OceanBase 0.5.2 系统描述

文档版本：Beta 04
发布日期：2014.10.14
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 0.5 的架构、功能和特性等信息。

读者对象

本文档主要适用于：

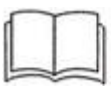
开发工程师。

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标志，它们所代表的含义如下。

标志	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。

在本文档中可能出现下列格式，它们所代表的含义如下。

格式	说明

宋体	表示正文。
黑体	标题、警告、注意、小窍门、说明等内容均采用黑
Calibri	表示代码或者屏幕显示内容。
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 04（2014-10-14）	第二次发布 Beta 版本，适用于 OceanBase 0.5.2。 完善各 Server 介绍。
Beta 03（2014-04-22）	第一次发布 Beta 版本，适用于 OceanBase 0.5。 OceanBase 0.5 各个 Server 进行了重构，在结构上与 0.4 版本发生了较大的变化。
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. 简介

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，并且在公司内部收藏夹、直通车报表、天猫评价等 OLTP 和 OLAP 在线业务上得到了广泛的应用。

1.1 背景信息

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奏效。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。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)，那么查询的时间会较长。假设平均每条查询时间是 5ms，那么 4000 条的查询时间可能达到 20s，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，就不需要再查询收藏表，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冗余表的大量修改，并压垮数据库。

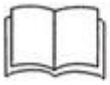
为了解决以上问题，淘宝急需一个适合互联网规模的分布式数据库，这个数据库不仅要能够解决收藏夹面临的业务挑战，还要能够做到可扩展、低成本、易用，并能够应用到更多的业务场景。因此，淘宝研发的千亿级海量数据库 OceanBase 应运而生。

1.2 系统架构

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。而经过对在线业务数据的分析，我们发现虽然其数据量十分庞大，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

单独的更新服务器来记录最近一段时间的修改增量，而以前的数据保持不变。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。

OceanBase 将写事务集中在独立的更新服务器上，可以避免复杂的分布式事务，高效地实现跨行跨表事务。更新服务器上的修改增量通过定期合并操作融合多台基准数据服务器中，从而避免其成为瓶颈，实现了良好的扩展性。当然，更新服务器的处理能力总是有一定的限制，因此,更新服务器对于内存、网卡及 CPU 等硬件配置要求较高。



- **基准数据又称静态数据，指某个时间点以前的数据快照，按 Range 划分以后存储多台基准数据服务器中。**
- **增量数据又称动态数据，在某个时间点以后更新数据，存储在更新服务器中的内存中。**

OceanBase 主要由以下四个部分(RootServer、UpdateServer、MergeServer、ChunkServer)组成，其整体架构如**图 1-1** 所示：

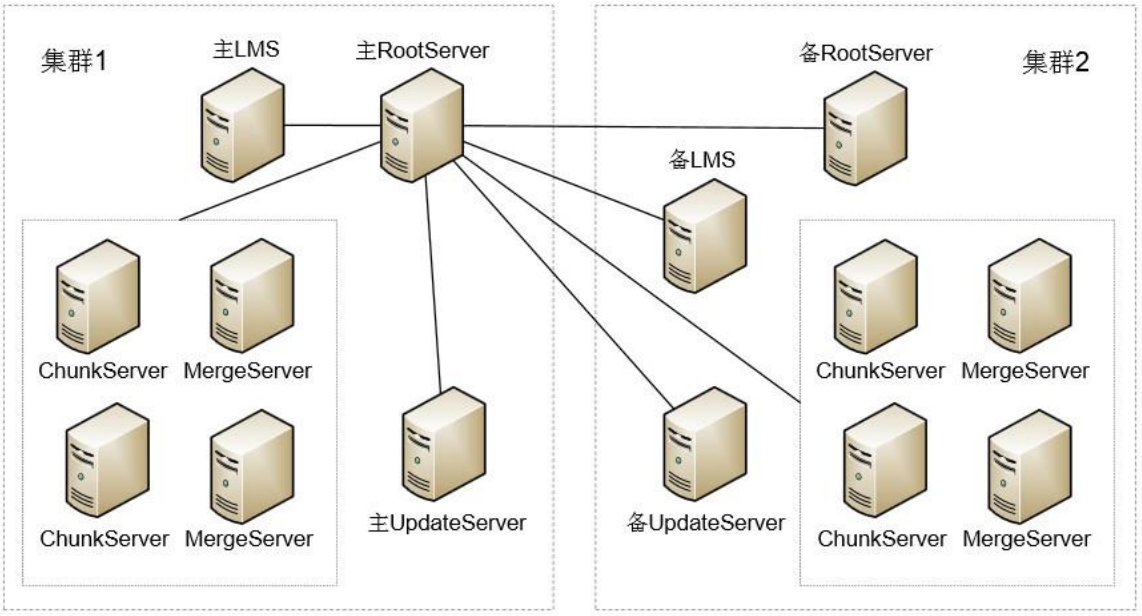


图 1-1 整体架构

- RootServer

  主控服务器，提供服务器管理和集群管理的功能。一般与 UpdateServer 部署在同一台服务器中。

- UpdateServer

  更新服务器，存储 OceanBase 系统的增量更新数据，是 OceanBase 中唯一的写入模块。一般与 RootServer 部署在同一台服务器中。

- ChunkServer

  基准数据服务器，存储 OceanBase 系统的基准数据。一般与 MergeServer 部署在同一台服务器中。

- MergeServer

  合并服务器，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。一般与 ChunkServer 部署在同一台服务器中。

- LMS（Listener MergeServer）

  OceanBase 集群内部特殊的 MergeServer 进程，只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。一般与 RootServer 部署在同一台服务器中。

0.5 版本的 OceanBase 不存在主备集群的概念，我们将包含两个或两个以上集群的 OceanBase 称为大集群。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，如图 1-2 所示。

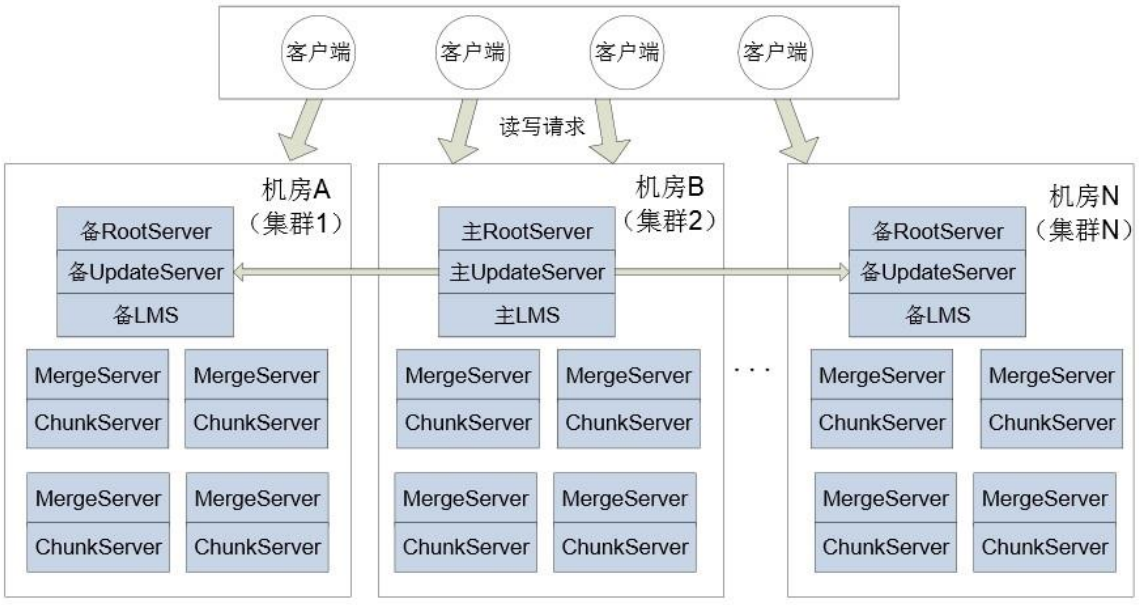
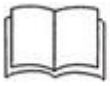


