OceanBase架构介绍

日照

2013/3/2

v0.1

# OceanBase架构初探

OceanBase是阿里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月为止，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。

从模块划分的角度看，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系统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。本章介绍OceanBase系统的设计思路和整体架构。

## 背景简介

淘宝是一家迅速发展的公司。全球网站排名公司Alexa提供的数据显示，2010年4月27日，Amazon、Ebay的用户占全球互联网用户的百分比分别为3.47%和2.68%，而淘宝的用户占全球互联网用户的百分比则达到了4.1%，淘宝网日独立访问量从此超过了Amazon和Ebay。

淘宝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。

淘宝收藏夹数据库包含了收藏info表（一条一条的收藏信息）和收藏item表(被收藏的宝贝和店铺)等：

* 收藏info表保存收藏信息条目，数百亿条
* 收藏item表保存收藏的宝贝和店铺的详细信息，数十亿条
* 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：假设如果平均每条item查询时间是5ms，则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可能导致收藏info表的大量修改，并压垮数据库。

为此，淘宝需要研发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。为此，淘宝研发了千亿级海量数据库OceanBase，并且已经于2011年8月底开源（<http://oceanbase.taobao.org/>）。虽然距离OceanBase开源已经超过一年多的时间，但OceanBase系统还有很多的问题，其中以易用性和可运维性最为严重。OceanBase团队一直在不断完善着系统，同时，我们也很乐意把设计开发过程中的一些经验分享出来。

## 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。

一种常见的做法是根据业务特点对数据库进行水平拆分，通常的做法是根据某个业务字段（通常取用户编号，user\_id）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：

1. 数据和负载增加后添加机器的操作比较复杂，往往需要人工介入；
2. 有些范围查询需要访问几乎所有的分区，例如，按照user\_id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；
3. 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。

 万事有其利必有一弊，分布式表格系统虽然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，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。这样，写事务都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相对较好，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，在软件层面也针对这种场景做了大量的优化。

## 系统架构

### 整体架构图

图1-1 OceanBase整体架构图

如图1-1，OceanBase有如下几个部分组成：

* 客户端：用户使用OceanBase的方式和Mysql数据库完全相同，支持JDBC、C客户端访问，等等。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，主备之间可以配置不同的同步模式。部署时，UpdateServer进程和RootServer进程往往共用物理服务器。
* 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。

图1-2 OceanBase双机房部署

如图1-2，OceanBase支持部署多个机房，每个机房部署一个包含RootServer、 MergeServer 、ChunkServer以及UpdateServer的完整OceanBase集群，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。

### 客户端

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

1. 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。

2. Java客户端：OceanBase内部部署了多台MergeServer，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，Java客户端首先按照一定的策略定位到某台MergeServer，接着调用Mysql JDBC Driver往这台MergeServer发送读写请求。Java客户端实现符合JDBC标准，能够支持Spring、iBatis等Java编程框架。

3. C客户端：OceanBase C客户端的功能和Java客户端类似。它首先按照一定的策略定位到某台MergeServer，接着调用Mysql 标准C客户端往这台MergeServer发送读写请求。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（与Oracle的系统表类似，存储OceanBase系统的元数据）内。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某台MergeServer，并负责对出现故障的MergeServer进行容错处理。

Java/C客户端访问OceanBase的流程大致如下：

1. 请求RootServer获取集群中MergeServer的地址列表。
2. 按照一定的策略选择某台MergeServer发送读写请求。客户端与MergeServer之间的通信协议兼容原生的Mysql协议，因此，只需要调用Mysql JDBC Driver或者Mysql C客户端这样的标准库即可。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重试；如果请求某台MergeServer失败超过一定的次数，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，往往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。

### 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成为主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）。OceanBase的数据分布方式与Bigtable一样采用顺序分布，不同的是，OceanBase没有采用RootTable + MetaTable两级索引结构，而是采用RootTable一级索引结构。



