同濟大學
TONGJI UNIVERSITY

数据库小组前沿探索
OceanBase 存储引擎研究

组　　长：2154312 郑博远

组　　员：2151531 栾佳浩

2151753 彭坤宇

2152595 张祖豪

2154318 蔡一锴

专　　业：计算机科学与技术

教　　师：李文根

完成日期：2023 年 12 月 12 日

OceanBase 数据库是由阿里巴巴集团完全自主研发的一款高性能、高可用的分布式数据库，兼顾分布式架构的扩展性与集中式架构的性能优势，用一套引擎同时支持 OLTP （联机事务处理）和 OLAP （联机分析处理）的混合负载。在可扩展性、可用性方面，OceanBase 已经连续 11 年稳定支撑双 11 海量数据的处理任务，并创新推出“三地五中心”城市级容灾新标准。同时，在性能方面，OceanBase 在被誉为“数据库世界杯”的 TPC-C 和 TPC-H 测试上都刷新过世界纪录。支撑 OceanBase 以如此强大的性能处理海量数据的，就是其背后的基于 LSM Tree 架构的数据库存储引擎。

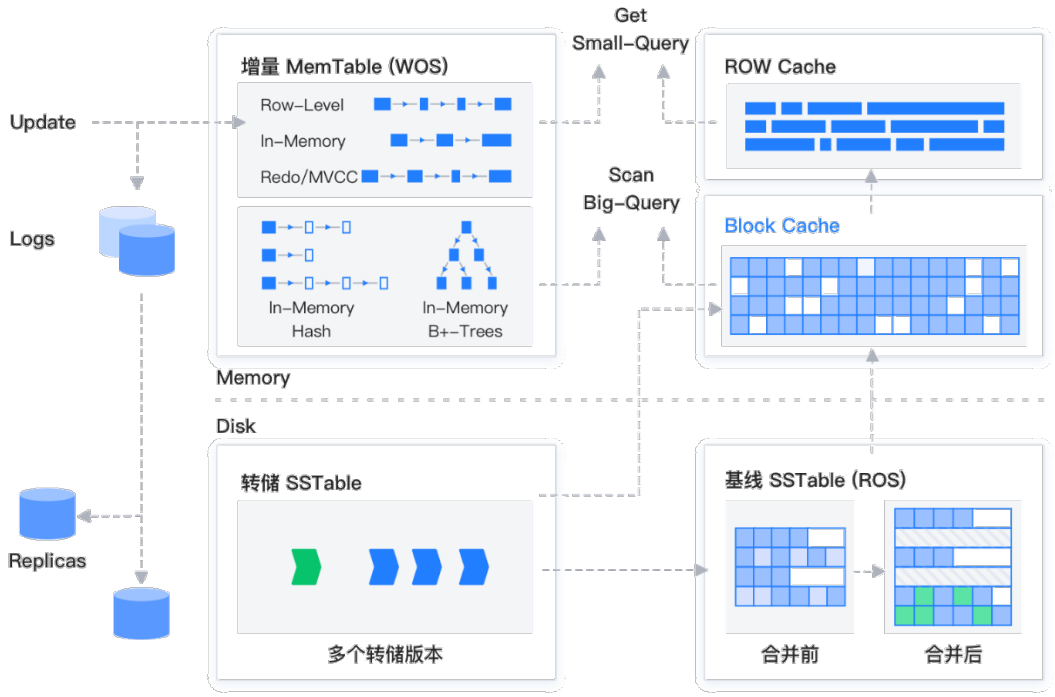

图 1　OceanBase 存储引擎总览

LSM Tree 全称是 Log Structured Merge Tree（日志结构化合并树），最早于 1996 年的《The Log-Structured Merge-Tree》一文中被提出。其核心思想充分利用了“磁盘批量的顺序写要远比随机写性能高出很多”的性质，围绕这一原理进行设计和优化，让写性能达到最优。LSM Tree 的写入全部都是追加写，不存在删除和修改。当然有得就有舍，这种结构虽然大大提升了数据的写入能力，却是以牺牲部分读取性能为代价，故此这种结构通常适合于写多读少的场景。也正因为 LSM Tree 的写入方式大大消除了磁盘上的小批量随机写操作，所以 LSM Tree 能够提供比传统的 B+树更好的写吞吐量。

想要支持更复杂和高效的读取，比如按 key 查询和按 range 查询，就得需要做一步的设计，这也是 LSM-Tree 除了利用磁盘顺序写之外，还划分了内存 MemTable 与磁盘 SSTable 多层的合并结构的原因。正是基于这种结构再加上

不同的优化实现，才造就了在这之上的各种独具特点的数据库，正如我们下文即将介绍的 OceanBase 数据库。

OceanBase 数据库的存储引擎基于 LSM Tree 架构，将数据分为静态基线数据（放在 SSTable 中）和动态增量数据（放在 MemTable 中）两部分，其中 SSTable 是不可变的，存储于磁盘；MemTable 支持读以及追加写，存储于内存。数据库 DML 操作插入、更新、删除等首先写入 MemTable，等到 MemTable 达到一定大小时转储到磁盘成为 SSTable。在进行查询时，需要分别对 SSTable 和 MemTable 进行查询，并将查询结果进行归并，返回给 SQL 层归并后的查询结果。同时在内存实现了 Block Cache 和 Row cache，来避免直接对硬盘上的基线数据进行大量随机读取。

当内存的增量数据达到一定规模的时候，会触发增量数据和基线数据的合并，把增量数据落盘。同时，由于合并操作需要占用极大量的 CPU 和 IO，因此每天凌晨的空闲时刻系统也会自动进行合并。