图 1-2　OceanBase 多机房部署

- 在大集群中只有一个主 RootServer，其余均为备 RootServer。主 RootServer 负责管理所有集群的 Server 列表，维护每个 Server 所属的集群 ID，并支持获取指定集群 ID 下的 Server 列表。在任何时刻，大集群中的 ChunkServer、MergeServer、UpdateServer 和 LMS 都只能看到一台 RootServer，即主 RootServer。
- 备 RootServer 定期从主 RootServer 同步 MergeServer 的列表，并利用这些 MergeServer 的地址访问内部表，定期刷新 Schema 和 Config。在进行主备切换时，新主 RootServer 通过内部表恢复集群所需的元数据信息。
- 在大集群中只有一个主 UpdateServer，其均为备他 UpdateServer。其中由主 UpdateServer 提供读写服务，并负责将操作日志同步到所有的备 UpdateServer。
- 客户端根据集群的流量分配比例将读写操作发往不同集群 MergeServer。MergeServer 将每个 Tablet 的读取请求发送到 Tablet 所在的 ChunkServer；ChunkServer 读取 SSTable 中包含的基准数据，然后请求 UpdateServer 获取相应的增量数据；MergeServer 将基准数据与增量数据融合后得到最终结果返回给客户端。


说明：

MergeServer 在读取 ChunkServer 的数据时，优先读本集群内的 ChunkServer。而 ChunkServer 在读取 UpdateServer 数据时如果未指定读主，则优先读本集群内的 UpdateServer，其次读其他集群的备 UpdateServer，最后读主 UpdateServer。

2. RootServer

RootServer 的功能主要包括：集群管理、数据分布以及副本管理。

RootServer 与所有 ChunkServer、MergeServer、UpdateServer 和 LMS 维持租约，提供服务器管理和集群管理的功能。所有集群内部同一时刻只允许一个 UpdateServer 提供写服务，这个 UpdateServer 称为主 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。

RootServer 功能细分：

- 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；
- 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 RootTable

RootServer 的中心数据结构为存储 Tablet 数据分布的一系列有序表格，称为 RootTable。其主要存储了集群中所有 Tablet 子表的数据量大小、数据校验码和位置分布等信息，是集群中最重要的元数据。

本小节主要介绍 RootTable 的结构和缓存机制。

2.1.1 结构

0.4 版本中，RootTable 以虚拟内存表的方式存在集群扩展性受限、对 RootServer 存在强依赖、不支持事务性读写 RootTable 等问题，因此 0.5 版本的 OceanBase 将内存表改造为系统内部表。其主要有如以下几个优点：

- 从根本上解决因受内存资源受限而导致集群最大数据量受限问题。
- MergeServer 可以像访问用户表一样，直接访问 RootTable 获取 Table 的位置分布，从而有效降低 RootServer 的负载。
- 主备 RootServer 不再需要通过同步 CommitLog 来保证内存数据的一致性和可靠性。
- 通过 OceanBase 本身提供的 ACID 事务特性，可以实现对 RootTable 访问的原子性。

每个 User Table 使用固定的三层结构来存储其 Tablet 位置信息等元数据，这三层 Table 一起构成系统全部的 RootTable，如**图 2-1** 所示。