图1-3 基准数据tablet划分

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。

RootServer采用一主一备的结构，主备之间数据强同步，并通过Linux HA（<http://www.linux-ha.org>）软件实现高可用性。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。

### 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。这样，ChunkServer故障是不会影响用户查询的。

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。

### ChunkServer

ChunkServer的功能包括：存储多个tablet、提供读取服务、执行定期合并以及数据分发。

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或者数据分发操作，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。

### 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集群通过定期合并和数据分发这两种机制将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。

### 定期合并&数据分发

定期合并和数据分发都是将UpdateServer中的增量更新分发到ChunkServer中的手段，二者的整体流程比较类似：

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启动定期合并或者数据分发操作，从UpdateServer获取每个tablet 对应的增量更新数据。

定期合并与数据分发两者之间的不同点在于，数据分发过程中ChunkServer只是将UpdateServer中冻结内存表中的增量更新数据缓存到本地，而定期合并过程中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），而数据分发可以不受限制。



图1-4 定期合并不停读服务

如图1-4，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

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

## 架构剖析

### 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，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 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日志同步到主备的情况下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主备同步也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。

### 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，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中持久化；
* 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；
* …

### 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，另外，每台机器都通过软件的方式实现了RAID1，将数据自动复制到多块磁盘，进一步增强了可靠性。
* ChunkServer的多个副本可以同时提供服务。Bigtable以及HBase这样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，备机几乎没有预热时间。
* 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，但存储系统单机通常只能服务少得多的数据量。

### 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获取需要读取的基线数据，并将物理执行计划和基线数据一起传给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传入的基线数据。

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。

### 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支持通过定期合并或者数据分发的方式将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磁盘，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，如果UpdateServer发生故障，比如机器突然停电，RAID卡能够确保将缓存中的数据刷入磁盘，不会出现丢数据的情况。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。

### 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磁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磁盘可以提供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，随机写入不能发挥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：除了操作日志总是顺序追加写入到普通SAS盘上，剩下的写请求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。

### 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。前面8.4.1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则更为常见。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。

### 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，这个存储引擎支持如下特性：

* 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；

二期的OceanBase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ChunkServer，实现数据复制、负载均衡、服务器故障检测与自动容错，等等；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。我们后来发现，有一个号称“世界上最快的内存数据库”MemSQL采用了和OceanBase UpdateServer类似的设计，在拥有64个CPU核心的服务器上实现了每秒150万次单行写事务。OceanBase相当于GFS + MemSQL，ChunkServer的实现类似GFS，UpdateServer的实现类似MemSQL，目标是成为可扩展的、支持每秒百万级单行事务操作的分布式数据库。

后续将分为两个章节，分别讲述分布式存储引擎层以及数据库功能层的实现细节。

# 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以及ChunkServer。其中，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，定期合并和数据分发的主要逻辑也由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。

## 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；
* 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；

### 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校验值以及基线数据版本。

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，我们可能会在2013年将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，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]。

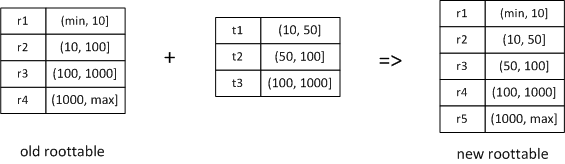


图2-1 RootTable修改

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。

### 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）以及负载均衡（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的个数以及所有tablet的大小总和，正在迁入的tablet个数、正在迁出的tablet个数以及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，步骤如下：

1. Tablet复制：扫描RootTable中的tablet，如果某个tablet的副本数小于阀值，选取某台包含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，比如，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，等等。

2. 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，以这台ChunkServer为迁移源，并选择一台符合要求的ChunkServer，生成tablet迁移任务。

Tablet复制以及负载均衡生成的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还有一个后台线程会扫描所有的ChunkServer，接着执行每台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2-1：某OceanBase集群包含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。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，选择某台ChunkServer（假设为ChunkServer3）作为迁移目的，生成迁移任务<ChunkServer1，ChunkServer3，A1>。接着，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，于是，将ChunkServer1作为迁移源，ChunkServer4作为迁移目的，选择某个tablet（假设为A2），生成迁移任务<ChunkServer1，ChunkServer4，A2>。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。最后，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。