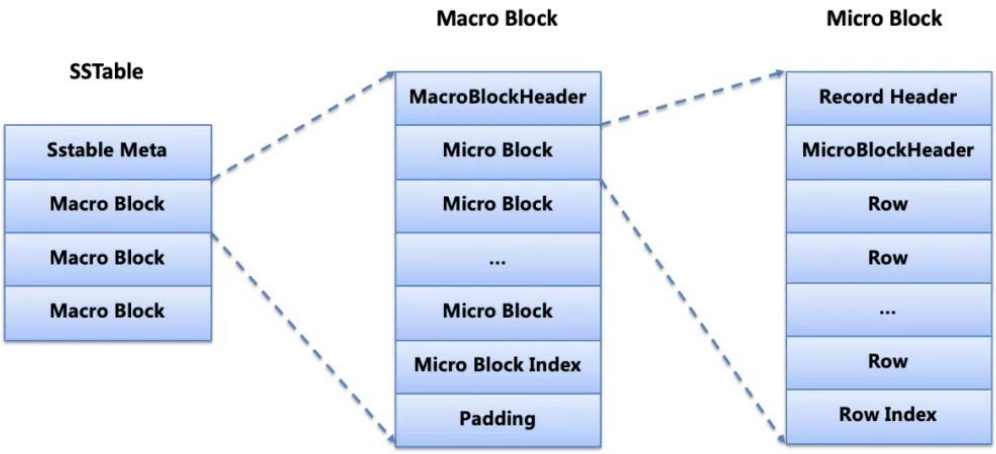

图 2 宏微块结构

OceanBase 数据库本质上是一个基线加增量的存储引擎，在保持 LSM-Tree 架构优点的同时也借鉴了部分传统关系数据库存储引擎的优点。传统数据库把数据分成很多页面，OceanBase 数据库也借鉴了传统数据库的思想，把数据文件按照 2MB 为基本粒度切分多一个个宏块，每个宏块内部继续拆分多个变长的微块。合并时数据会基于宏块的粒度进行重用，没有更新的数据宏块不会被重新打开读取，这样能够尽可能减少合并期间的写放大，相较于传统的 LSM-Tree 架构数据库显著降低合并代价。另外，OceanBase 数据库通过轮转合并的机制把正常服务和合并时间错开，使得合并操作对正常用户请求完全没有干扰。

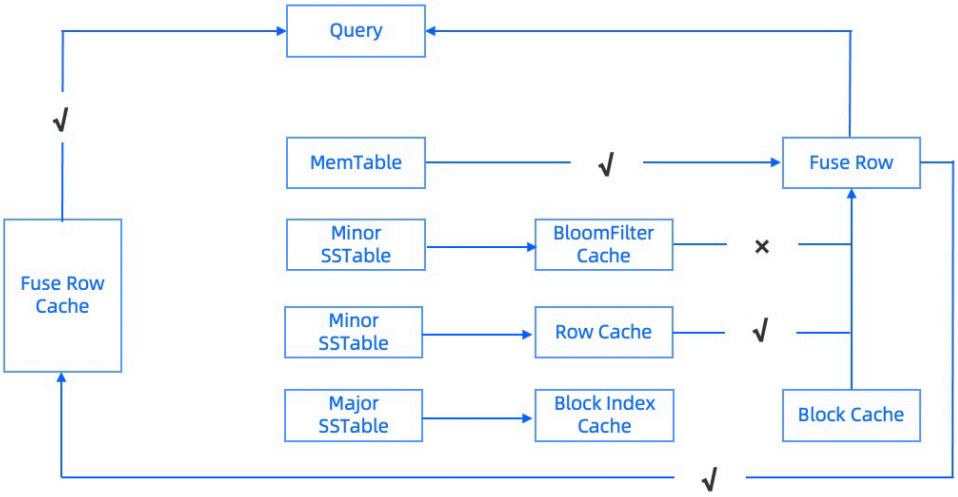

图 3　基线+增量设计

由于 OceanBase 数据库采用基线加增量的设计，一部分数据在基线，一部分在增量。每次查询原理上都既要读基线，也要读增量。为此，OceanBase 做了很多优化，尤其针对单行优化。OceanBase 的行缓存会极大加速对单行的查询性能。对于不存在行的“空查”，则会构建布隆过滤器并进行缓存。由于 OLTP 业务大部分操作为小查询，上述优化能避免传统数据库解析整个数据块的开销，达到接近内存数据库的性能。另外，由于基线是只读数据且内部采用连续存储，OceanBase 可以采用激进的压缩算法，在不影响查询性能前提下做到高压缩比。

OceanBase 结合借鉴经典数据库的优点，提供了更为通用的 LSM-tree 架构的关系型数据库存储引擎，具备低成本、易使用、高性能、高可靠的特性。从功能模块划分上，OceanBase 数据库存储引擎可以大致分为“存储架构”“转储合并”“读写查询”“数据校验”四个部分。接下来，我们将分模块进行介绍。

（一）存储架构

1.1 SSTable 静态基线数据

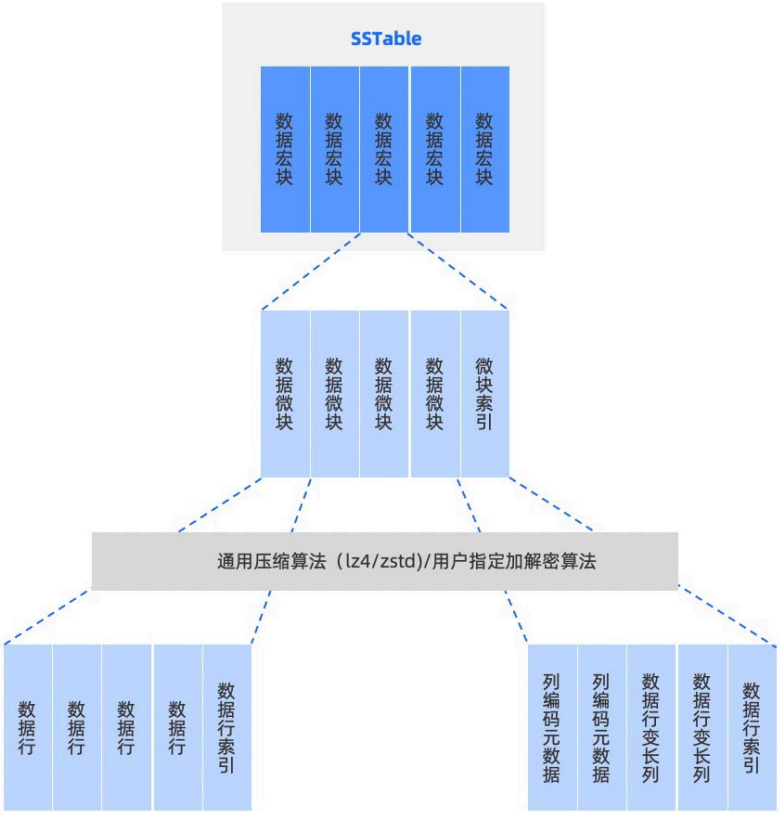

图 4 SSTable 结构

OceanBase 的磁盘存储文件称为 SSTable，其具体又被细分为多版本 SSTable 和基线 SSTable。其中，多版本 SSTable 包括 Mini SSTable 和 Minor SSTable，用于记录一段连续时间内写入的数据；基线 SSTable 则是 Major SSTable，包含某个快照点内所有完整提交数据。

