产品名称Product name	密级Confidentiality level
产品版本Product version	
V 3.0	

移动数据库
TMDB 详细设计

Prepared by 拟制		Date 日期	2023-01-07
Reviewed by 评审人		Date 日期	
Approved by 批准		Date 日期	

Revision Record 修订记录

Date 日期	Revision Version 修订 版本	Sec No. 修改 章节	Change Description 修改描述	Author 作者
2023.1.13	2.0	2	使用LSM-Tree实现TMDB的存储	王征权
2023.1.13	2.0	3	使用修改的Jsqlparser实现语法解析	袁来
2023.1.13	2.0	4	使用LSM-Tree存储格式设计日志并实现日志功能	王海月
2023.3.13	3.0	2	根据代码实现方案修改文档	王征权

目录

1 概述

1.1 TMDB 整体架构

TMDB（全称 totem mobile database）是一款针对移动端开发的支持对象代理模型的数据库，如图 1.1 为 TMDB 的整体体系结构，在此详细设计文档中的 2-4 章将分别具体介绍其中的存储管理、编译查询、日志管理模块。负责接收用户的读写请求的是 Transaction，这部分将不同请求类型分发给不同模块进行处理，同时协调各模块、处理业务逻辑。

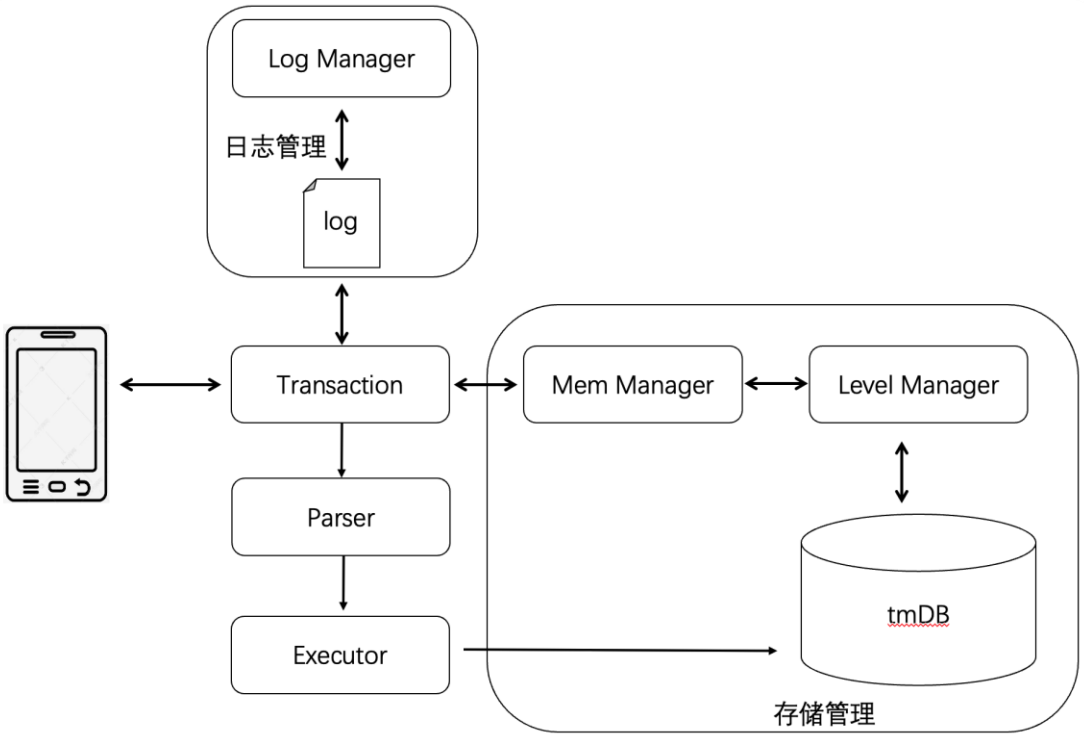


图 1.1 TMDB 整体架构图

1.2 存储管理概述

TMDB 是键值数据库，同时支持对象代理模型。数据分为系统表和数据表，系统表通过表 1.1-表 1.5 五种方式存放，数据表通过转化为键值对的形式存放于 LSM-Tree 中。