### 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线数据完全一致，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合并相对更加麻烦，步骤如下：

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2-2： 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。RootServer扫描RootTable后发现A1和A2满足tablet合并条件，首先发起Tablet迁移，假设将A1迁移到ChunkServer2，使得A1和A2在相同的ChunkServer上，接着分别向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。

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。

### 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。

### 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/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/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。

## 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。

### 存储引擎



图2-2 UpdateServer存储引擎

如图2-2，UpdateServer存储引擎与 Bigtable存储引擎看起来很相似，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磁盘中，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磁盘中。

* **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。

图2-3 MemTable内存结构

**如图2-3所示，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。**

**MemTable的主要对外接口可以归结如下：**

|  |
| --- |
| **// 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磁盘中。**

**SSTable的详细格式请参考9.3节ChunkServer实现机制，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或者数据分发的方式转移到ChunkServer中去，以后不再需要访问UpdateServer中的SSTable数据。**

### 任务模型

任务模型包括网络框架、任务队列、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，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**



图2-4 tbnet队列模型

如图2-4，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：网络读写线程以及超时检查线程，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。

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。



图2-5 Libeasy任务模型

如图2-5，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。

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。

### 主备同步

1.4.1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，主UpdateServer操作本地后返回客户端更新成功，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。



图2-6 UpdateServer主备同步原理

如图2-6，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。备UpdateServer只要接收到日志就可以回复主UpdateServer同步成功，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。

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。

## 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&数据分发获取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）。下面从SSTable开始介绍ChunkServer的内部实现。

### SSTable



图2-7 SSTable格式

如图2-7，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。

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：块索引以及行索引；而整个ChunkServer是一个三级索引结构：tablet索引、块索引以及行索引。

SSTable分为两种格式：稀疏格式以及稠密格式。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2-3 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。



图2-8 SSTable包含多个Table/Column Group

如图2-8所示，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以及Snappy。

SSTable的操作接口分为写入和读取两个部分，其中，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&数据分发过程将产生新的SSTable，步骤如下：

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。

### 缓存实现

ChunkServer中包含三种缓存：块缓存（Block Cache），行缓存（Row Cache）以及块索引缓存（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。

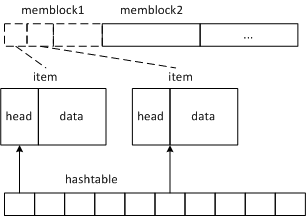


图2-8 Key-Value Cache实现

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

1. 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。

2. 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。

3. 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。

### 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2-4 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A；

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。

### 定期合并&数据分发

RootServer将UpdateServer上的版本变化信息通知ChunkServer后，ChunkServer将执行定期合并或者数据分发。

如果UpdateServer执行了大版本冻结，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。定期合并一般安排在每天凌晨业务低峰期（凌晨1:00开始）执行一次，因此也称为每日合并。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。

如果UpdateServer执行了小版本冻结，ChunkServer将执行数据分发。与定期合并不同的是，数据分发只是将UpdateServer冻结的数据缓存到ChunkServer，并不会生成新的SSTable文件。因此，数据分发对ChunkServer造成的压力不大。

数据分发由外部读取请求驱动，当请求ChunkServer上的某个tablet时，除了返回使用者需要的数据外，还会在后台生成这个tablet的数据分发任务，这个任务会获取UpdateServer中冻结的小版本数据，并缓存在ChunkServer的内存中。如果内存用完，数据分发任务将不再进行。当然，这里可以做一些改进，比如除了将UpdateServer分发的数据存放到ChunkServer的内存中，还可以存储到SSD磁盘中。

