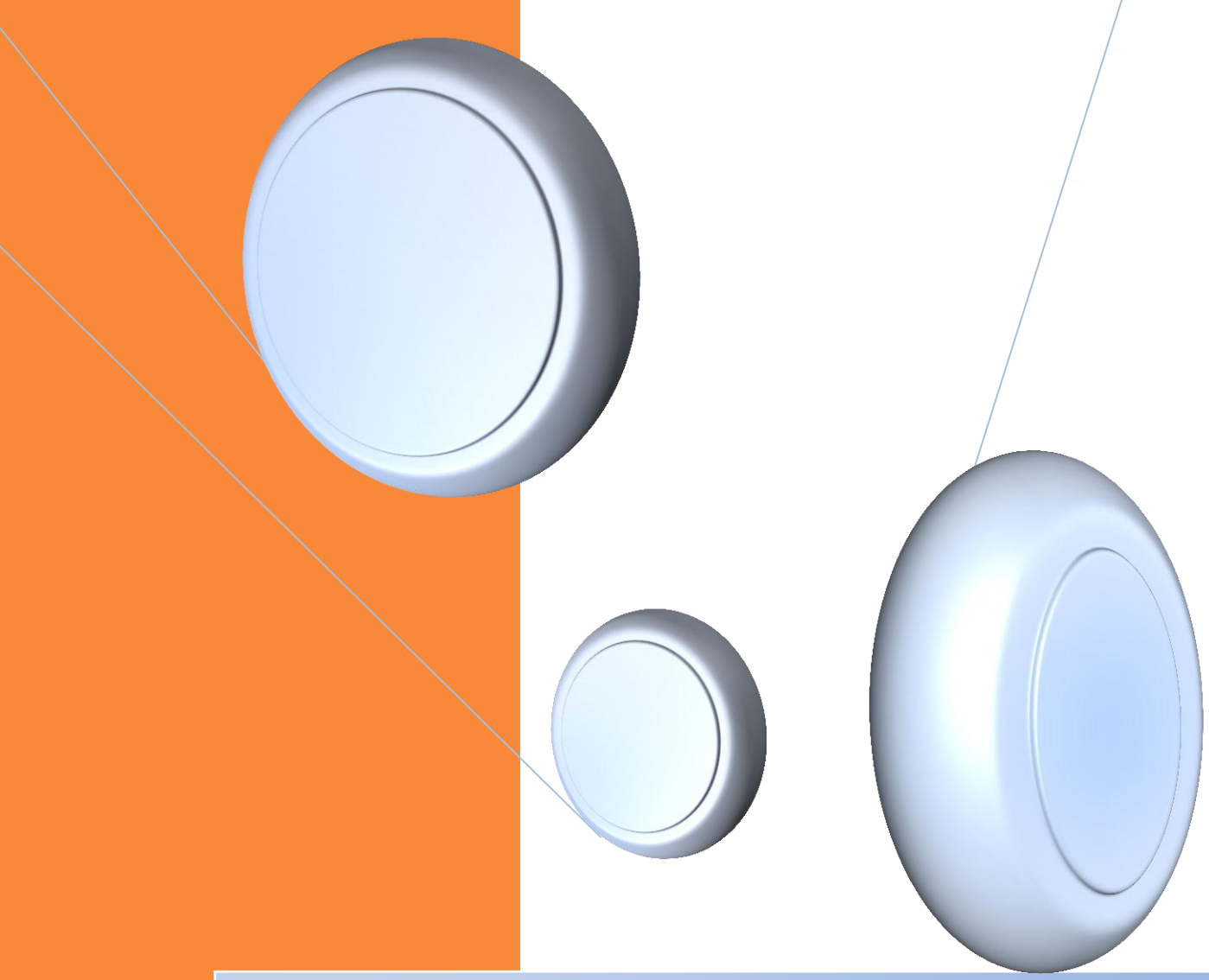


OceanBase 0.4.2

描述

文档版本：Beta 02

发布日期：2013.11.30

支付宝（中国）网络技术有限公司·OceanBase 团队

前 言

概述

本文档主要介绍OceanBase 0.4.2的架构、存储引擎和功能等信息。

读者对象

本文档主要适用于：

- 开发工程师。
- 安装工程师。
- 数据库管理工程师。

通用约定

在本文档中可能出现下列各式，它们所代表的含义如下。

格式	说明
警告	表示可能导致设备损坏、数据丢失或不可预知的结果。
注意	表示可能导致设备性能降低、服务不可用。
小窍门	可以帮助您解决某个问题或节省您的时间。
说明	表示正文的附加信息，是对正文的强调和补充。
宋体	表示正文。
粗体	表示命令行中的关键字（命令中保持不变、必须照输的部分）或者正文中强调的内容。
斜体	用于变量输入。
{ a | b | ... }	表示从两个或多个选项中选取一个。
[ ]	表示用“[ ]”括起来的部分在命令配置时是可选的。

修订记录

修改记录累积了每次文档更新的说明。最新版本的文档包含以前所有文档版本。

版本和发布日期	说明
Beta 02（2013-11-30）	第一次发布Beta版本，适用于OceanBase 0.4.2。
01（2013-10-30）	第一次正式发布，适用于OceanBase 0.4.1。

联系我们

如果您有任何疑问或是想了解 OceanBase 的最新开源动态消息，请联系我们：

支付宝（中国）网络技术有限公司·OceanBase 团队

地址：杭州市万塘路 18 号黄龙时代广场 B 座；邮编：310099

北京市朝阳区东三环中路 1 号环球金融中心西塔 14 层；邮编：100020

邮箱：alipay-oceanbase-support@list.alibaba-inc.com

新浪微博：http://weibo.com/u/2356115944

技术交流群（阿里旺旺）：853923637

目 录

1 架构初探

OceanBase 是阿里巴巴集团研发的可扩展的关系数据库，实现了数千亿条记录、数百 TB 数据上的跨行跨表事务。截止到 2012 年 8 月，OceanBase 支持了收藏夹、直通车报表、天猫评价等 OLTP 和 OLAP 在线业务，线上数据量已经超过一千亿条。

1.1 背景简介

OceanBase 最初是为了解决阿里巴巴集团旗下的淘宝网的大规模数据问题而诞生的。淘宝网的数据规模及其访问量对关系数据库提出了很大挑战：数百亿条的记录、数十 TB 的数据、数万 TPS、数十万 QPS 让传统的关系数据库不堪重负，单纯的硬件升级已经无法使得问题得到解决，分库分表也并不总是凑效。下面来看一个实际的例子。

淘宝收藏夹是淘宝线上的主要应用之一，淘宝用户在其中保存自己感兴趣的宝贝（即商品，此外用户也可以收藏感兴趣的店铺）以便下次快速访问、对比和购买等，用户可以展示和编辑（添加/删除）自己的收藏。因此，淘宝收藏夹数据库中的主要数据如下：

- 收藏 info 表中保存数百亿条收藏信息条目。
- 收藏 item 表中保存数十亿条收藏的宝贝和店铺的详细信息。
- 热门宝贝可能被多达数十万买家收藏。
- 每个用户可以收藏千个宝贝。
- 宝贝的价格、收藏人气等信息随时变化。

如果用户选择按宝贝价格排序后展示，那么数据库需要从收藏 item 表中读取收藏的宝贝的价格等最新信息，然后进行排序处理。如果用户的收藏条目比较多(例如 4000 条)，那么查询对应的 item 的时间会较长。假设平均每条 item 查询时间是 5ms，那么 4000 条的查询时间可能达到 20s。若果真如此，则用户体验会很差。

如果把收藏的宝贝的详细信息实时冗余到收藏 info 表，则上述查询收藏 item 表的操作就不再需要了。但是，由于许多热门商品可能有几千到几十万人收藏，这些热门商品的价格等信息的变动也可能导致收藏 info 表的大量修改，并压垮数据库。

为了解决以上问题，淘宝急需一个适合互联网规模的分布式数据库，这个数据库不仅要能够解决收藏夹面临的业务挑战，还要能够做到可扩展、低成本、易用，

并能够应用到更多的业务场景。因此，淘宝研发的千亿级海量数据库 OceanBase 应运而生。

OceanBase 开源地址：https://github.com/alibaba/oceanbase/wiki

1.2 设计思路

OceanBase 的目标是支持数百 TB 的数据量以及数十万 TPS、数百万 QPS 的访问量。无论是数据量还是访问量，即使采用非常昂贵的小型机甚至是大型机，单台关系数据库系统都无法承受。当前常见的处理方法主要有以下两种：

一种常见的做法是根据业务特点对数据库进行水平拆分。根据某个业务字段，通常取用户编号，哈希后取模，根据取模的结果将数据分布到不同的数据库服务器上，客户端请求通过数据库中间层路由到不同的分区。这种方式目前还存在一定的弊端：

第一，数据和负载增加后添加机器的操作比较复杂，需要人工介入。

第二，有些范围查询需要访问几乎所有的分区，例如，按照用户编号分区，查询收藏了一个商品的所有用户需要访问所有的分区。

第三，目前广泛使用的关系数据库存储引擎都是针对机械硬盘的特点设计的，不能够完全发挥新硬件（SSD）的能力。

另外一种做法是参考分布式表格系统的做法。例如 Google Bigtable 系统，将大表划分为几万、几十万甚至几百万个子表，子表之间按照主键有序，如果某台服务器发生故障，它上面服务的数据能够在很短的时间内自动迁移到集群中所有的其它服务器。这种方式解决了可扩展性的问题，少量突发的服务器故障或者增加服务器对使用者基本是透明的，能够轻松应对促销或者热点事件等突发流量增长。另外，由于子表是按照主键有序分布的，很好地解决了范围查询的问题。

虽然分布式表格系统解决了可扩展性问题，但往往无法支持事务。例如 Bigtable 只支持单行事务，针对同一个 user_id 下的多条记录的操作都无法保证原子性。而 OceanBase 希望能够支持跨行跨表事务，这样使用起来会比较方便。

最直接的做法是在 Bigtable 开源实现（如 HBase 或者 Hypertable）的基础上引入两阶段提交（Two-phase Commit）协议支持分布式事务，这种思路在 Google 的 Percolator 系统中得到了体现。然而，Percolator 系统中事务的平均响应时间达到 2~5 秒，只能应用在类似网页建库这样的半线上业务中。另外，Bigtable 的开源实现也不够成熟，单台服务器能够支持的数据量有限，单个请求的最大响应时间很难得到保证，机器故障等异常处理机制也有很多比较严重的问题。总体上看，这种做法的工作量和难度超出了项目组的承受能力，因此，我们需要根据业务特点做一些定制。

通过分析，我们发现，虽然在线业务的数据量十分庞大，例如几十亿条、上百亿条甚至更多记录，但最近一段时间（例如一天）的修改量往往并不多，通常不超过几千万条到几亿条，因此，OceanBase 决定采用单台更新服务器来记录最近一段时间的修改增量，称为增量数据；而以前的数据保持不变，称为基准数据。

基准数据以类似分布式文件系统的方式存储于多台基准数据服务器中，每次查询都需要把基准数据和增量数据融合后返回给客户端。因此，写事务都将集中在单台更新服务器上，避免了复杂的分布式事务，高效地实现了跨行跨表事务。另外，更新服务器上的修改增量能够定期分发到多台基准数据服务器中，避免成为瓶颈，实现了良好的扩展性。

当然，单台更新服务器的处理能力总是有一定的限制。因此，更新服务器的硬件配置较高，如内存较大、网卡及 CPU 较好。另外，最近一段时间的更新操作往往总是能够存放在内存中，所以在软件层面也针对这种场景做了大量的优化。

1.3 系统架构

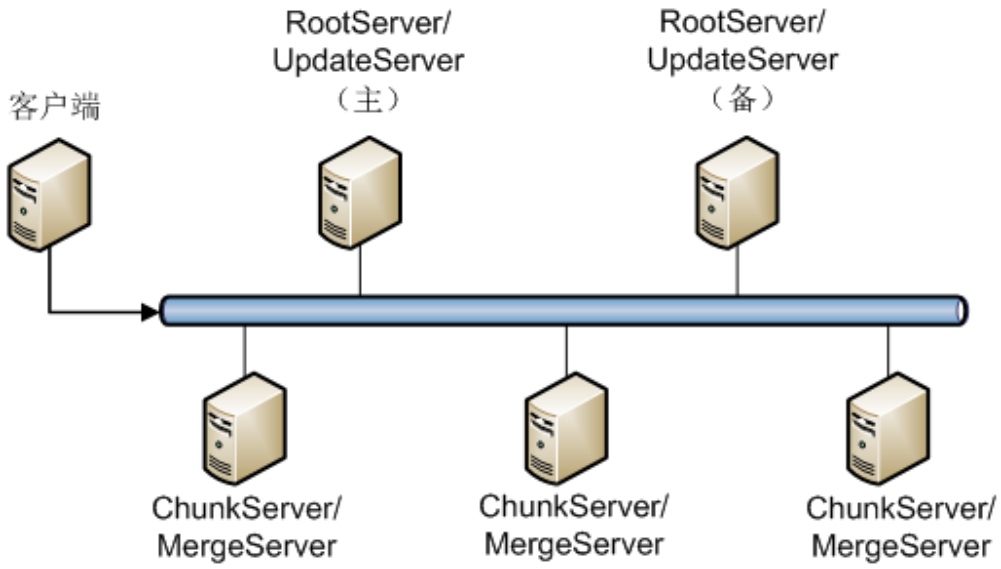
介绍 OceanBase 的整体架构和各个模块。

1.3.1 整体架构图

OceanBase 可以划分为四个模块：主控服务器 RootServer、更新服务器 UpdateServer、基准数据服务器 ChunkServer 以及合并服务器 MergeServer。

OceanBase 内部按照时间线将数据划分为基准数据和增量数据，基准数据是只读的，所有的修改更新到增量数据中，系统内部通过合并操作定期将增量数据融合到基准数据中。OceanBase 整体架构如图 1-1所示。

图 1-1 整体架构



- 客户端
  用户使用 OceanBase 的方式和 Mysql 数据库完全相同，支持 OceanBase Java 客户端和 OceanBase C 客户端访问等。基于 Mysql 数据库开发的应用程序、工具能够直接迁移到 OceanBase。

- RootServer：
  管理集群中的所有服务器，Tablet 数据分布以及副本管理。RootServer 一般为一主一备，主备之间数据强同步。

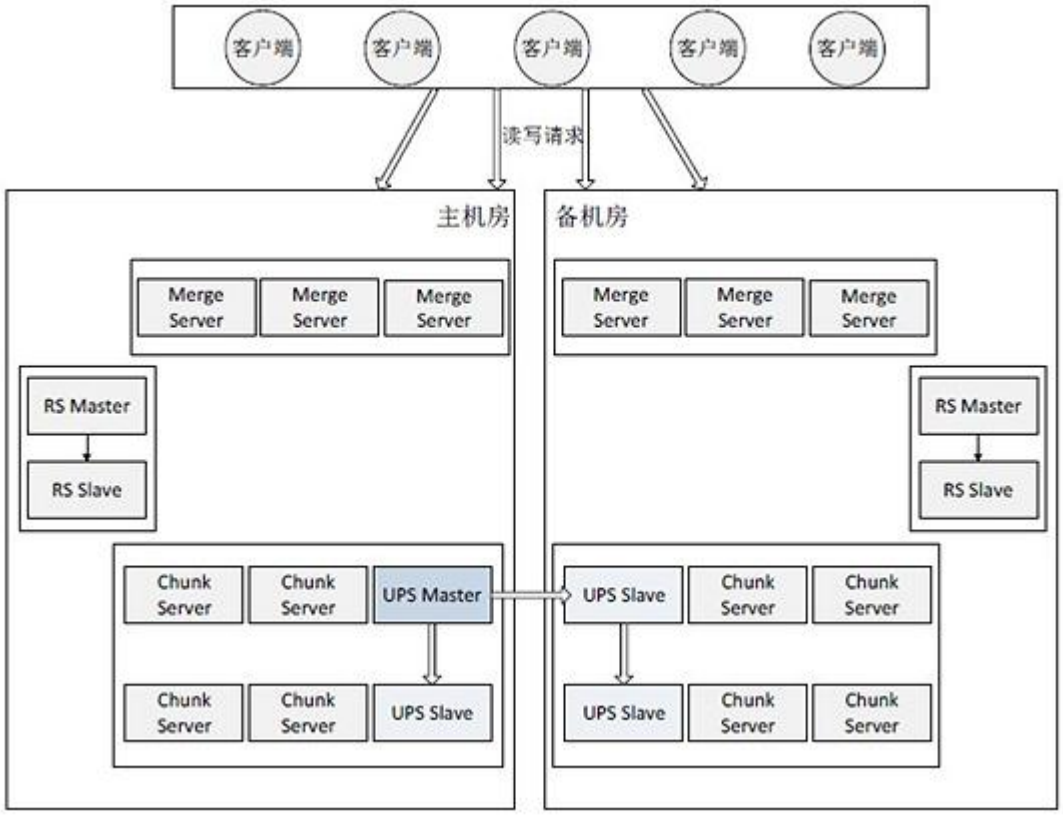
- UpdateServer
  存储 OceanBase 系统的增量更新数据。UpdateServer 一般为一主一备，目前的每个集群内部，同一时刻只允许主 UpdateServer 提供写服务。部署时，UpdateServer 和 RootServer 一般合设在同一服务器中。

- ChunkServer
  存储 OceanBase 系统的基准数据。基准数据一般存储两份或者三份，可配置。

- MergeServer
  接收并解析用户的 SQL 请求，经过词法分析、语法分析、查询优化等一系列操作后转发给相应的 ChunkServer 或者 UpdateServer。如果请求的数据分布在多台 ChunkServer 上，MergeServer 还需要对多台 ChunkServer 返回的结果进行合并。客户端和 MergeServer 之间采用原生的 MySQL 通信协议，MySQL 客户端可以直接访问 MergeServer。OceanBase 集群内部还有一个特殊的 MergeServer 进程，即 Listener，一般与 RootServer 合设。负责从集群的内部表中查询主备集群的流量分布信息和所有的其他 MergeServer 的地址列表。

OceanBase 支持部署多个机房，每个机房中均部署一个包含 RootServer、MergeServer、ChunkServer 和 UpdateServer 的完整 OceanBase 集群，如图 1-2 所示。每个集群由各自的 RootServer 负责数据划分、负载均衡，集群服务器管理等操作。集群之间数据同步通过主集群中的主 UpdateServer 往备集群同步增量更新操作日志实现。客户端配置了多个集群的 RootServer 地址列表，使用者可以设置每个集群的流量分配比例，客户端根据这个比例将读写操作发往不同的集群。

图 1-2 OceanBase 部署



1.3.2 OceanBase 客户端

OceanBase 客户端与 MergeServer 通信，目前主要支持如下几种客户端：

- Mysql 客户端
  MergeServer 兼容 Mysql 协议，MySQL 客户端及相关工具（如 Java 数据库访问方式 JDBC）只需要将服务器的地址设置为任意一台 MergeServer 的地址就可以直接使用。

- OceanBase Java 客户端
  OceanBase 内部部署多台 MergeServer，OceanBase Java 客户端提供对 MySQL 标准 JDBC Driver 的封装，并提供流量分配、负载均衡、MergeServer 异常处理等功能。简单来讲，OceanBase Java 客户端首先按照一定的策略选择一台 MergeServer，接着调用 Mysql JDBC Driver 往这台 MergeServer 发送读写请求。OceanBase Java 客户端实现符合 JDBC 标准，能够支持 Spring、iBatis 等 Java 编程框架。

- OceanBase C 客户端
  OceanBase C 客户端的功能和 OceanBase Java 客户端类似。它首先按照一定的策略选择一台 MergeServer，接着调用 MySQL 标准 C 客户端往这台 MergeServer 发送读写请求。OceanBase C 客户端的接口和 Mysql 标准 C 客户端接口完全相同，因此，能够通过 LD_PRELOAD 的方式将

应用程序依赖的 Mysql 标准 C 客户端替换为 OceanBase C 客户端，而无需修改应用程序的代码。

OceanBase 集群有多台 MergeServer，这些 MergeServer 的服务器地址存储在 OceanBase 服务器端的系统表内（与 Oracle 的系统表类似，存储系统的元数据）。OceanBase Java/C 客户端首先请求服务器端获取 MergeServer 地址列表，接着按照一定的策略将读写请求发送给其中一台 MergeServer，并负责对出现故障的 MergeServer 进行容错处理。

OceanBase Java/C 客户端访问 OceanBase 的流程如下：

1. OceanBase 客户端请求 RootServer 获取集群中 MergeServer 的地址列表。
2. 按照一定的策略选择一台 MergeServer 发送读写请求。
   说明：*客户端支持的策略主要有两种：随机以及一致性哈希。一致性哈希的主要目的是将相同的 SQL 请求发送到同一台 MergeServer，方便 MergeServer 对查询结果进行缓存。*
3. 如果请求 MergeServer 失败，则从 MergeServer 列表中重新选择一台 MergeServer。当请求某台 MergeServer 失败超过一定的次数时，则将这台 MergeServer 加入黑名单并从 MergeServer 列表中删除。另外，客户端会定期请求 RootServer 更新 MergeServer 地址列表。

如果 OceanBase 部署多个集群，客户端还需要处理多个集群的流量分配问题。使用者可以设置多个集群之间的流量分配比例，客户端获取到流量分配比例后，按照这个比例将请求发送到不同的集群。

OceanBase 程序升级版本时，先将备集群的读取流量调整为“0”。这时所有的读写请求都只发往主集群，接着升级备集群的程序版本。备集群升级完成后将流量逐步切换到备集群，并观察一段时间。如果没有出现异常，则将所有的流量切到备集群，并将备集群切换为主集群提供写服务。原来的主集群变为新的备集群，升级新的备集群的程序版本后重新分配主备集群的流量比例。

