计算得到内存使用上限。

若观察到 OBCServer 节点内存使用超限，可以查询 \_all\_virtual\_server\_stat 可以看到实际的 OBCServer 节点内的 limit，然后再次检查 memory\_limit、memory\_limit\_percentage 这两个参数是否配置正确。如果还存在内存超限且整体趋势持续增长的情况，那么有可能是遇到了内存泄露的场景，需要联系 OceanBase 数据库技术支持工程师来进行排查解决。

## 1.3 多租户线程 FAQ

### 1.3.26 OceanBase 数据库在运行过程中是单进程还是多进程？

OceanBase 数据库是单进程数据库，运行过程中的主要线程如下：

- election worker：选举线程。
- net io：处理网络 I/O 的线程。
- disk io：处理磁盘 I/O 的线程。
- clog writer：写 clog 的线程。

- **misc timer**：包括多个后台定时器线程，主要负责清理资源。
- **compaction worker**：处理 Minor Merge、Major Merge 的线程。
- **sql worker** 与 **transaction worker**：处理 SQL 和事务请求的线程。

## 1.3.27 OBServer 节点有哪些后台线程，主要是做什么用的？

在大部分场景下，用户无需关注后台线程实现细节。在 OBServer 节点版本迭代中，后台线程也会出现更新的情况。

如下列举出 OBServer 节点的部分常见后台线程，以及其相关功能介绍。

线程名	级别	归属模块	线程数量	功能描述
FrzInfoDet	租户	事务	2	周期性检查是否有新的 freeze_info
LockWaitMgr	租户	事务	1	用于周期性检查超时时间，唤醒等锁的事务
TenantWeakRe	租户	事务	1	租户级别备机读时间戳生成线程
TransService	租户	事务	1	处理事务模块内部若干后台处理的异步任务，推 Ls 的 Checkpoint 等
TransTimeWhe	租户	事务	$\max(\text{cpu\_num}/24, 2)$	处理事务 2PC 流程的定时任务
TsMgr	进程	事务	1	GTS 的后台任务处理线程：删除无用的租户，刷新各个租户的 GTS 等
TSWorker	进程	事务	1	处理远程 GTS 访问返回的结果，回调事务
TxLoopWorker	租户	事务	1	事务模块后台定时任务
ArbSer	进程	系统	1	仲裁 Server 定时从配置文件加载配置参数

Blacklist	进程	系统	2	负责探测与通信目的端 Server 之间的网络是否联通
ConfigMgr	进程	系统	1	用于配置项的刷新
L0_G0	租户	系统	$2 + \min\_cpu * cpu\_quota\_concurrency$	处理大部分该租户请求
L2_G0	租户	系统	1	专门处理嵌套层级为 2 的请求
L3_G0	租户	系统	1	专门处理嵌套层级为 3 的请求
L4_G0	租户	系统	1	专门处理嵌套层级为 4 的请求
L5_G0	租户	系统	1	专门处理嵌套层级为 5 的请求
L6_G0	租户	系统	1	专门处理嵌套层级为 6 的请求
L7_G0	租户	系统	1	专门处理嵌套层级为 7 的请求
L8_G0	租户	系统	1	专门处理嵌套层级为 8 的请求
L9_G0	租户	系统	1	专门处理嵌套层级为 9 的请求
LuaHandler	进程	系统	1	处理应急场景的 Lua 请求以读取 observer 进程内部状态
MemDumpTimer	进程	系统	1	定时打印 MEMORY 日志
MemoryDump	进程	系统	1	定时统计内存信息
MultiTenant	进程	系统	1	负责刷多租户 CPU 配比，用于资源调度
OB_PLOG	进程	系统	1	异步打印 observer 进程诊断日志
pnio	进程	系统	由 net_thread_count 配置参数决定	新网络框架 pkt-nio 的网络 IO 线程

pnlisten	进程	系统	1	监听 RPC 端口以及转发 RPC 连接到网络 IO 线程
SignalHandle	进程	系统	1	信号处理线程
SignalWorker	进程	系统	1	异步处理信号线程
L0_G2	租户	选举	min_cpu, 最少 8 个	专门处理选举请求的线程

### 1.3.28 OBCServer 节点的多租户架构中如何做到 CPU 资源的有效隔离？

在 OBCServer 节点目前发布的版本中，OBCServer 节点主要是依靠控制活跃线程数量（实际消耗 CPU 资源的线程）来控制 CPU 消耗的。OBCServer 节点在创建租户的时候会指定租户资源池，资源池定义 `max_cpu` 可以限制的租户级别的最大活跃线程使用，从而做到租户间 CPU 使用的隔离。在 OBCServer 节点的最新版本中，OBCServer 节点内核实现了 cgroup 的隔离来对 CPU 内存和资源进行有效地控制和限制。

### 1.3.29 如何读取 OBCServer worker 线程的数量呢？OBCServer 节点是否会在负载高的时候动态启动新的线程来执行负载呢？

在 `observer.log` 中以关键字 `dump tenant info` 为主的日志片段里，会描述现在 worker 线程的现有值以及最大值。具体如下：

```
token_count = 分配给该租户的 cpu_count (>min_cpu&&<max_cpu)
_cpu_quota_concurrency。
```

举例一个特定的场景，一台装有 OceanBase 数据库的机器上租户 T1 配置了 `unit_min_cpu = 2, unit_max_cpu=8`，但是在该台机器上配置的 CPU 资源存在超卖。分配给 T1 的实际 CPU 为 5。那么 `token_count = 5_cpu_quota_concurrency`。

### 1.3.30 大查询 CPU 资源分配原则是什么？OLAP 和 OLTP 同时存在的情况下会出现抢占 CPU 资源的情况么？

在使用 OceanBase 时，可以通过配置参数 `large_query_threshold` 来定义执行时间超过一定阈值的查询操作为大查询。如果系统中同时运行着大查询和小查询，OceanBase 数据库会将一部分 CPU 资源分配给大查询，并通过配置参数 `large_query_worker_percentage`（默认值为30%）来限制执行大查询最多可以使用的租户活跃工作线程数。

OceanBase 数据库通过限制大查询能使用的租户活跃工作线程数来约束大查询使用的 CPU 资源，以此来保证系统还会有足够的 CPU 资源来执行 OLTP（e.g.交易型的小事务）负载。通过这样的方式来保证对响应时间比较敏感的 OLTP 负载能够得到足够多的 CPU 资源尽快的被执行。此外需要注意的是，OceanBase 数据库可以做到大查询和 OLTP 资源的分配，`large_query_threshold` 参数也应设置在一个合理的范围内，不应该设置成为一个过大的值。否则大查询很容易抢占掉系统的 CPU 资源而挤进而引发 OLTP 响应过慢甚至队列堆积的问题。



## 2 SQL FAQ

### 2.1 SQL 操作执行 FAQ

#### 2.1.1 如何定位创建 PL 的错误？

可以通过 SHOW ERRORS 命令，查看创建存储过程时的错误信息。

#### 2.1.2 如何分析 PL 的错误日志？

用户只需要关心返回给客户端的错误日志即可。这是由于 OceanBase 数据库会将 Resolve 尝试过程中留下的错误信息记录到日志中；并且如果 PL 中包含了 EXPECTATION OTHERS 语句，则执行到此处时，会在日志中留下错误信息。因此，PL 错误日志中记录的信息是不准确的，可以忽略；用户只需要关心返回给客户端的错误日志即可。

#### 2.1.3 如何查询已经创建的 PL 对象的源码？

可以通过查询 DBA\_SOURCE、ALL\_SOURCE 或 USER\_SOURCE 视图来查询已经创建的 PL 对象的源码，其中，TEXT 列即为 PL 对象的源码。

#### 2.1.4 MySQL 模式是否支持 INSERT ALL INTO 这种语法？

不支持。当前仅 Oracle 模式支持 INSERT ALL INTO 语法，MySQL 模式暂不支持 INSERT ALL INTO 语法。

#### 2.1.5 SQL 引擎、事务引擎等对于租户而言，资源是如何分配和隔离的？

SQL 引擎和事务引擎都是区分租户的，不同租户之间完全隔离。其中：

- SQL 引擎的 Plan Cache 与事务引擎的锁是完全独立的。
- CPU：一个租户 SQL 线程的 CPU 占用，用于控制同时活跃的 SQL 线程。
- 内存：不同租户的 SQL 内存与事务内存分开管理，一个租户的内存耗光，不会影响另一个租户。
- 线程：不同租户的 SQL 引擎与事务引擎的线程完全独立，一个租户的线程挂起不会影响另一个租户。

#### 2.1.6 OceanBase 数据库的 Batch 执行是什么？

当我们使用 JDBC 和 OceanBase 数据库进行交互时，把多次请求放进一个组，从而进行一次网络传输可完成多次请求，这被称为 Batch 执行，通常也被称为“批处理”。

## 2.1.7 为什么要使用 Batch 执行？

有时候我们会因为数据的一致性而使用 Batch 执行，但更多时候，使用 Batch 执行最大的好处是可以提升性能，这体现在以下几个方面：

- Batch 语句会被改写以提高性能；
- Batch 执行可以减少和数据库的交互次数；
- OceanBase 数据库在收到 Batch 执行时可以做一些优化工作，进一步提升性能。

## 2.1.8 JDBC 中哪些 class/object 可以实现 Batch 执行？

无论使用 Statement 还是 PreparedStatement 都可以实现 Batch 执行。

## 2.1.9 使用 Batch 执行需要设置什么 JDBC 配置属性？

- 从功能上说，使用 Batch 执行必须要设置 `rewriteBatchedStatements=TRUE`；
- 从实现和行为上说，`useServerPrepStmts` 会决定 Batch 执行的不同行为；
- 从性能上说，`cachePrepStmt`、`prepStmtCacheSize`、`prepStmtCacheSqlLimit`、`maxBatchTotalParamsNum` 都会对性能有所提升。

如下是有关配置属性的说明：

配置属性	默认值	说明
------	-----	----

allowMultiQueries	FALSE	<p>决定了一条语句中是否可以用 “;” 分割多个请求，Batch 执行并不依赖于这个属性，而仅仅依赖于 rewriteBatchedStatements。</p> <p><b>说明</b></p> <ul style="list-style-type: none"><li>● 如果 JDBC 版本小于等于 1.1.9，那么 allowMultiQueries 必须开启 update 语句才会用 “;” 拼接，否则将报错：Not supported feature or function。</li><li>● 如果 JDBC 版本大于等于 2.2.6，那么 allowMultiQueries 是否开启是没有影响的。</li></ul>
rewriteBatchedStatements	FALSE	<p>决定了 Batch 执行中是否会重写 INSERT 语句。</p> <ul style="list-style-type: none"><li>● 对于 PreparedStatement 对象，会使用多个 VALUES 来拼接；</li><li>● 对于 Statement 对象，会使用分号来拼接多个 INSERT 语句。</li></ul>
useServerPrepStmts	FALSE	<p>决定了是否使用 Server 端的 Prepared 语句，仅对 PreparedStatement 对象有效。</p> <ul style="list-style-type: none"><li>● TRUE 表示使用 Server 端的 Prepared 语句，这需要在 OBCServer 端开启 <code>_ob_enable_prepared_statement</code>，同时意味着使用二进制协议进行通讯；</li><li>● FALSE 表示使用 Client 端的 Prepared 语句，同时意味着使用文本协议进行通讯。</li></ul>

cachePrepStmts	FALSE/TRUE	<p>决定了 JDBC Driver 是否缓存 Prepared 语句，针对 Client 端的 Prepared 语句和 Server 端的 Prepared 语句，缓存的内容稍有不同。</p> <p><b>说明</b> 这个参数在 OceanBase Connector/J 1.x 中默认是 FALSE；在 OceanBase Connector/J 2.x 中默认是 TRUE。</p>
prepStmtCacheSize	25/250	<p>如果开启了 cachePrepStmts，决定了可以缓存多少条 Prepared statements。</p> <p><b>说明</b> 这个参数在 OceanBase Connector/J 1.x 中默认是 25；在 OceanBase Connector/J 2.x 中默认是 250。</p>
prepStmtCacheSqlLimit	256/2048	<p>如果开启了 cachePrepStmts，决定了可以缓存最大的 SQL 是多大。</p> <p><b>说明</b> 这个参数在 OceanBase Connector/J 1.x 中默认是 256；在 OceanBase Connector/J 2.x 中默认是 2048。</p>

maxBatchTotalParamsNum	30000	<p>当使用 executeBatch，决定了最大可以拼接多少个参数。</p> <p><b>说明</b> 这个参数仅在 OceanBase Connector/J 2.2.7 及以后才有。</p>
------------------------	-------	--