例2-5 假设某台ChunkServer上有一个tablet t1，t1的主键范围为(1, 10]，只有一行数据：rowkey=8 => (<2, update, 20>, <3, update, 30>, <4, update, 40>)。UpdateServer的冻结版本有两行更新操作：rowkey=8 => (<2, update, 30>, <3, update, 38>) 和rowkey=20 => (<4, update, 50>)。

1. 如果是大版本冻结，那么，ChunkServer上的tablet t1执行定期合并后结果为：rowkey=8 => (<2, update, 30>, <3, update, 38>, <4, update, 40>)；
2. 如果是小版本冻结，那么，ChunkServer上的tablet t1执行数据分发后的结果为：rowkey=8 => (<2, update, 20>, <3, update, 30>, <4, update, 40>, <2, update, 30>, <3, update, 38>)。

## 消除更新瓶颈

UpdateServer单点看起来像是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。

### 读写优化回顾

OceanBase UpdateServer相当于一个内存数据库，其架构设计和“世界上最快的内存数据库”MemSQL比较类似，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

2.2.2节中提到，如果不经过优化，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 + Tree。为了最大程度地发挥多核的优势，B+ Tree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 B+ Tree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如下几点：

1. Group commit。将多个写操作聚合在一起，一次性刷入磁盘中。

2. 降低日志缓冲区的锁冲突（准备做）。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。

3. 日志文件并发写入（可能会做）。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会自动触发数据分发操作，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。

### 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，每个月）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。

* ChunkServer旁路导入

OceanBase的数据按照全局有序排列，因此，旁路导入的第一步就是使用Hadoop MapReduce这样的工具将所有的数据排好序，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。接着，再将SSTable文件并行拷贝到集群中所有的ChunkServer中。最后，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，ChunkServer加载完本地的SSTable文件后会向RootServer汇报，RootServer接着将汇报的子表信息更新到RootTable中。

例2-6 有4台ChunkServer：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。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

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

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，这里不会涉及这些细节。

## 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。

### 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。

1. 如果申请的内存不超过64KB，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，需要首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。

2. 如果申请的内存超过64KB，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，再从全局内存池中申请。

全局内存池的意义如下：

1. 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。

2. 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。

### 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2-7 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，那么：

a) 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。

b) 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。

### 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（简称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（简称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。

### 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：

1. 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。

2. 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用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。

### 缓存预热

UpdateServer中的Frozen MemTable将会以SSTable的形式转储到SSD磁盘中，如果内存不够需要丢弃Frozen MemTable，大量请求只能读取SSD磁盘，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。

# 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专门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。

## 整体结构



图3-1 数据库功能层整体结构

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。

如果是只读事务，MergeServer需要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。

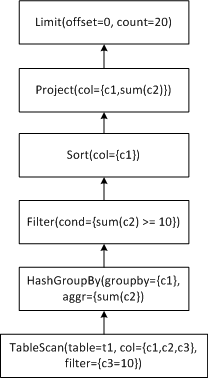
如果是读写事务，MergeServer需要首先从ChunkServer中读取需要的基线数据，接着将物理执行计划以及基线数据一起发送给UpdateServer，UpdateServer将调用UPS-SQL模块完成最终的写事务。

* 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UPS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。

## 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以及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3-1 假设有一个单表SQL语句：



|  |
| --- |
| select c1, sum(c2)  from t1  where c3=10  group by c1  having sum(c2) >= 10  order by c1  limit 0, 20 |

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。

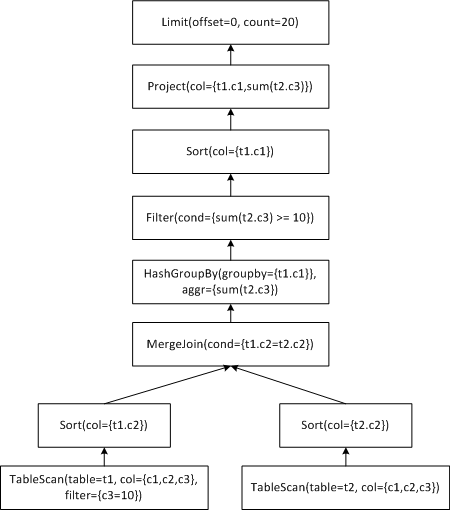
3. 调用Filter操作符，过滤分组后生成的结果，只返回上一层sum(c2) >= 10的行。