1.3.3 RootServer

RootServer 的功能主要包括：集群管理、数据分布以及副本管理。

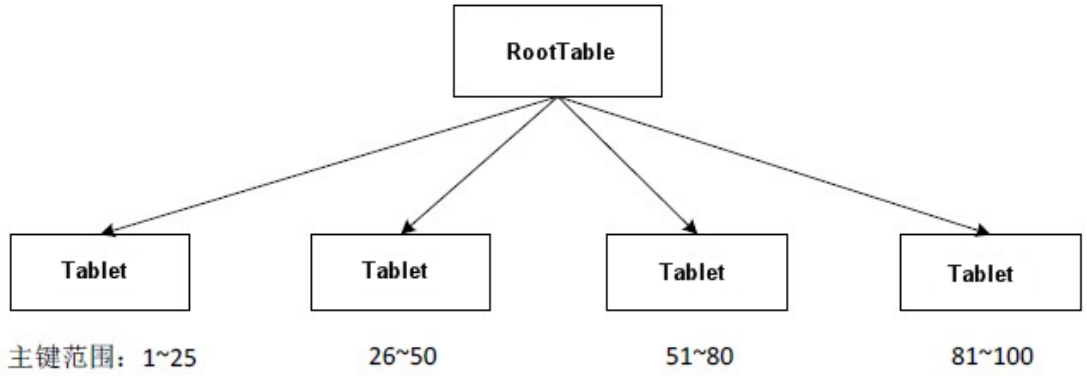
RootServer 管理集群中的所有 MergeServer、ChunkServer 以及 UpdateServer。每个集群内部同一时刻只允许一个 UpdateServer 提供写服务，这个 UpdateServer 为主 UpdateServer。这种方式通过牺牲一定的可用性获取了强一致性。RootServer 通过租约（Lease）机制选择唯一的主 UpdateServer，当原先的主 UpdateServer 发生故障后，RootServer 能够在原先的租约失效后选择一台新的 UpdateServer 作为主 UpdateServer。另外，RootServer 与 MergeServer、ChunkServer 之间保持心跳（heartbeat），从而能够感知到在线和已经下线的 MergeServer、ChunkServer 机器列表。

OceanBase 内部使用主键对表格中的数据进行排序和存储，主键由若干列组成并且具有唯一性。在 OceanBase 内部，基准数据按照主键排序并且划分为数据

量大致相等的数据范围，称为 Tablet。每个 Tablet 的缺省大小是 256MB（可配置）。目前，OceanBase 采用 RootTable 一级索引结构。

如图 1-3 所示，主键值在[1,100]之间的表格被划分为四个 Tablet：1~25，26~50，51~80 以及 81~100。RootServer 中的 RootTable 记录了每个 Tablet 所在的 ChunkServer 位置信息，每个 Tablet 包含多个副本（一般为三个副本，可配置），分布在多台 ChunkServer 中。当其中某台 ChunkServer 发生故障时，RootServer 能够检测到，并且触发对这台 ChunkServer 上的 Tablet 增加副本的操作。另外，RootServer 也会定期执行负载均衡，选择某些 Tablet 从负载较高的机器迁移到负载较低的机器。

图 1-3 基准数据 Tablet 划分



RootServer 建议采用一主一备的结构，主备之间数据强同步，并通过 Linux HA 实现高可用性。主备 RootServer 之间共享 VIP，当主 RootServer 发生故障后，VIP 能够自动漂移到备 RootServer 所在的机器，备 RootServer 检测到以后切换为主 RootServer 提供服务。

1.3.4 MergeServer

MergeServer 的功能主要包括：协议解析、SQL 解析、请求转发、结果合并、多表操作等。

OceanBase 客户端与 MergeServer 之间的协议为 MySQL 协议。MergeServer 首先解析 MySQL 协议，从中提取出用户发送的 SQL 语句，接着进行词法分析和语法分析，生成 SQL 语句的逻辑查询计划和物理查询计划，最后根据物理查询计划调用 OceanBase 内部的各种操作符。

MergeServer 缓存了 Tablet 分布信息，根据请求涉及的 Tablet 将请求转发给该 Tablet 所在的 ChunkServer。如果是写操作，还会转发给 UpdateServer。某些请求需要跨多个 Tablet，此时 MergeServer 会将请求拆分后发送给多台 ChunkServer，并合并这些 ChunkServer 返回的结果。如果请求涉及到多个表格，MergeServer 需要首先从 ChunkServer 获取每个表格的数据，接着再执行多表关联或者嵌套查询等操作。

MergeServer 支持并发请求多台 ChunkServer，即将多个请求发给多台 ChunkServer，再一次性等待所有请求的应答。另外，在 SQL 执行过程中，如

果某个 Tablet 所在的 ChunkServer 出现故障，MergeServer 会将请求转发给该 Tablet 的其他副本所在的 ChunkServer。

MergeServer 本身是没有状态的，因此，MergeServer 宕机不会对使用者产生影响，客户端会自动将发生故障的 MergeServer 屏蔽掉。

1.3.5 ChunkServer

ChunkServer 的功能包括：存储多个 Tablet、提供读取服务和执行定期合并等。

OceanBase 将大表划分为大小约为 256MB 的 Tablet，每个 Tablet 由一个或者多个 SSTable 组成（一般为一个），每个 SSTable 由多个块（Block，大小为 4KB ~ 64KB 之间，可配置）组成，数据在 SSTable 中按照主键有序存储。查找某一行数据时，需要首先定位这一行所属的 Tablet，接着在相应的 SSTable 中执行二分查找。SSTable 支持两种缓存模式，Block Cache 以及 Row Cache。Block Cache 以 Block 为单位缓存最近读取的数据，Row Cache 以行为单位缓存最近读取的数据。

MergeServer 将每个 Tablet 的读取请求发送到 Tablet 所在的 ChunkServer。ChunkServer 首先读取 SSTable 中包含的基准数据，接着请求 UpdateServer 获取相应的增量更新数据，并将基准数据与增量更新融合后得到最终结果。

由于每次读取都需要从 UpdateServer 中获取最新的增量更新。为了保证读取性能，需要限制 UpdateServer 中增量更新的数据量，最好能够全部存放在内存中。OceanBase 内部会定期触发合并，在这个过程中，ChunkServer 将从 UpdateServer 获取一段时间之前的更新操作。通常情况下，OceanBase 集群会在每天的服务低峰期（凌晨 1:00 开始，可配置）执行一次合并操作。这个合并操作往往也称为每日合并。

1.3.6 UpdateServer

UpdateServer 是集群中唯一能够接受写入的模块，每个集群中只有一个主 UpdateServer。UpdateServer 中的更新操作首先写入到内存表，当内存表的数据量超过一定值时，可以生成快照文件并转储到 SSD 中。快照文件的组织方式与 ChunkServer 中的 SSTable 类似，因此，这些快照文件也称为 SSTable。另外，由于数据行的某些列被更新，某些列没被更新，SSTable 中存储的数据行是稀疏的，称为稀疏型 SSTable。

为了保证可靠性，主 UpdateServer 更新内存表之前需要首先写操作日志，并同步到备 UpdateServer。当主 UpdateServer 发生故障时，RootServer 上维护的租约将失效，此时，RootServer 将从备 UpdateServer 列表中选择一台最新的备 UpdateServer 切换为主 UpdateServer 继续提供写服务。UpdateServer 宕机重启后需要首先加载转储的快照文件（SSTable 文件），接着回放快照点之后的操作日志。

由于集群中只有一台主 UpdateServer 提供写服务，因此，OceanBase 很容易地实现了跨行跨表事务，而不需要采用传统的两阶段提交协议。当然，这样也带来了一系列的问题。由于整个集群所有的读写操作都必须经过 UpdateServer，

UpdateServer 的性能至关重要。OceanBase 集群通过定期合并，将 UpdateServer 一段时间之前的增量更新源源不断地分散到 ChunkServer，而 UpdateServer 只需要服务最新一小段时间新增的数据，这些数据往往可以全部存放在内存中。另外，系统实现时也需要对 UpdateServer 的内存操作、网络框架、磁盘操作做大量的优化。

1.4 架构剖析

介绍了 OceanBase 的架构特点。

1.4.1 一致性选择

Eric Brewer 教授的 CAP 理论指出，在满足分区可容忍性的前提下，一致性和可用性不可兼得。

虽然目前大量的互联网项目选择了弱一致性，但我们认为这是底层存储系统。比如 MySQL 数据库，这是在大数据量和高并发需求压力之下的无奈选择。弱一致性给应用带来了很多麻烦，比如数据不一致时需要人工订正数据。如果存储系统既能够满足大数据量和高并发的需求，又能够提供强一致性，且硬件成本相差不大，用户将毫不犹豫地选择它。强一致性将大大简化数据库的管理，应用程序也会因此而简化。因此，OceanBase 选择支持强一致性和跨行跨表事务。

OceanBase 中的 UpdateServer 为主备高可用架构，更新操作流程如下：

1. 将更新操作发送到备机。
2. 将更新操作的 redo 日志写入主机硬盘。
3. 将 redo 日志应用到主机的内存表格中。
4. 返回客户端写入成功。

OceanBase 要求将 redo 日志同步到主备后才能够返回客户端写入成功。即使主机出现故障，备机自动切换为主机，也能够保证新的主机拥有以前所有的更新操作，严格保证数据不丢失。另外，为了提高可用性，OceanBase 还增加了一种机制：如果主机往备机同步 redo 日志失败，比如备机故障或者主备之间网络故障，主机可以将备机从同步列表中剔除，本地更新成功后就返回客户端写入成功。主机将备机剔除前需要通知 RootServer。后续如果主机故障，RootServer 能够避免将不同步的备机切换为主机。

OceanBase 的高可用机制保证主机、备机以及主备之间网络三者之中的任何一个出现故障都不会对用户产生影响。然而，如果三者之中的两个同时出现故障，系统可用性将受到影响，但仍然保证数据不丢失。如果应用对可用性要求特别高，可以增加备机数量，从而容忍多台机器同时出现故障的情况。

OceanBase 主备同步允许配置为异步模式，支持最终一致性。这种模式一般用来支持异地容灾。例如，用户请求通过杭州主站的机房提供服务，主站的 UpdateServer 内部有一个同步线程不停地将用户更新操作发送到青岛机房。如

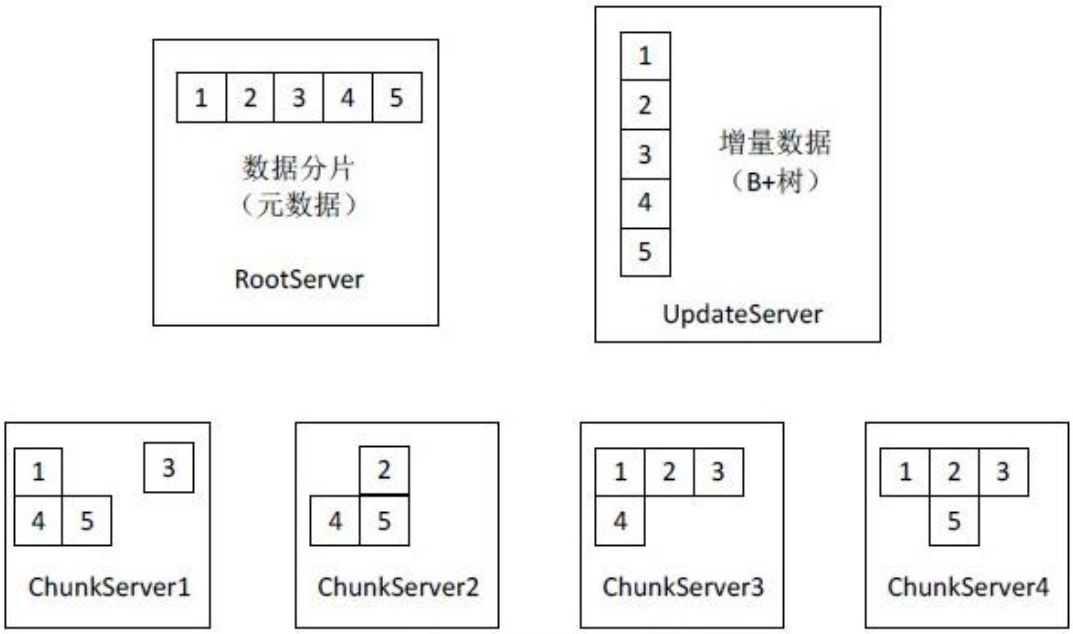
果杭州机房整体出现不可恢复的故障，比如地震，还能够通过青岛机房恢复数据并继续提供服务。

另外，OceanBase 所有写事务最终都落到 UpdateServer，而 UpdateServer 逻辑上是一个单点，支持跨行跨表事务，实现上借鉴了传统关系数据库的做法。

1.4.2 数据结构

OceanBase 数据分为基准数据和增量数据两个部分。基准数据分布在多台 ChunkServer 上，增量数据全部存放在一台 UpdateServer 上。如图 1-5 所示，系统中有 5 个 Tablet，每个 Tablet 有 3 个副本，所有的 Tablet 分布到 4 台 ChunkServer 上。RootServer 中维护了每个 Tablet 所在的 ChunkServer 的位置信息，UpdateServer 存储了这 5 个 Tablet 的增量更新。

图 1-5 OceanBase 数据结构



不考虑数据复制，基准数据的数据结构如下：

- 每个表格按照主键组成一颗分布式 B+树，主键由若干列组成。
- 每个叶子节点包含表格一个前开后闭的主键范围(rk1，rk2]内的数据。
- 每个叶子节点称为一个子表（Tablet），包含一个或者多个 SSTable。
- 每个 SSTable 内部按主键范围有序划分为多个块（block）并内建块索引（block index）。
- 每个块的大小通常在 4KB ~ 64KB 之间并内建块内的行索引。
- 数据压缩以块为单位，压缩算法由用户决定并可随时变更。
- 叶子节点可能合并或者分裂。
- 所有叶子节点基本上是均匀的，随机地分布在多台 ChunkServer 机器上。

- 通常情况下每个叶子节点有 2~3 个副本。
- 叶子节点时负载平衡和任务调度的基本单元。
- 支持 bloom filter 过滤。

增量数据的数据结构如下：

- 增量数据按照时间从旧到新划分为多个版本。
- 最新版本的数据为一颗内存中的 B+树，称为 Active MemTable。
- 用户的更新操作写入 Active MemTable。到达一定大小后，原有的 Active MemTable 将被冻结，并开启新的 Active MemTable 接受更新操作。
- 冻结的 MemTable 将以 SSTable 的形式转储到 SSD 中。
- 每个 SSTable 内部按主键范围有序划分为多个块并内建块索引，每个块的大小通常为 4KB ~ 8KB 并内建块内行索引，一般不压缩。
- UpdateServer 支持主备，增量数据通常为 2 个副本，每个副本支持 RAID1 存储。

1.4.3 可靠性与可用性

分布式系统需要处理各种故障，例如软件故障、服务器故障、网络故障、数据中心故障、地震、火灾等。与其它分布式存储系统一样，OceanBase 通过冗余的方式保障了高可靠性和高可用性。

- OceanBase 在 ChunkServer 中保存了基准数据的多个副本。单集群部署时一般会配置 3 个副本，主备集群部署时一般会配置每个集群 2 个副本，总共 4 个副本。
- OceanBase 在 UpdateServer 中保存了增量数据的多个副本。UpdateServer 主备模式下主备两台机器各保存一个副本。
- ChunkServer 的多个副本可以同时提供服务。Bigtable 以及 HBase 的系统服务节点不冗余，如果服务器出现故障，需要等待其它节点恢复成功才能提供服务，而 OceanBase 多个 ChunkServer 的 Tablet 副本数据完全一致，可以同时提供服务。
- UpdateServer 主备之间为热备，同一时刻只有一台机器为主 UpdateServer 提供写服务。如果主 UpdateServer 发生故障，OceanBase 能够在几秒中之内（一般为 3~5 秒）检测到并将服务切换到备机。
- OceanBase 存储多个副本并没有带来太多的成本。当前的主流服务器的磁盘容量通常是富余的，例如 300GB×12 或 600GB×12 的服务器有 3TB 或 6TB 左右的磁盘总容量，但关系数据库系统单机通常只能服务少得多的数据量。

1.4.4 读写事务

在 OceanBase 系统中，用户的读写请求，即读写事务，都发给 MergeServer。MergeServer 解析这些读写事务的内容，例如词法和语法分析、Schema 检查等。对于只读事务，由 MergeServer 发给相应的 ChunkServer 分别执行后再合并每个 ChunkServer 的执行结果；对于读写事务，由 MergeServer 进行预处理后，发送给 UpdateServer 执行。

只读事务执行流程如下：

1. MergeServer 解析 SQL 语句，词法分析、语法分析、预处理（Schema 合法性检查、权限检查、数据类型检查等），最后生成逻辑执行计划和物理执行计划。
2. 如果 SQL 请求只涉及单张表格，MergeServer 将请求拆分后同时发给多台 ChunkServer 并发执行。每台 ChunkServer 将读取的部分结果返回 MergeServer，由 MergeServer 来执行结果合并。
3. 如果 SQL 请求涉及多张表格，MergeServer 还需要执行联表、嵌套查询等操作。
4. MergeServer 将最终结果返回给客户端。

读写事务执行流程如下：

1. 与只读事务相同，MergeServer 首先解析 SQL 请求，得到物理执行计划。
2. MergeServer 请求 ChunkServer 获取需要读取的基准数据，并将物理执行计划和基准数据一起传给 UpdateServer。
3. UpdateServer 根据物理执行计划执行读写事务，执行过程中需要使用 MergeServer 传入的基准数据。
4. UpdateServer 返回 MergeServer 操作成功或者失败，MergeServer 接着会把操作结果返回客户端。

例如，假设某 SQL 语句为：“update t1 set c1 = c1 + 1 where rowkey=1”，即将表格 t1 中主键为 1 的 c1 列加 1，这一行数据存储在 ChunkServer 中，c1 列的值原来为 2012。那么，MergeServer 执行 SQL 时首先从 ChunkServer 读取主键为 1 的数据行的 c1 列，接着将读取结果（c1=2012）以及 SQL 语句的物理执行计划一起发送给 UpdateServer。UpdateServer 根据物理执行计划将 c1 加 1，即将 c1 变为 2013 并记录到 MemTable 中。当然，更新 MemTable 之前需要记录操作日志。

1.4.5 单点性能

OceanBase 架构的优势在于既支持跨行跨表事务，又支持存储服务器线性扩展。当然，这个架构也有一个明显的缺陷：UpdateServer 单点，这个问题限制了 OceanBase 集群的整体读写性能。

下面从内存容量、网络、磁盘等几个方面分析 UpdateServer 的读写性能。其实大部分数据库每天的修改次数相当有限，只有少数修改比较频繁的数据库才有每

天几亿次的修改次数。另外，数据库平均每次修改涉及的数据量很少，很多时候只有几十个字节到几百个字节。假设数据库每天更新 1 亿次，平均每次需要消耗 100 字节，每天插入 1000 万次，平均每次需要消耗 1000 字节，那么，一天的修改量为：1 亿 * 100 + 1000 万 * 1000 = 20GB，如果内存数据结构膨胀 2 倍，占用内存只有 40GB。而当前主流的服务器都可以配置 96GB 内存，一些高档的服务器甚至可以配置 192GB，384GB 乃至更多内存。

从上面的分析可以看出，UpdateServer 的内存容量一般不会成为瓶颈。然而，服务器的内存毕竟有限，实际应用中仍然可能出现修改量超出内存的情况。例如，淘宝双 11 网购节数据库修改量暴涨，某些特殊应用每天的修改次数特别多或者每次修改的数据量特别大，DBA 数据订正时一次性写入大量数据。为此，UpdateServer 设计实现了几种方式解决内存容量问题：UpdateServer 的内存表达到一定大小时，可自动或者手工冻结并转储到 SSD 中，另外，OceanBase 通过定期合并将 UpdateServer 的数据分散到集群中所有的 ChunkServer 机器中。这样不仅避免了 UpdateServer 单机数据容量问题，还能够使得读取操作往往只需要访问 UpdateServer 内存中的数据，避免访问 SSD 盘，提高了读取性能。

从网络角度看，假设每秒的读取次数为 20 万次，每次需要从 UpdateServer 中获取 100 字节，那么，读取操作占用的 UpdateServer 出口带宽为：20 万 * 100 = 20MB，远远没有达到千兆网卡带宽上限。另外，UpdateServer 还可以配置多块千兆网卡或者万兆网卡，例如，OceanBase 线上集群一般给 UpdateServer 配置 4 块千兆网卡。当然，如果软件层面没有做好，硬件特性将得不到充分发挥。针对 UpdateServer 全内存、收发的网络包一般比较小的特点，开发团队对 UpdateServer 的网络框架做了专门的优化，大大提高了每秒收发网络包的个数，使得网络不会成为瓶颈。

从磁盘的角度看，数据库事务需要首先将操作日志写入磁盘。如果每次写入都需要将数据刷入磁盘，而一块 SAS 磁盘每秒支持的 IOPS 很难超过 300，磁盘将很快成为瓶颈。为了解决这个问题，UpdateServer 在硬件上会配置一块带有缓存模块的 RAID 卡，UpdateServer 写操作日志只需要写入到 RAID 卡的缓存模块即可，延时可以控制在 1 毫秒之内。RAID 卡带电池或电容(BBU)，如果 UpdateServer 发生故障，比如机器突然停电，RAID 卡能够保证在一段时间内（例如 48 小时或者 72 小时）数据不丢失。另外，UpdateServer 还实现了写事务的 group commit 机制，将多个用户写操作凑成一批一次性提交，进一步减少磁盘 IO 次数。