## 2.1.10 哪些 OceanBase 数据库的配置项和 Batch 执行有关？

如下配置项和 Batch 执行有关系：

配置项	默认值	范围	生效方式	含义
ob_enable_batched_multi_statement	FALSE	租户	动态	用于设置是否启用批处理多条语句的功能。开启这个参数同时也意味着：在 Batch 执行场景下，当 Client/Server 使用文本协议进行通讯时，OceanBase 数据库会对格式一致的多条 UPDATE 语句当成一条语句进行解析，并根据对应的参数和数据分布，生成 Batch physical plan。
_ob_enable_prepared_statement	FALSE	集群	动态	表示是否可以使用 Server 端的 Prepared 语句。
_enable_static_typing_engine	TRUE	集群	动态	指定是否使用新的 SQL 引擎。新老的 SQL 引擎对于是否能处理 Batch UPDATE 有差别，老引擎只能处理包含全部主键的 Batch UPDATE，新引擎能处理不包含全部主键的 Batch UPDATE。

变量	默认值	级别	含义
_enable_dist_data_access_service	TRUE	SESSION/GLOBAL	打开或者关闭 SQL 以 DAS 的方式执行，若需要获得 Batch UPDATE 的优化能力，需要打开这个变量。

## 2.1.11 Statement 和 PreparedStatement 在行为和使用方法上有什么不同？

- 使用 Statement 对象：

```
conn = DriverManager.getConnection(obUrl);
conn.setAutoCommit(false);
Statement stmt = conn.createStatement();
String SQL = "INSERT INTO test1 (c1, c2) VALUES (1, 'test11')";
stmt.addBatch(SQL);
String SQL = "INSERT INTO test1 (c1, c2) VALUES (2, 'test12')";
stmt.addBatch(SQL);
String SQL = "INSERT INTO test1 (c1, c2) VALUES (3, 'test13')";
stmt.addBatch(SQL);
int[] count = stmt.executeBatch();
stmt.clearBatch();
conn.commit();
```

- 使用 PreparedStatement 对象：

```
conn = DriverManager.getConnection(obUrl);
conn.setAutoCommit(false);
String SQL = "INSERT INTO TEST1 (C1, C2) VALUES (?, ?)";
PreparedStatement pstmt = conn.prepareStatement(SQL);
int rowCount = 5, batchCount = 10;
for (int k=1; k<=batchCount; k++) {
    for (int i=1; i<=rowCount; i++) {
        pstmt.setInt(1, (k*100+i));
        pstmt.setString(2, "test value");
        pstmt.addBatch();
    }
    int[] count = pstmt.executeBatch();
    pstmt.clearBatch();
}
```



```
conn.commit();
pstmt.close();
```

下表列出了使用 PreparedStatement 和 Statement 对象时，Batch 执行的不同行为（前提是 `rewriteBatchedStatements=TRUE`）：

使用 PreparedStatement 对象：

useServerPrepStmts	INSERT	UPDATE	场景
TRUE	多条 INSERT 语句的 VALUES 会以多个 “?” 的形式，拼接在一条 INSERT 语句的多个 VALUES 中，形如： INSERT INTO TEST1 VALUES (?), (?), ..., (?)	多条单独的 UPDATE 语句，其中的变量用 “?” 替代	场景1
FALSE	多条 INSERT 语句的 VALUES 会以多个具体值的形式，拼接在一条 INSERT 语句的多个 VALUES 中，形如： INSERT INTO TEST1 VALUES (1), (2), ..., (10)	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景2

使用 Statement 对象：

useServerPrepStmts	INSERT	UPDATE	场景
TRUE	多条单独的 INSERT 语句用 “;” 拼接在一起	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景3
FALSE	多条单独的 INSERT 语句用 “;” 拼接在一起	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景4

## 2.1.12 OceanBase 数据库的 Batch 执行有哪些类型，分别对请求的优化处理有哪些？

从语句角度看，OceanBase 数据库的 Batch 执行针对 INSERT，UPDATE 和 DELETE 的处理是不同的，具体来说：

### 2.1.12.1 说明

以下所涉及的场景均以 `rewriteBatchedStatements=TRUE` 为前提。

## INSERT

### ● 场景1

使用 `PreparedStatement` 对象：

useServerPrepStmts	INSERT
TRUE	多条 INSERT 语句的 VALUES 会以多个 “?” 的形式，拼接在一条 INSERT 语句的多个 VALUES 中，形如：INSERT INTO TEST1 VALUES (?), (?),..., (?)

在场景1 中，OceanBase 服务器端会收到一次 INSERT 语句的 `COM_STMT_PREPARE` 请求（`request_type=5`）和 一次 INSERT 语句的 `COM_STMT_EXECUTE` 请求（`request_type=6`），从优化的角度看，有如下好处：

- 只需要发生两次通讯就完成了 INSERT 语句的 Batch 执行。
- `PreparedStatement` 天然的属性可以减少编译时间。
- 假设后续有更多的 `executeBatch`，且设置了合理的 `cachePrepStmts` 及相关参数，可以减少 `Prepare` 请求（`request_type=5`）的次数，而只需要执行 `execute` 请求（`request_type=6`）。

### ● 场景2

使用 `PreparedStatement` 对象：

useServerPrepStmts	INSERT
FALSE	多条 INSERT 语句的 VALUES 会以多个具体值的形式，拼接在一条 INSERT 语句的多个 VALUES 中，形如：INSERT INTO TEST1 VALUES (1), (2),..., (10)

在场景2 中，OceanBase 服务器端会收到一次 INSERT 语句的 `COM_QUERY` 请求（`request_type=2`），从优化的角度看，有如下好处：

- 只需要发生一次通讯就完成了 INSERT 语句的 Batch 执行。

### ● 场景3/4

使用 Statement 对象：

useServerPrepStmts	INSERT	场景
TRUE	多条单独的 INSERT 语句用 “;” 拼接在一起	场景3
FALSE	多条单独的 INSERT 语句用 “;” 拼接在一起	场景4

在场景3/4中，OceanBase 服务器端会收到一个由多条 INSERT 语句用 “;” 拼接在一起的请求，并依次执行它们，所以也具有如下好处：

- 只需要发生一次通讯就完成了 INSERT 语句的 Batch 执行。

## UPDATE

使用 PreparedStatement 对象：

useServerPrepStmts	UPDATE	场景
TRUE	多条单独的 UPDATE 语句，其中的变量用 “?” 替代	场景1
FALSE	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景2

使用 Statement 对象：

useServerPrepStmts	UPDATE	场景
TRUE	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景3
FALSE	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景4

- 如果没有开启 `ob_enable_batched_multi_statement`，场景1/2/3/4 的 UPDATE Batch 执行在 OceanBase Server 端都会被依次执行，不会有特别的优化。
- 如果开启了 `ob_enable_batched_multi_statement`，那么对于场景2/3/4 的 UPDATE Batch 执行，OceanBase Server 端会对格式一致的多条 UPDATE 语句当成一条语句进行解析，并根据对应的参数和数据分布，生成 batch physical plan，这可以有效的提升 Batch UPDATE 执行的效率。但在使用这个功能时需要开启显式事务。

DELETE 当前版本对于 Batch DELETE 语句没有优化效果。

## 2.1.13 如何在不同的场景下选择不同的配置？

## 2.1.13.2 说明

请尽量选择较新版本的 oceanbase-client jar 包进行配置。

下表列出了使用 PreparedStatement 和 Statement 对象时，Batch 执行的不同行为（前提是 `rewriteBatchedStatements=TRUE`）：

使用 PreparedStatement 对象：

useServerPrepStmts	INSERT	UPDATE	场景
TRUE	多条 INSERT 语句的 VALUES 会以多个 “?” 的形式，拼接在一条 INSERT 语句的多个 VALUES 中，形如： INSERT INTO TEST1 VALUES (?, ?),..., (?)	多条单独的 UPDATE 语句，其中的变量用 “?” 替代	场景1
FALSE	多条 INSERT 语句的 VALUES 会以多个具体值的形式，拼接在一条 INSERT 语句的多个 VALUES 中，形如： INSERT INTO TEST1 VALUES (1), (2),..., (10)	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景2

使用 Statement 对象：

useServerPrepStmts	INSERT	UPDATE	场景
TRUE	多条单独的 INSERT 语句用 “;” 拼接在一起	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景3
FALSE	多条单独的 INSERT 语句用 “;” 拼接在一起	多条单独的 UPDATE 语句用 “;” 拼接在一起	场景4

**Batch INSERT** 场景1/2 能更有效的发挥 Batch 执行的性能，是推荐的配置，也就是使用如下配置：

- 场景1
  - JDBC 对象：PreparedStatement 对象
  - Server 端参数：

```
_ob_enable_prepared_statement=TRUE
```

- JDBC 配置属性:

```
rewriteBatchedStatements=TRUE
useServerPrepStmts=TRUE
cachePrepStmts=TRUE
prepStmtCacheSize=<根据实际情况>
prepStmtCacheSqlLimit=<根据实际情况>
maxBatchTotalParamsNum=<根据实际情况>
```

- 场景2

- JDBC 对象: PreparedStatement 对象

- JDBC 配置属性:

```
rewriteBatchedStatements=TRUE
useServerPrepStmts=FALSE
```

**Batch UPDATE** 场景2/3/4 使用了文本协议进行通讯, 因此都能利用到多 UPDATE 语句批处理的功能, 是推荐的配置, 也就是使用如下配置:

- 场景2

- JDBC 对象: PreparedStatement 对象

- Server 端参数:

```
ob_enable_batched_multi_statement=TRUE
_enable_static_typing_engine=TRUE
```

- Server 端变量:

```
_enable_dist_data_access_service=1
```

- JDBC 配置属性:

```
rewriteBatchedStatements=TRUE
useServerPrepStmts=FALSE
allowMultiQueries=TRUE
--设置这个是为了避免 JDBC 驱动不同版本间的行为差异
```

- 场景3/4

- JDBC 对象：Statement 对象

- Server 端参数：

```
ob_enable_batched_multi_statement=TRUE
_enable_static_typing_engine=TRUE
```

- Server 端变量：

```
_enable_dist_data_access_service=1
```

- JDBC 配置属性：

```
rewriteBatchedStatements=TRUE
allowMultiQueries=TRUE
--设置这个是为了避免 JDBC 驱动不同版本间的行为差异
```