表 1.1 Class **系统表**

classname	类名
classid	类 id
attrnum	类属性个数
attrid	属性 id
attrname	属性名
attrtype	属性类型
classtype	类类型

表 1.2 Deputy 表

originid	类 id
deputyid	代理类 id
deputyrule	代理规则

表 1.3 Object 表

classid	类 id
tupleid	元组 id
sstsuffix	位于 SSTable 的后缀

表 1.4 Swiching 表

attr	源属性 id
depuattr	代理属性
rule	Swich 规则

表 1.5 BiPointer 表

classid	源类 id
objectid	源对象号
deputyclassid	代理类 id
deputyobjectid	代理对象号

有关于 LSM-Tree 的组织形式将在第二章展开介绍，在 2.1-2.5 小节中将会具体介绍有关 LSM-Tree 的基本知识及其组件，在 2.6 小节将会具体介绍存储层的整体结构以及插入数据、查询数据的相关流程。

1.3 编译查询概述

TOTEM 数据库采用了一个与 SQL 语言兼容的对象代理数据库语言 TOTEM-SQL[6]作为其工作语言。用户的各种 TOTEM-SQL 命令的执行是由 TOTEM 的查询系统完成的。TOTEM 移动端的查询系统采用了与其他数据库系统相似的处理流程。系统在接收了查询服务命令后，首先会在分析器中编译，形成查询树后，再经过重写和优化，进入执行器，由存储系统完成磁盘数据的获取，返回查询结果。

对于普通的查询（指不涉及到代理类或代理对象的查询），TOTEM 的查询系统沿用了一般数据库的实现方法。但在处理涉及代理类查询时，TOTEM 建立了一套完善的查询机制，使代理类的查询能采用与传统数据库一致的形式描述和实现。

查询执行操作对应语句的特点是需要查询满足条件的（代理）对象，然后返回给用户或者在（代理）对象上进行某些操作后写回到磁盘上。对象上的操作包括增、删、查、改。

1.4 日志管理概述

TMDB 采用 WAL(Write Ahead Log)预写日志方法，即对数据文件的修改必须只能发生在这些修改已经记录到日志之后，也就是说，在描述这些变化的日志记录刷写到存储器之后 ，保证数据操作的原子性和持久性。日志文件保存 redo 信息。采用这种方法，当系统“崩溃”需要恢复时，在重新启动时，程序可能需要知道当时执行的操作是成功了还是部分成功或者

是失败了。使用了 WAL，程序就可以检查日志文件，并对计划执行的操作内容跟实际上执行的操作内容进行比较，根据日志信息进行 redo 操作进行恢复。由于采用非重写技术，在不需要 undo 功能的情况下，仍能保证数据库的一致性。

2 存储管理

2.1 LSM-Tree 介绍

LSM-Tree[1]的英文全称为 Log-structured Merge Tree，即日志结构合并树，这是一种分层、有序、面向磁盘的 key-value 存储结构，其结构如图 2.1 所示。从其名称出发来理解 LSM-Tree，一方面，它是一种基于日志的结构，即所有的操作都要首先写入日志(称为 WAL: write ahead log)，这样方便进行数据恢复和异常处理；另一方面，在 LSM-Tree 中涉及大量合并(merge，也称 compaction)操作，数据首先写入内存，如图 2.1 中的 C0 所示，随后通过不断的合并过程，在磁盘上从小文件 C1 不断合并到大文件 CL 中。