为了平衡写和读的性能，SSTable 内部还按数据大小划分为宏块和微块。SSTable 以宏块（Macro Block）为单位组织数据，是大小为 2M 的定长数据块。在 OceanBase 中，宏块具有合并和分裂的能力。当由于数据删除导致相邻宏块中的所有行可以在一个宏块中容纳时，相邻的多个宏块会被合并成一个宏块。相反，当宏块内的插入和更新操作导致空间不足，需要将数据存放到多个宏块时，宏块就会进行分裂。

为了避免读一行要加载一个宏块的问题，宏块内部又划分出很多个大小为 16K（压缩前的大小）的微块（Micro Block）。如图 5 所示，宏块结构中首先存储

了包含着宏块大小、微块信息等元数据的 header，紧接着排放多个微块的具体数据，此外还存放着用于索引每一个微块的索引数据。微块是数据读取 I/O 的基本单位，为变长数据块，内部数据可以按行存储或列式编码存储。微块大小的选择影响压缩率和一次读 IO 的代价，OceanBase 默认微块大小为 16KB，适用于大多数场景。这种设计在压缩存储成本的同时，有效平衡了磁盘空间管理的问题。得益于宏块、微块的设计，在相同块大小、压缩算法与数据的情况下，OceanBase 相较于传统关系型数据库能够显著节省大量的存储空间。

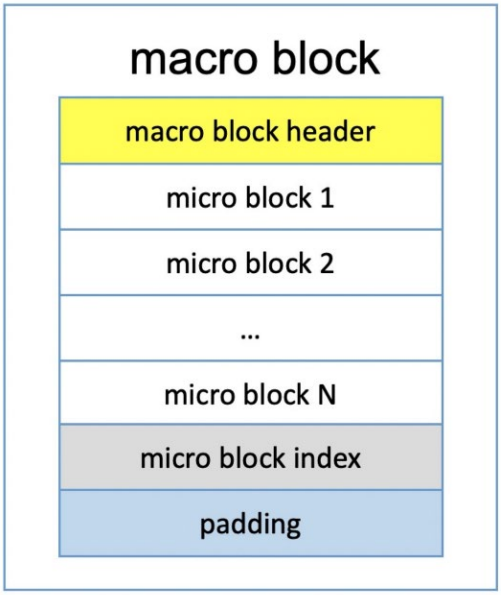

图 5 微块在宏块中的存储格式

此外，OceanBase 在微块层次上进行了数据的压缩与编码。一方面，OceanBase 支持 zlib、snappy、lz4 和 zstd 等四种通用压缩算法，能够在不了解数据内部结构的情况下直接对数据块进行压缩。另一方面，OceanBase 在通用压缩的基础上，通过自研的编码方式对数据块内部数据进行有针对性的压缩。编码基于对数据块内部数据格式和语义的理解，实现了更高效的压缩。OceanBase 数据库提供了多种按列进行压缩的编码格式，包括字典编码、游程编码、整形差值编码等，以适应不同数据特征。OceanBase 的编码还通过支持向量化执行和 filter 下压提高了过滤效率。然而，编码也带来了编解码开销与查询时解码复杂性等一些问题，但 OceanBase 通过诸如将解码器和对应的数据一起缓存在内存中等优化举措，尝试在性能、内存占用和存储成本之间达到了较好的权衡。

1.2 MemTable 动态增量数据

OceanBase 数据库的内存存储引擎 MemTable 由 BTree 和 Hashtable 组成，在插入/更新/删除数据时，数据被写入 MemTable ，在 HashTable 和 BTree 中

存储的均为指向对应数据的指针。

OceanBase 数据库中，所存放动态增量数据的内存存储引擎 MemTable 采用了由 BTree 和 HashTable 共同组成的高效结构。在对数据库中的表格执行插入、更新和删除数据的操作时，相关数据会被写入内存块，而将指向相应数据的指针存储到 HashTable 和 BTree 中。

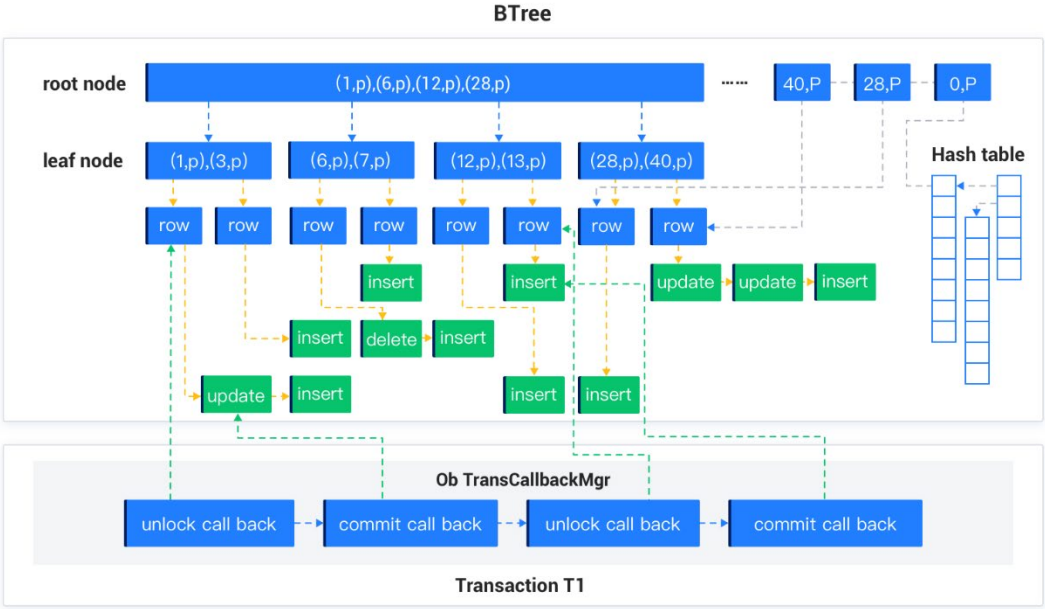

图 6　MemTable 中的数据结构

在冲突检查方面，HashTable 相较于 BTree 更为高效。HashTable 由于能够快速检查数据是否已存在，有效避免了重复插入，在数据插入时具有明显优势。尽管查询速率高，但 HashTable 也因其不适合处理范围查询操作有所不足。对于插入或更新数据等事务操作，HashTable 需要先找到相应的行（OceanBase 将行锁嵌入在行头数据结构中）并对其进行上锁，从而防止其他事务对该行的修改。