1.4.6 SSD 支持

磁盘随机 IO 是存储系统性能的决定因素，传统的 SAS 盘能够提供的 IOPS 不超过 300。关系数据库一般采用 Buffer Cache 的方式缓解这个问题，读取操作将磁盘中的页面缓存到 Buffer Cache 中，并通过 LRU 或者类似的方式淘汰不经常访问的页面。同样，写入操作也是将数据写入到 Buffer Cache 中，由 Buffer Cache 按照一定的策略将内存中页面的内容刷入磁盘。这种方式面临一些问题，例如 Cache 冷启动问题，即数据库刚启动时性能很差，需要将读取流量逐步切入。另外，这种方式不适合写入特别多的场景。

最近几年，SSD 盘取得了很大的进展，它不仅提供了非常好的随机读取性能，功耗也非常低，大有取代传统机械磁盘之势。一块普通的 SSD 盘可以提供 35000 IOPS 甚至更高，并提供 300MB/s 或以上的读出带宽。然而，SSD 盘的随机写性能并不理想。这是因为尽管 SSD 的读和写以页（page，例如 4KB，8KB 等）为单位，但 SSD 写入前需要首先擦除已有内容，而擦除以块（block）为单位，一个（block）由若干个连续的页（page）组成，大小通常在 512KB ~ 2MB 左右。假如写入的页（page）有内容，即使只写入一个字节，SSD 也需要擦除整个 512KB ~ 2MB 大小的块（block），然后再写入整个页（page）的内容，这就是 SSD 的写入放大效应。虽然 SSD 硬件厂商都针对这个问题做了一些优化，但整体上看，随机写入并非 SSD 的优势。

OceanBase 设计之初就认为 SSD 为大势所趋，整个系统设计时完全摒弃了随机写：操作日志总是顺序追加写入到普通 SAS 盘上，剩下的写请求也都是对响应时间要求不是很高的批量顺序写，SSD 盘可以轻松应对，而大量查询请求的随机读，则发挥了 SSD 良好的随机读的特性。摒弃随机写，采用批量的顺序写，也使得固态盘的使用寿命不再成为问题：主流 SSD 盘使用 MLC SSD 芯片，而 MLC 号称可以擦写 1 万次（SLC 可以擦写 10 万次，但因成本高而较少使用），即使按最保守的 2500 次擦写次数计算，而且每天全部擦写一遍，其使用寿命为 2500/365=6.8 年。

1.4.7 数据正确性

数据丢失或者数据错误对于存储系统来说是一种灾难。在本文档的“1.4.1 一致性选择”章节中已经提到，OceanBase 设计为强一致性系统，设计方案上保证不丢数据。然而，TCP 协议传输、磁盘读写都可能出现数据错误，程序 Bug 更为常见。为了防止各种因素导致的数据损毁，OceanBase 采取了以下数据校验措施：

- 数据存储校验
  每个存储记录（通常是几个 KB 到几十 KB）同时保存 64 位 CRC 校验码，数据被访问时，重新计算和比对校验码。
- 数据传输校验
  每个传输记录同时传输 64 位 CRC 校验码，数据被接收后，重新计算和比对校验码。
- 数据镜像校验
  UpdateServer 在机群内有主 UpdateServer 和备 UpdateServer，集群间有主集群和备集群。这些 UpdateServer 的内存表（MemTable）必须保持一致。为此，UpdateServer 为 MemTable 生成一个校验码，MemTable 每次更新时，校验码同步更新并记录在对应的 commit log 中。备 UpdateServer 收到 commit log 重放更新 MemTable 时，也同步更新 MemTable 校验码并与接收到的校验码对照。UpdateServer 重新启动后重放日志恢复 MemTable 时也同步更新 MemTable 校验码并与保存在每条 commit log 中校验码对照。
- 数据副本校验
  定期合并时，新的 Tablet 由各个 Chunk Server 独立地融合旧的 Tablet

与冻结的 MemTable 而生成，如果发生任何异常或者错误（比如程序 bug），同一 Tablet 的多个副本可能不一致，则这种不一致可能随着定期合并而逐步累积或扩散且很难被发现，即使被察觉，也可能因为需要追溯较长时间而难以定位到源头。为了防止这种情况出现，Chunk Server 在定期合并生成新的 Tablet 时，也同时为每个 Tablet 生成一个校验码，并随新 Tablet 汇报给 RootServer，以便 RootServer 核对同一 Tablet 不同副本的校验码。

1.4.8 分层结构

OceanBase 对外提供的是与关系数据库一样的 SQL 操作接口，而内部却实现成一个线性可扩展的分布式系统。系统从逻辑实现上可以分为两个层次：分布式存储引擎层以及数据库功能层。

OceanBase 一期只实现了分布式存储引擎，这个存储引擎支持的特性如下：

- 支持分布式数据结构，基线数据逻辑上构成一颗分布式 B+树，增量数据为内存中的 B+树。
- 支持目前 OceanBase 的所有分布式特性，包括数据分布、负载均衡、主备同步、容错、自动增加/减少服务器等。
- 支持根据主键更新、插入、删除、随机读取一条记录，另外，支持根据主键范围顺序查找一段范围的记录。

OceanBase 二期的版本在分布式存储引擎之上增加了 SQL 支持：

- 支持 SQL 语言以及 Mysql 协议，Mysql 客户端可以直接访问。
- 支持读写事务。
- 支持多版本并发控制。
- 支持读事务并发执行。

从另外一个角度看，OceanBase 融合了分布式存储系统和关系数据库这两种技术。通过分布式存储技术将基准数据分布到多台 Chunk Server 上，实现数据复制、负载均衡、服务器故障检测与自动容错等功能；UpdateServer 相当于一个高性能的内存数据库，底层采用关系数据库技术实现。

2 分布式存储引擎

分布式存储引擎层负责处理分布式系统中的各种问题，例如数据分布、负载均衡、容错、一致性协议等。与其它 NOSQL 系统类似，分布式存储引擎层支持根据主键更新、插入、删除、随机读取以及范围查找等操作，数据库功能层构建在分布式存储引擎层之上。

分布式存储引擎层包含三个模块：RootServer、UpdateServer 和 Chunk Server。

- RootServer 用于整体控制，实现 Tablet 分布、副本复制、负载均衡、机器管理以及 Schema 管理。
- UpdateServer 用于存储增量数据，数据结构为一颗内存 B+树，并通过主备实时同步实现高可用，另外，UpdateServer 的网络框架也经过专门的优化。
- ChunkServer 用于存储基准数据，基准数据按照主键有序划分为一个一个 Tablet，每个 Tablet 在 ChunkServer 上存储了一个或者多个 SSTable，另外，定期合并的主要逻辑也由 ChunkServer 实现。

OceanBase 实现时也采用了很多技巧，例如 Group Commit、双缓冲区预读、合并限速、内存管理等。本章将介绍分布式存储引擎的实现以及涉及到的实现技巧。

2.1 RootServer 实现机制

RootServer 是 OceanBase 集群对外的窗口，客户端通过 RootServer 获取集群中其它模块的信息。RootServer 实现的功能包括：

- 管理集群中的所有 ChunkServer，处理 ChunkServer 上下线。
- 管理集群中的 UpdateServer，实现 UpdateServer 选主。
- 管理集群中 Tablet 数据分布，发起 Tablet 复制、迁移以及合并等操作。
- 与 ChunkServer 保持心跳，接受 ChunkServer 汇报，处理 Tablet 分裂与合并。
- 接受 UpdateServer 汇报的大版本冻结消息，通知 ChunkServer 执行定期合并。
- 实现主备 RootServer，数据强同步，支持主 RootServer 宕机自动切换。

2.1.1 数据结构

RootServer 的中心数据结构为一张存储了 Tablet 数据分布的有序表格，称为 RootTable。每个 Tablet 存储的信息包括：Tablet 主键范围、Tablet 各个副本所在 ChunkServer 的编号、Tablet 各个副本的数据行数、占用的磁盘空间、CRC 校验值以及基准数据版本。

RootTable 是一个读多写少的数据结构，除了 ChunkServer 汇报、RootServer 发起 Tablet 复制、迁移以及合并等操作需要修改 RootTable 外，其它操作都只需要从 RootTable 中读取某个 Tablet 所在的 ChunkServer。因此，OceanBase 设计时考虑以写时复制的方式实现该结构.另外，考虑到 RootTable 修改特别少，实现时没有采用支持写时复制的 B+树或者 Skip List，而是采用相对更加简单的有序数组，以减少工作量。

往 RootTable 增加 Tablet 信息的操作步骤如下：

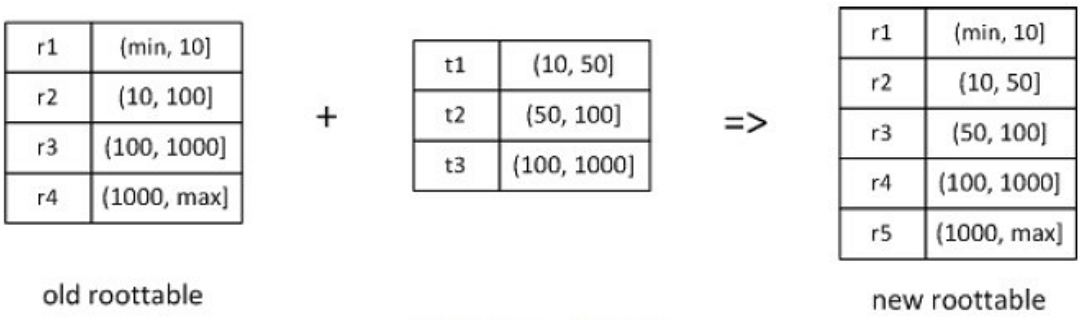
1. 拷贝当前服务的 RootTable 为新的 RootTable。
2. 将 Tablet 信息追加到新的 RootTable，并对新的 RootTable 重新排序。
3. 原子地修改指针使得当前服务的 RootTable 指向新的 RootTable。

ChunkServer 一次汇报一批 Tablet（默认一批包含 1024 个），如果每个 Tablet 修改都需要拷贝整个 RootTable 并重新排序，性能上显然无法接受。RootServer 实现时做了一些优化：拷贝当前服务的 RootTable 为新的 RootTable 后，将 ChunkServer 汇报的一批 Tablet 一次性追加到新的 RootTable 中并重新排序，最后再原子地切换当前服务的 RootTable 为新的 RootTable。采用批处理优化后，RootTable 的性能基本满足需求，OceanBase 单个集群支持的 Tablet 个数最大达到几百万个。当然，这种实现方式并不优雅，后续我们会将 RootTable 改造为 B+树或者 Skip List。

ChunkServer 汇报的 Tablet 信息可能和 RootTable 中记录的不同，比如发生了 Tablet 分裂。此时，RootServer 需要根据汇报的 Tablet 信息更新 RootTable。

如图 2-1 所示，假设原来的 RootTable 包含四个 Tablet：r1(min, 10]，r2(10, 100]，r3(100, 1000]，r4(1000, max]。ChunkServer 汇报的 Tablet 列表为：t1(10, 50]，t2(50, 100]，t3(100, 1000]，表示 r2 发生了 Tablet 分裂，那么，RootServer 会将 RootTable 修改为：r1(min, 10]，r2(10, 50]，r3(50, 100]，r4(100, 1000]，r5(1000, max]。

图 2-1 RootTable 修改



RootServer 中还有一个管理所有 ChunkServer 信息的数组，称为 ChunkServerManager。数组中的每个元素代表一台 ChunkServer，存储的信息包括：机器状态（已下线、正在服务、正在汇报、汇报完成等）、启动后注册时间、上次心跳时间、磁盘相关信息、负载均衡相关信息。OceanBase 刚上线时依据每台 ChunkServer 磁盘占用信息执行负载均衡，目的是为了尽可能确保每台 ChunkServer 占用差不多的磁盘空间。上线运行一段时间后发现这种方式效果并不好。目前的方式为按照每个表格的 Tablet 个数执行负载均衡，目的是尽可能保证对于每个表格，每台 ChunkServer 上的 Tablet 个数大致相同。

2.1.2 Tablet 复制与负载均衡

RootServer 中有两种操作都可能触发 Tablet 迁移：Tablet 复制（rereplication）和负载均衡（rebalance）。当某些 ChunkServer 下线超过一段时间后，为了防止数据丢失，需要拷贝副本数小于阀值的 Tablet。另外，系统也需要定期执行负载均衡，将 Tablet 从负载较高的机器迁移到负载较低的机器。

每台 ChunkServer 记录了 Tablet 迁移相关信息，包括：ChunkServer 上 Tablet 的个数、所有 Tablet 的大小总和、正在迁入的 Tablet 个数、正在迁出的 Tablet 个数和 Tablet 迁移任务列表。RootServer 包含一个专门的线程定期执行 Tablet 复制与负载均衡任务。

- Tablet 复制
  扫描 RootTable 中的 Tablet，如果某个 Tablet 的副本数小于阀值，则选取一台包含该 Tablet 副本的 ChunkServer 为迁移源，另外一台符合要求的 ChunkServer 为迁移目的地，生成 Tablet 迁移任务。迁移目的地需要符合一些条件有：不包含待迁移 Tablet、服务的 Tablet 个数小于平均个数减去可容忍个数（默认值为 10）、正在进行的迁移任务不超过阀值等。
- 负载均衡
  扫描 RootTable 中的 Tablet，如果某台 ChunkServer 包含的某个表格的 Tablet 个数超过平均个数以及可容忍个数（默认值为 10）之和，则以这台 ChunkServer 为迁移源，并选择一台符合要求的 ChunkServer 为迁移目的地，生成 Tablet 迁移任务。

Tablet 复制和负载均衡生成的 Tablet 迁移并不会立即执行，而是会加入到迁移源的迁移任务列表中。RootServer 中的一个后台线程会扫描所有的 ChunkServer，然后执行每台 ChunkServer 的迁移任务列表中保存的迁移任务。Tablet 迁移时限制了每台 ChunkServer 同时进行的最大迁入和迁出任务数，从而防止一台新的 ChunkServer 刚上线时，迁入大量 Tablet 而负载过高。

例如：某 OceanBase 集群中包含 4 台 ChunkServer：ChunkServer1（包含 Tablet A1，A2，A3），ChunkServer2（包含 Tablet A3，A4），ChunkServer3（包含 Tablet A2），ChunkServer4（包含 Tablet A4）。假设 Tablet 副本数配置为 2，最多能够容忍的不均衡 Tablet 的个数为 0。

1. RootServer 后台线程首先执行 Tablet 复制，发现 Tablet A1 只有一个副本，于是，将 ChunkServer1 作为迁移源，并选择一台 ChunkServer 作

为迁移目的地（假设为 ChunkServer3），生成迁移任务<ChunkServer1，ChunkServer3，A1>。

2. 执行负载均衡，发现 ChunkServer1 包含 3 个 Tablet，超过平均值（平均值为 2），而 ChunkServer4 包含的 Tablet 个数小于平均值。
3. 将 ChunkServer1 作为迁移源， ChunkServer4 作为迁移目的地，选择某个 Tablet（假设为 A2），生成迁移任务<ChunkServer1，ChunkServer4，A2>。
4. 如果迁移成功，A2 将包含 3 个副本，可以通知 ChunkServer1 删除上面的 A2 副本。
   此时，Tablet 分布情况为：ChunkServer1（包含 Tablet A1，A3），ChunkServer2（包含 Tablet A3，A4），ChunkServer3（包含 Tablet A1，A2），ChunkServer4（包含 Tablet A2，A4）。 每个 Tablet 包含 2 个副本，且平均分布在 4 台 ChunkServer 上。

2.1.3 Tablet 分裂与合并

Tablet 分裂由 ChunkServer 在定期合并过程中执行。由于每个 Tablet 包含多个副本，且分布在多台 ChunkServer 上，如何确保多个副本之间的分裂点保持一致成为问题的关键。OceanBase 采用了一种比较直接的做法：每台 ChunkServer 使用相同的分裂规则。由于每个 Tablet 的不同副本之间的基准数据完全一致，且定期合并过程中冻结的增量数据也完全相同，只要分裂规则一致，分裂后的 Tablet 主键范围也保证相同。

OceanBase 曾经有一个线上版本的分裂规则如下：只要定期合并过程中产生的数据量超过 256MB，就生成一个新的 Tablet。假设定期合并产生的数据量为 257MB，那么最后将分裂为两个 Tablet，其中，前一个 Tablet（记为 r1）的数据量为 256MB，后一个 Tablet（记为 r2）的数据量为 1MB。接着，r1 接受新的修改，数据量很快又超过 256MB，于是，又分裂为两个 Tablet。系统运行一段时间后，充斥着大量数据量很少的 Tablet。

为了解决分裂产生小 Tablet 的问题，需要确保分裂以后的每个 Tablet 数据量大致相同。OceanBase 对每个 Tablet 记录了两个元数据：数据行数 row_count 以及 Tablet 大小（occupy_size）。根据这两个值，可以计算出每行数据的平均大小，即：occupy_size/row_count。 根据数据行平均大小，可以计算出分裂后的 Tablet 行数，从而得到分裂点。

Tablet 合并过程如下：

1. 合并准备：RootServer 选择若干个主键范围连续的小 Tablet。
2. Tablet 迁移：将待合并的若干个小 Tablet 迁移到相同的 ChunkServer 机器。
3. Tablet 合并：往 ChunkServer 机器发送 Tablet 合并命令，生成合并后的 Tablet 范围。

例如：某 OceanBase 集群中有 3 台 ChunkServer：ChunkServer1（包含 Tablet A1，A3），ChunkServer2（包含 Tablet A2，A3），ChunkServer3（包含 Tablet A1，A2），其中，A1 和 A2 分别为 10MB，A3 为 256MB。

1. RootServer 扫描 RootTable 后发现 A1 和 A2 满足 Tablet 合并条件，则发起 Tablet 迁移。
2. 假设将 A1 迁移到 ChunkServer2，使得 A1 和 A2 在相同的 ChunkServer 上。
3. 分别向 ChunkServer2 和 ChunkServer3 发起 Tablet 合并命令。
   Tablet 合并完成以后，Tablet 分布情况为：ChunkServer1（包含 Tablet A3），ChunkServer2（包含 Tablet A4(A1, A2)，A3），ChunkServer3（包含 Tablet A4(A1, A2)），其中，A4 是 Tablet A1 和 A2 合并后的结果。

说明：*每个 Tablet 包含多个副本，只要某一个副本合并成功，OceanBase 就认为 Tablet 合并成功，其它合并失败的 Tablet 将通过垃圾回收机制删除掉。*

2.1.4 UpdateServer 选主

为了确保一致性，RootServer 需要确保每个集群中只有一台 UpdateServer 提供写服务，这个 UpdateServer 称为主 UpdateServer。

RootServer 通过租约（Lease）机制实现 UpdateServer 选主。主 UpdateServer 必须持有 RootServer 的租约才能提供写服务，租约的有效期一般为 3~5 秒。正常情况下，RootServer 会定期给主 UpdateServer 发送命令，延长租约的有效期。如果主 UpdateServer 出现异常，RootServer 等待主 UpdateServer 的租约过期后才能选择其它的 UpdateServer 为主 UpdateServer 继续提供写服务。

RootServer 可能需要频繁升级，升级过程中 UpdateServer 的租约将很快过期，系统也会因此停服务。为了解决这个问题，RootServer 设计了优雅退出的机制，即 RootServer 退出之前给 UpdateServer 发送一个有效期超长的租约（比如半小时），承诺这段时间不进行主 UpdateServer 选举，用于 RootServer 升级。

2.1.5 RootServer 主备

每个集群一般部署一主一备两台 RootServer，主备之间数据强同步，即所有的操作都需要首先同步到备机，接着修改主机，最后才能返回操作成功。

RootServer 主备之间需要同步的数据包括：RootTable 中记录的 Tablet 分布信息、ChunkServerManager 中记录的 ChunkServer 机器变化信息以及 UpdateServer 机器信息。Tablet 复制、负载均衡、合并、分裂以及 ChunkServer 和 UpdateServer 上下线等操作都会引起 RootServer 内部数据变化，这些变化都将以操作日志的形式同步到备 RootServer。备 RootServer 实时回放这些操作日志，从而与主 RootServer 保持同步。

OceanBase 中的其它模块，比如 ChunkServer 和 UpdateServer，以及客户端通过 VIP（Virtual IP）访问 RootServer，正常情况下，VIP 总是指向主 RootServer。

当主 RootServer 出现故障时，部署在主备 RootServer 上的 Linux HA（heartbeat）软件能够检测到，并将 VIP 漂移到备 RootServer。

Linux HA 软件的核心包含两个部分：心跳检测部分和资源接管部分。心跳检测部分通过网络链接或者串口线进行，主备 RootServer 上的 heartbeat 软件相互发送报文来告诉对方自己当前的状态。如果在指定的时间内未收到对方发送的报文，那么就认为对方失败。这时需启动资源接管模块来接管运行在对方主机上的资源，这里的资源就是 VIP。备 RootServer 后台线程能够检测到 VIP 漂移到自身，于是自动切换为主机提供服务。