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3-2 假设有一个需要联表的SQL语句：



|  |
| --- |
| select t1.c1, sum(t2.c3)  from t1, t2  where t1.c2=t2.c2  and t1.c3=10  group by t1.c1  having sum(t2.c3) >= 10  order by t1.c1  limit 0, 20 |

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。

### 物理操作符接口

2.3.1节介绍一期分布式存储引擎中的迭代器接口为ObIterator，通过它，可以将读到的数据以cell为单位逐个迭代出来。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，以及根据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。例3-1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就能够把需要的数据依次迭代出来。

### 单表操作

单表相关的物理运算符包括：

1. 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。

2. Filter：针对每行数据，判断是否满足过滤条件。

3. 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。

4. 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。

5. 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。

6. Limit：返回行号在[offset, offset + count)范围内的行。

7. Distinct：消除某些列相同的重复行。

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好序的数据。因此，首先将内存中的数据排好序。如果数据总量不超过排序器内存上限，那么将内存中已经排好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>对排序，这样才能使得key相同的元组能够连续迭代出来。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。

### 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列上排好序，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：

例3-3假设有两张表格：item（商品表，包括商品号item\_id，商品名item\_name，分类号category\_id，），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10-4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。

|  |
| --- |
| select item\_id, item\_name  from item  where category\_id **IN**  (  select category\_id  from category  ) |

|  |
| --- |
| select item\_id, item\_name  from item, category  where  item.category\_id  = category.category\_id |

图3-4 IN子查询转化为等值连接

例3-4例3-3中，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。

|  |
| --- |
| select item\_id, item\_name  from item  where category\_id **IN**  (  select category\_id  from category  ) |

|  |
| --- |
| select item\_id, item\_name  from item  where category\_id **IN**  {1,2,3,4,5,6,7,7,9,10} |

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。

### 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。

1. 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。

2. 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。

3. 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。

4. 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。

5. 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，并将排好序的部分数据返回MergeServer，再由MergeServer执行多路归并。

6. 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。

7. Distinct：Distinct与GroupBy类似。ChunkServer先完成部分数据的去重，再由MergeServer进行整体去重。

例3-5 图10-2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，过滤第3步生成的最终结果，只返回sum(c2) >= 10的行。

4. MergeServer调用Sort操作符将结果按照c1排序。

5. MergeServer调用Project操作符，只返回c1和sum(c2)这两列数据。

6. MergeServer调用Limit操作符执行分页操作，只返回前20条数据。

当然，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。

## 写事务

写事务，包括UPDATE、INSERT、DELETE、REPLACE（插入或者更新，如果行不存在则插入新行；否则，更新已有行），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，等等，这些基线数据也会由MergeServer传给UpdateServer。

### 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，将会执行条件判断，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3-6 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。

|  |
| --- |
| // 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； |

|  |
| --- |
| // 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”； |

图3-6 单行写事务（不带条件） 图3-7 单行写事务（带条件）

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>，接着，请求主键<1，2>所在的ChunkServer，获取基线数据中行是否存在，最后，将SQL执行计划和基线数据中行是否存在一起发送给UpdateServer。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3-7中的UPDATE和DELETE语句还带有item\_name = “item1”的条件，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”。

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3-7 有两张表格item（user\_id, item\_id, item\_status, item\_name）以及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。

|  |
| --- |
| // 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”； |

图3-8 复杂写事务举例

图3-8的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。

### 多版本并发控制

2.2.1节提到，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 并发修改B+树

图3-9说明了并发修改B+树的实现原理。

a) 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。

b) 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。

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10-9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：