反观 BTree，由于其内部数据的有序排列性，适合进行范围查询操作。这样的特性使得 BTree 在范围查询时只需搜索局部数据，从而能够很好地提高查询效率。然而 BTree 因为需要进行大量的主键比较操作，需要从 BTree 的根节点逐步寻找到叶子结点，在单行查找方面性能较差，理论性能显著逊于 HashTable。

（二）转储合并

OceanBase 数据库的存储引擎采用 LSM-Tree 架构。数据主要分为 MemTable 和 SSTable 两部分。当 MemTable 的大小超过一定阈值时，需要将其中的数据转移到 SSTable 中，以释放内存。这个过程被称为“转储”。每次转储都会生成一个新的 SSTable。当转储次数超过一定阈值，或者在每天的业务低峰期，系统会将基线 SSTable 与后续转储的增量 SSTable 合并成一个 SSTable。这个过程被称为“合并”。换言之，转储是将 MemTable 中的数据存储到 SSTable，而合并是将不同 SSTable 合并为一个，以优化数据库性能。

表 1　OceanBase 的转储合并

转储（Minor Compaction）	合并（Major Compaction）
Partition 或者租户级别，只是 MemTable 的物化。	全局级别，产生一个全局快照。
每个 OBServer 的每个租户独立决定自己 MemTable 的冻结操作，主备分区不保持一致。	全局分区一起做 MEMTable 的冻结操作，要求主备 Partition 保持一致，在合并时会对数据进行一致性校验。
可能包含多个不同版本的数据行。	只包含快照点的版本行。
转储只与相同大版本的 Minor SSTable 合并，产生新的 Minor SSTable，所以只包含增量数据，最终被删除的行需要特殊标记。	合并会把当前大版本的 SSTable 和 MemTable 与前一个大版本的全量静态数据进行合并，产生新的全量数据。

2.1 转储

转储可以分为分层转储和非分层转储。

非分层转储为 Oceanbase 数据库使用的方式，数据库在同一时间只会维护一个转储 SSTable，当 MemTable 需要转储时，就会将 MemTable 中数据与转储 SSTable 的数据进行归并。随着转储次数不断增加，转储 SSTable 的大小也越来越大，而一次转储需要操作的数据量也越来越多，导致转储速度越来越慢。在转

储期间 MemTable 无空内存，无法进行新数据的存储，从而导致错误。

因此，Oceanbase 从 v2.2 版本开始引入了分层转储策略，结构如下：

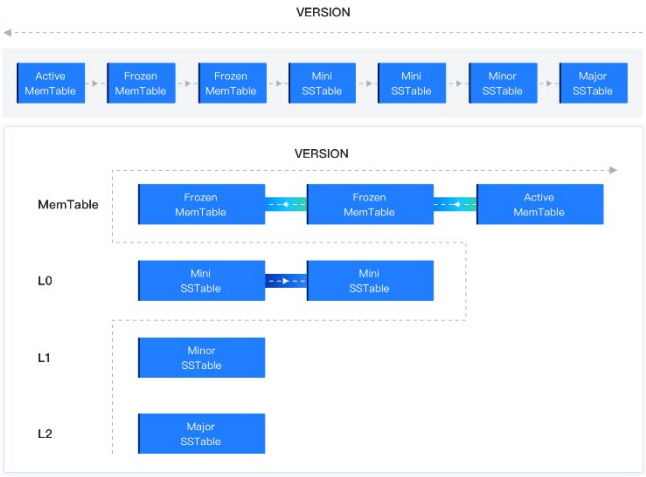

图 7　分层转储示意图

2.1.1 分层转储

• L0 层

L0 层内部称为 Mini SSTable。对于 L0 层提供 server 级配置参数来设置 L0 层内部分层数和每层最大 SSTable 个数。其中分为 level-0 到 level-n 层，当某一层 SSTable 达到上限时 compaction 为一个 SSTable 写入下一层。当 level-max 写满时做 L0 层到 L1 整体 compaction 释放空间。以下图例可以清楚解释：

图 8　L0 层到 L1 层释放

• L1 层

L1 层内部称为 Minor SSTable。每当 L0 层 Mini SSTable 达到 compaction

阈值后，L1 层 Minor SSTable 开始参与和 L0 层的 compaction。OceanBase 数据库内部提供写放大系数设置，来提升 compaction 效率。

• L2 层

L2 层内部称为基线 Major SSTable。为保持多副本间基线数据完全一致，日常转储过程中 Major SSTable 仍保持只读，不发生实际 compaction 动作。

2.1.2 转储触发

转储有两种触发方式：自动触发与手动触发。当一个租户的 MemTable 内存的使用量达到 memstore_limit_percentage * freeze_trigger_percentage 所限制使用的值时，就会自动触发冻结（转储的前置动作），然后系统内部再调度转储。也通过运维命令手动触发转储。

2.2 合并

与数据转储相比，合并通常是一项耗时较长、较为繁重的操作。在最佳实践中，通常期望每天只执行一次合并操作，并将其安排在业务低峰期进行。因此，有时候将合并操作称为每日合并。

合并操作涉及对动态数据和静态数据进行合并，这一过程相对费时。当由数据转储生成的增量数据积累到一定程度时，通过 Major Freeze 来执行大版本的合并。与数据转储的主要区别在于，合并是对租户上所有分区在一个统一的快照点和全局静态数据进行合并的全局操作，最终形成一个全局快照。

2.2.1 合并方式

• 全量合并

将静态数据全部读出并和动态数据合并为最终的静态数据。合并时间长，耗费 IO 和 CPU。

• 增量合并

在 OceanBase 数据库的存储引擎中，宏块是基本的 IO 写入单位。并非所有宏块都会被修改，当一个宏块没有增量修改时，可以直接重用这个数据宏块，这被称为增量合并。增量合并大大减少了合并的工作量，是 OceanBase 数据库目前默认的合并算法。