2.2 UpdateServer 实现机制

UpdateServer 用于存储增量数据，它是一个单机存储系统，由如下几个部分组成：

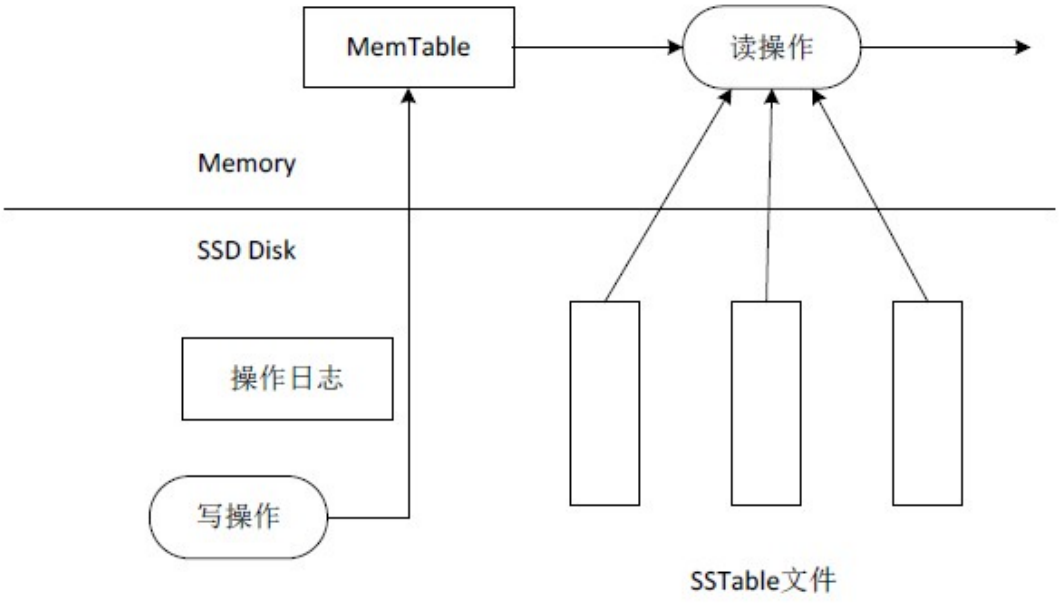
- 内存存储引擎：在内存中存储修改增量，支持冻结以及转储操作。
- 任务处理模型：包括网络框架、任务队列、工作线程等，针对小数据包做了专门的优化。
- 主备同步模块：将更新事务以操作日志的形式同步到备 UpdateServer。

UpdateServer 是 OceanBase 性能瓶颈点，核心是高效，实现时对锁（例如，无锁数据结构）、索引结构、内存占用、任务处理模型以及主备同步都需要做专门的优化。

2.2.1 内存存储引擎

UpdateServer 存储引擎如图 2-2所示。

图 2-2 UpdateServer 存储引擎



UpdateServer 存储引擎与 Bigtable 存储引擎相似，不同点在于：

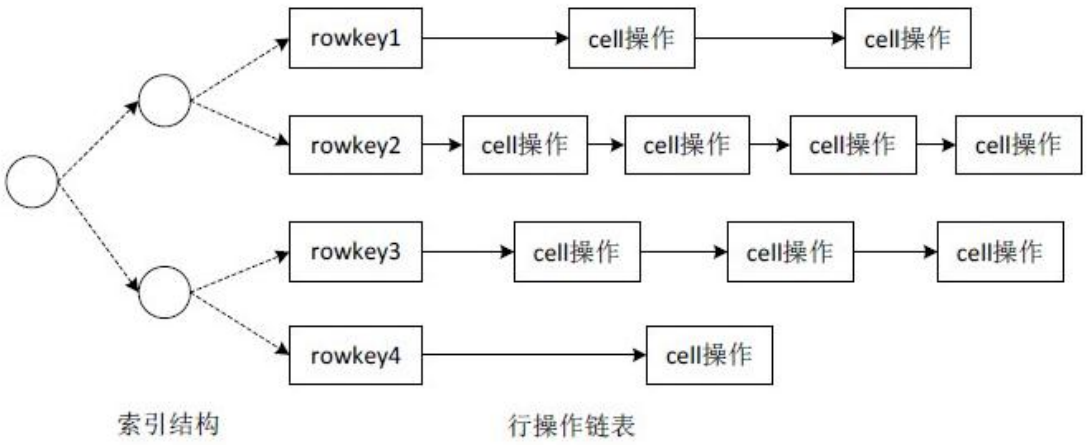
1. UpdateServer 只存储了增量更新数据，基准数据以 SSTable 的形式存储在 ChunkServer 上，而 Bigtable 存储引擎同时包含某个 Tablet 的基准数据和增量数据。
2. UpdateServer 内部所有表格共用 MemTable 以及 SSTable，而 Bigtable 中每个 Tablet 的 MemTable 和 SSTable 分开存放。
3. UpdateServer 的 SSTable 存储在 SSD 盘中，而 Bigtable 的 SSTable 存储在 GFS 中。

UpdateServer 存储引擎包含几个部分：操作日志、内存表（底层为一颗高性能的 B+树）以及 SSTable。更新操作首先记录到操作日志中，接着更新内存中活跃的 MemTable（Active MemTable），活跃的 MemTable 到达一定大小后将被冻结，成为 Frozen MemTable，同时创建新的 Active MemTable。Frozen MemTable 将以 SSTable 文件的形式转储到 SSD 盘中。

- 操作日志
  OceanBase 中有一个专门的提交线程负责确定多个写事务的顺序（即事务 id）。将这些写事务的操作追加到日志缓冲区，并将日志缓冲区的内容写入日志文件。为了防止写操作日志污染操作系统的缓存，写操作日志文件采用 Direct IO 的方式实现。
- MemTable
  MemTable 底层是一颗高性能内存 B+树。MemTable 封装了 B+树，对外提供统一的读写接口。

B+树中的每个元素对应 MemTable 中的一行操作，key 为行主键，value 为行操作链表的指针。每行的操作按照时间顺序构成一个行操作链表，如图 2-3 所示。

图 2-3 MemTable 内存结构



MemTable 内存结构包含两部分：索引结构以及行操作链表，索引结构为 B+树，支持插入、删除、更新、随机读取以及范围查询操作。行操作链表保存的是对某一行各个列（每个行和列确定一个单元，称为 Cell）的操作，例如，对主键为 1 的商品有 3 个操作，分别是：将商品购买人数修改为 100，删除该商品，将商品

名称修改为“女鞋”。那么，该商品的行操作链中将保存三个 Cell，分别为：<update，购买人数，100>、<delete，*> 以及<update，商品名，“女鞋”>。 也就是说，MemTable 中存储的是对该商品的所有操作，而不是最终结果。另外，MemTable 删除一行也只是往行操作链表的末尾加入一个逻辑删除标记，即<delete，*>，而不是实际删除索引结构或者行操作链表中的行内容。

MemTable 实现时做了很多优化，包括：

- Hash 索引
  针对主要操作为随机读取的应用，MemTable 不仅支持 B+树索引，还支持 Hash 索引，UpdateServer 内部会保证两个索引之间的一致性。
- 内存优化
  行操作链表中每个 cell 操作都需要存储操作列的编号（column_id）、操作类型（更新操作还是删除操作）、操作值以及指向下一个 cell 操作的指针，如果不做优化，内存膨胀会很大。为了减少内存占用，MemTable 实现时会对整数值进行变长编码，并将多个 cell 操作编码后序列到同一块缓冲区中，共用一个指向下一个 cell 操作缓冲区的指针。

```
// 开启一个事务
// @param =in] trans_type 事务类型，可能为读事务或者写事务
// @param =out] td 返回的事务描述符
int start_transaction(const TETransType trans_type,     MemTableTransDescriptor& td);

// 提交或者回滚一个事务
// @param =in] td 事务描述符
// @param =in] rollback 是否回滚，默认为 false
int end_transaction(const MemTableTransDescriptor td, bool rollback = false);

// 执行随机读取操作，返回一个迭代器
// @param =in] td 事务描述符
// @param =in] table_id 表格编号
// @param =in] row_key 待查询的主键
// @param =out] iter 返回的迭代器
int get(const MemTableTransDescriptor td,   const uint64_t table_id,   const
ObRowkey& row_key, MemTableIterator& iter);

// 执行范围查询操作，返回一个迭代器
// @param =in] td 事务描述符
// @param =in] range 查询范围，包括起始行、结束行，开区间或者闭区间
// @param =out] iter 返回的迭代器
int scan(const MemTableTransDescriptor td, const ObRange& range,
MemTableIterator& iter);

// 开始执行一次更新操作
// @param =in] td 事务描述符
```

```
int start_mutation(const MemTableTransDescriptor td);

// 提交或者回滚一次更新操作
// @param =in] td 事务描述符
// @param =in] rollback 是否回滚
int end_mutation(const MemTableTransDescriptor td, bool rollback);

// 执行更新操作
// @param =in] td 事务描述符
// @param =in] mutator 更新操作，包含一个或者多个对多个表格的 cell 操作
int set(const MemTableTransDescriptor td,  ObUpsMutator& mutator);
```

对于读事务，操作步骤如下：

1. 调用 start_transaction 开始一个读事务，获得事务描述符。
2. 执行随机读取或者扫描操作，返回一个迭代器。
3. 调用 end_transaction 提交或者回滚一个事务。

```
class MemTableIterator
    {
 public:
 // 迭代器移动到下一个 cell
 int next_cell();
 // 获取当前 cell 的内容
 // @param [out] cell_info 当前 cell 的内容，包括表名（table_id），行主键（row_key），
列编号（column_id）以及列值（column_value）
 int get_cell(ObCellInfo** cell_info);
 // 获取当前 cell 的内容
 // @param [out] cell_info 当前 cell 的内容
 // @param is_row_changed 是否迭代到下一行
 int get_cell(ObCellInfo** cell_info, bool * is_row_changed);
 };
```

读事务返回一个迭代器 MemTableIterator，通过它可以不断地获取下一个读到的 cell。上面的例子中，读取编号为 1 的商品可以得到一个迭代器，从这个迭代器中可以读出行操作链中保存的 3 个 Cell，依次为：<update，购买人数，100>，<delete，*>，<update，商品名，“女鞋”>。

写事务总是批量执行，步骤如下：

1. 调用 start_transaction 开始一批写事务，获得事务描述符。
2. 调用 start_mutation 开始一次写操作。
3. 执行写操作，将数据写入到 MemTable 中。
4. 调用 end_mutation 提交或者回滚一次写操作；如果还有写事务，转到“步骤 2”。

5. 调用 end_transaction 提交写事务。

- SSTable

  当活跃的 MemTable 超过一定大小或者管理员主动发起冻结命令时，活跃的 MemTable 将被冻结，生成冻结的 MemTable，并同时以 SSTable 的形式转储到 SSD 盘中。

  SSTable 的详细格式请参见本手册“2.3 ChunkServer 实现机制”章节。与 ChunkServer 中的 SSTable 不同的是，UpdateServer 中所有的表格共用一个 SSTable，且 SSTable 为稀疏格式。也就是说，每一行数据的每一列可能存在，也可能不存在更新操作。

  另外，OceanBase 设计时也尽量避免读取 UpdateServer 中的 SSTable。只要内存足够，冻结的 MemTable 会保留在内存中。系统会尽快将冻结的数据通过定期合并转移到 ChunkServer 中去，以后不再需要访问 UpdateServer 中的 SSTable 数据。

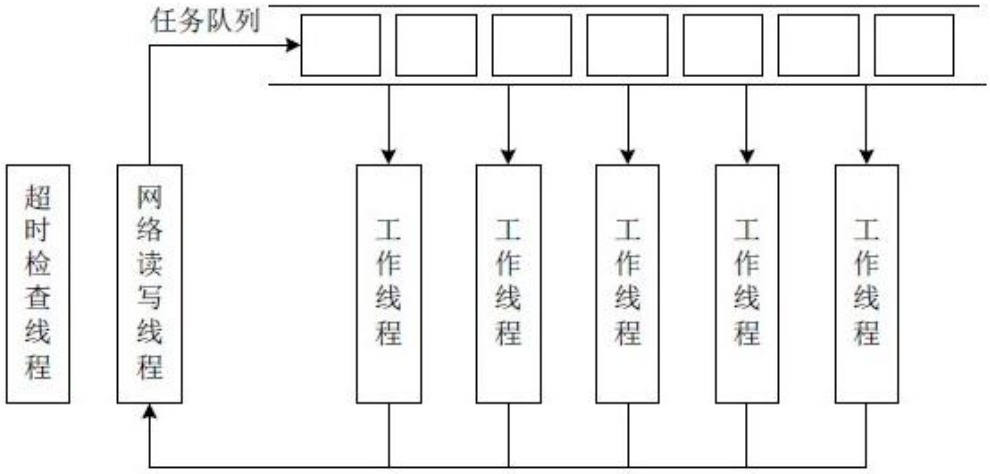
2.2.2 任务处理模型

任务模型包括网络框架、任务队列和工作线程。UpdateServer 最初的任务模型基于淘宝网实现的 Tbnet 框架（已开源，请参见 http://code.taobao.org/p/tb-common-utils/src/trunk/tbnet/）。Tbnet 封装得很好，使用比较方便，每秒收包个数最多可以达到接近 10 万，不过仍然无法完全发挥 UpdateServer 收发小数据包以及内存服务的特点。OceanBase 后来采用优化过的任务模型 Libeasy，小数据包处理能力得到进一步提升。

- Tbnet

  如图 2-4 所示，Tbnet 队列模型本质上是一个“生产者-消费者”队列模型，有两个线程：网络读写线程和超时检查线程。其中，网络读写线程执行事件循环，当服务器端有可读事件时，调用回调函数读取请求数据包，生成请求任务，并加入到任务队列中。工作线程从任务队列中获取任务，处理完成后触发可写事件，网络读写线程会将处理结果发送给客户端。超时检查线程用于将超时的请求移除。

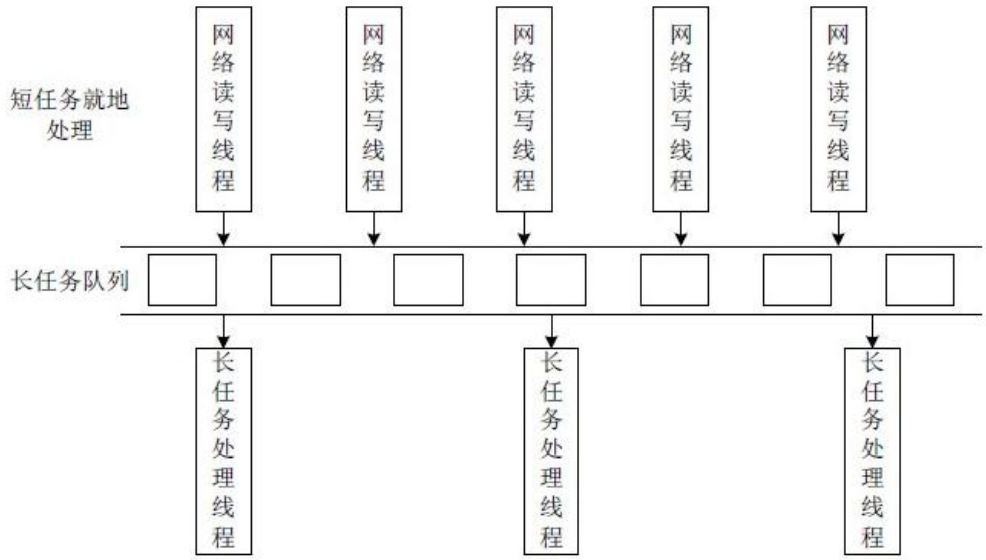
- **图 2-4 Tbnet 队列模型**



Tbnet 模型的问题在于多个工作线程从任务队列获取任务需要加锁互斥，这个过程将产生大量的上下文切换，测试发现，当 UpdateServer 每秒处理包的数量超过 8 万个时，UpdateServer 每秒的上下文切换次数接近 30 万次，在测试环境中已经达到极限（测试环境配置：Linux 内核 2.6.18，CPU 为 2 * Intel Nehalem E5520，共 8 核 16 线程）。

- Libeasy
  为了解决收发小数据包带来的上下文切换问题，OceanBase 目前采用 Libeasy 任务模型。Libeasy 采用多个线程收发包，增强了网络收发能力，每个线程收到网络包后立即处理，减少了上下文切换。
  如图 2-5 所示，UpdateServer 有多个网络读写线程。每个线程通过 Linux epool 监测一个套接字集合上的网络读写事件。每个套接字只能同时分配给一个线程。当网络读写线程收到网络包后，立即调用任务处理函数，如果任务处理时间很短，可以很快完成并回复客户端，不需要加锁，避免了上下文切换。UpdateServer 中大部分任务为短任务，比如随机读取内存表，另外还有少量任务需要等待共享资源上的锁，可以将这些任务加入到长任务队列中，交给专门的长任务处理线程处理。

- **图 2-5 Libeasy 任务模型**



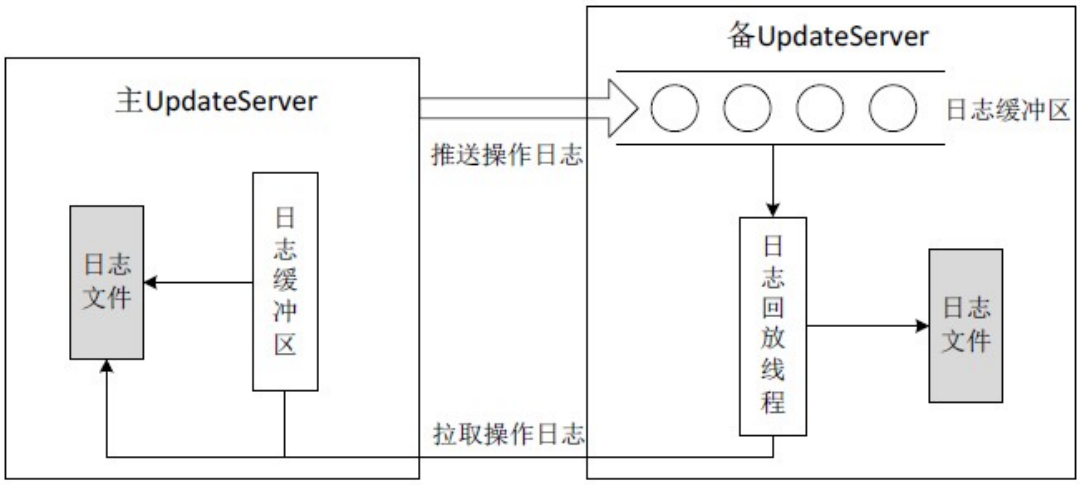
由于每个网络读写线程处理一部分预先分配的套接字，这就可能出现某些套接字上请求特别多而导致负载不均衡的情况。例如，有两个网络读写线程 thread1 和 thread2，其中 thread1 处理套接字 fd1、fd2，thread2 处理套接字 fd3、fd4，fd1 和 fd2 上每秒 1000 次请求，fd3 和 fd4 上每秒 10 次请求，两个线程之间的负载很不均衡。为了处理这种情况，Libeasy 内部会自动在网络读写线程之间执行负载均衡操作，将套接字从负载较高的线程迁移到负载较低的线程。

2.2.3 主备同步模块

在本手册的“1.4.1 一致性选择”章节已经介绍了 UpdateServer 的一致性选择。OceanBase 选择了强一致性，主 UpdateServer 往备 UpdateServer 同步操作日志，如果同步成功，则主 UpdateServer 读写操作生效，并将更新成功消息返回给客户端；否则，主 UpdateServer 会把备 UpdateServer 从同步列表中剔除。另外，剔除备 UpdateServer 之前需要通知 RootServer，从而防止 RootServer 将不一致的备 UpdateServer 选为主 UpdateServer。

如图 2-6 所示，主 UpdateServer 往备机推送操作日志，备 UpdateServer 的接收线程接收日志，并写入到一块全局日志缓冲区中。主 UpdateServer 接着更新本地内存并将日志刷到磁盘文件中，最后回复客户端写入操作成功。这种方式实现了强一致性，如果主 UpdateServer 出现故障，备 UpdateServer 包含所有的更新操作，因而能够完全无缝地切换为主 UpdateServer 继续提供服务。另外，主备同步过程中要求主机刷磁盘文件，备机只需要写内存缓冲区，强同步带来的额外延时也几乎可以忽略。

图 2-6 UpdateServer 主备同步原理



正常情况下，备 UpdateServer 的日志回放线程会从全局日志缓冲区中读取操作日志，在内存中回放并同时将操作日志刷到备机的日志文件中。如果发生异常，比如备 UpdateServer 刚启动或者主备之间网络刚恢复，全局日志缓冲区中没有日志或者日志不连续，此时，备 UpdateServer 需要主动请求主 UpdateServer 拉取操作日志。主 UpdateServer 首先查找日志缓冲区，如果缓冲区中没有数据，还需要读取磁盘日志文件，并将操作日志回复备 UpdateServer。