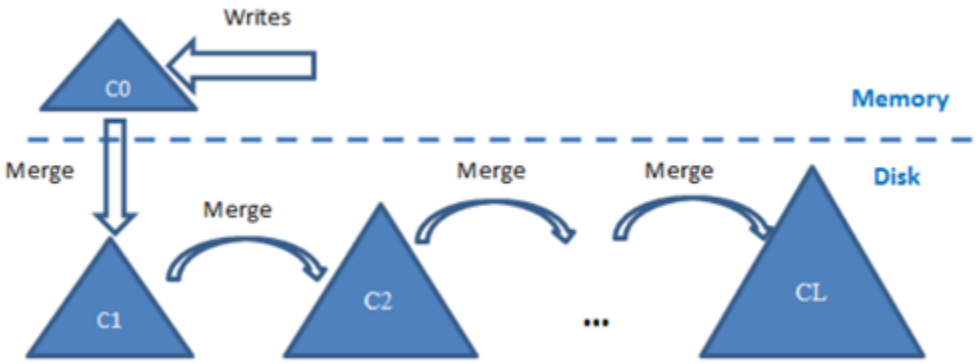


图 2.1 LSM-Tree 整体图

与传统数据库的就地更新策略(in-place update，即先找到对应数据存储的位置，然后就地修改)不同，LSM-Tree 采用一种追加更新(out-place update)的方式[2]，即所有修改操作，都首先以写数据的形式写到内存中，再从内存中顺序刷新(flush)到磁盘。

LSM-Tree 的具体结构如图 2. 2 所示。

在内存中，LSM-Tree 的数据保存在 MemTable 中，MemTable 又分为 immutable MemTable 和 mutable MemTable，前者不可修改，后者可修改。数据写入时，会写入 mutable MemTable 中，当 mutable MemTable 写满时会转化为 immutable MemTable，再写入数据需要等待空闲的 mutable MemTable。每个 immutable MemTable 都会等待合适的时机，进行 compaction 操作合并到外存中，随后转化为 mutable MemTable 再次变得可写入。

在外存中，LSM-Tree 的数据保存在 SSTable (Sorted-String Table)中，每个 SSTable 中的数据按照 key 排序。这些 SSTable 分层存储，从 level-0 到 level-n，对应 SSTable 的数量越来越多，大小越来越大。从内存中的 immutable MemTable 进行 compaction 时首先转换为 level-0 中的 SSTable，当 level-i 的大小达到上限时，会进行 compaction 到 level-(i+1)，经过一系列 compaction 最终到达 level-n。

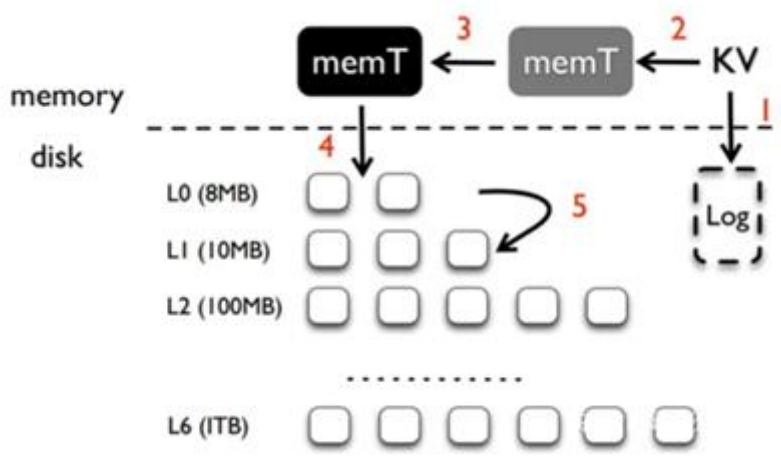


图 2.2 LSM-Tree 层级图

这样的设计带来了很多优点，比如：

• 卓越的写性能：out-place update 不需要首先找到旧数据的实际存储位置，直接写入一条新数据，充分利用磁盘顺序写比随机写效率高；

• 高空间利用率：按 level 组织数据，并通过 compaction 操作将新数据从上层传播到底层，这些 compaction 操作过滤掉旧版本数据，减少冗余数据大小，从而减少所需的存储空间，提高空间利用率；

• 版本控制的简化：不同的 level 提供了天然的版本控制信息，假设 level-i 中有数据 k1-v1， level-(i+1)中有相同 key 的数据 k1-v2，通过层级信息就能判断 level-1 中的数据是最新版本的数据；