• 渐进合并

某些 DDL 操作（如加减列）需将所有数据重写一遍，渐进合并将数据重写分散到多次每日合并中，减轻了 DBA 做 DDL 操作的负担，使 DDL 更加平滑。

• 并行合并

OceanBase 数据库 v1.0 中增加了对分区表的支持，对于不同的数据分区，合并可以并行进行。为了提高合并速度，特别是在数据倾斜的情况下，引入了分区内并行合并，将数据划分到不同线程中并行进行合并。

2.2.2 合并触发

合并触发有三种触发方式：自动触发、定时触发与手动触发。

• 自动触发

自动触发合并的条件是，当一个租户的 MemTable 的内存使用达到了 freeze_trigger_percentage 设置的阈值，且转储次数已达到了 major_compact_trigger 设置的上限时，此时不会进行转储，而是直接进行合并。

• 定时触发

开启全局合并开关后，系统时间达到了每日合并的时间点时，就会自动触发合并。每日合并的时间点通过配置项 major_freeze_duty_time 来控制，此外也可以在 OCP 上修改，默认是每天 02:00 进行合并。

• 手动触发

通过运维命令手动触发合并。

（三）读写查询

3.1 插入

3.1.1 聚簇索引

在 OceanBase 数据库中，所有的数据表都可以视为索引聚簇表，聚簇索引的叶子节点直接存储用户信息的内存地址，使用内存地址可以直接找到相应的行数据。作为比较，非聚簇索引的叶子节点上存储的并不是真正的行数据，而是主键 ID，所以当使用非聚簇索引进行查询时，首先会得到一个主键 ID，然后再使用主键 ID 去聚簇索引上找到真正的行数据，这个过程称为回表查询。

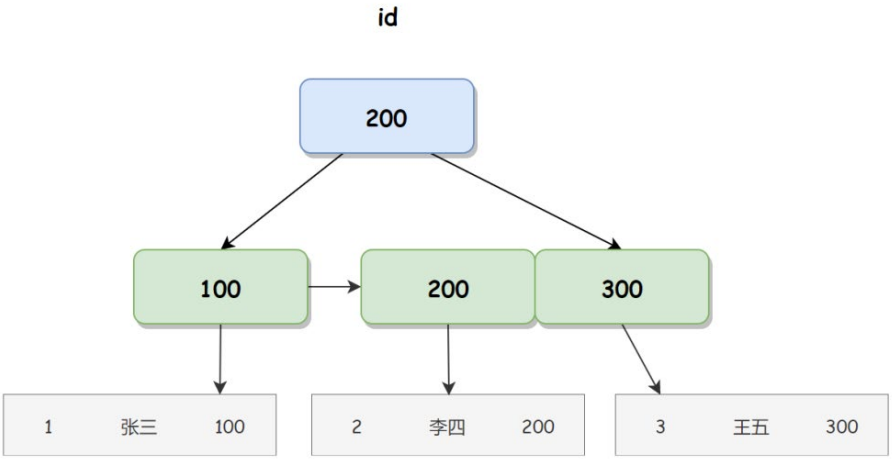

图 9　聚簇索引 id 对应的 B+树

以上面 student 表为例，在 student 中非聚簇索引 class_id 对应 B+ 树：

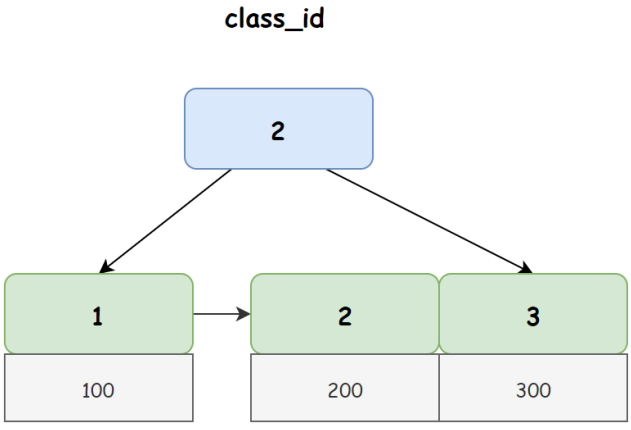

图 10　非聚簇索引 id 对应的 B+树

由于非聚簇索引需要进行回表查询，因此性能不如聚簇索引。

3.1.2 维护隐藏主键

对于无主键堆表，OceanBase 在内部会为其维护一个隐藏主键。因此当用户插入数据时，在向 MemTable 中写入新的用户数据前，需要先判断当前数据表中是否已经存在相同数据主键数据，为了加速这个重复主键查询性能，对于每个 SSTable 会由后台线程针对不同宏块判重频率来异步调度构建 Bloomfilter。

3.2 更新

作为LSM-Tree数据库，OceanBase数据库的每次更新都会插入一行新数据。与 Clog 不同，在 MemTable 中更新写入的数据只包含更新列的新值和对应的主键列，即更新行不一定包含表的全部列数据。通过不断进行后台 Compaction 动作，这些增量更新会被融合在一起，从而加速用户查询。

3.3 删除

和更新类似，删除操作并不直接作用在原数据上，而是通过使用删除行的主键写入一行数据，利用行头标记来标明删除动作。然而，大量的删除操作对于 LSM-Tree 数据库并不友好，因为这可能导致即使一个数据范围被完全删除，数据库仍需迭代该范围内的所有删除标记行，然后完成融合才能确认删除状态。为了规避这种情况，OceanBase 数据库提供了内在的范围删除标记逻辑。此外，数据库还支持用户明确指定表模式，从而通过特殊的转储合并方式提前回收这些删除行，以加速查询。

由于 OceanBase 支持二级分区机制，对于过期数据可以一级按用户分区，二级按时间分区。因此，如果要删除过期数据，可以通过 drop 分区的方式实现，从而有效解决了大用户扩展性的问题。

3.3 查询

由于增量更新的策略，查询每一行数据的时候需要根据版本从新到旧遍历所有的 MemTable 以及 SSTable，需要将 MemTable 和 SSTable 的数据进行归并，才能得到最终的查询结果。