2.3 ChunkServer 实现机制

ChunkServer 用于存储基线数据，它由如下基本部分组成：

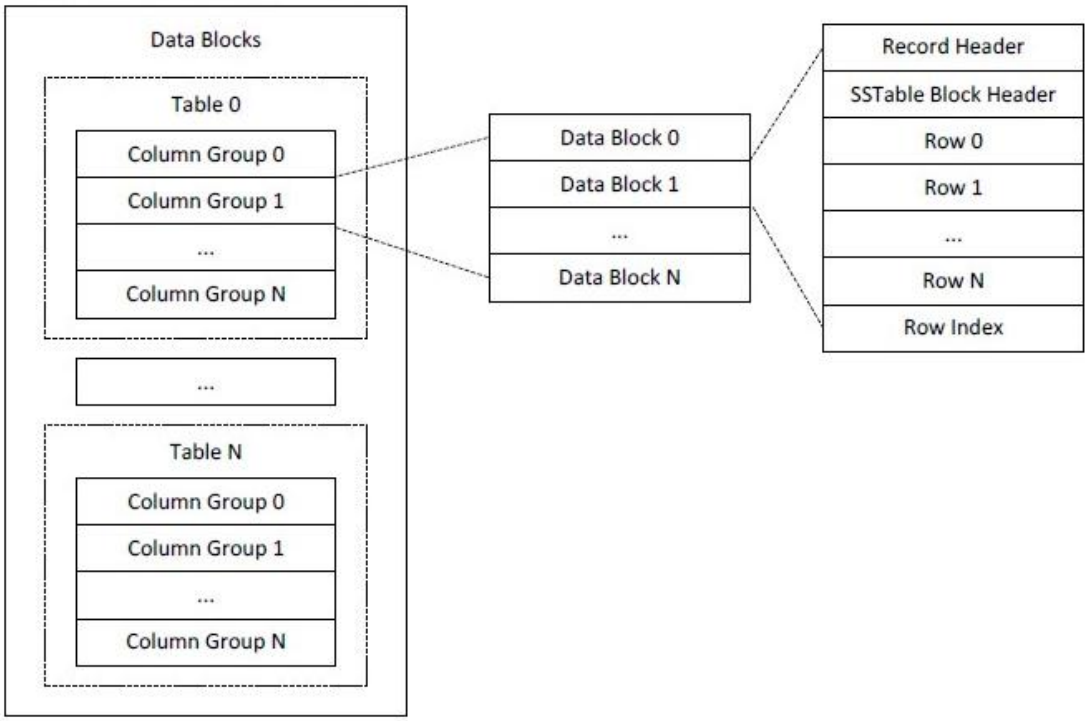
- SSTable，根据主键有序存储每个 Tablet 的基线数据。
- 基于 LRU（Least Recently Used）实现块缓存（Block cache）以及行缓存（Row cache）。
- 实现 Direct IO，磁盘 IO 与 CPU 计算并行化。
- 通过定期合并获取 UpdateServer 的冻结数据，从而分散到整个集群。
- 主动实现 Tablet 分裂，配合 RootServer 实现 Tablet 迁移、删除、合并。

每台 ChunkServer 服务着几千到几万个 Tablet 的基线数据，每个 Tablet 由若干个 SSTable 组成（一般为 1 个）。

2.3.1 SSTable

如图 2-7 所示，SSTable 中的数据按主键排序后存放在连续的数据块（Block）中，Block 之间也有序。接着，存放数据块索引（Block Index），由每个 Block 最后一行的主键（End Key）组成，用于数据查询中的 Block 定位。接着，存放 Bloom Filter 和表格的 Schema 信息。最后，存放固定大小的 Trailer 以及 Trailer 的偏移位置。

图 2-7 SSTable 包含多个 Table/Column Group



查找 SSTable 时，首先从 Tablet 的索引信息中读取 SSTable Trailer 的偏移位置，接着获取 Trailer 信息。根据 Trailer 中记录的信息，可以获取 Block Index 的大小和偏移，从而将整个 Block Index 加载到内存中。根据 Block Index 记录的每个 Block 的 End Key，可以通过二分查找定位到查找的 Block。最后将 Block 加载到内存中，通过二分查找 Block 中记录的行索引（Row Index）查找到具体某一行。本质上看，SSTable 是一个两级索引结构：块索引和行索引。而整个 ChunkServer 是一个三级索引结构：Tablet 索引、块索引和行索引。

SSTable 分为两种格式：稀疏格式和稠密格式。对于稀疏格式，某些列可能存在，也可能不存在，因此每一行只存储包含实际值的列，每一列存储的内容为：<Column ID，Column Value>；而稠密格式中每一行都需要存储所有列，每一列只需要存储 Column Value，不需要存储 Column ID，这是因为 Column ID 可以从表格 Schema 中获取。

例如：假设有一张表格包含 10 列，列 ID 为 1~10，表格中有一行的数据内容为：column_id=2 column_id =3 column_id =5 column_id =7 column_id =8 20 30 50 70 80
那么，如果采用稀疏格式存储，内容为：<2，20>，<3，30>，<5，50>，<7，70>，<8，80>；如果采用稠密格式存储，内容为：null，20，30，null，50，null，70，80，null，null。

ChunkServer 中的 SSTable 为稠密格式，而 UpdateServer 中的 SSTable 为稀疏格式，且存储了多张表格的数据。另外，SSTable 支持列组（Column Group），将同一个 Column Group 下的多个列的内容存储在一块。

当一个 SSTable 中包含多个 Table/Column Group 时，数据按照[table_id，column group id，row_key]的形式有序存储。另外，SSTable 支持压缩功能，压缩以

Block 为单位。每个 Block 写入磁盘之前调用压缩算法执行压缩，读取时需要解压缩。用户可以自定义 SSTable 的压缩算法，目前支持的算法包括 LZO 和 Snappy。

SSTable 的操作接口分为写入和读取两个部分：写入类为 ObSSTableWriter，读取类为 ObSSTableGetter（随机读取）和 ObSSTableScanner（范围查询）。

```
class ObSSTableWriter
    {
 public:
 // 创建 SSTable
 // @param [in] schema 表格 schema 信息
 // @param [in] path SSTable 在磁盘中的路径名
 // @param [in] compressor_name 压缩算法名
 // @param [in] store_type SSTable 格式，稀疏格式或者稠密格式
 // @param [in] block_size 块大小，默认 64KB
 int create_sstable(const ObSSTableSchema& schema, const ObString& path, const
ObString& compressor_name, const int store_type, const int64_t block_size);
// 往 SSTable 中追加一行数据
    // @param [in] row 一行 SSTable 数据
    // @param [out] space_usage 追加完这一行后 SSTable 大致占用的磁盘空间
    int append_row(const ObSSTableRow& row, int64_t& space_usage);
// 关闭 SSTable，将往磁盘中写入 Block Index，Bloom Filter，Schema，Trailer 等信息。
    // @param [out] trailer_offset 返回 SSTable 的 Trailer 偏移量
    int close_sstable(int64_t& trailer_offset); };
```

定期合并过程将产生新的 SSTable 的过程如下：

1. 调用 create_sstable 函数创建一个新的 SSTable。
2. 不断调用 append_row 函数往 SSTable 中追加一行一行数据。
3. 调用 close_sstable 完成 SSTable 写入。

与“2.2.1 内存存储引擎”节中的 MemTableIterator 一样，ObSSTableGetter 和 ObSSTableScanner 实现了迭代器接口，通过它可以不断地获取 SSTable 的下一个 cell。

```
class ObIterator
    {
 public:
 // 迭代器移动到下一个 cell
 int next_cell();
// 获取当前 cell 的内容
    // @param [out] cell_info 当前 cell 的内容，包括表名（table_id），行主键（row_key），
列编号（column_id）以及列值（column_value）
    int get_cell(ObCellInfo** cell_info);
// 获取当前 cell 的内容
```

```
// @param [out] cell_info  当前 cell 的内容
// @param is_row_changed  是否迭代到下一行
int get_cell(ObCellInfo** cell_info, bool * is_row_changed); };
```

OceanBase 读取的数据可能来源于 MemTable，也可能来源于 SSTable，或者是合并多个 MemTable 和多个 SSTable 生成的结果。无论底层数据来源如何变化，上层的读取接口总是 ObIterator。

2.3.2 缓存实现

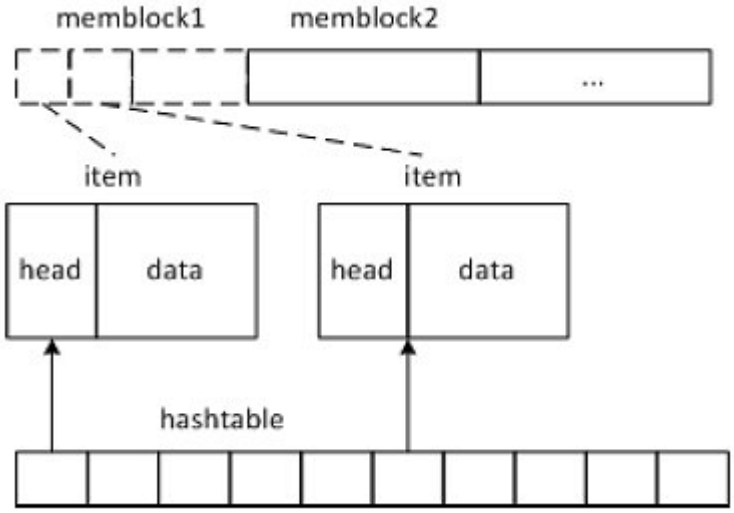
ChunkServer 中包含三种缓存：块缓存（Block Cache），行缓存（Row Cache）和块索引缓存（Block Index Cache）。不同缓存的底层采用相同的实现方式。

经典的 LRU 缓存实现包含两个部分：Hash 表和 LRU 链表。其中，Hash 表用于查找缓存中的元素，LRU 链表用于淘汰。每次访问 LRU 缓存时，需要将被访问的元素移动到 LRU 链表的头部，从而避免被很快淘汰，这个过程需要锁住 LRU 链表。

如图 2-8 所示，块缓存和行缓存底层都是一个 Key-Value Cache，实现如下：

- OceanBase 一次分配 1MB 的连续内存块（称为 memblock），每个 memblock 包含若干缓存项（item）。添加 item 时，只需要简单地将 item 追加到 memblock 的尾部；另外，缓存淘汰以 memblock 为单位，而不是以 item 为单位。
- OceanBase 没有维护 LRU 链表，而是对每个 memblock 都维护了访问次数和最近访问时间。淘汰 memblock 时，对所有的 memblock 按照访问次数和最近访问时间排序，淘汰访问次数少且长时间没有访问的 memblock。这种实现方式通过牺牲 LRU 算法的精确性，来规避访问 LRU 链表的全局锁。
- 每个 memblock 维护了引用计数，读取缓存项时所在 memblock 的引用计数加 1，淘汰 memblock 时引用计数减 1，引用计数为 0 时 memblock 可以回收重用。通过引用计数，实现读取 memblock 中的缓存项不加锁。

图 2-8 Key-Value Cache 实现



2.3.3 IO 实现

OceanBase 没有使用操作系统本身的 page cache 机制，而是自己实现缓存。相应地，IO 也采用 Direct IO 实现，并且支持磁盘 IO 与 CPU 计算并行化。

ChunkServer 采用 Linux 的 Libaio 实现异步 IO，并通过双缓冲区机制实现磁盘预读与 CPU 处 理并行化，步骤如下：

1. 分配 current 以及 ahead 两个缓冲区，current 为当前缓冲区，ahead 为预读缓冲区。
2. 使用 current 缓冲区读取数据，current 缓冲区通过 Libaio 发起异步读取请求，接着等待异步读取完成。
3. 异步读取完成后，将 current 缓冲区返回上层执行 CPU 计算，同时，原来的 ahead 变为新的 current，发送异步读取请求将数据读取到新的 current 缓冲区。CPU 计算完成后，原来的 current 缓冲区变为空闲，成为新的 ahead，准备作为下一次预读的缓冲区。
4. 重复“步骤 3”，直到所有数据全部读完。

例如：假设需要读取的数据范围为(1, 150]。分三次读取：(1, 50], (50, 100], (100, 150]，current 和 ahead 缓冲区分别记为 A 和 B。

1. 发送异步请求将(1, 50]读取到缓冲区 A，等待读取完成。
2. 对缓冲区 A 执行 CPU 计算，发送异步请求，将(50, 100]读取到缓冲区 B。
3. 如果 CPU 计算先于磁盘读取完成，那么，缓冲区 A 变为空闲，等到(50, 100]读取完成后将缓冲区 B 返回上层执行 CPU 计算，同时，发送异步请求，将(100, 150]读取到缓冲区。

4. 如果磁盘读取先于 CPU 计算完成，那么，首先等待缓冲区 A 上的 CPU 计算完成，接着，将缓冲区 B 返回上层执行 CPU 计算，同时，发送异步请求，将(100,150]读取到缓冲区 A。
5. 等待(100, 150]读取完成后，将缓冲区 A 返回给上层执行 CPU 计算。

2.3.4 定期合并

定期合并也称为每日合并，主要将 UpdateServer 中的增量更新分发到 ChunkServer 中。其主要流程如下：

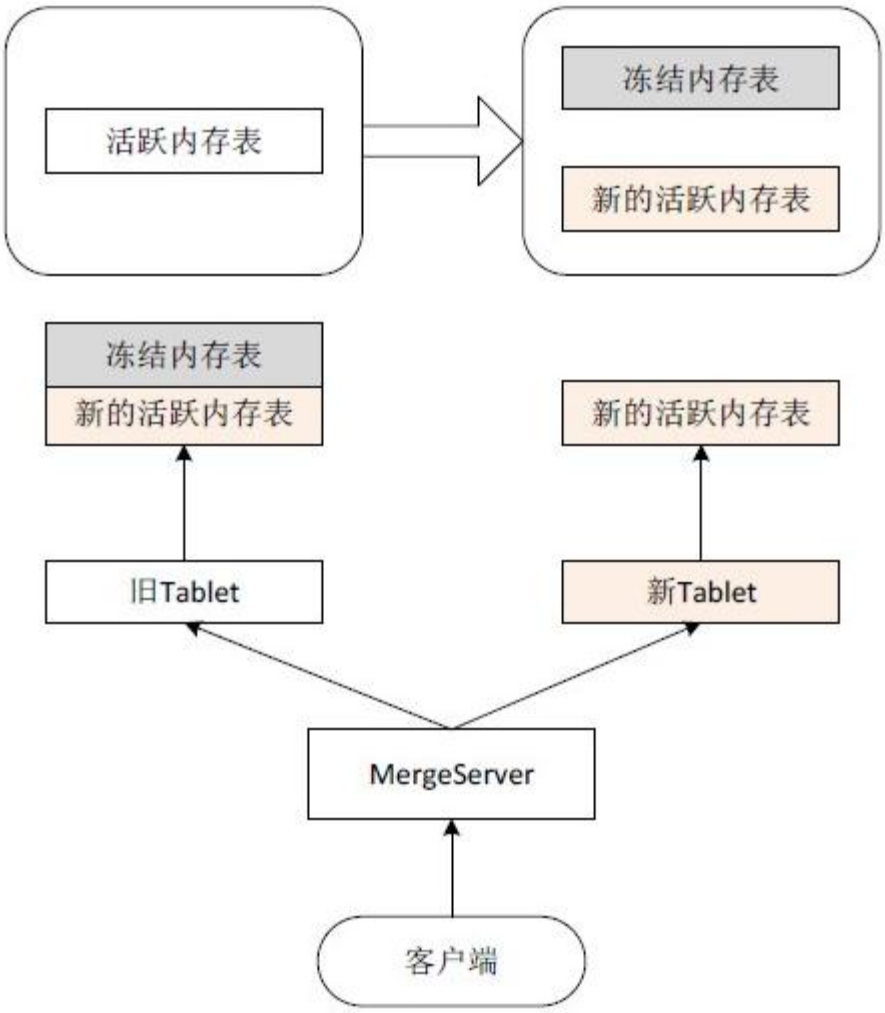
1. UpdateServer 冻结当前的活跃内存表（Active MemTable），生成冻结内存表，并开启新的活跃内存表。后续的更新操作都写入新的活跃内存表。
2. UpdateServer 通知 RootServer 数据版本发生了变化。之后，RootServer 通过心跳消息通知 ChunkServer。
3. 每台 ChunkServer 启动定期合并，从 UpdateServer 获取每个 Tablet 对应的增量更新数据。

定期合并过程中 ChunkServer 需要将本地 SSTable 中的基准数据与冻结内存表的增量更新数据执行一次多路归并，融合后生成新的基准数据并存放到新的 SSTable 中。定期合并对系统服务能力影响很大，往往安排在每天服务低峰期执行（例如凌晨 1 点开始）。

如图 2-9 所示，活跃内存表冻结后生成冻结内存表，后续的写操作进入新的活跃内存表。定期合并过程中 ChunkServer 需要读取 UpdateServer 中冻结内存表的数据、融合后生成新的 Tablet，即：

```
新 Tablet = 旧 Tablet + 冻结内存表
```

图 2-9 定期合并



虽然定期合并过程中各个 ChunkServer 的各个 Tablet 合并时间和完成时间可能都不相同，但并不影响读取服务。如果 Tablet 没有合并完成，那么使用旧 Tablet，并且读取 UpdateServer 中的冻结内存表以及新的活跃内存表；否则，使用新 Tablet，只读取新的活跃内存表，即：

```
查询结果 = 旧 Tablet + 冻结内存表 + 新的活跃内存表 = 新 Tablet + 新的活跃内存表
```

UpdateServer 进行数据冻结可以分为大版本冻结和小版本冻结：假设 UpdateServer 中的数据版本为“111”，其允许的小版本数为“3”。在 UpdateServer 中会先产生一个小版本数据“111-1”，当小版本的数据达到一定大小时会进行冻结（数据大小与 UpdateServer 服务器内存相关），我们称为“小版本冻结”。同时将产生一个新的小版本“111-2”，以此类推。当冻结的小版本数达到 UpdateServer 允许的最大小版本数时，即所有小版本数据均冻结，我们称为“大版本冻结”。

当 UpdateServer 执行了大版本冻结时，ChunkServer 将执行定期合并。ChunkServer 唤醒若干个定期合并线程（比如 10 个），每个线程执行如下流程：

1. 加锁获取下一个需要定期合并的 Tablet。

2. 根据 Tablet 的主键范围读取 UpdateServer 中的更新操作。
3. 将每行数据的基线数据和增量数据合并后，产生新的基线数据，并写入到新的 SSTable 中。
4. 更改 Tablet 索引信息，指向新的 SSTable。

等到 ChunkServer 上所有的 Tablet 定期合并都执行完成后，ChunkServer 会向 RootServer 汇报，RootServer 会更新 RootTable 中记录的 Tablet 版本信息。另外，定期合并过程中 ChunkServer 的压力比较大，需要控制合并速度，否则可能影响正常的读取服务。

2.4 消除更新瓶颈

从表面上看，UpdateServer 单点像是 OceanBase 架构的软肋，然而经过 OceanBase 团队持续不断地的性能优化以及旁路导入功能的开发，单点的架构在实践过程中经受住了线上考验。每年淘宝网“双十一”光棍节，OceanBase 系统都承载着核心的数据库业务，系统访问量出现 5 到 10 倍的增长，而 OceanBase 只需简单地增加机器即可。

本节介绍 OceanBase 的优化工作，包括读写性能优化以及旁路导入功能。

2.4.1 读写优化回顾

OceanBase 中的 UpdateServer 相当于一个内存数据库，能够支持每秒数百万次单行读写操作，这样的性能对于目前关系数据库的应用场景都是足够的。为了达到这样的性能指标，我们已经完成或正在进行的工作如下：

- 网络框架优化
  在本手册的“2.2.2 任务处理模型”章节中提到，如果不经过优化，UpdateServer 单机每秒最多能够接收的数据包个数只有 10 万个左右，而经过优化后的 libeasy 框架对于千兆网卡每秒最多收包个数超过 50 万，对于万兆网卡则超过 100 万。另外，UpdateServer 内部还会在软件层面实现多块网卡的负载均衡，从而更好地发挥多网卡的优势。通过网络框架优化，使得单机支持百万次操作成为可能。
- 高性能内存数据结构
  UpdateServer 的底层是一颗高性能内存 B+树。为了最大程度地发挥多核的优势，B+树实现时大部分情况下都做到了无锁（lock*free）。测试数据表明，即使在普通的 16 核机器上，OceanBase 的 B+树每秒支持的单行修改操作都超过 150 万次。
- 写操作日志优化
  在软件层面，写操作日志涉及到的工作主要有以下三点：
  - Group commit。将多个写操作聚合在一起，一次性刷入磁盘中。
  - 降低日志缓冲区的锁冲突。多个线程同时往日志缓冲区中追加数据，实现时需要尽可能地减少追加过程的锁冲突。追加过程包含两个阶

段：第一个阶段是占位，第二个阶段是拷贝数据。相比较而言，拷贝数据比较耗时。实现的关键在于只对占位操作互斥，而允许多线程并发拷贝数据。例如，有两个线程，线程 1 和线程 2，他们分别需要往缓冲区追加大小为 100 字节和大小为 300 字节的数据。假设缓冲区初始为空，那么，线程 1 可以首先占住位置 0~100，线程 2 接着占住 100~300。最后，线程 1 和线程 2 同时将数据拷贝到刚才占住的位置。

- 日志文件并发写入。UpdateServer 中每个日志缓冲区的大小一般为 64MB，如果写入太快，那么，很快会产生多个日志缓冲区需要刷入磁盘，可以并发地将这些日志缓冲区刷入不同的磁盘。

在硬件层面，UpdateServer 机器需要配置较好的 RAID 卡。这些 RAID 卡自带缓存，而且容量比较大（例如 1GB），从而进一步提升写磁盘性能。

- 内存容量优化
  随着数据不断写入，UpdateServer 的内存容量将成为瓶颈。因此，有两种解决思路。第一种思路是精心设计 UpdateServer 的内存数据结构，尽可能地节省内存使用；另外一种思路就是将 UpdateServer 内存中的数据很快地分发出去。
  OceanBase 实现了这两种思路。首先，UpdateServer 会将内存中的数据编码为精心设计的格式。例如，100 以内的 64 位整数在内存中只需要占用两个字节。这种编码格式不仅能够有效地减少内存占用，而且往往使得 CPU 缓存能够容纳更多的数据。因此，也不会造成太多额外的 CPU 消耗。另外，当 UpdateServer 的内存使用量到达一定大小时，OceanBase 会触发数据合并操作，将 UpdateServer 的数据分发到集群中的 ChunkServer 中，从而避免 UpdateServer 的内存容量成为瓶颈。

2.4.2 数据旁路导入

虽然 OceanBase 内部实现了大量优化技术，但是 UpdateServer 单点写入对于某些 OLAP 应用仍然可能成为问题。这些应用往往需要定期（例如每天、每月）导入大批数据，对导入性能要求很高。为此，OceanBase 专门开发了旁路导入功能，本节介绍直接将数据导入到 ChunkServer 中的方法。