• 简易的数据恢复：由于 WAL 的存在，在遇到异常情况时可以通过 WAL 快速恢复内存中的数据。

当然，LSM-Tree 这样的设计在获得高效的写性能时，也牺牲了其他方面的性能：

• 数据的最新程度 mutable Memtable > immutable Table > Level-0 > Level-1 > Level-2 >…，因此在查询数据时，需要先扫描 MemTable，如果没有命中，则需要从 level-0 开始一直扫描到 level-6 直到命中或者扫描完所有 SSTable；

• LSM-Tree 的合并操作(compaction)将若干个 SSTable 的数据读到内存中，进行更新、去重、排序等一系列操作后重新写回 SSTable，涉及到数据的解码、编码、比较、合并,是一个计算密集型的操作。一方面，合并操作在整理数据的过程中会移动数据的位置，导致数据对应的缓存失效，造成了读性能的抖动问题。另一方面，合并操作在整理 LSM-Tree 数据的同时造成了写放大,严重影响了 LSM-Tree 的写性能。

2.2 SSTable 结构

SSTable (Sorted String Table) 是排序字符串表的简称，它是一个种高效的 key-value 型文件存储格式，其结构如图 2.3 所示。

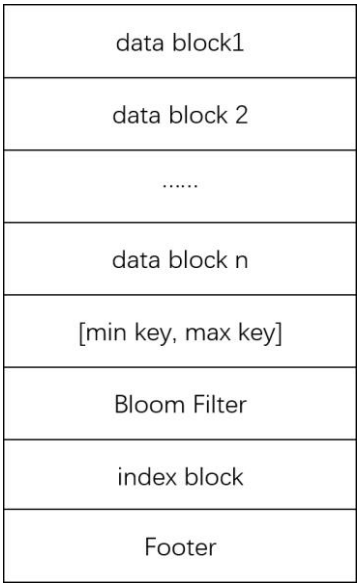


图 2.3 SSTable 结构图

首先在 SSTable 尾部有个 Footer，它给出整个 SSTable 中其他各区域的物理偏移和长度，也是整个 SSTable 的访问入口。数据在 SSTable 中分为不同 data block，每个 data block 为 4KB。由于数据在 SSTable 中按照 key 排序，因此不同 data block 之间也严格按照 key 排序，并把每个 data block 的最大 key 连同该 data block 的开始偏移保存到 index block 中，方便快速定位。同时，为了优化读性能，我们还额外保存最大 key、最小 key，和一个 Bloom Filter (将在 2.4 小节进行详细介绍)。

结合以上对 SSTable 结构的设计，查询 SSTable 中指定 key 的过程如下：

①首先访问 Footer，获得其他组件的偏移及长度；

②访问最小 key 和最大 key，判断要查询的数据在不在这个范围内，若不在则无须继续访问此 SSTable；

③访问 Bloom Filter，输入 key 值返回 true/false，判断对应的数据在不在此 SSTable 中；

④访问 index block，其中记录的每个数据块的最大 key、偏移、长度，可以根据这些信息定位到目标 key 可能存在的数据块，随后遍历访问该数据块寻找目标 key。

2.3 B 树实现索引

在 TMDB 中，我们使用 B-Tree 作为索引。

B-Tree 是一种自平衡的树，能够保持数据有序。这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成，为系统大块数据的读写操作做了优化。B-Tree 通过减少定位记录时所经历的中间过程，从而加快存取速度。

一个 m 阶 B-Tree 有以下几个特征：

- 每个节点最多只有 m 个子节点；
- 每个非叶子节点（除了根）具有至少⌈m/2⌉子节点；
- 如果根不是叶节点，则根至少有两个子节点；
- 具有 k 个子节点的非叶节点包含 k -1 个键；
- 所有叶子都出现在同一水平，没有任何信息（高度一致）。