3.3.1 查询流程

OceanBase 数据库目前使用拉取模式的 table_scan 迭代流程，流程如下：

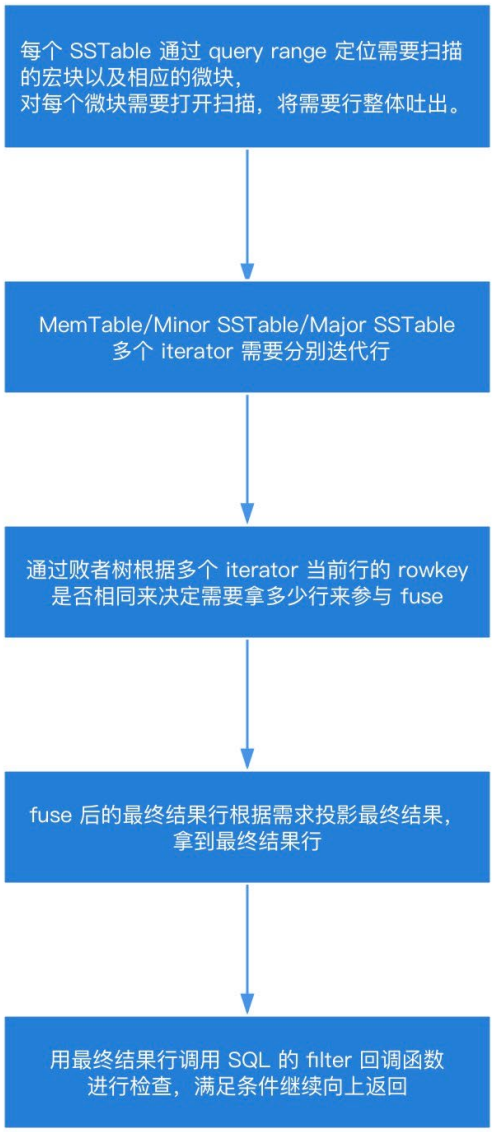

图 11　OceanBase 数据库查询流程

3.3.2 查询下压

对于 LSM-Tree 架构的存储引擎，在查询的流程一般都会遇到一系列问题，需要完成以下优化：

1. 减少迭代和主键比较的频率： 在较多 OLAP 场景中，由于大部分表很少更新，数据主要存在于主要 SSTable 中，因此应该尽量减少迭代和主键比较的频率。优化思路是直接从主要 SSTable 进行快速扫描，仅在主键可能存在交

集时进行融合。当前实现中逐行检查主键是否相等的方式效率较低，同时也限制了向量化扩展的可能性。

2. 优化过滤算子计算时间： 目前过滤算子的计算时间较晚，为了保证最终结果的正确性，所有行必须经过融合后才能确保数据是最新状态。这导致过滤算子的检查被放在每行迭代的最后进行，可能影响性能。优化的方向是尽早进行过滤，减少不必要的迭代。

3. 减少无用的投影： 从微块扫描吐出的行需要包含最终用户所需的所有列，但在多次迭代和融合之后，经过过滤条件不符合的行可能会浪费额外的投影列。优化方向是在不符合过滤条件的情况下，减少除过滤条件列外的其他投影列，以降低资源浪费。

OceanBase 数据库通过实现过滤算子下压到存储层来完成优化，如下图所示：

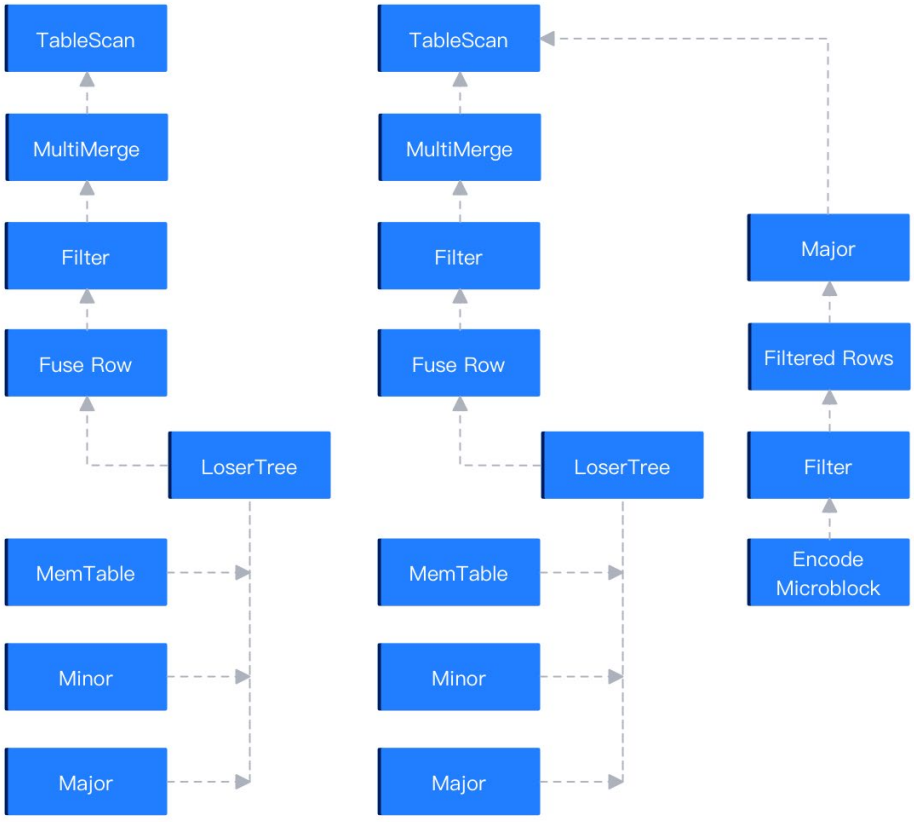

图 12 查询下压优化

1. 算子下压：OceanBase 数据库具备快速区分数据无交集状态的能力，即能够感知 major SSTable 和增量数据无交集的数据区间。由于该区间的数据只需访问 major SSTable 即可获取最新状态，因此可以直接将过滤算子下压至这段数据。

2. 算子过滤：对于每个下压的过滤表达式，OceanBase 数据库进行解析并拆分成存储层能够理解的表达式树。这个树结构包含了对应列信息以及相应的过滤条件表达式。