OceanBase 的数据按照全局有序排列，因此，ChunkServer 旁路导入的过程如下：

1. 使用工具（例如：Hadoop MapReduce）将所有的数据排序，并且划分为一个一个有序的范围，每个范围对应一个 SSTable 文件。
2. 将 SSTable 文件并行拷贝到集群内所有的 ChunkServer 中。
3. 通过 RootServer 要求每个 ChunkServer 并行加载这些 SSTable 文件。每个 SSTable 文件对应 ChunkServer 的一个子表。

4. ChunkServer 加载完本地的 SSTable 文件后向 RootServer 汇报，RootServer 接着将汇报的子表信息更新到 RootTable 中。

例如：有 4 台 ChunkServer 分别为 A、B、C 和 D。所有的数据排好序后划分为 6 个范围：r1(0~100]，r2(100~200]，r3(200~300]，r4(300~400]，r5(400~500]，r6(500~600]，对应的 SSTable 文件分别记为 sst1，sst2，…，sst6。

1. 假设每个子表存储两个副本，那么，拷贝完 SSTable 文件后，可能的分布情况为：
   A：sst1，sst3，sst4
   B：sst2，sst3，sst5
   C：sst1，sst4，sst6
   D：sst2，sst5，sst6
2. 接着，每个 ChunkServer 分别加载本地的 SSTable 文件，完成后向 RootServer 汇报。RootServer 最终会将这些信息记录到 RootTable 中，如下：
   r1(0~100]：A、C
   r2(100~200]：B、D
   r3(200~300]：A、B
   r4(300~400]：A、C
   r5(400~500]：B、D
   r6(500~600]：C、D
3. 如果导入的过程中 ChunkServer 发生故障，例如拷贝 sst1 到机器 C 失败，那么，旁路导入模块会自动选择另外一台机器拷贝数据。

当然，实现旁路导入功能时还需要考虑很多问题。例如：如何支持将数据导入到多个数据中心的主备 OceanBase 集群等。

2.5 实现技巧

OceanBase 开发过程中使用了一些小技巧，这些技巧说起来相当朴实，却实实在在地解决了当时面临的问题。本节通过几个例子介绍开发过程中用到的实现技巧。

2.5.1 内存管理

内存管理是 C++高性能服务器的核心问题。一些通用的内存管理库，比如 Google TCMalloc 在内存申请/释放速度、小内存管理、锁开销等方面都已经做得相当卓越了，然而，我们并没有采用。这是因为，通用内存管理库在性能上毕竟不如专用的内存池，更为严重的是，它鼓励了开发人员忽视内存管理的陋习，比如在服务器程序中滥用 C++标准模板库（STL）。

在分布式存储系统开发初期，内存相关的 Bug 相当常见，比如内存越界，服务器出现 Core Dump，这些 Bug 都非常难以调试。因此，这个时期内存管理的首要问题并不是高效，而是可控性，并防止内存碎片。

OceanBase 系统有一个全局的定长内存池，这个内存池维护了由 64KB 大小的定长内存块组成的空闲链表。

- 如果申请的内存不超过 64KB，则尝试从空闲链表中获取一个 64KB 的内存块返回给申请者。如果空闲链表为空，则先从操作系统中申请一批大小为 64KB 的内存块加入空闲链表。释放时将 64KB 的内存块加入到空闲链表中以便下次重用。
- 如果申请的内存超过 64KB，则直接调用 Glibc 的 malloc 函数，向操作系统申请用户所需大小的内存块。释放时直接调用 Glibc 的 free 函数，将内存块归还操作系统。

OceanBase 的全局内存池实现简单，但内存使用率比较低，即使申请几个字节的内存，也需要占用大小为 64KB 的内存块。因此，全局内存池不适合管理小块内存，每个需要申请内存的模块，比如 UpdateServer 中的 MemTable，ChunkServer 中的缓存等，都只能从全局内存池中申请大块内存，每个模块内部再实现专用的内存池。每个线程处理读写请求时需要使用临时内存，为了提高效率，每个线程会缓存若干个大小分别为 64KB 和 2MB 的内存块，每个线程总是首先尝试从线程局部缓存中申请内存，如果申请不到，则再从全局内存池中申请。

全局内存池的意义如下：

- 全局内存池可以统计每个模块的内存使用情况。如果出现内存泄露，可以很快定位到发生问题的模块。
- 全局内存池可用于辅助调试。例如，可以将全局内存池中申请到的内存块按字节填充为某个非法的值（比如 0xFE），当出现内存越界等问题时，服务器程序会很快在出现问题的位置 Core Dump，而不是带着错误运行一段时间后才 Core Dump，从而方便问题定位。

总而言之，OceanBase 的内存管理没有采用高深的技术，也没有做到通用或者最优，但是很好地满足了系统初期的两个最主要的需求：可控性以及没有内存碎片。

2.5.2 成组提交

为了提高写性能，UpdateServer 会将多个写操作的日志组成一批，一次性写到日志文件中，这种技术称为成组提交（Group Commit）。

考虑如下模型：生产者不断地将写任务加入到任务队列中，有一个批处理线程从任务队列中每次取一批写任务进行批量处理。由于写操作的时间消耗主要在于写日志文件，批处理 1 个写任务与批处理 10 个写任务花费的时间相差不大。因此，批处理线程总是尽量提高一次处理的任务数。假设一批任务最多包含 1024 个，常见的 Group Commit 做法为：批处理线程尝试从任务队列中取出 1024 个任务，如果队列中任务不够，那么等待一段时间（比如 5ms），直到取到 1024 个任务或者超时为止。

这种做法的问题在于延时，当系统比较空闲时，批处理线程经常需要额外等待一段时间。然而仔细观察可以发现，这里其实是不需要等待的。如果批处理线程前

一次处理的任务数较少，下一次任务队列中自然会积攒较多的任务，相应地，批处理线程也能处理得更快。

例如：假设生产者每隔 1ms 会往任务队列中加入一个新的任务，批处理线程处理 1 个任务和 10 个任务的时间都是 5ms。

- 方式 1（等待 5ms）：5ms 的时候开始处理第一批共 5 个任务，10ms 的时候处理完成。接着等待 5ms，直到 15ms 的时候开始处理第二批任务共 10 个任务，25ms 的时候处理完成。依次类推。
- 方式 2（不等待）：1ms 的时候开始处理第一批共 1 个任务，6ms 的时候处理完成。接着开始处理第二批共 5 个任务，11ms 的时候处理完成。依次类推。

“方式 1”每隔 10ms 处理 10 个任务，“方式 2”每隔 5ms 处理 5 个任务。无论采用哪种方式，批处理线程的处理能力为 1ms 一个任务，与生产者产生任务的速率相同。“方式 1”和“方式 2”处理的并发数相同，而“方式 2”的任务响应时间更短。

2.5.3 双缓冲区

双缓冲区广泛用于“生产者/消费者”模型。ChunkServer 中使用了双缓冲区异步预读的技术，生产者为磁盘，消费者为 CPU，磁盘中生产的原始数据需要给 CPU 计算消费掉。

所谓“双缓冲区”，顾名思义就是两个缓冲区（假设为 A 和 B）。这两个缓冲区，总是一个用于生产者，一个用于消费者。当两个缓冲区都操作完，再进行一次切换，先前被生产者写入的被消费者读取，先前消费者读取的转为生产者写入。为了做到不冲突，给每个缓冲区分配一把互斥锁（假设为 La 和 Lb）。生产者或者消费者如果要操作某个缓冲区，必须先拥有对应的互斥锁。

双缓冲区包括如下几种状态：

- 双缓冲区都在使用的状态（并发读写）
  大多数情况下，生产者和消费者都处于并发读写状态。不妨设生产者写入 A，消费者读取 B。在这种状态下，生产者拥有锁 La；同样地，消费者拥有锁 Lb。由于俩缓冲区都是处于独占状态，因此每次读写缓冲区中的元素都不需要再进行加锁、解锁操作。这是节约开销的主要来源。
- 单个缓冲区空闲状态
  由于两个并发实体的速度会有差异，必然会出现一个缓冲区已经操作完，而另一个尚未操作完。不妨假设生产者快于消费者。在这种情况下，当生产者把 A 写满的时候，生产者要先释放 La（表示它已经不再操作 A），然后尝试获取 Lb。由于 B 还没有被读空，Lb 还被消费者持有，所以生产者进入等待（wait）状态。
- 缓冲区的切换
  过了若干时间，消费者终于把 B 读完。这时候，消费者也要先释放 Lb，然后尝试获取 La。由于 La 刚才已经被生产者释放，所以消费者能立即拥

有 La 并开始读取 A 的数据。而由于 Lb 被消费者释放，所以刚才等待的生产者会苏醒过来（wakeup）并拥有 Lb，然后生产者继续往 B 写入数据。

2.5.4 定期合并限速

定期合并期间系统的压力较大，需要控制定期合并的速度，避免影响正常服务。定期合并限速的措施包括：

- ChunkServer：ChunkServer 定期合并过程中，每合并完成若干行（默认 2000 行）数据，就查看本机的负载（查看 Linux 系统的 Load 值）。如果负载过高，一部分定期合并线程转入休眠状态；如果负载过低，唤醒更多的定期合并线程。另外，RootServer 将 UpdateServer 冻结的大版本通知所有的 ChunkServer，每台 ChunkServer 会随机等待一段时间再开始执行定期合并，防止所有的 ChunkServer 同时将大量的请求发给 UpdateServer。
- UpdateServer：定期合并过程中 ChunkServer 需要从 UpdateServer 读取大量的数据，为了防止定期合并任务占满带宽而阻塞用户的正常请求，UpdateServer 将任务区分为高优先级（用户正常请求）和低优先级（定期合并任务），并单独统计每种任务的输出带宽。如果低优先级任务的输出带宽超过上限，降低低优先级任务的处理速度；反之，适当提高低优先级任务的处理速度。

如果 OceanBase 部署了两个集群，还能够支持主备集群在不同时间段进行“错峰合并”：一个集群执行定期合并时，把全部或大部分读写流量切到另一个集群。该集群合并完成后，把全部或大部分流量切回，以便另一个集群接着进行定期合并。两个集群都合并完成后，恢复正常的流量分配。

2.5.5 缓存预热

UpdateServer 中的 Frozen MemTable 将会以 SSTable 的形式转储到 SSD 盘中。如果内存不够需要丢弃 Frozen MemTable，则大量请求只能读取 SSD 盘，UpdateServer 性能将大幅下降。因此，希望能够在丢弃 Frozen MemTable 之前将 SSTable 的缓存预热。

UpdateServer 的缓存预热机制实现如下：在丢弃 Frozen MemTable 之前的一段时间（比如 10 分钟），每隔一段时间（比如 30 秒），将一定比率（比如 5%）的请求发给 SSTable，而不是 Frozen MemTable。这样，SSTable 上的读请求将从 5%到 10%，再到 15%，依次类推，直到 100%，很自然地实现了缓存预热。

另外，ChunkServer 定期合并后需要使用生成的新的 SSTable 提供服务，这里也需要缓存预热。OceanBase 最初的版本实现了主动缓存预热：扫描原来的缓存，根据每个缓存项的 key 读取新的 SSTable 并将结果加入到新的缓存中。例如，原来缓存数据项的主键分别为 100、200、500，那么只需要从新的 SSTable 中读取主键为 100、200、500 的数据并加入新的缓存。扫描完成后，原来的缓存可以丢弃。

线上运行一段时间后发现，定期合并基本上都安排在凌晨业务低峰期，合并完成后 OceanBase 集群收到的用户请求总是由少到多（早上 7 点之前请求很少，9 点以后请求逐步增多），能够很自然地实现被动缓存预热。由于 ChunkServer 在主动缓存预热期间需要占用两倍的内存，因此，目前的线上版本放弃了这种方式，转而采用被动缓存预热。

3 数据库功能

数据库功能层构建在分布式存储引擎层之上，实现完整的关系数据库功能。

对于使用者来说，OceanBase 与 Mysql 数据库并没有什么区别，可以通过 Mysql 客户端连接 OceanBase，也可以在程序中通过 JDBC/ODBC 操作 OceanBase。OceanBase 的 MergeServer 模块支持 Mysql 协议，能够将其中的 SQL 请求解析出来，并转化为 OceanBase 系统的内部调用。

OceanBase 定位为全功能的关系数据库，但这并不代表我们会同等对待所有的关系数据库功能。关系数据库系统中优化器是最为复杂的，这个问题困扰了关系数据库几十年，更不可能是 OceanBase 的长项。因此，OceanBase 支持的 SQL 语句一般比较简单，绝大部分为针对单张表格的操作，只有很少一部分操作涉及到多张表格。OceanBase 内部将事务划分为只读事务和读写事务，只读事务执行过程中不需要加锁，读写事务最终需要发给 UpdateServer 执行。相比传统的关系数据库，OceanBase 执行简单的 SQL 语句要高效得多。

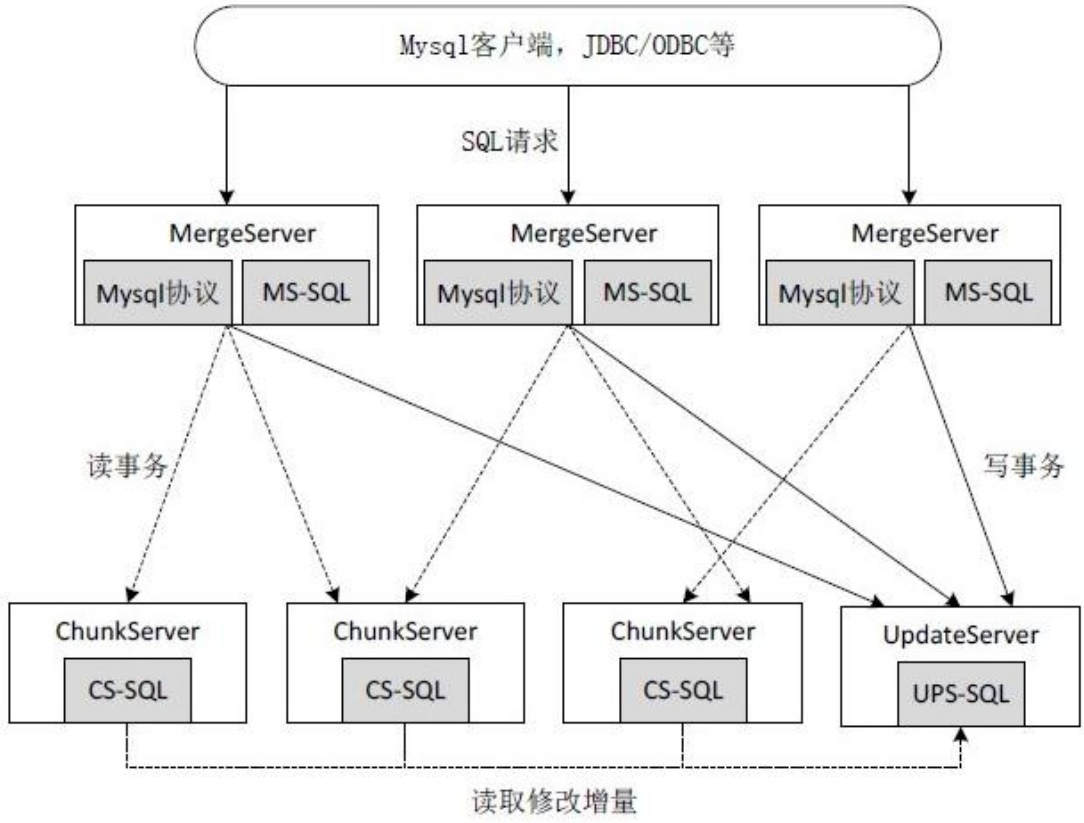
除了支持 OLTP 业务，OceanBase 还能够支持 OLAP 业务。OLAP 业务的查询请求并发数不会太高，但每次查询的数据量都非常大。为此，OceanBase 设计了并行计算框架和列式存储来处理 OLAP 业务面临的大查询问题。

最后，OceanBase 还针对实际业务的需求开发了很多特色功能，例如：用于淘宝网收藏夹的大表左连接功能，数据自动过期以及批量删除功能。这些功能在关系数据库中要么不支持，要么效率很低，不能满足业务的需求，我们将这些需求通用化后集成到 OceanBase 系统中。

3.1 整体结构

如图 3-1，用户可以通过兼容 Mysql 协议的客户端，JDBC/ODBC 等方式将 SQL 请求发送给某一台 MergeServer。MergeServer 的 Mysql 协议模块将解析出其中的 SQL 语句，并交给 MS-SQL 模块进行词法分析（采用 GNU Flex 实现）、语法分析（采用 GNU Bison 实现）、预处理、并生成逻辑执行计划和物理执行计划。

图 3-1 数据库功能层整体结构



如果是只读事务，MergeServer 首先定位请求的数据所在的 ChunkServer，接着往相应的 ChunkServer 发送 SQL 子请求，每个 ChunkServer 将调用 CS-SQL 模块计算 SQL 子请求的结果，并将计算结果返回给 MergeServer。最后，MergeServer 需要整合这些子请求的返回结果，执行结果合并、联表、子查询等操作，得到最终结果并返回给客户端。

如果是读写事务，MergeServer 首先从 ChunkServer 中读取需要的基线数据，接着将物理执行计划以及基线数据一起发送给 UpdateServer，UpdateServer 将调用 UPS-SQL 模块完成写事务。

- CS-SQL：实现针对单个 Tablet 的 SQL 查询，包括表格扫描（table scan）、投影（projection）、过滤（filter）、排序（order by）、分组（group by）、分页（limit），支持表达式计算、聚集函数（count/sum/max/min 等）。执行表格扫描时，需要从 UpdateServer 读取修改增量，与本地的基准数据合并。
- UPS-SQL：实现写事务，支持的功能包括多版本并发控制、操作日志多线程并发回放等。
- MS-SQL：SQL 语句解析，包括词法分析、语法分析、预处理、生成执行计划，按照 Tablet 范围合并多个 ChunkServer 返回的部分结果，实现针对多个表格的物理操作符，包括联表（Join），子查询（subquery）等。

3.2 只读事务

只读事务（SELECT 语句），经过词法分析、语法分析、预处理后，转化为逻辑查询计划和物理查询计划。逻辑查询计划的改进和物理查询计划的选择（即查询优化器）是关系数据库最难的部分，OceanBase 目前在这一部分的工作不多，因此本节不会涉及太多关于如何生成物理查询计划的内容。下面仅以两个例子说明 OceanBase 的物理查询计划。

例 1：有一个单表 SQL 语句，如图 3-2 所示。

图 3-2 单表物理查询计划示例

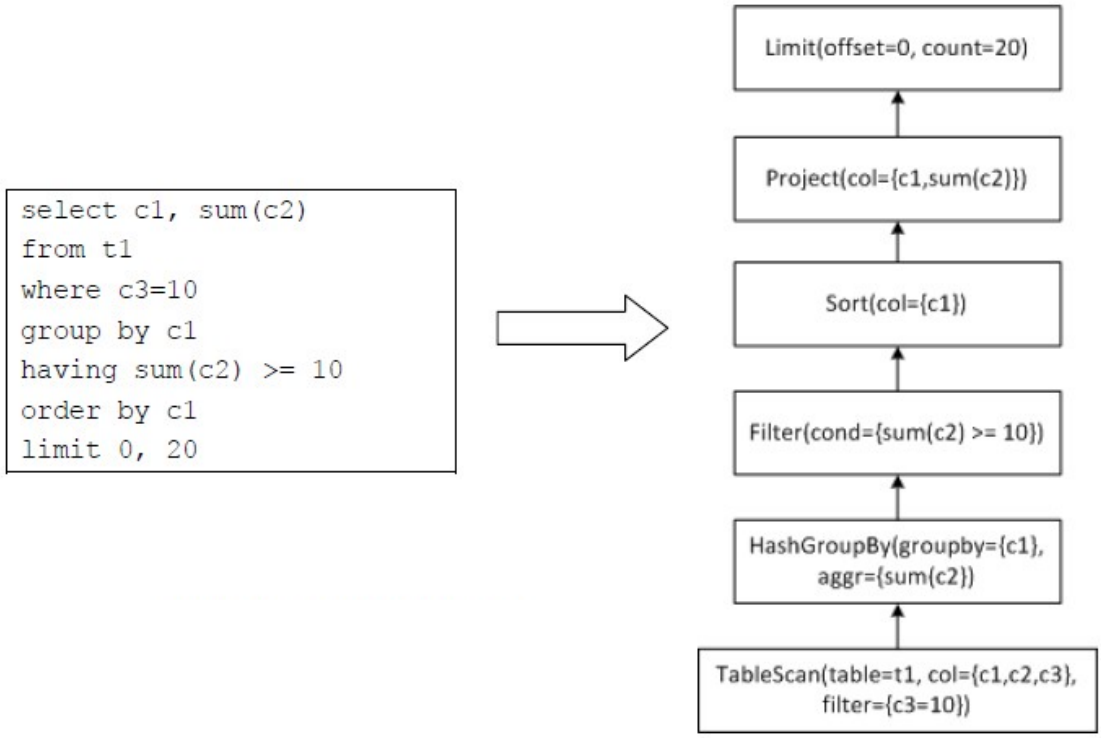


图 3-2 中的单表 SQL 语句执行过程如下：

1. 调用 TableScan 操作符，读取 table t1 中的数据，该操作符还将执行投影（Project）和过滤（Filter），返回的结果只包含 c3=10 的数据行，且每行只包含 c1、c2、c3 三列。
2. 调用 HashGroupBy 操作符（假设采用基于哈希的分组算法），按照 c1 对数据分组，同时计算每个分组内 c2 列的总和。
3. 调用 Filter 操作符，过滤分组后生成的结果，只返回上一层 sum(c2) >= 10 的行。
4. 调用 Sort 操作符将结果按照 c1 排序。
5. 调用 Project 操作符，只返回 c1 和 sum(c2)这两列数据。
6. 调用 Limit 操作符执行分页操作，只返回前 20 条数据。