## 2.1.14 如何查看 OceanBase 数据库的 Batch 执行是否生效？

最常用的方法就是通过 `gv$sql_audit` 来观察 Batch 执行是否生效，以下分几种场景举例：

- 场景1 的 Batch INSERT 如果生效，会在 `gv$sql_audit` 中看到如下记录：

```
query_sql: insert into test_multi_queries (c1, c2) values (?, ?)
request_type: 5
ps_stmt_id: 1

query_sql: insert into test_multi_queries (c1, c2) values (?, ?),(?, ?),(?, ?)
request_type: 5
ps_stmt_id: 2

query_sql: insert into test_multi_queries (c1, c2) values (?, ?),(?, ?),(?, ?)
request_type: 6
ps_stmt_id: 2
```

- 场景2 的 Batch INSERT 如果生效，会在 `gv$sql_audit` 中看到如下记录：

```
query_sql: insert into test_multi_queries (c1, c2) values (1, 'PreparedStatement; re
writeBatchedStatements=true&allowMultiQueries=true&useLocalSessionState=true'
),(2, 'PreparedStatement; rewriteBatchedStatements=true&allowMultiQueries=true
&useLocalSessionState=true'),(3, 'PreparedStatement; rewriteBatchedStatements=t
rue&allowMultiQueries=true&useLocalSessionState=true')
```

- 场景2 的 Batch UPDATE 如果生效，会在 `gv$sql_audit` 中看到如下记录：

```
query_sql: update test2 set c2='batch update1' where c1=1;update test2 set
c2='batch update2' where c1=2;update test2 set c2='batch update3' where c1=3
ret_code: 0
is_batched_multi_stmt: 1
```

### 2.1.14.3 注意

如果 `ret_code = -5787`，这表明 Batch UPDATE 没有生效，需要根据上述说明查找原因。

## 2.1.15 Batch 执行时，executeBatch 方法返回的值是多少？

`executeBatch` 方法被调用后，会返回一个整型数组 `int []`。对于 Batch INSERT 和 Batch UPDATE 来说：

- 如果在 OceanBase 客户端最终是依次执行的，那么这个数组会返回 Batch 中每一个 Operation 所修改的行数。
- 如果在 OceanBase 客户端最终是作为一个整体执行的，比如 JDBC 驱动把多条 INSERT 语句改成了一个 INSERT 语句的多个 values（场景1/2）；又比如 UPDATE 语句作为一个 Batch physical plan 被执行（场景2），那么这个数组的每个元素会返回 -2，表示执行成功但更新行数未知。

## 2.1.16 如何解决 SQL 查询“大小账号”的问题

- 什么是“大小账号”问题：



“大小账号”问题是指在SQL语句解析和执行过程中，由于相同的SQL ID对应不同的执行计划，导致不同查询条件下的性能表现差异显著。例如，当某条SQL语句被参数化后，计划缓存中可能存在一个不适合所有情况的执行计划。这会造成以下影响：

当系统首次执行某条SQL（如 `select * from items where store = 'taobao';`）时，执行计划（p1）可能是全表扫描，适用于选择率极低情况。

如果之后执行另一条参数化SQL（如 `select * from items where store = 'xiaomaibu';`），且选择率很高，原本的计划p1可能就不再是最优方案，导致性能下降。

#### ● 如何识别“大小账号” SQL：

识别“大小账号” SQL 的关键是监控逻辑读行数、影响行数和返回行数的波动。具体的识别规则包括：

- 逻辑读行数的波动超过 1000 行。
- 影响行数的波动超过 1000 行。
- 返回行数的波动超过 10000 行。

用户可以通过手动统计 `v$sql_audit` 等视图的执行记录和统计信息来识别出存在“大小账号”的SQL。

#### ● 如何解决“大小账号” SQL 的问题：

- 关闭计划缓存

##### 1. 单条 SQL 关闭 plan cache：

- 使用 `USE_PLAN_CACHE` hint，例如：

```
SELECT /*+ USE_PLAN_CACHE(none) */ * FROM items WHERE store = 'taobao';
```

- 或设置会话变量 `ob_enable_plan_cache`：

```
SET ob_enable_plan_cache = 0;
```

上述**关闭缓存计划**这种方式均需对应用程序中的 SQL 语句进行修改。如果问题是在应用上线后才被发现，则必须发布新版本的应用程序以解决该问题。您可以参考通过**使用 SQL Outline 绑定 Hint**的方式，由数据库管理员为 SQL 语句添加控制。

- ##### 2. 使用 SQL Outline 绑定 Hint：
- 通过 SQL outline 的方式，由 DBA 控制 SQL 的执行计划。

```
CREATE OUTLINE otl_no_plan_cache1  
ON select /*+ USE_PLAN_CACHE(NONE) */ * from items where store = 'taobao';
```

```
# 或使用 SQL_ID; SQL_ID 可以从 V$OB_SQL_AUDIT 等视图中查到
CREATE OUTLINE otl_idx_c2
ON 'ED570339F2C856BA96008A29EDF04C74'
USING HINT /*+ USE_PLAN_CACHE(NONE) */;
```

## • 开启 SQL Plan Management (SPM)

在 OceanBase V4.2.1 及之后版本中，建议开启 SPM。该机制在某些情况下能够缓解“大小账号”问题，特别是在多个执行计划中存在一个可以接受的单一计划时。SPM 自动演进会保证在“清空 Plan Cache”时计划会有正向演进的效果，改善执行性能。

## • 解决“大小账号”SQL 带来的性能下降问题

■ 人工调优方式：在没有适合的单一执行计划时，可以通过以下方式进行人工调优：

■ **加索引**：通过构建索引来提升查询性能。

■ **cursor\_sharing\_exact hint**：对大账号添加这个 hint，以便让查询得到独立的执行计划。

■ **限制资源消耗**：为了防止“大账号”查询影响系统性能，可以采用以下方法来限制资源消耗：

1. **最大并发数控制**：使用 `MAX_CONCURRENT` hint，例如：

```
CREATE OUTLINE otl_sql_throttle1 ON 'SQL_ID' USING HINT /*+
MAX_CONCURRENT(10) */;
```

2. **SQL级资源隔离**：将大账号查询绑定到资源组，限制其执行中的资源消耗，示例：

```
CALL DBMS_RESOURCE_MANAGER.SET_CONSUMER_GROUP_MAPPING(
ATTRIBUTE => 'column',
VALUE => 'items.store = \'taobao\'',
CONSUMER_GROUP => 'slow_group');
```

您可以通过 Hint 的方式，给特定的 SQL 语句指定资源组：

```
select /*+ RESOURCE_GROUP('slow_group') */ * from items where store =
'taobao';
```

3. **大查询任务自动隔离**: 通过配置项 `large_query_threshold` 来判定大查询, 并限制其 CPU 占用, 以确保小查询得到优先处理。如果同时有大查询和小查询, 大查询最多占用 30% 的租户工作线程, 30% 这个百分比值可以通过配置项 `large_query_worker_percentage` 来设置。

## 2.2 SQL 调优 FAQ

### 2.2.17 数据库物理设计降低查询性能

查询的性能很大程度上取决于数据库的物理设计, 包括所访问对象的 Schema 信息等。例如, 对于二级索引, 如果所需的投影列没有包括在索引列之中, 则需要使用回表的机制访问主表, 查询的代价会增加很多。此时, 可以考虑将用户的投影列加入到索引列中, 构成所谓的"覆盖索引", 避免回表访问。

### 2.2.18 系统负载影响单条 SQL 的响应时间

系统的整体负载除了会影响系统的整体吞吐量, 也会引起单条 SQL 的响应时间变化。OceanBase 数据库的 SQL 引擎采用队列模型, 针对用户请求, 如果可用线程全部被占用, 则新的请求需要在请求队列中排队, 直到某个线程完成当前请求。请求在队列中的排队时间可以在 (G)V\$OB\_SQL\_AUDIT 中看到。

### 2.2.19 代价模型缺陷导致的执行计划选择错误

OceanBase 数据库内建的代价模型是服务器的固有逻辑, 最佳的执行计划依赖此代价模型。因此, 一旦出现由代价模型导致的计划选择错误, 用户只能通过执行计划绑定来确保选择"正确"的执行计划。

### 2.2.20 客户端路由与服务器之间出现路由反馈逻辑错误

obproxy 的一个主要功能是将 SQL 查询路由到恰当的服务器节点。具体来说, 如果用户查询没有指定使用弱一致性读属性, Proxy 需要将其路由到所涉及的表 (或具体分区) 的主节点上, 以避免服务器节点之前的二次转发; 否则, Proxy 会根据预先设置好的规则将其转发到恰当的节点。由于 Proxy 与服务器之间采用松耦合的方式, Proxy 上缓存的数据物理分布信息刷新可能不及时, 导致错误的路由选择。可能导致路由信息变化的场景有:

- 负载均衡导致重新选主
- 网络不稳导致服务器间重新选主
- 由服务器上下线、轮转合并等导致的重新选主

当在 SQL Audit 或执行计划缓存中发现有大量远程执行时，需要考虑是否与上述场景吻合。客户端与服务器之间有路由反馈逻辑，一旦发生错误，客户端会主动刷新数据物理分布信息，随后路由的选择也将恢复正常。

## 2.3 索引监控 FAQ

### 2.3.21 索引监控默认启动，会不会影响性能？

会影响到，但默认（SAMPLED）模式下影响很小，几乎可以忽略。

### 2.3.22 采样模式下的数据是否准确？例如查询了 1 次，是否一定会被记录下来？

不一定，采样的目的是为了过滤部分数据，因此某次查询的记录可能会被丢弃。

### 2.3.23 备库的索引监控是只统计自己还是同步主库的？

统计是租户级的，备库可以查到主库同步的数据，但不能写入。

## 2.4 分区交换 FAQ

### 2.4.24 Oceanbase 数据库 Oracle 模式下，分区表和非分区表都创建了同样的全局索引，为什么分区交换报错，是索引不匹配？

Oceanbase 数据库 Oracle 模式下，在创建索引时，如果不添加关键字 `local`，就会默认创建全局存储的索引，分区交换的前提是分区表的局部存储的索引和非分区表的全部索引一一匹配，否则不能进行分区交换。

### 2.4.25 为什么不能交换二级分区表的一级分区？

当前版本 Oceanbase 数据库的一级分区只是逻辑概念，实际存储层是按照二级分区组织 Tablet，所以对于二级分区表来说，要求二级分区是 Range/Range Columns 分区，对一级分区键格式不要求。

## 2.5 复制表 FAQ

### 2.5.26 什么是复制表？

复制表是 OceanBase 数据库支持的一种特殊表，这种表可以在任意一个副本上读到数据的最新修改。推荐应用于写入频率较低、对读操作延迟和负载均衡要求较高的业务场景。

## 2.5.27 复制表适用场景？

### 2.5.27.1 适用场景一：写入频率较低、读操作延迟和负载均衡要求高的场景

当表的数据量不大，访问又特别频繁的情况下，使用普通表，数据会集中在一个节点，形成热点，影响性能，可以选择用复制表。

典型复制表场景：

- 配置表，业务通过该表读取配置信息。
- 金融场景中的存储汇率的表，该表不会实时更新，每天只更新一次。
- 银行分行或者网点信息表，该表很少新增记录项。

复制表最佳实践：

1. 创建复制表时，建议按需选择，而不是在租户中创建大量复制表。
2. 写入复制表数据时，不建议复制表的写跟读在同一个事务中。
3. 查询复制表时，如果有 JOIN 查询，则按照普通表 JOIN 复制表的顺序设计查询（Query）SQL。