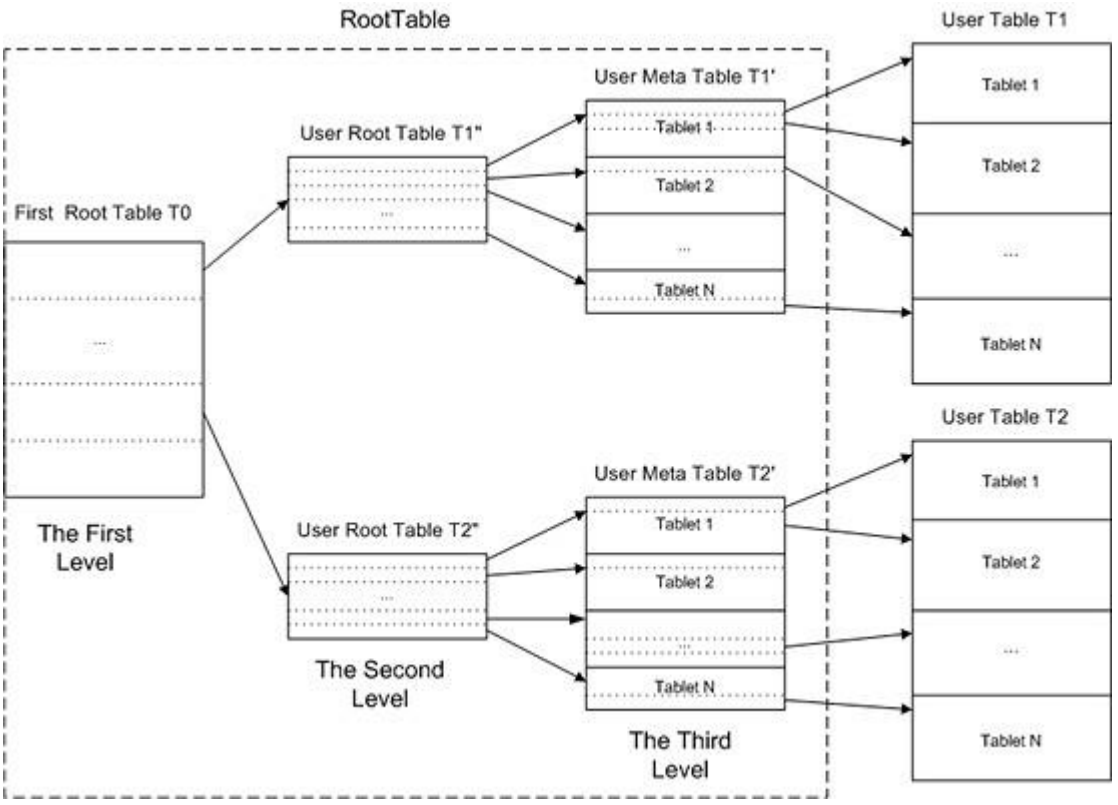


图 2-1 RootTable 结构

- 最右侧为 User Table，不属于 RootTable。每个 User Table 按照配置大小在每日合并中分裂成多个 Tablet，每个 Tablet 存储部分 User Table 的数据，所有 Tablet 分片构成整个 User Table 的所有数据。
- 第三层为 User Table 的元数据表，称为“User Meta Table”。每个 User Table 对应一个，每一行对应 User Table 中的一个 Tablet，存储 User Table 的 Tablet 位置信息。同时可根据配置大小在每日合并分裂为多个 Tablet，每个 Tablet 存储部分 User meta Table 数据。
- 第二层为 User Meta Table 的元数据表，称为“User Root Table”。每一行对应第三层中的一个 Tablet，存储 User Meta Table 的 Tablet 位置信息。“User Root Table”只有一个 Tablet，且该 Tablet 不允许分裂。
- 第一层是所有 User Root Table 的入口，称为“First Root Table”。每一行对应第二层中的一个 User Root Table，存储 User Root Table 的位置信息。“First Root Table”只有一个 Tablet，且该 Tablet 不允许分裂。

假设每行元数据大小为 1K，每个 Tablet 最大为 256M，则每个 User Table 最多允许存在（2^{18}）×（2^{18}）=（2^{36}）个 Tablet，则每个 User Table 的总数据量最大可达(2^{36}) × (2^{28}) = (2^{64})B,同时系统最大允许存在的 User Table 数约为(2^{28}/3)≈85K 个。

2.1.2 缓存机制

RootTable 内部表化后，所有对 RootTable 的读写操作都要经过 RootServer 的 RootTable 管理模块，对外提供统一的读写接口。为了提高对 RootTable 的访问效率，RootTable 按照层级进行多级缓存，每一级缓存上一级的部分或者全部行数据，每一层级单独进行管理，分别使用不同的超时控制。

- 集群中，First Root Table 有且只有一个 Tablet，因此其位置元数据信息只有一项，称为第 0 级缓存，需常驻内存，不使用淘汰机制。
- First Root Table 中存储了所有 User Root Table 的所有 Tablet 元数据，因为 User Root Table 不会分裂，所以每 User Root Table 只有一个 Tablet，为了简化定位逻辑，降低查询中对 OceanBase 模块线程资源的占用，这部分缓存保存本集群的全部 User Root Table 元数据，不进行淘汰，并支持动态扩展。
- User Root Table 和 User Meta Table 分别存储了所有 User Meta Table 和 User Table 的所有 Tablet 元数据，User Meta Table 和 User Table 一般都是可以分裂的，而且数据量相对前两层指数级放大，因此通过缓存淘汰算法降低对内存资源的占用。

2.1.3 Tablet 汇报处理

RootServer 重启恢复或者迁移、复制、删除、每日合并、Tablet 合并等都会涉及 Tablet 元数据的变化。RootServer 根据 ChunkServer 汇报的 Tablet 副本更新 RootTable。除了迁移、复制不会更改 RootTable 中的 Rowkey，其他分裂、合并的更新通常会涉及到多行事务。同时更新 RootTable 后的结果仍然能够保证 RootTable 的连续和完整性。如果 Tablet 的 Range 分裂点不一致，则需要回滚 RootTable 的修改。

对所有汇报的每个 Tablet A (R1, R2]处理的过程为：查询 RootTable 获取内部表对应与此 Range 存在交集的 Tablet 记录，如果不存在交集，则直接插入一行新记录；如果存在交集的情况，则分为五种情形，如**图 2-2** 所示。

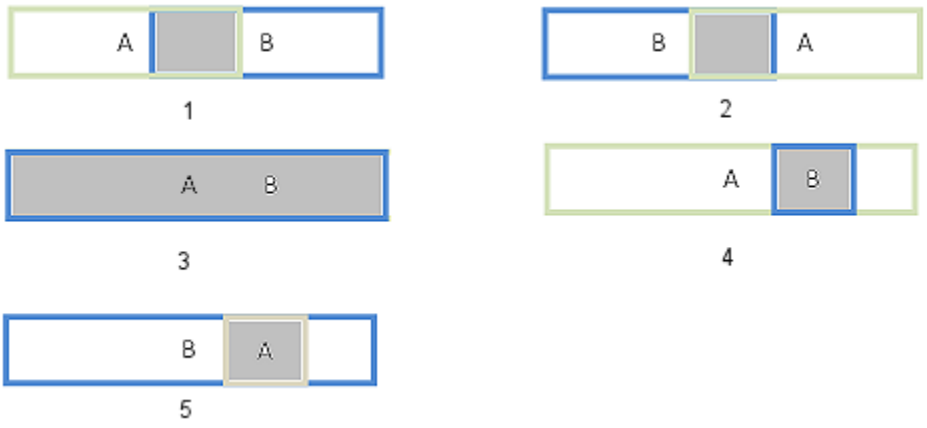
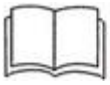


图 2-2 Range 存在交集

- “1”和“2”为非法异常情况（start_key > A.start_key && start_key < A.end_key && end_key > A.start_key && end_key > A.end_key）和（start_key < A.start_key && end_key > A.start_key && end_key < A.end_key）。


说明：

一个 Table 会划分成很多个 Tablet，这些 Tablet 中包含的数据是不重叠的，例如：（a,b)(b,c)(c,d)。当一个 Tablet 中数据过多时，就会进行分裂，且分裂出来的 Range 仍然是有序的。这些新的 Range 进行汇报，并和老的 Range 比较时，不会出现新 Range 与老 Range 交叉的情况，否则 RootServer 报错。

- “3”表示存在单行 Range B，而且与 Range A 恰好相同（start_key = A.start_key && end_key == A.end_key），则只需锁住 B 行更新对应的其他列数据。
- “4”表示 RootTable 存在一行或者多行内容（start_key >= A.start_key && start_key < A.end_key && end_key <= A.end_key），而且 Range A 是此多行的并集，则判定为合并操作。（目前系统中没有 tablet 的合并，出现这种情况可能是 CS 下线之后再重新上线，在此期间 tablet 已经分裂过）
- “5”表示存在单行 Range B，而且 Range A 是此行的子集（start_key <= A.start_key && end_key >= A.end_key），然后根据版本号判定为分裂操作，锁住 B 行，将其拆分成三行或者两行；但是考虑到汇报是按序的，可以将这部操作查询优化为（start_key = A.start_key && end_key > A.end_key）。

2.1.4 Tablet 复制与负载均衡

RootServer 中有两种操作都可能触发 Tablet 迁移：Tablet 复制和负载均衡。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 迁移任务列表等。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。

2.2 用户表的创建和删除

2.2.1 用户表的创建

在使用 DDL 语句创建 User Table 时，OceanBase 会自动创建对应的 User Root Table 和 User Meta Table，其创建过程如下：