1. 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。

2. Get：根据主键读取一行的内容。

3. 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



图3-10 MemTable实现MVCC

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。

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。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 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 |

**图3-11 读写事务并发执行实例**

**如图3-11，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>，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。**

* 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。

1. 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。

2. 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，整个事务执行失败。

3. 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。

|  |
| --- |
| 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 |

图3-12 select…for update示例

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。

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



图3-13 备机多线程并发日志回放

3.2.3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。

## 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实现了两个主要功能：并发查询以及列式存储。并行计算功能允许将SQL请求拆分为多个子请求同时发送给多台机器并发执行，列式存储能够提高压缩率，大大降低SQL执行时读取的数据量。本节首先介绍并发查询功能，接着介绍OceanBase的列式存储引擎。

### 并发查询

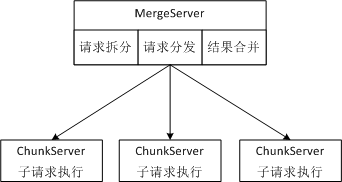


图3-14 OceanBase并发查询

如图3-14，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。

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10.2.4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。

### 列式存储

列式存储主要的目的有两个：1. 大部分OLAP查询只需要读取部分列而不是全部列数据，列式存储可以避免读取无用数据；2. 将同一列的数据在物理上存放在一起，能够极大地提高数据压缩率。

### 列式存储

列式存储主要的目的有两个：1. 大部分OLAP查询只需要读取部分列而不是全部列数据，列式存储可以避免读取无用数据；2. 将同一列的数据在物理上存放在一起，能够极大地提高数据压缩率。

* Column Group

OceanBase通过Column Group支持行列混合存储，每个Column Group存储多个经常一起访问的列。

图3-15 OceanBase Column Group设计

如图3-15，OceanBase SSTable首先按照Column Group存储，每个Column Group内部再按行存储。分为几种情况：

1. 所有列属于同一个Column Group。数据在SSTable中按行存储，OLTP应用往往配置为这种方式。

2. 每列对应一个Column Group。数据在SSTable中按列存储，这种方式在实际应用中比较少见。

3. 每个Column Group对应一行数据的部分列。数据在SSTable中按行列混合存储，OLAP应用往往配置为这种方式。

OceanBase还允许一个列属于多个Column Group，通过冗余存储这些列，能够提高访问性能。例如，某表格总共包含5列，用户经常一起访问（1，3，5）或者（1，2，3，4）列。如果将（1，3，5）和（1，2，3，4）存储到两个Column Group中，那么，大部分访问只需要读取一个Column Group，避免了多个Column Group的合并操作。

列式存储提高了数据压缩比，然而，实践过程中我们发现，由于OceanBase最初的几个版本内存操作实现得不够精细，例如数据结构设计不合理，数据在内存中膨胀很多倍，导致大查询的性能瓶颈集中在CPU，列式存储的优势完全没有发挥出来。这就告诉我们，列式存储的前提是设计好内存数据结构，把CPU操作优化好，否则，后续的工作都是无用功。为了更好地支持OLAP应用，新版的OceanBase将重新设计列式存储引擎。

## 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，然而，传统的关系数据库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，分别为大表左连接以及数据过期与批量删除。

### 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以及商品表collect\_item，其中，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。

自然想到的做法是直接采用关系数据库Join操作实现，即根据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，商品表数据量太大，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。

另外一种常见的做法是做冗余，即在collect\_info表中冗余商品的价格、人气等信息，读取时就不需要读取collect\_item表了。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的基线数据中冗余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表格的基线数据（冗余了collect\_item信息）。

2. 从UpdateServer读取collect\_info表格的修改增量，并融合到第1步的结果中。

3. 从UpdateServer读取collect\_item表格中每个收藏商品的修改增量，并融合到第2步的结果中。

4. 对第3步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表格的基线数据中，生成新的基线数据。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。

### 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。由于导入的数据量很大，一条一条删除其中的每行数据是不现实的。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。