如图 2.4 所示为一个 4 阶 B-Tree。

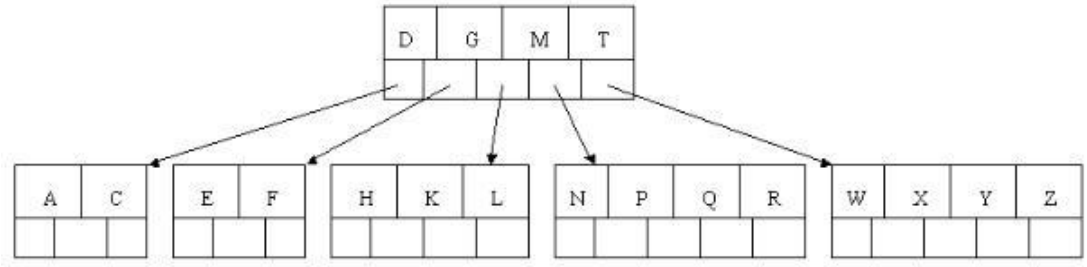


图 2.4 一个 4 阶 B-Tree 例图

在 SSTable 的结构中，数据以 4KB 的大小分为不同 data block，在将 MemTable 持久化为 SSTable 时，先将每个 data block 的最大 key 以及该 data block 在 SSTable 中的偏移量插入 B-Tree 中，得到一个能够记录此 SSTable 索引信息的 B-Tree，如图 2.5 所示，图中每结点的第一行记录的是 data block 的最大 key，第二行记录的是对应的偏移量，第三行记录的是该结点的孩子结点指针。在此例中 k1-k19 是递增的。在查询时，假设我们的目标 key 在 k7-k8 之间，那么我们首先搜索 B-Tree 定位到 k7 和 k8，同时确定了目标数据存在于 data block 7 中，通过偏移量即可进一步访问该 data block。

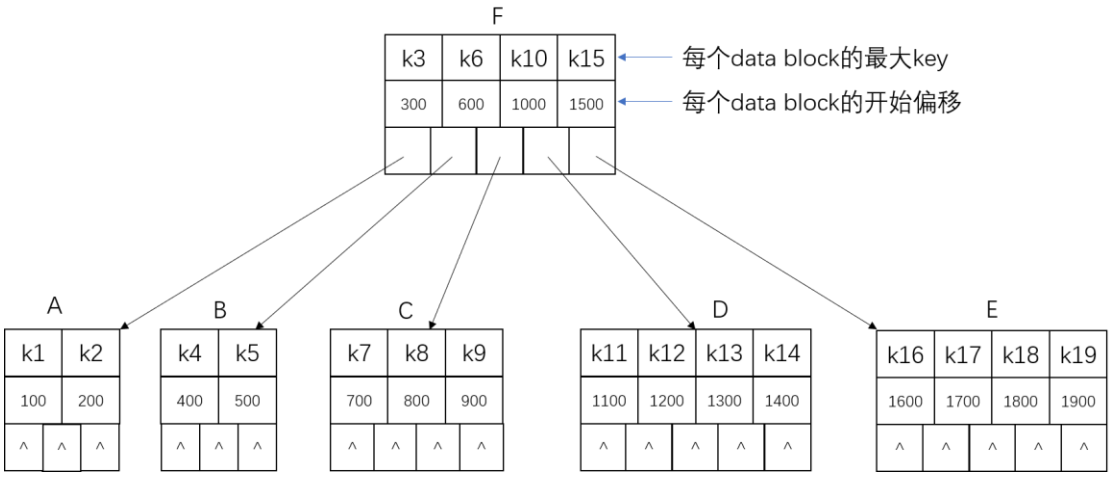


图 2.5 记录偏移的 B-Tree 例图

在将 B-Tree 持久化保存到 SSTable 中的 index block 时，需要解决孩子结点如何保存的问题。在内存中的 B-Tree，可以很方便地通过指针记录其孩子节点，但是在持久化到磁盘时，指针记录的是在内存中的地址，而我们需要的是孩子结点的磁盘上的物理地址，因此，我们先根遍历 B-Tree，即先保存其孩子结点，得到每个孩子结点在 SSTable 中的物理偏移之后，再保存父节点。例如图 2.5 所示的 B-Tree 结构，我们先保存 A、B、C、D、E 结点，同时得到这些节点的偏移量，再记录 F 结点，同时将第三行的指针替换为实际的物理偏移量。

2.4 Bloom Filter

Bloom Filter 是由 Bloom 在 1970 年提出的一种多哈希函数映射的快速查找算法。通常应用在一些需要快速判断某个元素是否属于集合，但是并不严格要求 100%正确的场合。