1.分配三个 Table ID，分别用于 User Root Table、User Meta Table 和 User Table。

2.创建 User Root Table。

a) 根据 User Meta Table 构造 User Root Table 的 Schema。
b) 选择 ChunkServer，创建对应的 Tablet。

3.创建 User Meta Table

c) 根据 User Table 构造 User Meta Table 的 Schema。
d) 选择 ChunkSever，创建对应的 Tablet。

4.选择 ChunkSever，创建 User Table 对应的 Tablet。

5.更新 Schema 相关的内部表。

e) 将 User Root Table、User Meta Table 和 User Table 的 Schema 记录到 Schema 相关的内部表。
f) 刷新 Schema Manager，并同步到各个 Server。

6.将 User Root Table、User Meta Table、User Table 的 Tablet Location 更新到上一级的 RootTable 内部表。

2.2.2 用户表的删除

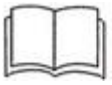
User Table 的删除操作相对比较简单，并不需要同步方式进行数据删除，而是异步延迟删除 Tablet 数据，只需从 Schema 表中清除对应的三张关联表即可，过程如下：

1. 删除三张核心表（核心表为：__all_table，__all_all_column, __all_join_info）中 User Table Schema 相关的内容。

2. 删除三张核心表中 User Meta Table Schema 相关的内容。

3. 删除三张核心表中 User Root Table Schema 相关的内容。

4. 刷新 Schema Manager，并同步到各个 Server。


说明:除了用户表之外的表都不允许删除。

非实时删除 User Table 的 Tablet 以及各级 RootTable 内部表中的数据，可以有效降低“Drop Table”的响应时间，同时可以解决 ChunkSever 重启异常情况下汇报了不存在的 Tablet 问题，也可以用于误操作之后的数据恢复，清理 RootTable 内部表数据交由后端一致性检查线程处理。

2.3 BootStrap 流程

BootStrap 是系统的初始化安装过程，主要用于创建系统内部表，并初始化系统配置。主集群 BootStrap 的流程如下：

1. 创建 First Root Table，并记录 First Root Table 唯一 Tablet 的位置信息，同时更新内存 First Root Table 的 RootTable 元数据位置信息。

2. 创建系统的内部表中与 Schema 相关的核心表“__all_table”、“__all_column”、“__all_join_info”和“__all_dll_operation”。

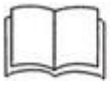
a) 选择合适的 ChunkServer 创建空 Tablet。
b) 修改 First Root Table 内部表加入一条新的记录。

3. 创建系统的其他内部表。

a) 选择合适的 ChunkServer 创建空 Tablet。
b) 修改 First Root Table 内部表加入一条新的记录。
c) 修改 Schema 相关的核心表，将内部表记录在 Schema 表中。

4. 读取内部表，刷新系统目前的 Schema Manager，同步到在线的 ChunkServer、MergeServer 和主 UpdateServer。

5. 初始化部分内部表的数据，包括“__all_cluster”、“__all_server”和“__all_user”等。


说明：

内部表的 Tablet 位置元数据直接写到 First Root Table，不创建类似于 User MetaTable 和 User Root Table 的表。
如果是 OceanBase 多集群部署，BootStrap 流程在创建空 Tablet 时，会在每个集群上分别选取合适的 ChunkServer 进行创建。

3. UpdateServer

UpdateServer 是集群中唯一能够接受写入的模块，主要用于存储增量数据。

3.1 基本信息

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。

OceanBase 集群通过定期合并，将 UpdateServer 一段时间之前的增量更新源不断地分散到 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，从而保证 UpdateServer 的读写性能。另外，系统实现时对 UpdateServer 的内存操作、网络框架、磁盘操作进行了大量的优化。

外部读写请求都直接发给主 UpdateServer，每日合并或者指定版本读取请求（比 如全量 dump）发给备 UpdateServer，并按照如下策略执行：

1. 随机选择本集群的某个备 UpdateServer。
2. 随机选择其它集群的某个备 UpdateServer。
3. 选择主 UpdateServer。

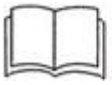
3.2 主备同步

UpdateServer 采用一主多备的高可用架构。主备 UpdateServer 之间使用 Paxos 协议进行 CommitLog 的一致性同步，以加强 OceanBase 的数据安全性。

主 UpdateServer 向备 UpdateServer 同步日志的请求采用异步请求方式，但是写本地硬盘采用同步的方式，即前一个日志块没有响应不会写下一个日志块。且任何一份数据需要在三 UpdateServer 中的两台确认写盘成功才认为 Commit 成功，保证任何一台机器故障都不会丢失数据，并且系统依然可以正常运行。当两台 UpdateServer 故障时，只要硬盘不出问题，数据仍不会丢失，但是系统会停止运行。

CommitLog 同步流程如下：

1. 主 UpdateServer 将 CommitLog 发送到备 UpdateServer。
2. 备 UpdateServer 收到主 UpdateServer 同步的 CommitLog 后，先将 CommitLog 存储在一个环形缓冲区。
3. 备 UpdateServer 应答主 UpdateServer，并在应答中携带上“nsync”标志。



说明：

此次应答只作为心跳机制，主要是告诉主机自己在追改日志过程中，保证不被主机踢掉。则在确认的日志连续后写盘。

4. 备 UpdateServer 状态为“sync”，且判断缓冲区中的日志是连续的，再将这些日志以同步的方式顺序写入日志文件。

5. 备 UpdateServer 针对已经写入硬盘的事务，应答主 UpdateServer。通过上述步骤不难看出：备 UpdateServer 写日志的方式也采用同步写日志的方式。这种方式可以有效防止 UpdateServer 故障时数据丢失。

如果主 UpdateServer 向备 UpdateServer 同步 CommitLog 全部超时，主 UpdateServer 就停止刷日志，并且把自己的状态设置成备，由 RootServer 重新选主。

UpdateServer 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。

3.3 选主流程

UpdateServer 选主流程需要保证至少有两台 UpdateServer 都向 RootServer 汇报了各自的最大日志时间戳，RootServer 选择其中时间戳最大的作为主 UpdateServer。

1. UpdateServer 在启动后，由 RootServer 根据 UpdateServer 最大日志时间选主。

2. 主备 UpdateServer 通过 lease 和 RootServer 保持通讯，保证自身活跃。

3. 主备 UpdateServer 之间，由主 UpdateServer 将系统每一次变更而产生的 CommitLog 发送给备 UpdateServer。

使用时间戳而不是日志 ID 是为了避免以下情况，假设有三台 UpdateServer，其中 UpdateServer01 为主，UpdateServer02 和 UpdateServer03 为备：

1. 当前，三台机器的日志 ID 都写到了 10，这时 UpdateServer01 发起了一次 Group Commit，要 CommitLog ID 为 11~20 的 10 条日志，但 UpdateServer02 和 UpdateServer03 都没有收到同步日志的请求，导致 UpdateServer01 的状态变成备。此时，对于客户端来说，11~20 的这些事务的状态是未知的。