### 2.5.27.2 适用场景二：业务上不能设置为分区表、同时频繁参与跟分区表 JOIN 的场景

出于业务逻辑考虑，某些表不能拆分为分区表或者没有必要拆分（比如查询没有明显的分区条件），但它又频繁的跟分区表有关联关系。为了减少不必要的跨机分布式查询，可以将这类表设置为复制表，每个节点都支持强读，不管分区表的分区的主副本在哪个机器上，它都可以在同机跟复制表的副本做表连接。

## 2.5.28 复制表误用场景？

### 2.5.28.3 误用场景一：为了避免两个分区表的跨节点连接，将其中一个分区表设成复制表

为了提升性能，无脑将 Query 中的分区表设置为复制表，并不科学，这些表不一定是复制表适用场景，可能达不到预期的效果。

- 表存在频繁的写入操作，复制表的写是有性能代价的，需要同步到所有副本，理论上同步是并发的，如果有个别节点，因为网络等原因，同步非常慢，会导致整个复制表的写性能下降。
- 如果复制表数据量非常大，复制表在每个节点都有副本，会占用大量的存储空间。

## 2.5.28.4 误用场景二：事务中含有复制表的写和读

复制表的纯写事务或者纯读事务，都是推荐的，只需要关注业务是否能接受写入延迟。

如果事务中同时存在复制表的写和读，因为需要读 Leader，则会导致事务中所有复制表都退化为普通表的性能，无法发挥复制表可以读取本地副本的优势。

## 2.5.28.5 误用场景三：复制表 JOIN 普通表

ODP (obproxy) 对多表 JOIN 的路由规则是，按照解析到的第一张表路由，如果是复制表，会随机选择一个副本路由。如果此节点恰好不是普通表的 Leader，则会发生远程路由，影响性能。

如果业务中涉及复制表跟普通分区表的 JOIN，建议调整 JOIN 顺序为普通分区表 JOIN 复制表，按照普通分区表路由，按预期生成本地计划，优化性能。

## 2.5.28.6 误用场景四：复制表设置为分区表

复制表本身就是各个节点都会创建副本的，不需要再设置为分区表。

## 2.6 自增列 FAQ

### 2.6.29 什么情况下需要将自增列数据类型设置为 BIGINT?

- 业务本身数据增长快，保留数据周期长。
- 业务不在意建表使用 BIGINT 还是 INT。
- Leader 打散，机器切主的概率变大（例如宕机、random 负载均衡等），自增列数值跳变的概率变大。
- NOORDER 模式，需要显式指定自增列值。

## 2.6.30 什么情况下允许自增列数据类型保持为 INT?

- 业务数据量本身远小于 INT 上限。
- 从 MySQL 迁移的业务，需要使用 INT 类型，否则应用可能存在兼容性问题。
- 单机场景，切主的场景很少。
- 有一定的监控运维能力，自增值个数接近上限时进行运维处理，如重建表导数、或 INT 改 BIGINT（从 V4.2.2 版本开始，列类型由 INT 修改为 BIGINT 为 Online DDL 操作）等。

## 2.6.31 什么情况下可以将自增列改为 NOORDER?

- 当用户无自增列表级有序需求，并期望优化高并发操作的性能时，可以将 ORDER 修改为 NOORDER。
- 当用户有自增列表级有序需求，但是 Leader 都在一个 OBCServer 上时，且期望优化高并发操作的性能时，可以将 ORDER 修改为 NOORDER。

## 2.6.32 什么情况下可以改小自增列 auto\_increment\_cache\_size?

- 跳变问题比较突出。
- 业务流量很低。
- 性能不敏感。
- 对性能有要求，但是为单机模式，Leader 集中在 1 个节点。



## 3 部署 FAQ

### 3.1 安装 OceanBase 服务器对系统有什么要求吗？

服务器应满足的最低配置要求如下表所示。

服务器类型	数量	功能最低配置	性能最低配置
OCP 管控服务器	1台	16C, 32G, 1.5TB 存储	32C, 128G, 1.5TB SSD 存储, 万兆网卡
OceanBase 数据库计算服务器	3台	4C, 16G, <b>3.1.0.1 说明</b> 磁盘中日志盘需要内存的 3 倍以上, 数据盘要满足目标数据量的存储。	32C, 256G, <b>3.1.0.2 说明</b> 磁盘中日志盘需要内存的 3 倍以上, 数据盘要满足目标数据量的存储。 , 万兆网卡

如果需要 OCP 管控服务提供高可用能力, 则需要 3 台管控服务器, 并提供负载均衡软件或者硬件, 例如 F5、阿里云 SLB, 或者使用 OceanBase 数据库提供的 ob\_dns 软负载组件, 进行三节点部署。



支持在下表所示的 Linux 操作系统中安装 OceanBase 数据库。

Linux 操作系统	版本	服务器架构
Rocky Linux	9	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
Alibaba Cloud Linux	2、3	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
龙蜥 AnolisOS	8	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
KylinOS	V10、V11	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
统信 UOS	V20	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
中科方德 NFSCina	4.0	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
浪潮 Inspur kos	5.8	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
CentOS / Red Hat Enterprise Linux	7、8、9	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
SUSE Enterprise Linux	12SP5	x86_64（包括海光）
Debian	12	x86_64（包括海光）
openEuler	20.03 LTS、22.03 LTS	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
凝思 Linux	V6.0.99、V6.0.100	x86_64（包括海光），ARM_64（鲲鹏、飞腾）
Ubuntu	22.04 LTS、24.04 LTS	x86_64（包括海光）

### 3.1.0.3 说明

操作系统需要配置网络和软件管理器（yum 或 zypper 源）。

## 3.2 在生产环境中，OceanBase 数据库会如何部署？

部署方案如下：

方案名称	方案特点	基础设施要求	适用场景
单机房 3 副本	RPO=0, RTO 低, 故障自动切换。可抵御个别硬件故障, 无法抵御机房级灾难或者城市级灾难。	单机房	对机房级容灾能力和城市级容灾能力没有要求。
同城 3 机房 3 副本	RPO=0, RTO 低, 故障自动切换。可抵御个别硬件故障和机房级灾难, 无法抵御城市级灾难。	同城 3 机房。同城机房间网络时延低。	需要机房级容灾能力, 但对城市级容灾能力没有要求。
3 地 5 机房 5 副本	RPO=0, RTO 低, 故障自动切换。可抵御个别硬件故障、机房级灾难和城市级灾难。	3 地 5 机房。其中 2 个城市距离较近, 网络时延低。	需要机房级容灾能力和城市级容灾能力。
同城 2 机房, 集群间数据复制	RPO>0, RTO 高, 人工决策切换。可抵御个别硬件故障、机房级灾难, 无法抵御城市级灾难。	同城 2 机房。	有 2 个同城机房, 同时有机房级容灾的要求。
2 地 3 机房, 5 副本+集群间数据复制	机房级故障: RPO=0, RTO 低, 故障自动切换。城市级故障: RPO>0, RTO 高, 人工决策切换。可抵御个别硬件故障、机房级灾难和城市级灾难。	2 地 3 机房。	有 2 个城市 3 个机房, 同时有机房级容灾和城市级容灾的要求。

## 3.3 什么是 LSE?

LSE 是 ARMv8.1 中引入的特性, 即 "Large System Extensions" (大型系统扩展)。LSE 提供了一组原子操作, 用于支持多处理器环境下的同步和互斥访问。这些原子操作包括 Load-Exclusive (LDE)、Store-Exclusive (STE) 和条件比较交换指令 (Conditional Compare Exchange, CCXE)。LSE 还提供了一些新的指令和内存屏障, 用于维护数据一致性和顺序性。

使用 LSE 指令, 可以实现对共享内存的高效访问, 并提供更细粒度的锁机制和更低的开销。相比于传统的同步指令, LSE 指令可以减少锁竞争、提高并发性能, 并提供更好的可伸缩性。

### 3.3.1 OCP 什么版本开始适配带 nolse 的包?

OCP 从 V4.3.0 版本开始适配具有 nolse 标记的 OceanBase RPM 软件包，主要解决 ARM 架构下 Large System Extensions (LSE) 扩展指令功能识别的问题。

### 3.3.2 OBD 部署时注意事项

可以将带 nolse 和不带 nolse 的 OceanBase 数据库 RPM 包同时上传，OBD 会自适应的根据系统支持的情况进行部署。或者按需上传支持的包进行安装。

### 3.3.3 OCP 部署时注意事项

虽然可以同时将带 nolse 和不带 nolse 的包上传到 OCP，但是不支持自适应的安装，在部署集群选择版本的时候需要按需选择是否支持 LSE 的包。

## 4 集群管理 FAQ

### 4.1 选举 FAQ

#### 4.1.1 OceanBase 集群的选举有哪几种，分别是由谁发起的？

OceanBase 集群在实现上主要包含无主选举、有主连任、有主改选(切主)这几个部分。其中无主选举是在集群启动或者 Leader 连任失败的时候才会进行。除了手动切主以及无主选举，其他的自动选举均是由 Paxos 组的 Leader 发起的。在组成了一个 Paxos 集合的 3 个成员副本中，切主动作由每个 Paxos 组的 Leader 控制，当 Leader 在续约流程中发现某个 Follower 比它的优先级更高时它就执行切主流程“让贤”。

#### 4.1.2 什么样的副本可以参与选举？

在 V4.x 版本中，OceanBase 数据库的副本类型有全能型副本（FULL）、只读型副本（ReadOnly）、列存副本（COLUMNSTORE）。只有全能型副本可以作为选举中 Leader 的候选者（Candidate），变为主提供数据库服务。

#### 4.1.3 OceanBase 集群的选举模块的对基础设施的要求有哪些？

从 OceanBase 数据库 4.0.0 版本开始，选举对于时钟偏差没有要求，由于 OceanBase 集群采用强 Leader 模式，有租约时长的限制，默认设置下，租约时长为 4s，要求环境中任意两副本的应用层最大单程消息延迟不可超过 1s。但是在实际环境中由于网络分层的原因以及操作系统和上层应用的实现问题，IP 层的消息延迟会在应用层中被放大，而且这个放大的倍数没有指标可以参考，因此需要保证环境中的 ping 延迟尽量低。

#### 4.1.4 OceanBase 集群的选举模块是如何保证 RTO 不大于 8s 的？

在默认设置下，OceanBase 集群中选举 Leader 的租约时长为 4s，当 Leader 宕机后，最长可能延迟 4s 的时间直至其租约到期，多数派副本在等待租约到期后会尽快开启下一轮选举，选出参考优先级后集群中最适合当选的副本成为新的 Leader。同时选举模块为 Leader 宕机后的无主选举做了特殊优化，选举层 Leader 宕机后的平均无主时间约等于租约时长。需要注意的是从选举成功到集群服务恢复之间，还需要一系列额外的工作流程：如新 Leader 需进行 Paxos 的 reconfirm 阶段补全日志（该阶段的耗时取决于新 Leader 落后于旧 Leader 的日志数量，选举保证尽可能不选到日志落后过多的副本）；在新 Leader 上恢复尚未提交的旧 Leader 的事务状态；向 Root Service 汇报 Leader 的位置信息，如果配置了 obproxy 还需

要等待 Proxy 更新正确 Leader 位置信息后才能将后面的 SQL 语句路由到正确的 Leader 位置上。OceanBase 集群选举模块能够做到：如果原主副本连任成功，原主可以连续提供主副本服务；如果原主副本下线，Leader 连任失败，OceanBase 集群进行无主选举，新主上任的时间在 8s 内完成。在 8s 的 RTO 时间内，选举层面将会消耗约 4s 的时间，剩余时间用于恢复其他阶段。其他阶段的恢复流程尽管和日志的数量差、未提交事务的规模呈正相关，但通常各阶段都不超过百 ms，总体的服务恢复时间可以保证在 8s 内完成。