例 2：有一个需要联表的 SQL 语句，如图 3-3 所示。

图 3-3 多表物理查询计划示例

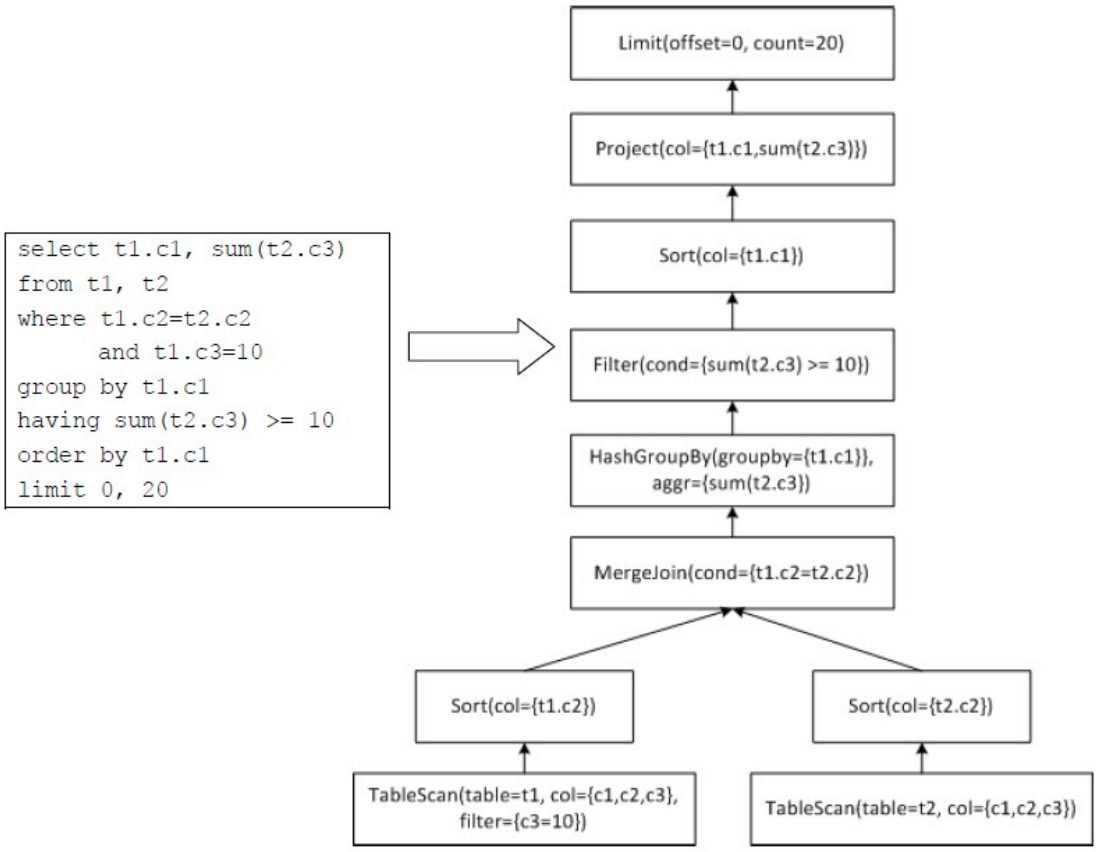


图 3-3 中的多表 SQL 语句执行过程如下：

1. 调用 TableScan 分别读取 t1 和 t2 的数据。对于 t1，使用条件 c3=10 对结果进行过滤，t1 和 t2 都只需要返回 c1、c2、c3 这三列数据。
2. 假设采用基于排序的表连接算法，t1 和 t2 分别按照 t1.c2 和 t2.c2 排序后，调用 MergeJoin 运算符，以 t1.c2=t2.c2 为条件执行等值连接。
3. 调用 HashGroupBy 运算符（假设采用基于哈希的分组算法），按照 t1.c1 对数据分组，同时计算每个分组内 t2.c3 列的总和。
4. 调用 Filter 运算符，过滤分组后的生成的结果，只返回上一层 sum(t2.c3) >= 10 的行。
5. 调用 Sort 操作符将结果按照 t1.c1 排序。
6. 调用 Project 操作符，只返回 t1.c1 和 sum(t2.c3)这两列数据。
7. 调用 Limit 操作符执行分页操作，只返回前 20 条数据。

3.2.1 物理操作符接口

在本手册的“2.3.1 SSTable”中介绍了一期分布式存储引擎的迭代器接口为 ObIterator。通过它，可以将读到的数据以 cell 为单位逐个迭代。然而，数据库

操作总是以行为单位的，因此二期实现数据库功能层时考虑将基于 cell 的迭代器修改为基于行的迭代器。

行迭代器接口如下：

```
// 物理运算符接口
   class ObPhyOperator
   {
      public:
   // 添加子运算符，所有非叶子节点物理运算符都需要调用该接口    。
      virtual int set_child(int32_t child_idx, ObPhyOperator &child_operator);

   // 打开物理运算符。申请资源，打开子运算符等。
      virtual int open() = 0;
   // 关闭物理运算符。释放资源，关闭子运算符等。
      virtual int close() = 0;

   // 获得下一行数据内容
   // @param=out] row  下一行数据内容的引用
   // @return  返回码，包括成功、迭代过程中出现错误以及迭代完成
      virtual int get_next_row(const ObRow *&row) = 0;
     };
// ObRow 表示一行数据内容
   class ObRow
   {
      public:
   // 根据表 ID 以及列 ID 获得指定 cell
   // @param =in] table_id  表格 ID
   // @param =in] column_id  列 ID
   // @param =out] cell  读到的 cell
      int get_cell(const uint64_t table_id, const uint64_t column_id, ObObj *&cell);

   // 获取低 cell_idx 个 cell
      int raw_get_cell(const int64_t cell_idx, const ObObj *&cell, uint64_t &table_id, uint64_t
&column_id);

   // 获取本行的列数
      int64_t get_column_num() const;
};
```

每一行数据（ObRow）包括多个列，每个列的内容包括所在的表 ID（table_id），列 ID（column_id）以及列内容（cell）。ObRow 提供两种访问方式：“根据 table_id 和 column_id 随机访问某个列”和“根据 cell_idx 获取下一个列”。

ObPhyOperator 每次获取一行数据，使用方法如下：

```
ObPhyOperator root_operator = root_operator_;// 根运算符
root_operator->open();
ObRow *row = NULL;
while (OB_SUCCESS == root_operator->get_next_row(row))
{
        Output(row); //输出本行
}
root_operator->close();
```

为什么 ObPhyOperator 类中有一个 set_child 接口呢？这是因为所有的物理运算符构成一颗树，每个物理运算的输出结果都可以认为是一个临时的二维表，树中孩子节点的输出总是作为它的父亲节点的输入。在“3.2 只读事务”的“例 1”中，叶子节点为一个 TableScan 类型的物理运算符（称为 table_scan_op），它的父亲节点为一个 HashGroupBy 类型的物理运算符（称为 hash_group_by_op），接下来依次为 Filter 类型物理运算符 filter_op，Sort 类型物理运算符 sort_op，Project 类型物理运算符 project_op，Limit 类型物理运算符 limit_op。其中，limit_op 为根运算符。那么，生成物理运算符时将执行如下语句：

```
limit_op->set_child(0, project_op);
project_op->set_child(0, sort_op);
sort_op->set_child(0, filter_op);
filter_op->set_child(0, hash_group_by_op);
hash_group_by_op->set_child(0, table_scan_op);
root_operator = limit_op;
```

SQL 最终执行时，只需要迭代 root_operator 就可以依次迭代出所需的数据。

3.2.2 单表操作

单表相关的物理运算符包括：

- TableScan
  扫描某个表格。MergeServer 将扫描请求发给请求的各个 Tablet 所在的 ChunkServer，并将 ChunkServer 返回的结果按照 Tablet 范围拼接起来作为输出。如果请求涉及到多个 Tablet，TabletScan 可由多台 ChunkServer 并发执行。
- Filter
  针对每行数据，判断是否满足过滤条件。
- Projection
  对输入的每一行，根据定义的输出表达式，计算输出结果行。
- GroupBy
  把输入数据按照指定列进行聚集，对聚集后的每组数据可以执行 count、sum、min、max、avg 等聚集操作。

- Sort
  对输入数据进行整体排序，如果内存不够，需要使用外排序。
- Limit
  返回行号在=offset, offset + count)范围内的行。
- Distinct
  消除某些列的重复行。

GroupBy、Distinct 物理操作符可以通过基于排序的算法实现，也可以通过基于哈希的算法实现，分别对应 HashGroupBy&MergeGroupBy，以及 HashDistinct&MergeDistinct。下面分别讨论排序算法和哈希算法：

- 排序算法
  MergeGroupBy、MergeDistinct 以及 Sort 都需要使用排序算法。通用的<key, value>排序器可以分为两个阶段：
  1. 数据收集。
     在数据收集阶段，调用者将<key, value>对依次加入到排序器。如果数据总量超过排序器的内存上限，需要首先将内存中的数据排好序，并存储到外部磁盘中。
  2. 迭代输出。
     迭代第一行数据时，内存中可能有一部分未排序的数据，磁盘中也可能有几路已经排序的数据。因此，首先将内存中的数据排序。如果数据总量不超过排序器的内存上限，那么将内存中已经排序的数据按行迭代输出（内排序）；否则，对内存和磁盘中的部分有序数据执行多路归并，一边归并一边将结果迭代输出。
- 哈希算法
  HashGroupBy 以及 HashDistinct 都需要使用哈希算法。假设需要对<key, value>对按照 key 分组，那么首先使用 key 计算哈希值 K，并将这个<key, value>对写入到第 K 个桶中。不同的 key 可能对应相同的哈希桶，因此，还需要对每个哈希桶内的<key, value>排序，使 key 相同的元组能够连续迭代。哈希算法的难点在于数据总量超过内存上限的处理，由于篇幅有限，请自行思考。

3.2.3 多表操作

多表相关的物理操作符主要是 Join。最为常见的 Join 类型包括两种：内连接（Inner Join）和左外连接（Left Outer Join），而且基本都是等值连接。如果需要 Join 多张表，可以先 Join 前两张表，再将前两张表 Join 生成的结果（相当于一张临时表）与第三张表格 Join，以此类推。

两张表实现等值连接方式主要分为两类：基于排序的算法（MergeJoin）以及基于哈希的算法（HashJoin）。对于 MergeJoin，首先使用 Sort 运算符分别对输入表格预处理，使得两张输入表都在 Join 列上排序，接着按顺序迭代两张输入表，合并 Join 列相同的行并输出；对于 HashJoin，首先根据 Join 列计算哈希值

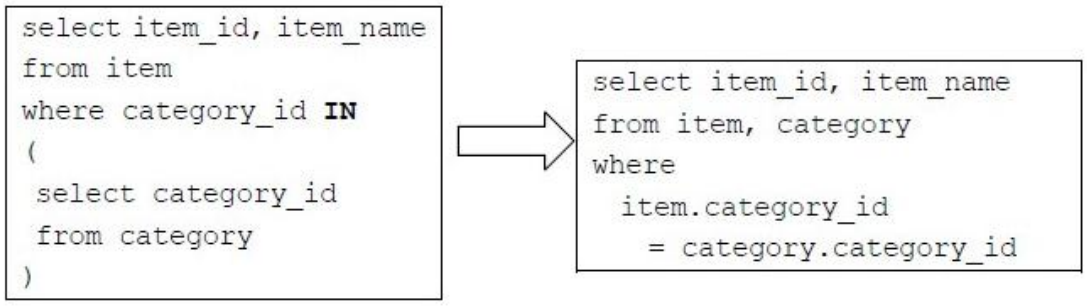
K，并分别将两张输入表格的数据写入到第 K 个桶中。接着，对每个哈希桶按照 Join 列排序。最后，依次对每个哈希桶合并 Join 列相同的行并输出。

子查询分为两种：关联子查询和非关联子查询，其中比较常用的是使用 IN 子句的非关联子查询。举例如下：

假设有两张表分别为 item（商品表，包括商品号 item_id，商品名 item_name，分类号 category_id）和 category（类别表，包括分类号 category_id，分类名 category_name）。

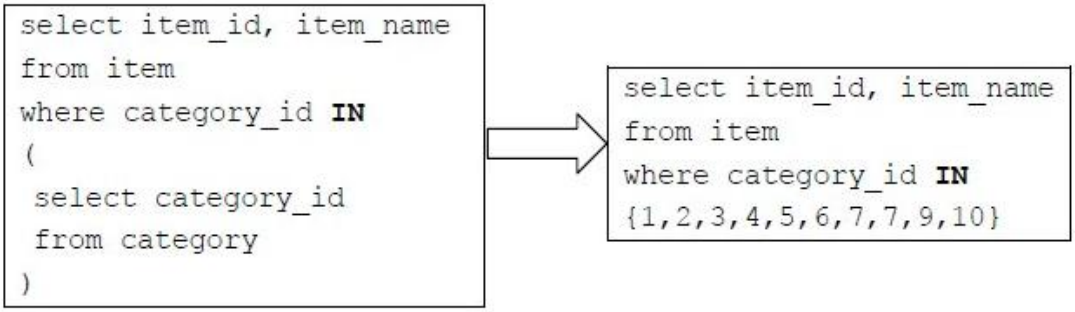
如果需要查询分类号出现在 category 表中商品，可以采用图 3-4 左边的 IN 子查询，而这个子查询将被自动转化为图 3-4 右边的等值连接。如果 category 表中的 category_id 列有重复，表连接之前还需要使用 distinct 运算符来删除重复的记录。

图 3-4 IN 子查询转化为等值连接



如果 category 表只包含 category_id 为 1~10 的记录，那么，可以将 IN 子查询转化为图 3-5 右边的常量表达式。

图 3-5 IN 子查询转化为常量表达式



转化为常量表达式后，MergeServer 执行 SQL 计算时，可以将 IN 后面的常量列表发送给 ChunkServer，ChunkServer 只返回 category_id 在 category 表中的商品记录，而不是将所有的记录返回给 MergeServer 过滤，从而减少二者之间传输的数据量。

3.2.4 SQL 执行本地化

MergeServer 包含 SQL 执行模块 MS-SQL，ChunkServer 也包含 SQL 执行模块 CS-SQL，那么如何区分二者的功能呢？多表操作由 MergeServer 执行；对于单表操作，OceanBase 设计的基本原则是尽量支持 SQL 计算本地化，保持数

据节点与计算节点一致，也就是说，只要 ChunkServer 能够实现的操作，原则上都应该由它来完成。

- TableScan
  每个 ChunkServer 扫描各自 Tablet 范围内的数据，由 MergeServer 合并 ChunkServer 返回的部分结果。

- Filter
  对基本表的过滤集成在 TableScan 操作符中，由 ChunkServer 完成。对分组后的结果执行过滤（Having）集成在 GroupBy 操作符中，一般情况下由 MergeServer 完成；但是，如果能够确定每个分组的所有数据行只属于同一个 Tablet，比如 SQL 请求只涉及一个 Tablet，那么，分组以及分组后的过滤操作符可以由 ChunkServer 完成。

- Projection
  对基本表的投影集成在 TableScan 操作符中，由 ChunkServer 完成，对最终结果的投影由 MergeServer 完成。

- GroupBy
  如果 SQL 读取的数据只在一个 Tablet 上，那么由该 Tablet 所在的 ChunkServer 完成分组操作；否则每台 ChunkServer 各自完成部分数据的分组操作，执行聚合运算后得到部分结果，再由 MergeServer 合并所有 ChunkServer 返回的部分结果，对于属于同一个分组的数据再次执行聚合运算。某些聚合运算需要做特殊处理，比如 avg，需要转化为 sum 和 count 操作发送给 ChunkServer，MergeServer 合并 ChunkServer 返回的部分结果后计算出最终的 sum 和 count 值，并通过“sum/count”得到 avg 的最终结果。

- Sort
  如果 SQL 读取的数据只在一个 Tablet 上，那么由该 Tablet 所在的 ChunkServer 完成排序操作；否则每台 ChunkServer 各自完成部分数据的排序，并将排序部分数据返回 MergeServer，再由 MergeServer 执行多路归并。

- Limit
  Limit 操作一般由 MergeServer 完成，但是如果请求的数据只在一个 Tablet 上，可以由 ChunkServer 完成，这往往会大大减少 MergeServer 与 ChunkServer 之间传输的数据量。

- Distinct
  Distinct 与 GroupBy 类似。ChunkServer 先完成消除部分数据的的重复行，再由 MergeServer 进行整体消除数据的重复行。

例如：图 3-2 中的 SQL 语句为“select c1, sum(c2) from t1 where c3 = 10 group by c1 having sum(c2) >= 10 orderby c1 limit 0, 20”。执行步骤如下：

1. ChunkServer 调用 TableScan 操作符，读取 table t1 中的数据，该操作符还将执行投影（Project）和过滤（Filter），返回的结果只包含 c3=10 的数据行，且每行只包含 c1、c2、c3 三列。

2. ChunkServer 调用 HashGroupBy 操作符（假设采用基于哈希的分组算法），按照 c1 对数据分组，同时计算每个分组内 c2 列的总和 sum(c2)。
3. 每个 ChunkServer 将分组后的部分结果返回 MergeServer，MergeServer 将来自不同 ChunkServer 的 c1 列相同的行合并在一起，再次执行 sum 运算。
4. MergeServer 调用 Filter 操作符，过滤“步骤 3”生成的最终结果，只返回 sum(c2) >= 10 的行。
5. MergeServer 调用 Sort 操作符将结果按照 c1 排序。
6. MergeServer 调用 Project 操作符，只返回 c1 和 sum(c2)这两列数据。
7. MergeServer 调用 Limit 操作符执行分页操作，只返回前 20 条数据。

如果能够确定请求的数据全部属于同一个 Tablet，那么，所有的物理运算符都可以由 ChunkServer 执行，MergeServer 只需要将 ChunkServer 计算得到的结果转发给客户端。

3.3 写事务

写事务包括 UPDATE、INSERT、DELETE 和 REPLACE。由 MergeServer 解析后生成物理执行计划，这个物理执行计划最终将发给 UpdateServer 执行。写事务可能需要读取基线数据，用于判断更新或者插入的数据行是否存在，判断某个条件是否满足等，这些基线数据也会由 MergeServer 传给 UpdateServer。

3.3.1 写事务执行流程

大部分写事务都是针对单行的操作，如果单行写事务不带其它条件：

- REPLACE
  REPLACE 事务不关心写入行是否已经存在，因此 MergeServer 直接将修改操作发送给 UpdateServer 执行。
- INSERT
  MergeServer 首先读取 ChunkServer 中的基线数据，并将基线数据中行是否存在信息发送给 UpdateServer。UpdateServer 接着查看增量数据中行是否被删除或者有新的更新操作，融合基线数据和增量数据后，如果行不存在，则执行插入操作；否则，返回行已存在错误。
- UPDATE
  与 INSERT 事务执行步骤类似，不同点在于，行已存在则执行更新操作；否则返回行不存在错误。
- DELETE
  与 UPDATE 事务执行步骤类似。如果行已存在则执行删除操作；否则返回行不存在错误。

如果单行写事务带有其它条件：

- UPDATE
  如果 UPDATE 事务带有其它条件，那么 MergeServer 除了从基线数据中读取行是否存在，还需要读取用于条件判断的基线数据，并传给 UpdateServer。UpdateServer 融合基线数据和增量数据后，执行条件判断，如果行存在且判断条件成立则执行更新操作，否则返回行已存在或者条件不成立错误。
- DELETE
  与 UPDATE 事务执行步骤类似。

例 1：有一张表格 item（user_id, item_id，item_status，item_name），其中 <user_id, item_id>为联合主键。

1. MergeServer 首先解析图 3-6 的 SQL 语句产生执行计划，确定待修改行的主键为<1，2>。
2. 请求主键<1，2>所在的 ChunkServer，获取基线数据中行是否存在。
3. 将 SQL 执行计划和基线数据中行是否存在一起发送给 UpdateServer。
4. UpdateServer 融合基线数据和增量数据，如果行已存在且未被删除，UPDATE 和 DELETE 语句执行成功，INSERT 语句执行返回“行已存在”；如果行不存在或者最后被删除，INSERT 语句执行成功，UPDATE 和 DELETE 语句返回“行不存在”。

图 3-6 单行写事务（不带条件）

```
// 插入语句
insert into item
values(1, 2, 0, "item1");
// 更新语句
update item
set item_status=1
where user_id=1
  and item_id=2;
// 删除语句
delete from item
where user_id=1
 and item_id=2;
```

如图 3-7 所示，UPDATE 和 DELETE 语句带有 item_name = “item1”的条件时，MergeServer 除了请求 ChunkServer 获取基线数据中行是否存在，还需要获取 item_name 的内容，并将这些信息一起发送给 UpdateServer。UpdateServer 融合基线数据和增量数据，判断最终结果中行是否存在，以及 item_name 的内

容是否为“item1”，只有两个条件同时成立，UPDATE 和 DELETE 语句才能够执行成功；否则，返回“行不存在或者 item_name 列的最终值”。

图 3-7 单行写事务（带条件）

```
// 更新语句
update item
set item_status=1
where user_id=1
  and item_id=2
  and item_name="item1";
// 删除语句
delete from item
where user_id=1
 and item_id=2
 and item_name="item1";
```