2. RootServer 重新选主 UpdateServer，恰好 UpdateServer01 没有参与，RootServer 选择了 UpdateServer02 作为主。
3. UpdateServer02 发起了一次 commit，写入了 11~15 的 5 条日志，并且应答了用户。
4. 之后这三台 UPS 全部宕机重启了，重新进行选主。此时，如果按照最大日志 ID 进行选择的话，UpdateServer01 会被选择成主，那么 UpdateServer02 和 UpdateServer03 上的 11~15 日志就会被 UpdateServer01 上面的日志覆盖。但是 UpdateServer02 和 UpdateServer03 上的 11~15 号日志已经应答了用户。因此，使用最大日志时间戳，可以保证后写入数据的 UpdateServer 被选择成主。在这种情况下，就是 UpdateServer02 选择成主 UPS，那么 UpdateServer01 需要把自己最后一个日志文件头部记录的 committed id 之后的所有日志都删除，并且从 UpdateServer02 处同步新的数据。

3.4 日志回放策略

备 UpdateServer 在回放日志时，首先检查缓冲区中有无需要回放的日志，如果没有则检查日志文件中是否包含需要回放的日志，再没有则从主机处获取日志，并且写盘，然后再回放取到的日志。

在 CommitLog 的每个文件的头部存在该文件生成时主机的 Committed ID，用于回放日志的时候明确知道生成日志的时间，且哪些数据是已经 Commit。假设一个集群主 UpdateServer01 宕机，之后作为备机启动。在进行日志回放时，只能回放到最后一个文件头所记录的 Committed ID 处，超过 Committed ID 位置的日志需要从新主 UpdateServer 中获取；而 UpdateServer01 剩余的日志文件都进行改名，并逻辑删除。

3.5 Tablet 冻结

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。定期合并过程请参见“5.2.3 每日合并”。

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。

如图 4-1 所示，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。

在 OceanBase 系统中，用户的读写请求都发给 MergeServer。MergeServer 缓存了 Tablet 分布信息，并根据请求涉及的 Tablet 将请求转发给相应的 ChunkServer；对于写操作，由 MergeServer 进行预处理后，发送给 UpdateServer 执行。

如果读写请求需要跨多个 Tablet，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。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。

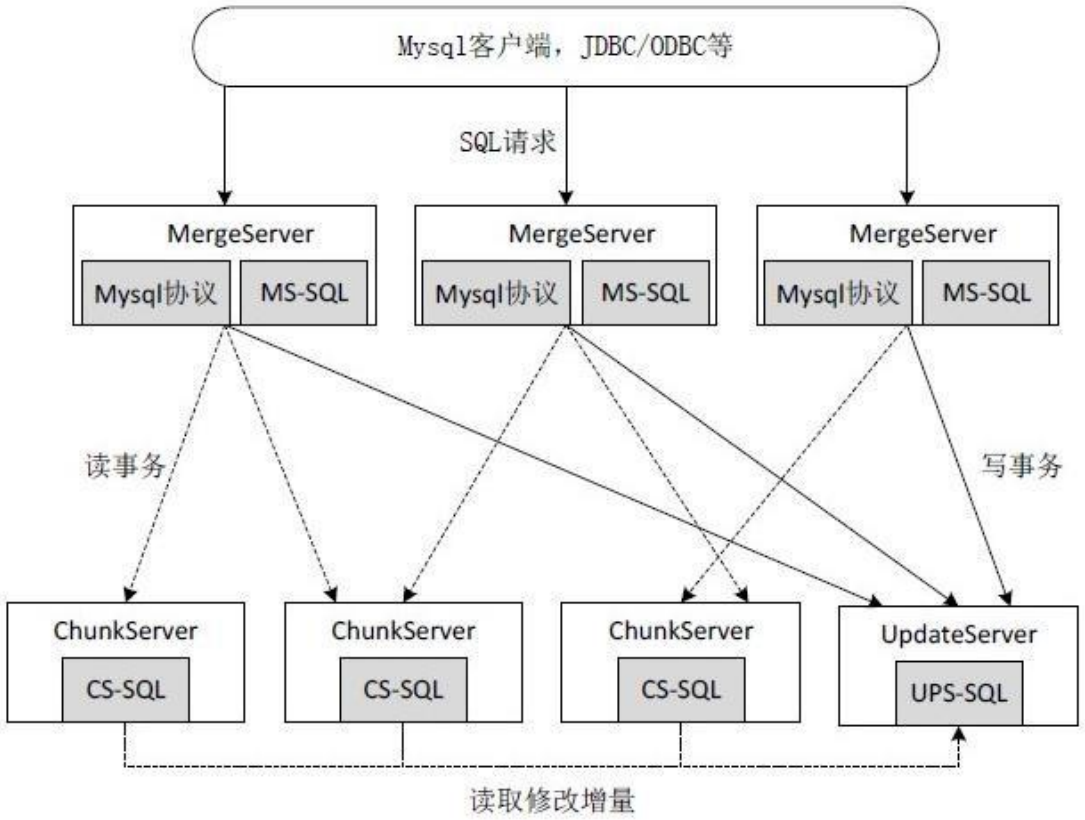


图 4-1 读写事务

4.1 读操作

如图 4-1 所示，读操作执行流程如下：

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。

4.2 写操作

如图 4-1 所示，写操作执行流程如下：

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。

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。

5. ChunkServer

ChunkServer 主要用于存储基准数据，基准数据按照主键有序划分为一个一个 Tablet、并提供读取服务和执行定期合并等。

5.1 SSTable 数据文件

SSTable 数据文件为 Tablet 的物理形式，是基准数据的存储方式，是一种按主键(rowkey，也称为 primary key)排序的文件格式。每个 Tablet 由一个或者多个 SSTable 组成（一般为一个）。

5.1.1 文件结构

SSTable 数据文件为 Tablet 的物理形式，是基准数据的存储方式，是一种按主键(rowkey，也称为 primary key)排序的文件格式。每个 Tablet 由一个或者多个 SSTable 组成（一般为一个）。

在 0.4 版本中，OceanBase 的合并是一个基于 Tablet 为单位的过程，每个 ChunkServer 会取一个 Tablet，然后将这个 Tablet 涵盖的 Range 在 UpdateServer 上进行扫描，如果扫描到的结果有增量数据，那么将 Tablet 的所有行读出来以后与 UpdateServer 上的扫描结果行进行归并排序，相同的行予以合并，然后存为新的 SSTable。即便在 UpdateServer 扫描对应 Range 的结果只有一行，也需要对整个 Tablet 进行全部扫描并合并和重新写入的过程。

如**图 5-1** 所示，由于更新数据较于全量数据比例较小，Tablet 中的更新其实远远小于 Tablet 本身的大小，因此 0.5 版本的 OceanBase 将 Tablet 进一步切分为更小的 Range，使更多的数据无需进行更新与合并。