### 4.1.5 如果多数派副本宕机，OceanBase 集群的选举模块是否还可以选出 Leader 副本保证高可用？

当 OceanBase 集群中的多数派副本同时宕机时，OceanBase 集群无法再选举出主副本（Leader 副本）对外提供数据服务。

### 4.1.6 OceanBase 集群的选举模块是否需要做任何运维操作？

实际上，OceanBase 数据库没有对用户提供任何干预选举模块的方法或者接口，比如控制多数派的选举规则等。在集群的正常运行过程中，不需要数据库管理员或者管理员进行任何人为的操作或者干预来控制或影响 OBServer 节点的选举模块完成其功能。在进行集群管理动作时，比如说停掉 OBServer，上线新的 OBServer，修改租户的 Primary Zone 设置等，都有可能触发选举模块进行选举，以保证租户中的数据副本存在主副本作为 Leader 对外提供数据服务。目前为止没有任何的系统运行情况需要管理员直接对选举模块进行运维干预。

### 4.1.7 在选举中哪个副本会被选为主，如何保证选举到的 Leader 副本是更好的选择？

OceanBase 集群的选举会比较副本的优先级，保证尽量选出来一个健康状况更好更优的机器作为 Leader 副本。目前 OceanBase 集群的优先级主要考虑了副本是否是 Leader 的合法候选人，副本健康分，副本成员列表版本号，以及副本日志同步状况四个维度。这也意味着通过 OceanBase 集群自动触发的选举最终会让更健康，版本推进最新的合法选举副本被选为主。在 OceanBase 数据库 4.0.0 版本中，无主选举和有主改选优先级统一，成员变更版本号作为最高优先级，同时也是唯一一个内嵌在选举协议内部的优先级，版本号大的副本不会给比自己版本号小的副本投票。

### 4.1.8 在什么情况下会有自动切主？

Leader 副本在延续租约的过程中，会周期性的与 Follower 副本进行交互，在交互过程中如果发现 Follower 副本具备更高的优先级，Leader 就会触发切主动作，“让位”给对应的 Follower。常见发生切主的原因有：

- `primary_zone` 发生变更。
- 发生 Stop Server/Stop Zone 的操作。
- 发生 OBServer Stopped（即 kill -15）的操作。
- 发生日志流的迁移。

## 4.1.9 如何查看触发切主的原因？

可以通过 `DBA_OB_SERVER_EVENT_HISTORY` 视图来查看某个日志流选举模块发生的事件。

## 4.1.10 如何查看无主选举？

可以通过 `DBA_OB_SERVER_EVENT_HISTORY` 视图查看，表里记录了所有日志流的所有选举事件，包括无主选举、切主、成员变更等。

## 4.1.11 OceanBase 集群报错 -4038 是什么含义？

4038 错误码的定义是 `not master`，在发生 4038 错误后有两种可能：

1. 当前 Leader 存在，但是 Leader 不在运行进程所在的 Server 上(需要排查上层模块问题)。
2. 当前 Leader 不存在。根据排查问题的经验基本上是网络连通状态有问题（这里指的是日志流级别的连通状态），例如：
  - 因为租户没有内存接收不到消息，但是还能发送消息，导致单向网络连通。
  - 因为工作队列积压导致无法接收消息。
  - 因为 Server 处于 Stop 状态无法接收消息。
  - 其他原因导致的网络连通性问题。

## 4.1.12 CLOG 模块会影响副本选举吗，是如何影响的？

OceanBase 数据库 V4.0.0 版本 `clog` 的异常状态会作为选举优先级的一部分，Leader 的 `clog` 如果出现异常将会降低 Leader 选举优先级，当有 Follower 的优先级比 Leader 更高时 Leader 将会触发切主。

## 4.1.13 CLOG 在发生主动卸主后会重新选主么？



OceanBase 数据库 V4.0.0 版本不再有 clog 发起的主动卸任动作，如果 clog 写盘 hung 住，将会记录 failure 事件，Leader 的优先级会被降低，其后会自动选择最优副本直接切主，切主事件将会记录在 DBA\_OB\_SERVER\_EVENT\_HISTORY 视图中。

## 4.2 CLOG FAQ

### 4.2.14 OceanBase CLOG 同步压缩功能中不同的压缩算法有什么区别，应该怎么选择？

OceanBase 数据库通过 `log_transport_compress_func` 来控制事务日志同步的压缩算法，默认为 `lz4_1.0`。在开启日志同步压缩后，clog 会按照算法来对日志进行压缩传输。目前 clog 支持的压缩算法有 `lz4_1.0`，`zstd_1.0`，`zstd_1.3.8`。在不同的压缩算法下会对 clog 进行不同比例的压缩并传输，进而减轻对带宽的压力。但与此同时，由于 clog 的处理都需要先进行压缩操作，所以会带来一定的性能代价。在不同的场景下，不同压缩算法带来的压缩收益以及性能代价会有不相同。默认值为 `lz4_1.0` 压缩算法是在保证可控的性能代价下相对能够带来较大的压缩比例。因此 OceanBase 数据库的配置参数 `log_transport_compress_func` 设置了默认值为 `lz4_1.0`。如果在不同的场景下，有更高的压缩比率追求或者能够容忍更程度的性能代价，可以选择不同的压缩算法。

### 4.2.15 事务执行到提交的过程中，数据更改是如何提交到 MemStore，持久层并保存在 clog 中的？

当一个数据库事务包含了 DML 的请求，数据更改会更新到 Leader 副本的 MemTable 中。在事务 Commit 成功以后，OceanBase 数据库的事务处理能够保证数据更改在多数派的 clog 中落盘并会回放到 Follower 副本的 MemStore 中。在 OceanBase MemStore 的使用超过一定阈值之后，会进行转储，将 MemStore 持久化到 SSTable 上。下面简单介绍在存储层（MemTable，SSTable）以及 clog 层所发生的关键动作：

1. 应用发起一个事务 Tnx1；
2. 假设 Tnx1 事务中会顺序执行 DML 请求 DML1 和 DML2；
3. DML1 和 DML2 通过执行 SQL 层，更新事务的上下文，并将数据更改更新到 MemTable 中；
4. DML1 和 DML2 在 SQL 层执行成功，应用程序发起 COMMIT，Tnx1 进入事务提交阶段；
5. Leader 副本发起 clog 落盘，并同时 will clog 同步到 Follower 副本；
6. 当 Leader 和 Follower 形成 clog 多数派落盘时，事务 Commit 成功，返回应用端；

7. Follower 副本在接收到 clog 且落盘后，还要等待 Leader 副本发送的“clog 已完成多数派落盘”的确认消息，才会开始异步回放，将 clog 的内容回放到 MemStore 为弱一致读提供数据服务。

## 4.2.16 如果 OBServer 节点的滑动窗口卡住或者满掉，会出现什么报错，会造成什么后果，应当如何调优？

clog 滑动窗口（Sliding Window）满主要分为下面两种情况：

clog 滑动窗口满掉场景	后果及关键日志
Leader 分区的 clog 滑动窗口满	Leader 分区无法再继续分配 log_id 给事务层，并且会在 observer.log 中打印 submit log error.*\\-4023 消息。如果 Leader 的 clog 滑动窗口出现满掉且持续没有自动缓解掉的情况，会导致事务提交无法再将请求放入到 clog 滑动窗口，导致事务的响应时间（RT）值整体升高的问题。
Follower 分区的 clog 滑动窗口满	Follower 分区会暂停接收 Leader 发过来的 clog，并且会在 observer.log 中打印 check_can_receive_log, now can not receive log 消息，直至有 clog 从窗口中滑出。如果长时间的 Follower 节点滑动窗口满掉且没有自动缓解，会影响多数派的 clog 提交，进而会影响事务提交。如影响了少数派的 clog 提交，会导致 Follower 节点的 clog 一直落后。如果 Follower 的 clog 持续落后那么在其他少数派副本出现故障的时候，会影响该副本不能被有限选举为 Leader 副本。

从整体上来说，如果 clog 滑动窗口满了，会影响 clog 的持久化的推进。这种情况下，应用端会感受到 slow trans 或者“锁等待”（日志中会出现 -6005 和 lock\_for\_write conflict.\*\\-4023）等问题，性能也会受到明显影响。分析的思路有如下几种：

- 分析 clog 提交写入是否大幅的增长变化，比如系统新上线一个高并发的跑批场景，对 clog 滑动窗口的并发压力非常大，那么需要先从业务逻辑检查高并发的跑批是否是业务需要的，如果业务允许，可以尝试降低并发测试问题进行缓解。这通常是比较安全且最快得到优化效果的一条路径。
- 如果业务并发并没有突然的激增，但是遇到了该问题，需要诊断 clog 滑动窗口的滑出是否有瓶颈，比如遇到 Follower 由于转储慢内存不足，日志回放不进内存，有可能导致 clog 滑动窗口满掉，如果遇到该种场景，需要联系 OceanBase 数据库技术支持进行分析。



- 拆分热点分区

由于所有的滑动窗口大小都是以分区为单位设置的，显然数据的热点越集中就越容易把滑动窗口撑满，因此将数据的热点打散到多个表或者分区，可以有效降低滑动窗口满的概率。除了我们经常关注的主表热点分区之外，还要特别关注“全局”索引的热点情况，包括：

- 全局“非分区”索引，索引数据集中在一个分区上；
- 分区键不合适的全局分区索引，比如索引键上的 NDV 过小，或者有严重的数据倾斜（某些键值的记录数过多），也会导致索引数据集中在少量甚至一个分区上。

## 4.2.17 CLOG reconfirm 失败的场景以及失败后有什么影响？如何分析 CLOG reconfirm 失败的情景

OceanBase 数据库是一个分布式数据库，分区要正常提供读写服务，必须选出一个 Leader，整个选主的流程大致如下：

1. OceanBase 数据库的选举模块选出副本 Leader。
2. clog 模块从选举模块得知当前 Leader，Leader 在上任前执行 clog reconfirm 过程，确保数据是最新的。
3. Leader 执行 Takeover 等操作后，正式上任。

如果新主从 OceanBase 数据库选举模块已经被选出，但是 clog reconfirm 由于一些原因导致失败了，那最终对应的数据分区也是无法提供数据服务。

clog reconfirm 过程的主要目的是对于滑动窗口中遗留的未确认日志进行重确认，并同步给 Follower 副本。OceanBase 数据库在事务提交的过程中，如果 clog 已经成功同步到了多数派，事务就已经提交成功，若此时 Leader 宕机，有可能还在线的副本里就有 clog 还未完成确认回放到 MemTable。那么 clog reconfirm 就保证了在 Leader 切换的场景下，所有副本的 clog 都已经完成了确认，还可以保证多数派的 clog 同步一致状态，保证接下来的事务提交不受影响。clog reconfirm 过程包含如下几个阶段：

阶段	阶段名称	阶段动作
1	INITED 阶段	尝试将已确认日志提交回放，本地写 prepare 日志和本次 Leader 选举的标记信息。
2	FLUSHING_PREPARE_LOG 阶段	发送 Leader 选举的标记信息给所有 Follower，等待 follower 回复各自的日志，收到多数派回复后进入下个阶段。
3	FETCH_MAX_LSN 阶段	等待副本最大提交日志 <code>max_flushed_log_id</code> 达到了多数派(含自己)，则统计收到的最大 <code>max_flushed_id</code> 值，并做一些准备工作。
4	RECONFIRMING 阶段	首先尝试将滑动窗口左侧已确认的 log 从滑动窗口滑出并提交回放，直到遇到第一条未确认 log 为止，然后对所有未确认日志执行重确认过程，即从多数派成员收集、确定该日志内容，然后同步给所有 follower；最后进入 START_WORKING 状态，写一条 start_working 日志。
5	START_WORKING 阶段	等待 start_working 日志形成多数派。
6	FINISHED 阶段	结束。