3.4 多级缓存

OceanBase 数据库引入了多级缓存系统以提升新能，包括数据微块的 Block Cache、每个 SSTable 的 Row Cache、查询融合结果的 Fuse Row Cache 以及插入判空检查的 Bloomfilter Cache 等。这一系统采用共享内存机制，同一个租户下的所有缓存共享内存，使得不同缓存对象能够相互挤占内存以满足当前需求。由于数据库存储引擎基于 LSM-Tree 架构，修改只写入 MemTable，而 SSTable 是只读的，因此 Cache 是只读的，无需处理刷脏页。然而，由于 SSTable 内的数据进行了数据编码和压缩，需要处理变长数据，使得 Cache 的内存管理更为复杂。同时，作为分布式多租户数据库系统，OceanBase 通过 Cache 框架实现了不同租户的不同类型的 Cache 的统一管理，根据优先级和数据访问热度进行相互挤占，实现了内存隔离。这些特性共同为 OceanBase 数据库的性能提供了更好的支持。

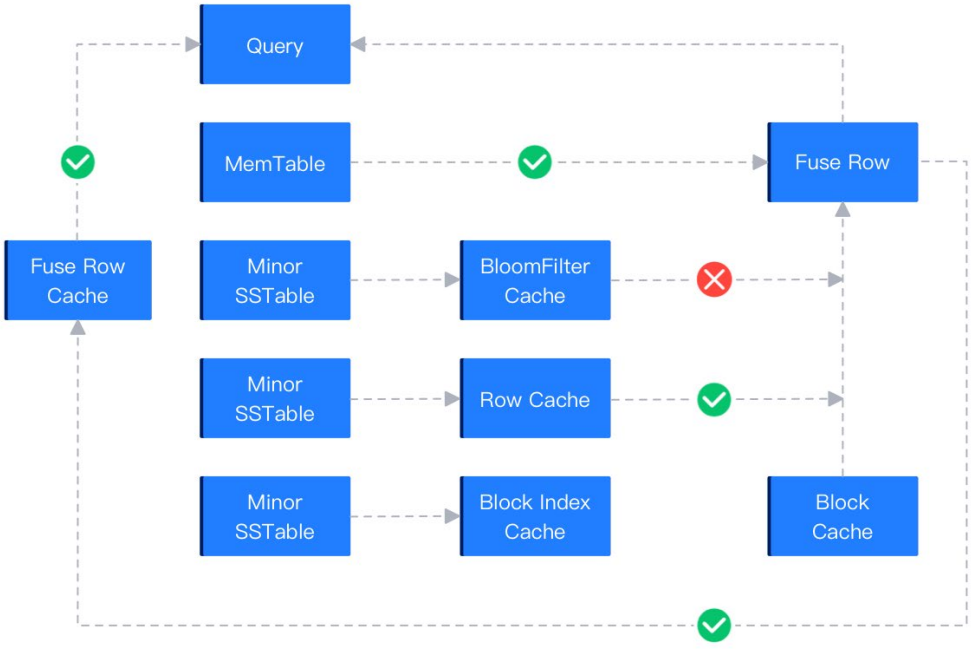

图 13 多级缓存示意图

为了能够更加通用的支持多种类的 Cache，需要处理变长数据的问题，OceanBase 数据库的底层 Cache 框架将内存划分为多个 2MB 大小的内存块，对内存的申请和释放都以 2MB 为单位进行。变长数据被简单 pack 在 2MB 大小的内存块内。为了支持对数据的快速定位，在 Hashmap 中存储了指向对应数

据的指针，整体结构如下图所示。

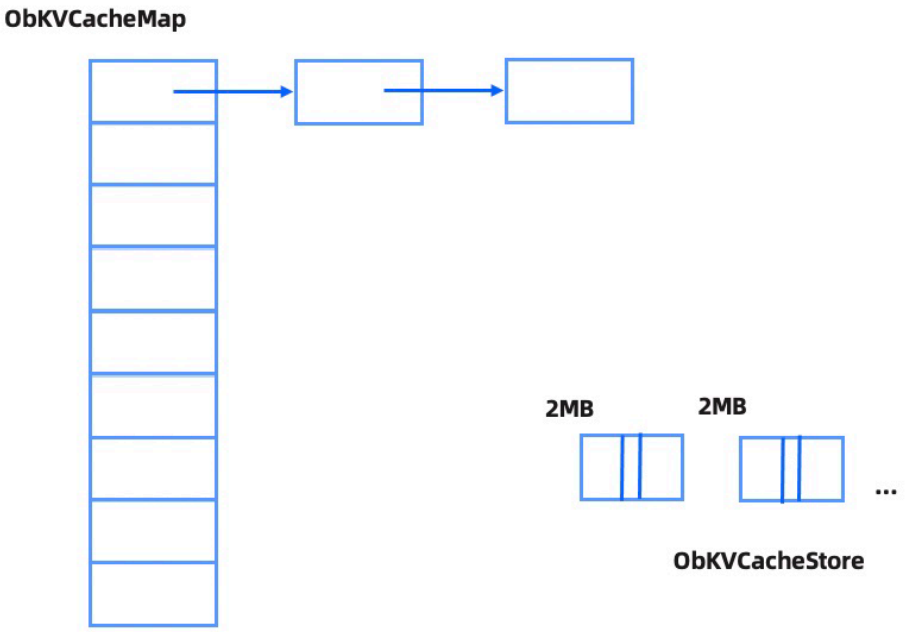

图 14 变长数据处理

OceanBase 数据库的 Cache 内存以 2MB 为单位整体淘汰，根据每个内存块的访问热度计算分值，访问越频繁的内存块分值越高。通过后台线程定期对所有内存块的分值排序，淘汰分值较低的块。对于整体分值不高但内部存在热点数据的内存块，会将热点数据从“冷块”移动到“热块”，避免淘汰热点数据。OceanBase 数据库的 Cache 在数据结构上不维护类似于 Oracle、MySQL 的 LRU 链表，因此对于读取数据时对 Cach 访问几乎无锁，对于热点数据的高并发访问更加友好。