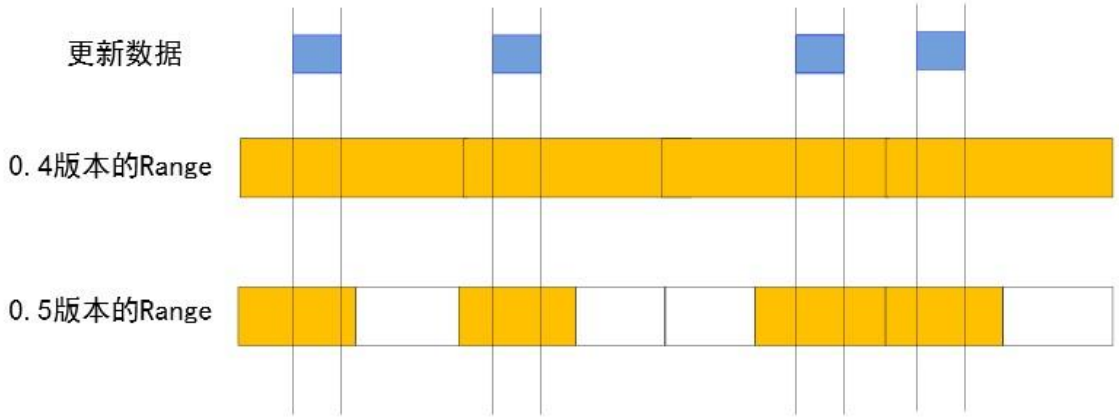


图 5-1 Range 对比

将 Range 切分更细以后带来一个新的问题，那就是 SSTable 个数会增大很多。例如，原来按 256MB 大小划分的 SSTable，现在分为 2MB，文件数量膨胀 128 倍，如果原来有 Tablet10 万个，现在则到 1280 万个。这个级别的元数据，将使得 RootTable 非常庞大，另外 ChunkServer 的磁盘上的文件个数膨胀以后，文件系统的元数据也同步加大。

为解决这一问题，0.5 版本的 OceanBase 仍将大表划分为大小约为 256MB 的 Tablet，然后在 Tablet 内部增加一级索引，索引到缺省为 2MB 的块。我们称这个块为宏块（MacroBlock），宏块由 4KB/8KB/16KB/64KB 等尺寸的块组成，我们称之为微块（MicroBlock）。

每日合并以宏块为单位，如果有更新数据，则重写宏块并更新索引，如果没有更新数据则保持不变。宏块大小缺省为 2MB，在系统初始化时设置，以后不能修改。ChunkServer 汇报给 RootServer 的 Tablet 仍按照原来的 256MB（或是配置）来进行，因此在 RootServer 端，看到的 Tablet 数保持不变。

SSTable 数据文件是一个大文件，划分为固定大小的宏块，类似于一个裸磁盘设备文件。其主要包括头文件和数据宏块，如图 5-2 所示。

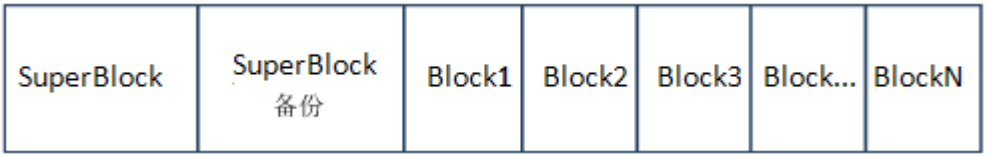


图 5-2 SSTable 结构

SuperBlock 是数据文件的文件头，记录了数据文件的关键信息。

SuperBlock 的有效数据如下：

```
struct SuperBlock
{
int32_t   header_size;        // SuperBlock 的尺寸，单位：Byte。
int32_t   version;  // 数据文件版本。
uint32_t magic;     // magic number，建议为“常量+磁盘号+机群唯一值”。
int32_t   attr;       // 暂时未用，必须置“0”。
int64_t   row_store_type;   // SSTable 行存储格式“1/2/3”: 稠密/稀疏/混合格式。
int64_t   macro_block_size;// 宏块大小，系统初始化时设定，缺省 2MB。
int64_t   total_macro_block_number;          // 该文件所有的宏块个数。
int64_t   used_macro_block_number;           // 已经使用的宏块个数。
int64_t   sstable_macro_block_number;        // 已经使用的 SSTable 宏块个数。
int64_t   first_macro_block_offset;   // 第一个宏块的相对于本结构头的偏移
值(单位：Byte)，后续的宏块顺序存放。
int64_t   total_sstable_data_size;     // 当前版本的 SSTable 的数据总量。
int64_t   tablet_number;    // 当前版本的 Tablet 个数。
int64_t   schema_entry_block_index;// Schema 宏块的入口。
int64_t   schema_number;  // 内存化后的 Schema 数组的长度。
int64_t   compressor_entry_block_index;       // Compressor name 的入口宏块。
int64_t   compressor_number;         // 内存化后的 Compressor 数组的长度。
int64_t   cur_tablet_meta_pos;        // 当前 Tablet 版本在以下数组中的位置。
int64_t   tablet_meta_entry_block_index[MAX_TABLET_VER_NUM];         // Tablet 元
数据的入口，次数组循环使用，下个每日合并用(cur_tablet_meta_pos+1)%数组长度。
int64_t   macro_block_meta_entry_block_index; // 宏块元数据的入口宏块。
```

```
int64_t   first_free_block_index;      // 第一个可用的宏块，“-1”表示没有。
int64_t   free_block_number;           // 可用的宏块个数(用于迁入 Tablet
时的可用等空间判断)。
int64_t   reserved[];          // 保留字段。
};
```

“Block1~BlockN”是数据宏块，即 MacroBlock，主要存放各种元数据和管理信息和 SSTable 数据。

5.1.2 宏块

MacroBlock 中主要存放 SSTable 数据，另外有少数 MacroBlock 存放各种元数据和管理信息。

* 元数据和管理信息

MacroBlock 存放的元数据和管理信息主要包括 Schema 数据、Compressor 数据、宏块元数据、Tablet 元数据，如 **图 5-3** 所示。