在上面 6 阶段中的关键阶段会将日志打印到 observer.log，例如：阶段 2 收到多数派的回复的标志日志是（如果无法看到该日志说明本阶段被卡住了）：

```
[20XX-XX-XX 20:28:06.142985] INFO [CLOG] ob_log_reconfirm.cpp:598 [127240][Y0-000000000000000000] [lt=25] max_log_ack_list majority(partition_key={tid:1099511627777, partition_id:0, part_cnt:1}, majority_cnt=2, max_log_ack_list=1 {server: "10.101.XXX.XXX:261XXX", timestamp:-1, flag:0})
```

单条 clog reconfirm 过程的超时时间是 10s，即如果 10s 没有执行完会打印 ERROR 级别的超时日志，关键字为 `is_reconfirm_role_change_or_sync_timeout`。clog reconfirm 通常由于卡在如上各个阶段而超时，卡住的原因主要有：无法提交回放、网络不通、日志盘满等。clog reconfirm 卡住的主要原因和关键日志信息有

主要原因	原因解释	关键日志及诊断方法
clog 回放卡住	reconfirm 过程中需要将滑动窗口左侧已经确认的日志全部提交回放，一旦回放由于各种异常（内存分配失败、回放出错）卡住的话，滑动窗口就会发生积压，积压的日志数量达到阈值(目前是 2万)后，滑动窗口就满了，无法接收新的日志，reconfirm 也就无法正常执行下去。	<p>如果滑动窗口中有已确认的日志需要提交回放，则可以看到 observer.log CLOG 模块报 [ERROR] 级别错误码 -4023，日志的关键字是：</p> <pre>there are confirmed logs in sw, try again(partition_key= {tid:1106108697592670, partition_id:0, part_cnt:1}, ret=-4023, new_start_id=371801833, start_id=371802703)</pre> <p>回放卡主的原因可能很多，诸如 alloc memory failed 这样的内存分配错误、replay error 这种回放错误。需要进一步通过 observer.log 进行排查 <code>grep ERROR observer.log   grep STORAGE   grep 'alloc memory'</code>、<code>grep ERROR observer.log   grep STORAGE   grep replay</code></p>

clog 满盘	clog reconfirm 过程中需要写 prepare、start_working 等日志，且需要达成多数派确认，如果 Server 的 clog 盘满了导致无法写这些日志，且余下的正常 Server 不够多数派，那么 reconfirm 也会失败。	<p>如果 clog 发生满盘，observer.log 里 CLOG 模块报 [ERROR] 级别错误码 -4264，日志的关键字是：</p> <pre>log outof disk space (partition_key={tid: 1102810162709414, partition_id:46, part_cnt:0}, server= "xx.xx.xx.xx:29432" , ret=-4264, free_quota=- 3381817344)</pre>
clog 同步网络不通	clog reconfirm 过程中候选 Leader 需要与各个 Follower 在多个阶段进行通信，因此如果网络出现问题，导致无法收发包或收发包延迟太大也可能导致 reconfirm 失败。	<p>网络不通会导致 reconfirm 多个阶段均会失败，比如发送完 PREPARE 日志后一直收不到 Follower 的回复，会导致本机一直打印如下关键字的日志直至超时失败：</p> <pre>max_log_ack_list not majority</pre> <p>后续的写 start_working 日志也可能由于网络异常而导致无法收到多数派 ack，进而超时失败。很多原因可能导致网络不通，排查步骤：</p> <ol style="list-style-type: none"> <li>1. 执行 ping，如果能通说明网络正常；</li> <li>2. 执行 sudo iptables -L，看是否有网络隔离；</li> <li>3. 查看对端租户队列日志，看是否队列满了，无法处理请求。</li> </ol>

500 租户请求队列堆积	<p>如果 500 请求队列发生堆积，无法处理新的 RPC 请求，reconfirm 相关的消息处理也会停滞。Leader 端日志显示，Follower 一直没有回复任何请求。这个时候需要进一步分析队列堆积情况和队列堆积原因。</p>	<p>查看队列堆积情况：搜索 Leader/Follower 的 observer.log，关键字是 <code>dump tenant info</code> (<code>tenant={id:500}</code>)，通过搜索关键字，可以看到 500 租户的队列情况，其中 <code>req_queue</code> 字段后跟的内容就是队列情况。</p> <p>通常在看出 500 租户队列请求队列堆积的情况下还需要一进步分析队列堆积的原因，这时候可以用 <code>obstack</code> 来打印当前的 OBSERVER 节点堆栈信息，并将信息一并发给 OceanBase 数据库技术支持进行问题诊断。</p>
--------------	---	---

工作线程不足	-	<p>看是否存在如下 easy 包处理超时的日志：</p> <pre>[20XX-XX-XX 11:12:12.103] easy_request.c:420(tid: 7fe2cdc9f700)[57136] rpc time, packet_id :2425446521, pcode :4096, client_start_time :1517541127590356, start_send_diff      :8, send_connect_diff:    0, connect_write_diff:   5, request_fly_ts:       188, arrival_push_diff:    2, wait_queue_diff:     4508603, pop_process_start_diff :2, process_handler_diff: 4184, process_end_response_diff: 1, response_fly_ts:   447, read_end_diff:        1, client_end_time       : 1517541132103797, dst: 10.101.X.X:XXXXX</pre> <p>可以看到 wait_queue_diff 达到了几秒级别，说明工作线程处理比较慢，检查 cpu_quota （cpu_quota_concurrency、workers_per_cpu_quota）相关配置项是否配置过小。</p>
--------	---	--

如果根据以上诊断方法无法排查出 clog reconfirm 失败的原因，那么请联系 OceanBase 数据库技术支持进行诊断。

## 4.2.18 CLOG 满盘应该如何进行问题分析和应急操作？

对于分区数多、写入压力大的集群，如果转储慢（卡住）、或者租户 Unit 规格异构，那么可能导致部分副本的 clog 回收不及时导致盘空间达到 95%，此时 OBServer 节点会自动停写，这样这台机器上会有大量副本不同步。正常情况下，clog 盘空间会维持在 80% 左右，一旦超过 80%，说明日志文件回收慢或者卡住了，clog 文件回收的条件是：该文件中所有分区日志对应的数据都已经转储到 SStable 中。因此通常导致 clog 盘满的原因是部分分区转储卡住。在 clog 盘空间开始报警之后，根据下面的步骤确认阻塞回收的分区：

1. 从以下命令的结果中查看：

```
grep can_skip_base observer.log
```

- 从搜索结果中查找 `need_record` 为 True 的记录，里面有 `partition_key`，表示这个分区的日志无法回收。
- 从报警日志附近时间点开始搜索，搜索结果中的分区就是当前阻塞回收的分区，拿到其中的 `partition_key`；

2. 利用上面的 `partition_key` 去搜索该分区的日志：

```
grep partition_key observer.log
```

通过日志分析它转储位点未推进的原因。

## 4.2.18.1 应急手段

在出现了 clog 满盘的情况下，如果已经影响到多数派节点，影响了事务提交，即整个系统都出现了 Hang 和无响应的情况，业务侧还有大量并发 DML 的事务涌入，建议先将业务负载暂停，恢复 clog 服务。

如果 clog 满盘的操作只影响了少数派，请先快速分析节点是否有 IO 层，网络层（基础设施层）的重大瓶颈。

若有可将少数派节点先隔离后将 IO 瓶颈和网络瓶颈进行解决。

如果没有 IO 层和网络层（基础设施层）的瓶颈，仍然可能在少数派节点上出现问题，请联系技术支持人员协助排查。

## 4.2.18.2 可能原因

- 运维问题。

clog 盘所在空间被其他用户的文件占用，导致可以利用的空间不足。

- 转储持续失败是 clog 满盘最常见的原因。

当 clog 中涉及的分区数据已经全部转储到 SSTable 中，且 partition 的元信息已经持久化宕机重启起始位点时，clog 文件就可以进行回收。所以，如果转储不成功，宕机重启 clog 起始回放位点不推进，CLOG 文件是无法进行回收的。因转储持续失败的原因场景和具体原因逻辑相较比较复杂，遇到该种情景，请联系 OceanBase 数据库技术支持进行介入诊断。

## 4.2.19 如果 OBServer 节点重启时在初始化 CLOG 失败会报什么错，如果 OBServer 节点启动时初始化失败报错应当如何操作恢复系统？

报错描述	报错日志	恢复方式
clog_shm（共享内存文件中记录 flush_pos_ 大于 clog 最后一个文件的有效位置）	observer.log CLOG 模块报出 [ERROR] 级别 -4016 错误码，报错关键字是 The clog start pos is unexpected, (ret=-4016, file_id=1018, offset=51658630	删除 store 目录下的 clog_shm 文件之后重新启动。
		这类报错通常是因为 clog 文件出了问题，或者迭代 clog 文件出了问题： <ul style="list-style-type: none"><li>如果是少数派副本出现如上问题，可以通过删除出错副本恢复。</li></ul> <p><b>说明</b></p> <p>需要特别注意的是：只有当少数派副本出现这样的问题后，可以通过该方式恢复。</p> <ul style="list-style-type: none"><li>如果多数派出现了上述问题，请联系 OceanBase 数据库技术支持，切勿私自操作，该动作操作不当可引发数据丢失。</li></ul>
	observer.log CLOG 模块报出 [ERROR] 级别 -4016 错误码，日志报错关键字有：	



迭代 clog 文件出错，这通常是由于 clog 文件本身出了问题

- `invalid`  
`scan_confirmed_log_cnt`  
`(ret=-4016, ret="`  
`OB_ERR_UNEXPECTED",`
- `notify_scan_finished_`  
`finished(ret=-4016,`  
`cost_time=154)`
- `notify_scan_finished_ failed`  
`(ret=-4016)`
- `log scan runnable exit error`  
`(ret=-4016)`

例如 3 副本的环境中，其中一个副本出错。

#### 恢复方式1：

找一台类似机器快速加入到 OceanBase 集群中，OceanBase 集群自动补齐第三副本。需要确认配置项

`enable_rereplication` 处于打开状态。出问题的机器最终会永久下线。

#### 恢复方式2：

找出问题的机器走永久下线流程并安全清空故障机器上的数据，重新以新机器方式重新加入集群。

1. 永久下线，执行如下命令下线故障机器，查询

`DBA_OB_SERVERS`，没有下线机器对应记录，则下线操作完成。

```
alter system delete server
'$obs0_ip:$obs0_port';
```

2. 清空故障机器上的数据。
3. 以新机器的方式重新加入集群。

```
alter system add server
'$obs0_ip:$obs0_port';
```

4. 之后集群会发起补副本操作，补齐第三副本，同样需要确认 `enable_rereplication` 处于打开状态。

## 4.3 存储 FAQ

### 4.3.20 OceanBase 数据库的存储引擎和传统数据库有什么不一样的地方？

OceanBase 数据库采用了一种读写分离的架构，把数据分为基线数据和增量数据。其中增量数据放在内存里（MemTable），基线数据放在 SSD 盘（SSTable）。对数据的修改都是增量数据，只写内存。所以 DML 是完全的内存操作，性能非常高。读的时候，数据可能会在内