我们在判断某个元素是否属于集合时，通常采用 hash set 的方法，能够在 O(1)的时间复杂度内查询元素是否属于集合。但是随着数据量的增大，会出现大量的哈希冲突，若要减少哈希冲突的概率，则需要扩大哈希数组的大小。若要降低冲突发生的概率到 1%以下，则

需要将哈希数组的大小设置为 100n 以上(n 为元素总个数)。如此庞大的空间消耗是我们不能接受的，为了拥有 O(1)的时间效率，又希望尽量降低空间消耗，我们使用 Bloom Filter。

Bloom Filter 通过使用多个哈希函数，每插入一个元素，将多个 bit 置为 1，在查询时也相应检查对应的多个 bit 是否全为 1，若是则返回 true，否则返回 false。如图 2.6 所示，假设我们使用 3 个哈希函数、18 个 bit 来实现 Bloom Filter。当插入元素 x 时，3 个哈希函数计算得到的值分别为 1、5、13，相应的我们将这三个 bit 置为 1，同理插入 y 和 z 时也会将哈希函数计算得到的 bit 置为 1。此时，如果我们需要判断元素 w 是否存在集合中，3 个哈希函数计算得到的值为 4、13、15，我们检查对应的这三个 bit，发现不全为 1，因此给出结论，元素 w 不在集合中。

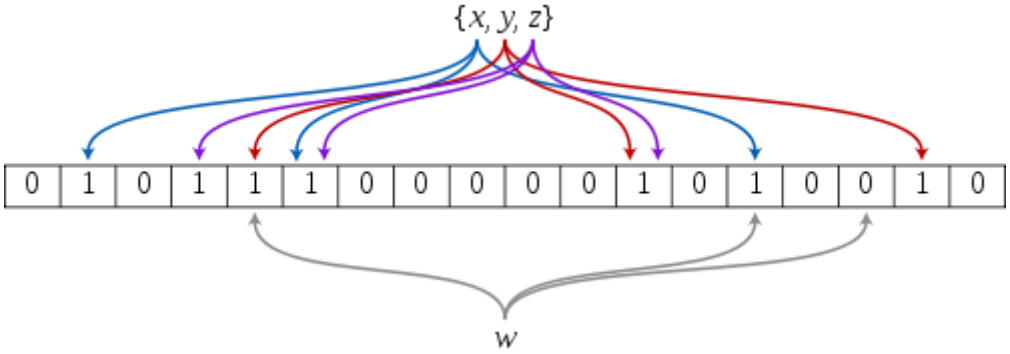


图 2.6 Bloom Filter 原理图

当然，Bloom Filter 并不能达到 100%的正确率。假如在图 2.6，我们需要判断的元素 t 恰好通过 3 个哈希函数计算得到的值为 1、3、4，而这三个 bit 都为 1，因此 Bloom Filter 返回 true。但实际上元素 t 并不存在于集合中。这种情况我们称为假阳性(false positive)。

设 n 为元素个数，p 为假阳性概率。为了使 p<1%，我们需要合理选择 Bloom Filter 的参数：

- 位数组大小 m = $\frac{n \times lnp}{(ln2)^2} \approx 20n$；
- 哈希函数的个数 k = $\ln 2 \times \frac{m}{n} \approx 14$。

哈希函数的个数是固定的，只有位数组大小需要随着数据量的变化而变化。因此对于图 2.3 中单个 SSTable 的 Bloom Filter 部分，我们只需要记录这 20n 个 bit，即⌈20n/8⌉个字节。

2.5 compaction 实现

compaction 在以 LSM-Tree 为架构的系统中是非常关键的机制，内存中的数据到达上限后不断刷盘，数据范围互相交叠的层越来越多，相同 key 的数据不断积累，引起读性能下降和空间膨胀。因此，compaction 机制被引入，通过周期性的后台任务不断的回收旧版本数据和将多层合并为一层的方式来优化读性能和空间占用问题。而 compaction 的策略和任务调度成为新的难题，看似简单的功能，实则需要各方面的权衡，涉及空间、I/O、CPU 资源和缓存等多个层面。