当然，并不是所有的写事务都这么简单。复杂的写事务可能需要修改多行数据，事务执行过程也可能比较复杂。

例 2：有两张表格分别为 item（user_id, item_id, item_status, item_name）和 user（user_id，user_name）。其中<user_id, item_id>为 item 表格的联合主键，user_id 为 user 表格的主键。

如图 3-8 所示，UPDATE 语句可能会更新多行。MergeServer 首先从 ChunkServer 获取编号为“1”的用户，包含的全部 item（可能包含多行），并传给 UpdateServer。接着，UpdateServer 融合基线数据和增量数据，更新每个存在且未被删除的 item 的 item_status 列。

图 3-8 中的 DELETE 语句更加复杂，执行时需要首先获取 user_name 为“张三”的用户的 user_id，考虑到事务隔离级别，这里可能需要锁住 user_name 为“张三”的数据行（防止 user_name 被修改为其它值）甚至锁住整张 user 表（防止其它行的 user_name 修改为“张三”或者插入 user_name 为“张三”的新行）。接着，获取用户名为“张三”的所有用户的所有 item，最后，删除这些 item。这条语句执行的难点在于如何降低锁粒度以及锁占用时间，具体的做法请读者自行思考。

图 3-8 复杂写事务举例

```
// 更新多行
update item
set item_status=1
where user_id=1;
// 复杂条件
delete item
where user_id in
  select user_id
  from user
  where user_name="张三";
```

3.3.2 多版本并发控制

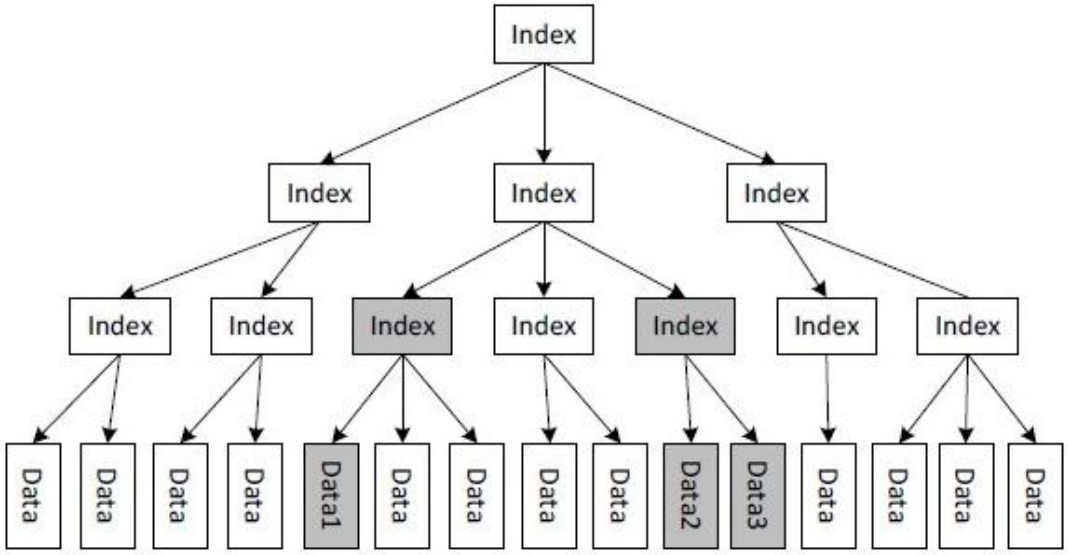
OceanBase MemTable 底层数据结构为一颗内存 B+树，支持多线程并发修改。

*** 并发修改 B+树**

MemTable 的索引结构是一颗支持多线程并发修改的 B+树。图 3-9 说明了并发修改 B+树的实现原理。

- 两个线程分别插入 Data1 和 Data2：由于 Data1 和 Data2 用于不同的父亲节点，插入 Data1 和 Data2 将影响 B+树的不同部分，两个线程可以并发执行，不会产生冲突。
- 两个线程分别插入 Data2 和 Data3：由于 Data2 和 Data3 拥有相同的父亲节点，因此，插入操作将产生冲突。其中一个线程会执行成功，另外一个线程失败后将重试。

图 3-9 并发修改 B+树



另外，每个索引节点满了以后将分裂为两个节点，并触发对该索引节点的父亲节点的修改操作。分裂操作将增加插入线程冲突的概率，在图 3-9 中，假设 Data1 和 Data2 的父亲节点都需要分裂，那么插入线程需要分别修改 Data1 和 Data2 的祖父节点，从而产生冲突。

B+树结构以行为单位索引 MemTable 中的数据，支持的功能如下：

- Put
  插入一行数据。MemTable 每次插入一行数据需要往 B+树索引结构中增加一个<key, value>对，其中，key 为该行数据的主键，value 为该行实际内容的头部信息。
- Get
  根据主键读取一行的内容。
- Scan
  扫描一段范围内的数据行。

细心的读者可能会发现，这里的 B+树不支持更新（Update）以及删除操作，这是由 MVCC 存储引擎的实现机制决定的。MVCC 存储引擎内部将删除操作实现为标记删除，即在行的末尾追加一个单元记录行的删除时间，而不会物理删除某行数据。

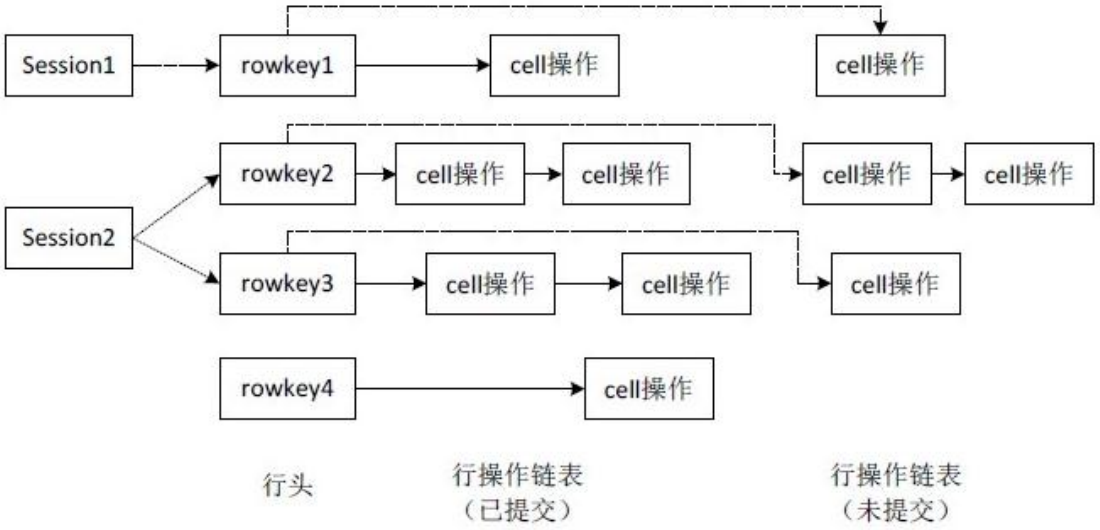
*** 多版本并发机制**

OceanBase 支持多线程并发修改，写操作拆分为两个阶段：

1. 预提交（多线程执行）。
   事务执行线程首先锁住待更新数据行。接着，将事务中针对数据行的操作追加到该行的未提交行操作链表中。最后，往提交任务队列中加入一个提交任务。
2. 提交（单线程执行）。
   提交线程不断地扫描并取出提交任务队列中的提交任务，将这些任务的操作日志追加到日志缓冲区中。如果日志缓冲区到达一定大小，将日志缓冲区中的数据同步到备机，同时写入主机的磁盘日志文件。操作日志写成功后，将未提交行操作链表中的 cell 操作追加到已提交行操作链表的末尾，释放锁并回复客户端写操作成功。

如图 3-10 所示，MemTable 行操作链表包含两个部分：已提交部分和未提交部分。另外，每个 Session 记录了当前事务正在操作的数据行的行头，每个数据行的行头包含已提交和未提交行操作链表的头部指针。在预提交阶段，每个事务会将 cell 操作追加到未提交行操作链表中，并在行头保存未提交行操作链表的头部指针以及锁信息，同时，将行头信息记录到 Session 中；在提交阶段，根据 Session 中记录的行头信息找到未提交行操作链表，链接到已提交行操作链表的末尾，并释放行头记录的锁。

图 3-10 MemTable 实现 MVCC



每个写事务会根据提交时的系统时间生成一个事务版本，读事务只会读取在它之前提交的写事务的修改操作。

如图 3-11 所示，对主键为 1 的商品有 2 个写事务，事务 T1（提交版本号为 2）将商品购买人数修改为 100，事务 T2（提交版本号为 4）将商品购买人数修改为 50。那么，事务 T2 预提交时，T1 已经提交，该商品的已提交行操作链包含一个 cell：<update，购买人数，100>，未提交操作链包含一个 cell：<update，购买人数，50>。事务 T2 成功提交后，该商品的已提交行操作链将包含两个 cell：<update，购买人数，100>以及<update，购买人数，50>，以及，未提交行操作链为空。对于只读事务：

(1) T3：事务版本号为 1，T1 和 T2 均未提交，该行数据为空。
(2) T4：事务版本号为 3，T1 已提交，T2 未提交，读取到<update，购买人数，100>。尽管 T2 在 T4 执行过程中将购买人数修改为 50，T4 第二次读取时会过滤掉 T2 的修改操作，因而两次读取将得到相同的结果。
(3) T5：事务版本号为 5，T1 和 T2 均已提交，读取到<update，购买人数，100>以及<update，购买人数，50 >，购买人数最终值为 50。

图 3-11 读写事务并发执行实例

时间戳	T1	T2	T3	T4	T5
1			READ		
2	WRITE(购买人数，100)				
3				READ	
4		WRITE(购买人数，50)			
5				READ	READ

*** 锁机制**

OceanBase 锁定粒度为行锁，默认情况下的隔离级别为 read committed。另外，读操作总是读取某个版本的快照数据，不需要加锁。

- 只写事务（修改单行）
  事务预提交时对待修改的数据行加写锁，事务提交时释放写锁。
- 只写事务（修改多行）
  事务预提交时对待修改的多个数据行加写锁，事务提交时释放写锁。为了保证一致性，采用两阶段锁的方式实现，即需要在事务预提交阶段获取所有数据行的写锁，如果获取某行写锁失败，则整个事务执行失败。
- 读写事务（read committed）
  读写事务中的读操作读取某个版本的快照，写操作的加锁方式与只写事务相同。为了保证系统并发性能，OceanBase 暂时不支持更高的隔离级别。另外，为了支持对一致性要求很高的业务，OceanBase 允许用户显式锁住某个数据行。例如，有一张账务表 account（account_id, balance），其中 account_id 为主键。假设需要从 A 账户（account_id=1）向 B 账户（account_id=2）转账 100 元，那么 A 账户需要减少 100 元，B 账户需要增加 100 元，整个转账操作是一个事务，执行过程中需要防止 A 账户和 B 账户被其它事务并发修改。

如图 3-12 所示，OceanBase 提供了”select … for update”语句用于显示锁住 A 账户或者 B 账户，防止转账过程中被其它事务并发修改。

图 3-12 select … for update 示例

```
select balance as balance_a
  from account
  where account_id=1
  for update; // 锁住 A 账户
select balance as balance_b
  from account
  where account_id=2
  for update; // 锁住 B 账户
```

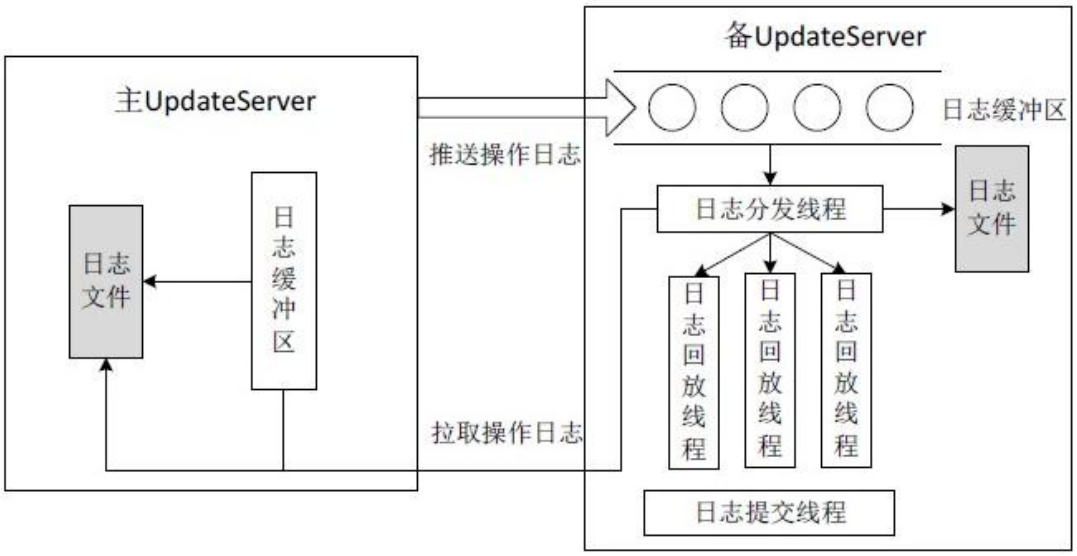
事务执行过程中可能会发生死锁，例如事务 T1 持有账户 A 的写锁并尝试获取账户 B 的写锁，事务 T2 持有账户 B 的写锁并尝试获取账户 A 的写锁，这两个事务因为循环等待而出现死锁。OceanBase 目前处理死锁的方式很简单，事务执行过程中如果超过一定时间无法获取写锁，则自动回滚。

*** 多线程并发日志回放**

在本手册的“3.2.3 多表操作”章节介绍了主备同步原理。引入多版本并发控制机制后，UpdateServer 备机支持多线程并发回放日志功能。如图 3-13 所示，有一个日志分发线程每次从日志源读取一批日志，拆分为单独的日志回放任务交给不

同的日志回放线程处理。一批日志回放完成时，日志提交线程会将对应的事务提交到 MemTable 并将日志内容持久化到日志文件。

图 3-13 备机多线程并发日志回放

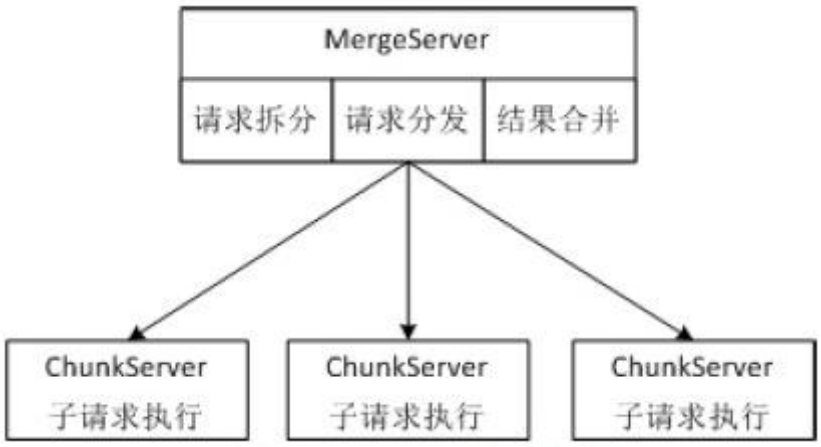


3.4 OLAP 业务支持

OLAP 业务的特点是 SQL 每次执行涉及的数据量很大，需要一次性分析几百万行甚至几千万行的数据。另外， SQL 执行时往往只读取每行的部分列而不是整行数据。为了支持 OLAP 计算，OceanBase 主要实现并发查询功能。

如图 3-14 所示，MergeServer 将大请求拆分为多个子请求，同时发往每个子请求所在的 ChunkServer 并发执行，每个 ChunkServer 执行子请求并将部分结果返回给 MergeServer。MergeServer 合并 ChunkServer 返回的部分结果并将最终结果返回给客户端。

图 3-14 OceanBase 并发查询



MergeServer 并发查询执行步骤如下：

1. MergeServer 解析 SQL 语句，根据本地缓存的 Tablet 位置信息获取需要请求的 ChunkServer。
2. 如果请求只涉及一个 Tablet，将请求发送给该 Tablet 所在的 ChunkServer 执行；如果请求涉及多个 Tablet，将请求按照 Tablet 拆分为多个子请求，每个子请求对应一个 Tablet，并发送给该 Tablet 所在的 ChunkServer 并发执行。MergeServer 等待每个子请求的返回结果。
3. ChunkServer 执行子请求，计算子请求的部分结果。SQL 执行遵从“3.2.4 SQL 执行本地化”章节提到的本地化原则，即能让 ChunkServer 执行的尽量让 ChunkServer 执行，包括 Filter、Project、子请求部分结果的 GroupBy、OrderBy、聚合运算等。
4. 每个子请求执行完成后，ChunkServer 将执行结果回复 MergeServer，MergeServer 首先将每个子请求的执行结果保存起来。如果某个子请求执行失败，MergeServer 会将该子请求发往 Tablet 其它副本所在的 ChunkServer 执行。
5. 等到所有的子请求执行完成后，MergeServer 会对全部数据排序、分组、聚合并将最终结果返回给客户。

OceanBase 支持 multiget 操作一次性读取多行数据，且读取的数据可能在不同的 ChunkServer 上。对于这样的操作，MergeServer 会按照 ChunkServer 拆分子请求，每个子请求对应一个 ChunkServer。假设客户端请求 5 行数据，其中第 1、3、5 行在 ChunkServer A 上，第 2、4 行在 ChunkServer B 上。那么，该请求将被拆分为（1、3、5）和（2、4）两个子请求，分别发往 ChunkServer A 和 B。

细心的读者可能会发现，OceanBase 这种查询模式虽然解决了绝大部分大查询请求的延时问题，但是如果查询的返回结果特别大，MergeServer 将成为性能瓶颈。因此，新版的 OceanBase 系统将对 OLAP 查询执行逻辑进行升级，使其能够支持更加复杂的 SQL 查询。

3.5 特色功能

虽然 OceanBase 是一个通用的分布式关系数据库，然而在阿里巴巴集团落地过程中，为了满足业务的需求，也实现了一些特色功能。这些功能在互联网应用中很常见，但在传统的关系数据库中往往实现得比较低效。本节介绍其中两个具有代表性的功能，分别为“大表左连接”和“数据过期与批量删除”。

3.5.1 大表左连接

大表左连接需求来源于淘宝收藏夹业务。简单来讲，收藏夹业务包含两张表格：收藏表 collect_info 和商品表 collect_item。collect_info 表存储了用户的收藏信息，比如收藏时间、标签等；collect_item 存储了用户收藏的商品或者店铺的信息，包括价格、人气等。Collect_info 的数据条目达到 100 亿条，collect_item 的数据条目接近 10 亿条，每个用户平均收藏了 50~100 个商品或者店铺。用户

可以按照收藏时间浏览收藏项，也可以对收藏项按照价格、人气排序。常见的实现方法有以下两种：

- 直接采用关系数据库 Join 操作实现
  根据 collect_info 中存储的商品编号（item_id），实时地从商品表读取商品的价格、人气等信息。然而，当商品表数据量较大时，需要分库分表后分布到多台数据库服务器，即使是同一个用户收藏的商品也会被打散到多台服务器。某些用户收藏了几千个商品或者店铺，如果需要从很多台服务器读取几千条数据，整体延时是不可接受的，系统的并发能力也将受限。

- 冗余
  在 collect_info 表中冗余商品的价格、人气等信息，那么读取时则不需要读取 collect_item 表。然而，热门商品可能被数十万个用户收藏，每次价格、人气发生变化时都需要修改数十万个用户的收藏条目。显然，这是不可接受的。

这个问题本质上是一个大表左连接（Left Join）的问题，连接列为 item_id，即右表（商品表）的主键。对于这个问题，OceanBase 的做法是在 collect_info 的基准数据中冗余 collect_item 信息，修改增量中将 collect_info 和 collect_item 两张表格分开存储。商品价格、人气变化信息只需要记录在 UpdateServer 的修改增量中，读取操作步骤如下：

1. 从 ChunkServer 读取 collect_info 表格的基准数据（冗余了 collect_item 信息）。
2. 从 UpdateServer 读取 collect_info 表格的修改增量，并融合到“步骤 1”的结果中。
3. 从 UpdateServer 读取 collect_item 表格中每个收藏商品的修改增量，并融合到“步骤 2”的结果中。
4. 对“步骤 3”生成的结果执行排序（按照人气、价格等），分页等操作并返回给客户端。

OceanBase 的实现方式得益于每天业务低峰期进行的每日合并操作。每日合并时，ChunkServer 会将 UpdateServer 上 collect_info 和 collect_item 表格中的修改增量融合到 collect_info 表格的基准数据中，生成新的基准数据。因此，collect_info 和 collect_item 的数据量不至于太大，从而能够存放到单台机器的内存中提供高效查询服务。

3.5.2 数据过期与批量删除

很多业务只需要存储一段时间（比如三个月或者半年的数据），更早之前的数据可以被丢弃从而节省存储成本。OceanBase 支持数据自动过期功能。

OceanBase 线上每个表格都包含创建时间（gmt_create）和修改时间（gmt_modified）列。使用者可以设置自动过期规则，比如只保留创建时间或修改时间不晚于某个时间点的数据行，读取操作会根据规则过滤这些失效的数据行，每日合并时这些数据行会被物理删除。

批量删除需求来源于 OLAP 业务。这些业务往往每天导入一批数据，由于业务逻辑复杂，上游系统很可能出错，导致某一天导入的数据出现问题，需要将这部分出错的数据删除掉。由于导入的数据量较大，一条一条删除其中的数据是不现实的。因此，OceanBase 实现了批量删除功能，具体做法和数据自动过期功能类似。使用者可以增加一个删除规则，比如删除创建时间在某个时间段的数据行，以后所有的读操作都会自动过滤这些出错的数据行，每日合并时这些出错的数据行会被物理删除。