存里有更新过的版本，在持久化存储里有基线版本，需要把两个版本进行合并，获得一个最新版本。同时在内存实现了 Block Cache 和 Row Cache，来避免对基线数据的随机读。当内存的增量数据达到一定规模的时候，会触发增量数据和基线数据的合并，把增量数据落盘。同时每天晚上的空闲时刻，系统也会自动每日合并。

## 4.3.21 数据在 OceanBase 数据库中是如何存储的？

OceanBase 数据库的数据文件以宏块（Macro Block）为单位组织数据，每个宏块大小为 2 MB。宏块内部又划分出很多个 16 KB（压缩前的大小）大小的微块（Micro Block），而每个微块里面包含多个行（Row）。OceanBase 数据库内部 IO 的最小单位是微块。

## 4.3.22 应该使用分区表吗？

是否使用分区表，可以参考以下信息进行判断：

- 如果一张表的数据量在可预期的未来会突破 200 GB，建议使用分区表。
- 如果表具有比较明显的时间周期，建议使用分区表。

## 4.3.23 在 OceanBase 数据库中，局部索引与全局索引在实现上的区别是什么？

局部索引与全局索引的区别如下：

- 局部索引：和单表或者分区表最小子分区绑定在一起，不会在 OceanBase 数据库系统租户中创建额外的分区记录，所有基于局部索引的 SQL 基本都是本地执行。
- 全局索引：需要在 OceanBase 数据库系统租户中创建额外的一个分区记录，可以理解为另外一张表，占用额外的分区配额。全局索引可以有自己的分区策略，Locality 也不与主表绑定在一起，即使是单表，全局索引也可以与主表分别存储在不同的节点上，当然这样也更容易使 SQL 成为分布式 SQL。

## 5 高可用 FAQ

### 5.1 什么是系统的高可用？

IT 系统的高可用（High Availability）指的是系统无中断地执行其功能的能力，代表系统的可用性程度。高可用是在系统设计、工程实践、产品能力各个环节上的综合作用，保证业务的持续连续性。而保障系统高可用最重要的焦点就是尽量消除或者冗余单点故障（Single Point of Failure），并且在单点或者系统出现不可用之后提供急速恢复的能力。企业级应用为了保证业务的连续可用，通常会对系统的可用性有很高的要求，需要保证在故障或者灾难发生时，系统的 RTO 尽量低，RPO 尽量接近 0。

高可用是分布式系统设计中必须考虑的因素，OceanBase 数据库作为一款原生的分布式数据库能够对外提供一致的、高可用的数据服务。OceanBase 数据库的事务一致性，存储持久化保证在出现 OBServer 节点退出重启情况下能够恢复到重启前相同的数据和状态。另外，OceanBase 数据库的备份恢复、主备库解决方案也保证了在不同场景下 OceanBase 数据库的高可用能力。

### 5.2 OceanBase 数据库有哪些产品能力保证了数据服务的高可用？

产品能力及解决方案	故障场景	工作原理
-----------	------	------

OceanBase 分布式选举	<p>OceanBase 集群少数派因为各种原因造成不可用的故障恢复：</p> <ul style="list-style-type: none"><li>● 比如3副本3台 OBCServer 节点组成的 OceanBase 集群中1台 OBCServer 节点宕机、退出。</li><li>● 部署在不同机房的少数派副本机房故障；</li><li>● 部署在不同城市的少数派副本城市发生灾难。</li></ul>	<p>OceanBase 集群的选举模块保证选举出唯一的主副本对外提供数据服务。同时，通过 Paxos 协议实现了多数派 clog 强一致同步持久化，在 Paxos 组中任意少数派副本发生故障的情况下，剩下的所有多数派 clog 合在一起后，保证有完整的 Clog，因此就能避免个别硬件故障带来的数据损失，保证了数据可靠性。</p> <p>当整个 OceanBase 集群中的少数派出现故障时，如果是非 Leader 副本的少数派不可用，不会影响系统的可用性和数据库的性能。如果少数派故障中有 Leader 副本，OceanBase 集群能够保证从剩余副本选出唯一新主提供数据服务，主要基于 OceanBase 集群的搭建部署模式，如果 OBCServer 节点的多个 Zone 分布在不同机房、不同城市时，通过 OceanBase 集群的分布式选举加上 OceanBase Paxos clog 同步就可以达到跨机房高可用或者是跨城市的高可用方案。</p>
		<p>OceanBase 集群的存储引擎将基线数据保存在 SSTable，增量数据保存在 MemTable 中。OceanBase clog 是数据的 redo log。当有数据修改发生时，请求会到达 Leader 副本所在节点，将数据更改请求更新到 MemTable 中，事务进行提交，Leader 更新本地 clog 并通过 Paxos 协议将日志同步写到其他 Follower 的副本节点上。当有多数派节点的日志写盘成功后，数据修改成功，返回客户端。</p> <p>Follower 副本会将 clog 回放放到本地 MemTable 上提供弱一致读服务。当 MemTable 到达阈值后，会触发冻结和转储，持久到 SSTable 层。而此时的 clog 回放位点会推进，类似于做了 Checkpoint。在 OBCServer 节点重启时，能够做到从 SSTable 中源数据还原、更新到最新信息。</p>

OceanBase clog 及存储引擎	<ul style="list-style-type: none"><li>● OBServer 节点多数派故障需要重启;</li><li>● 维护计划内的 OBServer 节点重启。</li></ul>	<p>然后，从分区的元信息中获取 clog 的回放位点，开始回放 clog 日志生成 MemTable 中。至此，OceanBase 集群可以将磁盘中的持久化信息恢复到宕机前的状态，保证了数据的完整性。</p> <p>当 OceanBase 集群由于故障（软件退出、异常重启、断电、机器故障等）重启或者计划内停机维护重启时，OBServer 节点能够在启动中恢复，将 OBServer 节点 store 目录下的日志和数据还原到内存中，将进程的状态恢复到宕机前的状态。如果多数派副本故障并需要重启，数据服务会发生中断，OceanBase 集群保证在多数派宕机重启后数据可以完全恢复到宕机之前。</p> <p>OBServer 节点对于多数派宕机重启也进行了进一步的优化处理，加速数据副本恢复加载到 MemTable 的速度，尽快对外提供数据服务。在整个集群重启的情景下，需要将磁盘上最新的 clog 重新回放到 MemTable 里后对外提供数据服务，在 OceanBase 集群能够重新恢复数据服务时，数据恢复到集群重启前。</p>
OceanBase 备份恢复	OceanBase 集群出现数据损坏、节点 Crash 或者集群故障，OceanBase 集群可以从备份的基线数据和 clog 备份中恢复。	当 OceanBase 集群出现数据损坏、节点 crash 或者集群故障时，OceanBase 集群可以从备份的基线数据和 clog 备份中恢复。

<p>OceanBase 主备库解决方案</p>	<p>机房级的故障或城市级的灾难恢复：</p> <ul style="list-style-type: none"> <li>● 故障恢复是指当系统出现短时不可恢复故障时，系统恢复可用性、恢复数据服务的一系列过程；</li> <li>● 灾难恢复是指当机房甚至城市出现灾难或故障事件导致机房或者该城市内机房长时间的故障，不能立即恢复可用时，整个系统恢复可用的一系列过程。</li> </ul>	<p>OceanBase 集群也支持传统的主备库架构。OceanBase 集群的多副本机制可以提供丰富的容灾能力，在机器级、机房级、城市级故障情况下，可以实现自动切换，并且不丢数据，<math>RPO = 0</math>。当主集群出现计划内或计划外（多数派副本故障）的不可用情况时，备集群可以接管服务，并且提供无损切换（<math>RPO = 0</math>）和有损切换（<math>RPO &gt; 0</math>）两种容灾能力，最大限度降低服务停机时间。</p> <p>OceanBase 集群支持创建、维护、管理和监控一个或多个备集群。备集群是生产库数据的热备份。管理员可以选择将密集型只读业务操作分配到备集群，以便提高系统的性能和资源利用率。</p>
--------------------------	--	--

## 5.3 OceanBase 数据库如果少数派宕机会发生什么？多久可以恢复？

OceanBase 数据库利用了基于 Paxos 分布式一致性协议保证了在任一时刻只有当多数派副本达成一致时，才能推选一个 Leader，保证主副本的唯一性来对外提供数据服务。如果正在提供服务的 Leader 副本遇到故障而无法继续提供服务，只要其余的 Follower 副本满足多数派并且达成一致，就可以推选一个新的 Leader 来接管服务，而正在提供服务的 Leader 自己无法满足多数派条件，将自动失去 Leader 的资格。当 Leader 副本出现故障时，Follower 能在多长时间内感知到 Leader 的故障并推选出新的 Leader，这个时间直接决定了 RTO 的大小。

OceanBase 数据库的选举模型是基于 Paxos 重新设计的选举方案，由同日志流的多副本进行共识协商，结合各副本优先级选出唯一的主副本。当选举成功后，每个副本会签订认定 Leader 的租约（Lease）。在租约过期前，Leader 会不断发起连任，正常情况下能够一直连任成功。如果 Leader 没有连任成功，在租约到期后会周期性的发起无主选举，保证副本的高可用。

在三副本（全能副本）场景下，当一个副本出现异常（比如说该节点机器故障，OBServer 节点下线等单点故障），OceanBase 数据库选举模块能够做到：如果原主副本连任成功，原主可以连续提供主副本服务；如果原主副本下线，Leader 连任失败，OceanBase 数据库进行无主选举，新主上任的时间在 8s 内完成。

同时，OceanBase 数据库通过 Paxos 协议实现了多数派 clog 强一致同步持久化，在 Paxos 组中任意少数派副本发生故障的情况下，剩下的多数派副本都能保证有最新的 Clog，因此就能避免个别硬件故障带来的数据损失，保证了数据可靠性。

## 5.4 OceanBase 数据库选举如何避免脑裂问题？



在高可用方案中，一个经典的脑裂（Split Brain）问题：如果两个数据副本因为网络问题互相不知晓对方的状态，并且分别认为各自应当作为主副本提供数据服务，那么这时候就会出现典型的脑裂现象，最后会导致系统混乱，数据损坏。OceanBase 数据库利用了 Paxos 协议中的多数派共识机制来保证数据的可靠性以及主副本的唯一性：在任一时刻只有多数派副本达成一致时，才能推选一个 Leader。如果正在提供服务的 Leader 副本遇到故障而无法继续提供服务，只要其余的 Follower 副本满足多数派并且达成一致，他们就可以推选一个新的 Leader 来接管服务，而正在提供服务的 Leader 自己无法满足多数派条件，将自动失去 Leader 的资格。

因此，我们可以看到 OceanBase 数据库的分布式选举在高可用方面有明显的优势：从理论上就保证了任一时刻至多有一个 Leader，彻底杜绝了脑裂的情况。由于不再担心脑裂，当 Leader 故障而无法提供服务时，Follower 可以自动触发选举来产生新的 Leader 并接管服务，全程无须人工介入。这样一来，不但从根本上解决了脑裂的问题，还可以利用自动重新选举大大缩短 RTO。当然，这里面还有一个很重要的因素，那就是 Leader 出现故障时，Follower 能在多长时间内感知到 Leader 的故障并推选出新的 Leader，这个时间直接决定了 RTO 的大小。