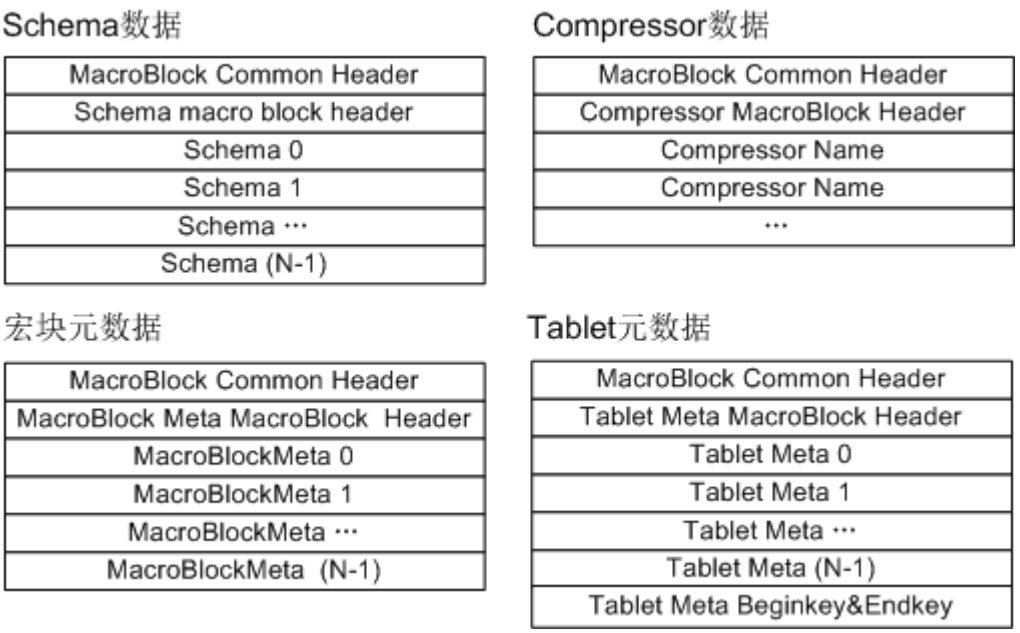


图 5-3 元数据和管理信息

* SSTable 数据

大部分 MacroBlock 中存放实际的 SSTable 数据，每个 SSTable 宏块是某个 Tablet 的一段连续的主键范围并且存放于一个宏块，包含 MicroBlocks 和 MicroBlock Index，如图 5-4 所示。

- Schema 数据

Schema 数据是 ChunkServer 转换后的必要信息，包括表 ID、列数、列 ID 及值的类型、主键列数及主键序号、压缩库名称的索引和 microblock 大小等。

Schema 用一个或多个宏块存储，入口宏块由 SuperBlock 中的“schema_entry_block_index”决定，多个 Schema 时用链表从后向前链接。每日合并时，ChunkServer 会保存变化后的 Schema，没有变化的 Schema 不会多次保存。如果 Schema 数据占用空间太大(例如保存的 Schema 总数超过表的总数的某个倍数)，则启用回收机制，删除不再使用的 Schema 数据。

Schema 的修改采用延期更新到磁盘的方式，例如 ChunkServer 正常退出或者每日合并前后把更新的 Schema 写到磁盘。

- Compressor 数据

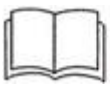
Compressor 数据用一个或多个宏块存储(一般只用一个宏块存储)。Compressor 数据宏块存放 ChunkServer 所有 SSTable 宏块使用的 CompressorName。Compressor 内存数组中还包含了该 Compressor 加载到内存的压缩库。

- 宏块元数据

表示该宏块存储的数据类型和版本号等信息。宏块元数据用一个或多个宏块(通常是多块)存储。

- Tablet 元数据

存储 Table ID、Tablet 总行数、实际大小、SSTable 宏块个数、此 Tablet 使用的 Schema、UpdateServer 冻结时的 CommitLog ID 和时间等信息。Tablet 元数据用一个或多个宏块(通常是多个)存储。



每个 MacroBlock 都会在头部存放一个公共的头信息 MacroBlock Common Header，用于标示该宏块本身的状态以及类型等信息。

Schema 的修改采用延期更新到磁盘的方式，例如 ChunkServer 正常退出或者每日合并前后把更新的 Schema 写到磁盘。

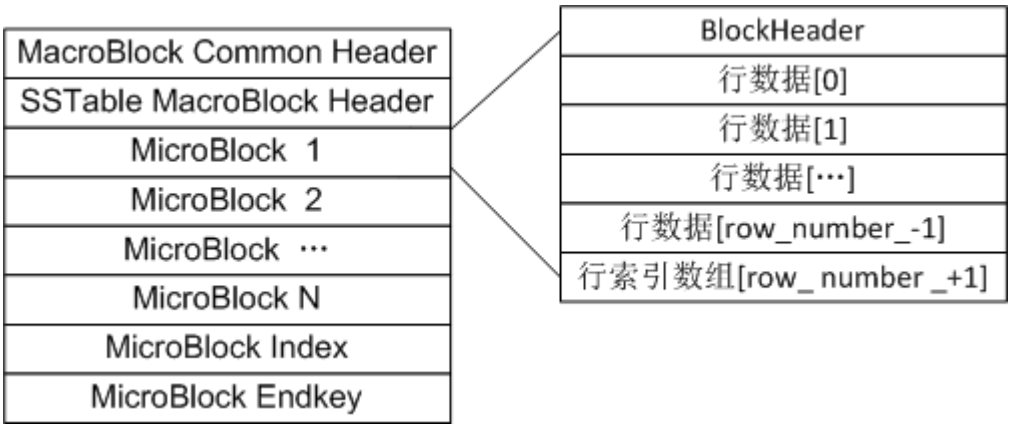


图 5-4 SSTable 数据

- MacroBlock Common Header

  公共头信息，用于标示该宏块本身的状态以及类型等信息。

- SSTable MacroBlock Header

  描述了该 Block 的元信息，包括 Block Index 和 Block Data 的位置大小，以及 Range 等信息。

- MicroBlock(1~N)

  每个 MicroBlock 大小为配置的 Block Size,存放行数据。

- MicroBlock Index

  索引了分块中所有的 Block，包括块数据长度和 Endkey 长度。

- MicroBlock Endkey

  N+1 个 Endkey 组成的数组，第一个是整个的 Beginkey，最后一个即为整个的 Endkey

  MicroBlock 数据由 BlockHeader、行数据和行索引数组组成：

- BlockHeader

  主要记录行索引数组的偏移量、行数等信息。

- 行数据

  主要由主键列和其余列组成，列顺序及列数由表的 schema 定义，并以行结束符分隔离。非主键部分有以下三种格式：

  a) 稠密格式

  每一行的列顺序和列数由表的 schema 定义，此处按 schema 的顺序保存了每个列的值，不保存对应的列 ID。这个格式用作 ChunkServer 每日合并的 SSTable 的存储。

  b) 稀疏格式

  列顺序和列数都不确定，column_id 和 column_value 总是成对地出现（最后的行结束符除外），通常表示对该列的修改操作。这个格式用作对 UpdateServer 的转储。

  c) 混合格式 暂时未定义。

- 行索引数组

  row_index_array_[row_number_+1]是[row_number_+1]个元素的数组，记录了每行行首相对于该 block 的偏移值。最后一个元素指向最后一行的尾部。