（四）数据校验

在 OceanBase 数据库中，备份数据的校验主要是对数据的有效性和完整性进行校验。数据的有效性主要是指数据文件是否有 bit 翻转等情况，可以通过计算 Checksum 校验和来判断数据的有效性。备份数据的完整性主要是指数据文件是否有丢失。OceanBase 始终将数据质量和安全放在第一位，对于全数据链路每一个涉及持久化的数据部分都会增加数据校验保护，同时利用多副本存储的内在优势，还会 增加副本间的数据校验进一步验证整体数据一致。

Oceanbase 主要支持以下几种场景的数据校验：

1. 本集群内备份数据的校验；
2. 本集群内指定路径的备份数据的校验或本集群内指定路径备份的备份数据的校验；
3. 跨集群指定路径的备份数据的校验或跨集群指定路径备份的备份数据的校验。

4.1 校验方式

OceanBase 数据库的校验方式分为两种：逻辑校验和物理校验。

4.1.1 逻辑校验

在常见部署模式下，OceanBase 数据库的每个用户表都会存在多副本，在租户每日合并时，所有的副本都会基于全局统一的快照版本生成一致的基线数据，利用这个特性，所有副本的数据会在合并完成时比对数据的校验和，保证完全一致。更进一步，基于用户表的索引，还会继续比对索引列的校验和，确保最后返回用户的数据不会因为程序内在问题出错。

4.1.2 物理校验

针对数据存储，OceanBase 数据库从数据存储最小 I/O 粒度微块开始，在每个微块/宏块 /SSTable/ 分区上都记录了相应的校验和，每次数据读取时都会进行数据校验；为了防止底 OceanBase 数据层存储硬件问题，在转储合并写入数据宏块时也会在写入后马上重新对数据进行校验；最后每个 Server 后台会有定期的数据巡检线程对整体数据扫描校验，以提前发现磁盘静默错误。

4.2 校验机制

4.2.1 多副本机制

OceanBase 作为分布式数据库，采用了多副本的容灾方式。由于磁盘静默错误发生的概率并不高，所以同一个数据块在多个副本同时出现静默错误的概率微乎其微。只要能知道某个副本出现了磁盘静默错误，就可以从剩余的正常副本中拷贝数据来修复这个错误。目前 OceanBase 数据库支持副本粒度的修复，出现磁盘静默错误时，可以先删除错误副本，再从其他机器补齐正确副本。

OceanBase 分布式数据库更多的是在软件层面引入保障机制，OceanBase 充分利用了 Paxos 协议，并将 Paxos 协议和传统的 WAL 机制结合起来，每一次 Redo Log 落盘时，都会以强一致方式同步到 Paxos 组中多数派（leader+若干 follower）副本的磁盘中，这样做有两个好处：首先， 在 Paxos 组中任意少数派副本发生故障的情况下，剩下的多数派副本都能保证有最新的 Redo Log，因此就能避免个别硬件故障带来的数据损失，保证 RPO=0；其次，Paxos 协议中的数据强一致是针对“多数派”副本而言，如果 Paxos 组中有少数派 follower 副本发生故障，剩下的多数派副本（leader+若干 follower）之间的数据强一致完全不受影响，这就解决了主从热备模式下备副本故障拖累主副本的可用性。

综上，OceanBase 利用 Paxos 协议可以保证 RPO=0，且不会影响系统性能，这也是 OceanBase 和传统数据库在数据可靠性方面最显著的不同点。

4.2.2 RedoLog 的校验机制

在每条 RedoLog 的头部，OceanBase 会记录这条日志的校验和。在做网络传输和日志回放时，都会强制对每条日志的校验和进行校验。这样 OceanBase 保证了三副本同步到的日志是正确且一致的，如果一条日志中的数据出现了静默错误，那么这条日志一定不会被同步到其他副本。

4.2.3 SSTable 的校验机制

SSTable 的数据存放在一个个宏块中，宏块的长度固定为 2MB，在宏块的头部会记录这个宏块的校验和。宏块内部会拆分多个微块，微块长度不固定，通常为 16KB，在微块的头部也会记录这个微块的校验和。SSTable 的读 IO 以微块为基本单位，写 IO 以宏块为基本单位。在读取微块时，会强制校验微块头的校验和，保证用户读到的微块数据是正确的。在迁移、备份等复制宏块的场景，目的端写宏块前，也会强制校验宏块的校验和，保证写入的数据是正确的，防止

磁盘静默错误的扩散。除了在读写时检查数据块的正确性，OceanBase 还希望尽早发现磁盘静默错误。OceanBase 数据库可以在后台开启巡检任务，周期性扫描全部宏块并检查其校验和，一旦发现磁盘静默错误便会告警。

4.2.4 冷备机制

在没有多副本的部署场景，或者是在多个副本同时发生了磁盘静默错误的情况下， OceanBase 数据库针对这些场景提供了备份恢复的功能，可以将数据备份到 NFS、OSS 等外部介质。因此发生磁盘静默错误后，OceanBase 可以从外部介质将正确的数据再恢复到数据库中。

4.2.5 副本间的检查点一致性校验

OceanBase 会在特定的检查点（即每日合并时）对多个副本之间的数据盘做一致性检查。主要的原因是每日合并动作本身就要对大量数据做归并和重新写入，刚好可以利用这个时机做数据的一致性检查。通过一致性检查，OceanBase 能够进一步在存储层确保了多个副本之间的数据一致性，从而提高数据可靠性。

4.2.6 数据表和索引表之间的数据一致性校验

对于有关联关系的数据对象，OceanBase 会做额外的检查以保证它们之间的数据一致性。例如对于索引和它的数据表，OceanBase 会在一些特定的检查点（如每日合并点）做索引和数据表之间的一致性检查，进一步提高数据可靠性。

与传统数据库一样，OceanBase 提供了完善的备份/恢复机制，包括全量备份功能和增量备份功能。此外，OceanBase 的增量备份是以不间断的后台 daemon 任务形式持续进行，完全不影响在线业务，从而降低了运维操作的复杂度。不过从分布式数据库的运行实践来看，在实际系统中极少发生 Paxos 组中多数派副本同时毁坏的情况，因此基本不会真正用到备份来恢复数据。

（五）小组分工

栾佳浩（2151531）：数据校验部分探究撰写

彭坤宇（2151753）：转储合并部分探究撰写，课堂汇报展示

张祖豪（2152595）：读写查询部分探究撰写

郑博远（2154312）：存储架构部分探究撰写，报告排版

蔡一锴（2154318）：文稿统筹总结，演示文稿制作