## 5.5 OceanBase 集群当多数派失败的时候是否依旧可以服务？

当 OceanBase 集群中多数派失败，那么对应的分区是无法对外提供数据服务的。为了保证数据的高可用，在整个系统架构设计和运维时，还应当考虑和优化两个副本不相关的连续故障发生的时间。最终保证 MTTF (Mean Time To Failure) 应当相较于故障的修复时间 MTTR (Mean Time To Repair) 足够的小，以保证整个数据服务的高可用性。当 OceanBase 集群多数派失败的时候，在保留问题分析需要的信息和日志前提下，应当尽快采取应急措施将集群尽快恢复可用。

## 5.6 OceanBase 集群的副本自动补齐功能是如何工作的，副本自动补齐是否能够保证即使在节点宕机的情况下，相应的分区副本依旧齐全？

当 observer 进程异常终止，若终止时间小于 `server_permanent_offline_time`，则不作处理，此时有些分区的副本数只有 2 个了（三副本情况下）；当终止时间超过 `server_permanent_offline_time` 时，则对该 Server 做永久下线处理，OceanBase 集群会在同个 Zone 的其他 Server 开辟区域（在资源充裕的 Server 下），维持副本个数。当有足够的资源时就会发起 Unit 迁移。

## 5.7 OceanBase 集群是否支持多机房多地区的部署方式，在部署的时候有哪些基础设施的要求，应该如何选择方案？

在 OceanBase 集群的搭建中，允许将多个副本（节点）分散到多个机房、多个城市中，跨机房/跨城市实现 Paxos 组。在选择多机房，多城市集群架构时，通常也是为了获取机房级容灾甚至是城市级容灾的能力。从技术上来说，只要是支撑 OceanBase 集群不同副本节点的基础设施足够好，在合理的副本分布下，OceanBase 集群就可以将数据分布在不同的节点上对外提供数据服务。通常情况下，进行多机房/多城市的部署方案最重要的是考虑业务和各方面产生的架构需求。因为多机房/多城市的部署方式也会直接带来整个系统成本的大幅提高（比如是跨城市的网络专线部署等等）。所以不同的部署方案是一个基于成本、需求、产品能力、方案可行性等多个维度的权衡结果产出。

从技术上来讲，可以解决机房级容灾或者城市级容灾的集群解决方案有：

- 同城三机房三副本部署
  - 同城 3 个机房组成一个集群（每个机房是一个 Zone），机房间网络延迟一般在 0.5 ~ 2 ms 之间。
  - 机房级灾难时，剩余的两份副本依然是多数派，依然可以同步 Redo-Log 日志，保证 RPO=0。
  - 无法应对城市级的灾难。



- 三地五中心五副本部署：
  - 三个城市，组成一个 5 副本的集群。



- 任何一个 IDC 或者城市的故障，依然构成多数派，可以确保 RPO=0。
- 由于 3 份以上副本才能构成多数派，但每个城市最多只有 2 份副本，为降低时延，城市 1 和城市 2 应该离得较近，以降低同步 Redo-Log 的时延。
- 为降低成本，城市 3 可以只部署日志型副本（只有日志）。



在实际的部署方案中，OceanBase 集群也根据客户的需求定制过同城双机房和两地三中心的方案。两种部署方案都各自解决了一部分相应场景对于系统高可用的需求，但是同时也存在一定容灾能力的短板。在一定时期有可能作为系统架构的中间态成为客户在初始部署的选择，客户也有可能参与 OceanBase 主备库架构搭配使用一起达到期望的容灾能力。具体的方案探讨需要考虑多个因素和方面，如果有实际具体的场景，请联系 OceanBase 数据库技术架构师进行方案讨论和对接。

## 6 列存 FAQ

### 6.1 列存允许增删列吗？

- 允许增加列和删列。
- 支持 Varchar 列字符数改大、改小。
- 列存支持多种 Offline DDL，和行存表无异。

有关列存修改的详细信息请参见[表行存列存转换（MySQL 模式）](#)和[表行存列存转换（Oracle 模式）](#)。

### 6.2 列存表的查询有何特点？

- 冗余行存表中，列存表查询逻辑默认采用 Range Scan 走列存模式，而 Point Get 查询则会回退到行存模式。
- 纯列存表中，任何查询都采用列存模式。

### 6.3 列存支持事务吗，对事务大小有什么限制？

和行存表一样，支持事务，并且事务大小无限制，具备高一致性。

### 6.4 列存表的日志同步、备份恢复等有什么特别之处吗？

没有任何特别之处，和行存表一致。同步的日志都是行存模式。

### 6.5 是否支持将行存表用 DDL 变成列存表？

支持。通过加列存、删行存实现。相关语法示例如下：

```
create table t1( pk1 int, c2 int, primary key (pk1));

alter table t1 add column group(all columns, each column);
alter table t1 drop column group(all columns, each column);

alter table t1 add column group(each column);
alter table t1 drop column group(each column);
```

## 说明

`alter table t1 drop column group(all columns, each column);` 执行后，不用担心没有任何 Group 来承载数据，所有列会被放到一个叫做 `DEFAULTL COLUMN GROUP` 的默认 Group 中。

## 6.6 列存里支持多个列集合在一起吗？

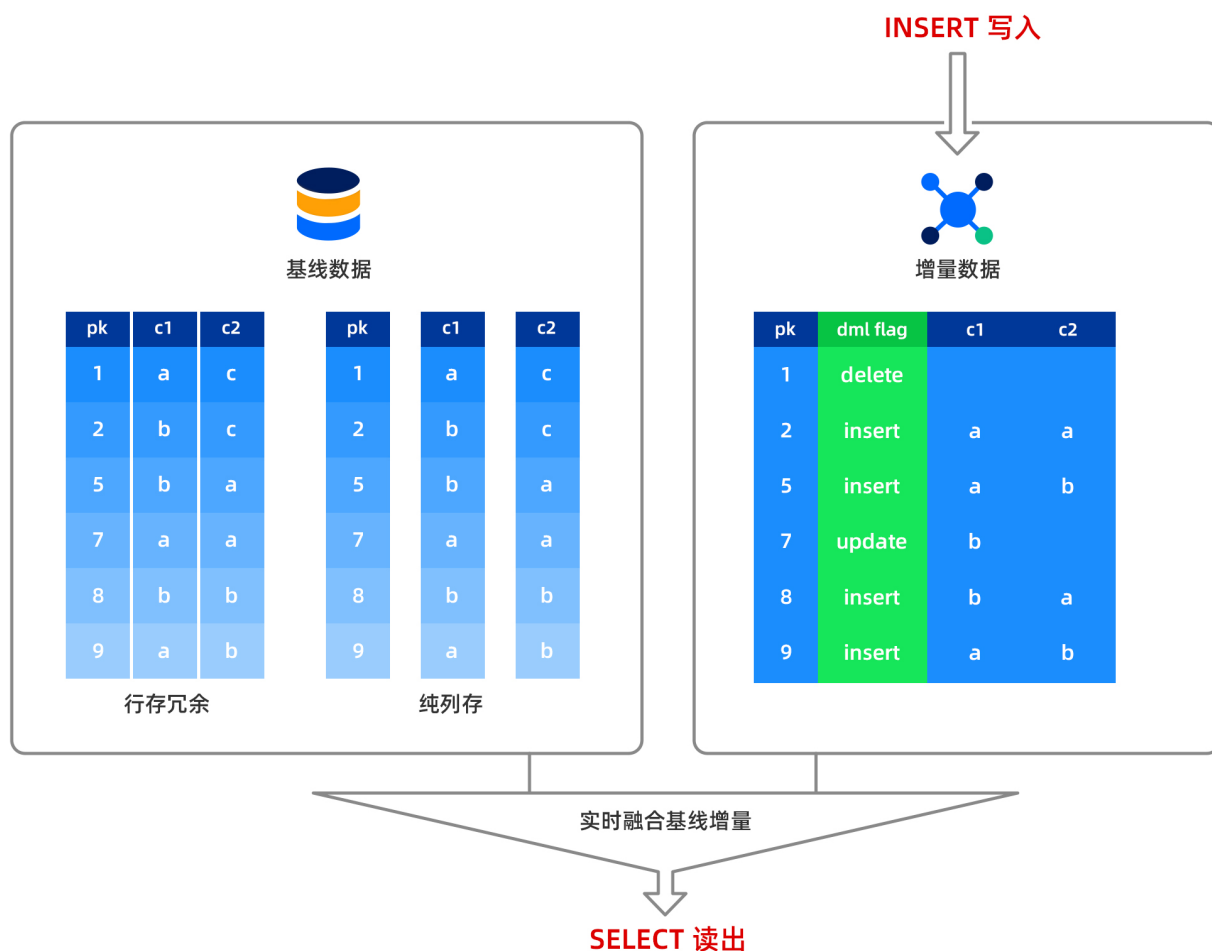
在 OceanBase 数据库 V4.3.0 版本中，仅支持要么每个列独立存储，要么所有列一起作为行存储。暂不支持任选若干列集合在一起存储。

## 6.7 列存支持更新吗，以及 MemTable 里的结构是怎样的？

在 OceanBase 数据库中，增删改操作都在内存里完成，数据以行存的形式保存在 MemTable 里；而基线数据是只读的，以列存的形式保存在磁盘上。当读取一行数据时，会实时地融合 MemTable 中的行存数据和磁盘里的列存数据，输出给用户。这意味着，OceanBase 数据库支持强一致读列存，不会有数据延迟。写入 MemTable 的数据支持转储，转储数据依然以行存的形式保存。合并后，行存数据和基线列存数据融合，形成新的基线列存数据。

## 注意

对于列存表来说，如果存在大量的更新操作，并且没有及时合并，查询性能会受到影响。因此，推荐在批量导入数据后发起一次合并操作，以获得最优的查询性能。而少量更新操作对性能影响不大。



## 6.8 支持对列存的某一列建索引吗？

支持。OceanBase 数据库不区分是对列存建索引，还是对行存建索引，创建的索引结构是一样的。

对列存某一列或几列建索引的意义在于可以构造一个覆盖索引，提升点查询性能，或者对特定列做排序以提升排序性能。

## 6.9 “列存索引” 是什么？

OceanBase 数据库还支持列存索引的概念，不同于“对列存建索引”。列存索引指的是索引表的结构是列存格式。

例如，我们已经有行存表 `t6`，希望对 `c3` 求和且性能最好，这时可以对 `c3` 建一个列存索引：

```
create table t6(
c1 TINYINT,
```

```
c2 SMALLINT,  
c3 MEDIUMINT  
);  
  
create /*+ parallel(2) */ index idx1 on t6(c3) with column group (each column);
```

除此之外，我们还支持更多索引创建方式：

- 支持索引中冗余行存。

```
create index idx1 on t1(c2) storing(c1) with column group(all columns, each  
column);  
alter table t1 add index idx1 (c2) storing(c1) with column group(all columns, each  
column);
```

- 支持索引中纯列存。

```
create index idx1 on t1(c2) storing(c1) with column group(each column);  
alter table t1 add index idx1 (c2) storing(c1) with column group(each column);
```

在数据库索引中使用 `STORING` 子句的目的是存储额外的非索引列数据到索引中。这可以为特定的查询提供性能优化，既可以避免回表，也可以降低索引排序的代价。当查询仅需要访问存储在索引中的列，而不需要回表查询原始行时，可以大幅提升查询效率。

## 6.10 列存表与列存副本的区别是什么？

列存表是指表的分区 Leader & Follower 的 Schema 均为列存格式，并且 OLAP 的查询可以是强读；而列存副本是在保证表的分区 Leader & Follower 的 Schema 为行存格式的前提下，只读副本 Learner 为列存格式，并且 OLAP 的查询只能是弱读。