5.2 读写逻辑

主要介绍 ChunkServer 上 Tablet 的读写逻辑，其中写逻辑包括迁移写入新的 Tablet 和每日合并。

5.2.1 读逻辑

ChunkServer 上的 SSTable 组织起来是一个自然的 BTree 结构，如**图 5-5** 所示。

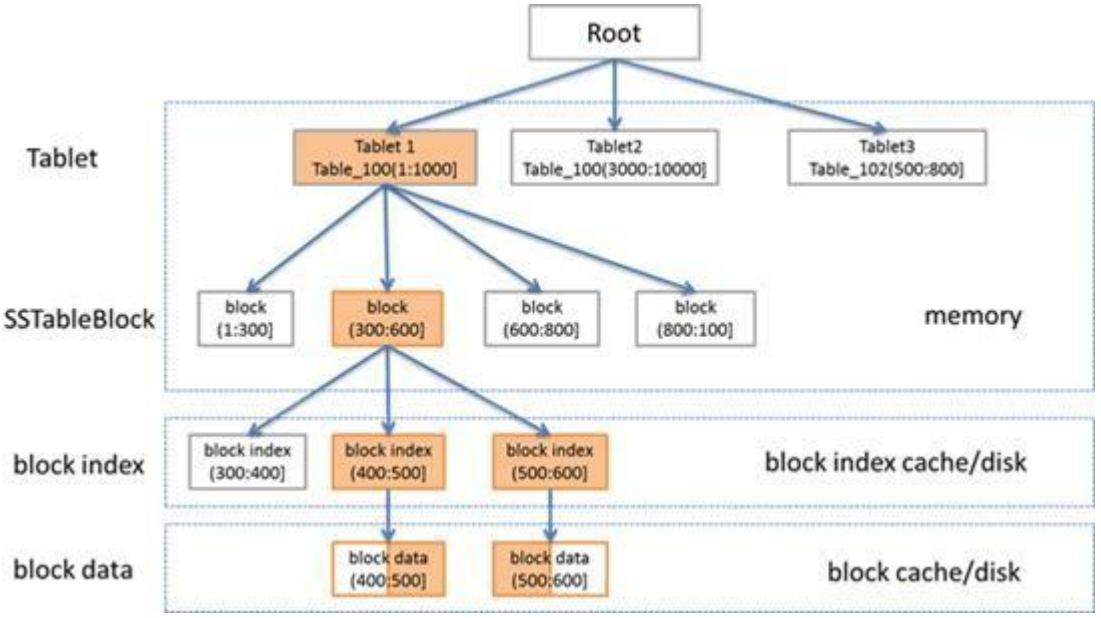


图 5-5 read

- Tablet 层节点是 ChunkServer 向 RootServer 汇报的单位。
- 每次系统启动时从索引文件中加载到内存。
- MacroBlock 索引层也是从索引文件中的 Tablet Descriptor 信息中读取加载到内存中。
- Block Index 是根据上两层定位结果按需装载进 Block Index Cache。
- 根据 Block Index 定位结果读取 MacroBlock 存入 Block Cache。假设查询表 100 中的范围[470,520]；可以看到沿着树的每层节点中做二分查找的过程。

5.2.2 迁移写入新的 Tablet

迁移写入新的 Tablet 的过程如下：

1. 每次从源端读入一个 MacroBlock 的数据。

2. 选择一个本地磁盘，找到对应的数据文件。

3. 分配一个 Tablet 元数据对象，并在数据文件中分配一个 MacroBlock。

4. 写入 MacroBlock，修改宏块元数据，将 MacroBlock 加入 Tablet 元数据的 MacroBlock 列表。

5. 将 Tablet 元数据加入 TabletImage，并记录 CommitLog。

5.2.3 每日合并

为了保证 UpdateServer 性能，OceanBase 内部会定期触发合并，主要将 UpdateServer 中的增量更新分发到 ChunkServer 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），因此定期合并也称为每日合并。

每日合并的主要流程如下：

1. 建立两个宏块大小的内存。

2. 从 TabletImage 中选取一个旧版本 Tablet，遍历其中的每一个 MacroBlock。

   - 如果当前 MacroBlock 有更新，则执行如下步骤。

   a) 与 UpdateServer 数据合并以后，按顺序写入“步骤 1”中建立的内存块。
   b) 如果两块内存都已经写满，那么刷第 1 块内存到数据文件(分配新的 MacroBlock，更新宏块元信息，加入新版本的 Tablet 元数据)，并清空第 1 块内存；调换两块内存的顺序，继续写入数据。
   c) 重复步骤“a”~步骤“c”，直到没有更新的 MacroBlock。

   - 如果 MacroBlock 没有更新，则执行如下操作。如果内存块中已经写入了数据，刷新“步骤 1”中的两块内存到数据文件，把当前未发生变化的宏块元数据加入新版本 Tablet 元数据。

3. 重复上述步骤直到所有 MacroBlock 处理完毕。

4. 在新的宏块元数据列表中计算分裂的点，将 Tablet 元数据划分为多个分裂的宏块元数据列表。

5. 将建立的新版本 Tablet 元数据加入新版本的 TabletImage。

5.2.4 宏块元数据的重整

当修改的元数据达到一定量时，元数据的内存池由于补丁内存块的不断增长会变得庞大，需要定期对元数据进行重整，一般在每日合并的间隙进行元数据的重整。

元数据的重整按磁盘依次进行，以减少内存使用量。

1. 发起重整时，首先切换当前 CommitLog 文件，记录当前重整开始的点，以作为下次重启系统后开始日志重放的点。

2. 建立一个新的元数据内存分配池。

3. 将数组中的指针指向的内存块（元数据）拷贝到新的内存池分配的内存中。

4. 每写满一个 2MB 内存块，即可刷到磁盘中，形成一个元数据链表。

注意:写元数据磁盘块本身也会修改元数据内容，按正常修改元数据逻辑进行。

5. 当指针数据遍历完成以后，即所有新的元数据都已刷到磁盘以后，修改 SuperBlock 中的元数据索引指向最后一个内存块。

6. 在等待足够长的时间以后(假定我们所有的读请求都不会超过这个时间)，可以确定不会有线程引用到老的元数据内存，则将老的元数据内存池一次性释放。

6. OceanBase 特性

主要通过分层结构、可靠性与可用性、数据正确性、单点性能和 SSD 支持等方面介绍 OceanBase 特性。

6.1 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。

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 ChunkServer 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。

6.2 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。

- 在 ChunkServer 中按表保存了基准数据的多个副本。
- 在 UpdateServer 中按表保存了增量数据的多个副本。
- ChunkServer 某个表的多个副本可以同时提供服务。
- 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。

在 0.5 版本的 OceanBase 中，按表配置副本数。例如，可配置“表 a”的副本数为 “3”，“表 b”的副本数为“1”。但是，一个表的副本数不会超过本集群 ChunkServer 的个数。例如，配置“表 a”的副本数为“3”，但集群 1 中只有 1 台 ChunkServer，那么“表 a”实际的副本数为“1”。

6.3 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。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。它们的内存表需要保持一致。为此，UpdateServer 为内存表生成一个校验码，内存表每次更新时，校验码同步更新并记录在对应的 CommitLog 中。备 UpdateServer 收到 CommitLog 重放更新内存表时，也同步更新内存表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内存表时也同步更新内存表校验码并与保存在每条 CommitLog 中校验码对照。

- 数据副本校验

定期合并时，新的 Tablet 由各个 ChunkServer 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，ChunkServer 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。

6.4 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。
- 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。针对 UpdateServer 全内存、收发的网络包一般比较小的特点，OceanBase 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 会在硬件上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。

6.5 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 35000IOPS 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”。