2.5.1 compaction 策略

首先介绍两种通用的 compaction 策略：

①size-tiered compaction：其思路非常直接，每层允许的 SST 文件最大数量都有个相同的阈值，随着 MemTable 不断 flush 成 SSTable，当某层的 SSTable 数达到阈值时，就把该层所有 SSTable 全部合并成一个大的新 SSTable，并放到较高一层去。size-tiered compaction 的优点是简单且易于实现，并且 SST 数目少，定位到文件的速度快。当然，单个 SST 的大小有可能会很大，较高的层级出现数百 GB 甚至 TB 级别的 SST 文件都是常见的。它的缺点是空间放大比较严重。

②leveled compaction：其思路是，不是像 size-tiered compaction 中那样直接合并成一个大 SSTable，而是拆分成多个 key 互不相交的小 SSTable 的序列，我们把这样的序列叫做“run”。每次做 compaction 时不必再选取一层内所有的数据，并且每层中 SSTable 的 key 区间都是不相交的，重复 key 减少了，所以很大程度上缓解了空间放大的问题。

在 TMDB 中，我们综合使用以上两种策略：

• 对于 level-0 层使用 size-tiered compaction，level-0 层是从 MemTable 直接 flush 过来的新 SSTable，该层各个 SSTable 的 key 是可以相交的，并且其数量阈值单独控制。TMDB 中规定 level-0 的最大 SSTable 个数为 4，当超过此阈值时，触发 compaction，将 level-0 中的所有 SSTable 合并成一个新的 SSTable 放到 level-1；

• 对于 level-1 及以上使用 leveled compaction，随着 SSTable 不断写入，level-1 的数据量会超过阈值。这时就会选择 level-1 中的至少一个 SSTable，将其数据合并到 level-2 层中与其 key 有交集的那些 SSTable 中，并从 level-1 删除这些数据。该次 compaction 完成后，level-2 的数据量又有可能超过阈值，进而触发 level-2 到 level-3 的 compaction，如此往复。在 TMDB 中规定 level-1 的最大大小为 10MB，往后每层大小上限为上一层的 10 倍。

2.5.2 compaction 选择

首先要选择进行 compaction 的 level。结合 2.5.1 节中 compaction 的策略，我们对每个 level 都计算评分。其中 level-0 的评分为当前 level-0 中 SSTable 的个数除以 4；level-1 及往后的每层的评分为当前该层的所有 SSTable 的总大小除以该层大小上限。选择评分最高的一个 level 来进行 compaction。

然后选择进行 compaction 的 SSTable。如果上一步选择的是 level-0，则直接将 level-0 中所有 SSTable 都合并成一个新 SSTable 并放到 level-1，然后将 level-0 中的所有 SSTable 删除。如果选择的不是 level-0，则需要选择若干个 SSTable。选择在优先级是，首先选择最长时间未进行 compaction 的 SSTable，然后选择所有与此 SSTable 在 key 上有重叠的 SSTable。在 TMDB 中，SSTable 文件名的后缀使用自增的数字，因此文件名后缀最大的表示最新进行 compaction，而最小的则表示最久未进行 compaction，因此选择特定 level 中进行 compaction 的 SSTable 就变得很简单，只需要选择文件名后缀最小的 SSTable 即可。

综合以上两点，TMDB 中 compaction 选取 level 和 SSTable 的整体流程如下[3]：

①计算各 level 的评分，选择评分最高的 level-i 进行 compaction；

②若 i=0，则将 level-0 所有 SSTable 进行 compaction 成新 SSTable 并加入 level-1，并删除 level-0 中所有旧 SSTable，否则进行步骤③；

③设 level-i 中需要进行 compaction 的文件集合 files=[ ]，选择 level-i 中文件名后缀最

小的 SSTable x 加入 files 中，files=[x]；

④在 level-i 中寻找所有与 x 有重叠的 SSTable，假设为 y 与 z，将其加入 files 中，files=[x, y, z]；

⑤设 level-(i+1)中需要进行 compaction 的文件集合 files_2=[ ]，选择 level-(i+1)中与 files 存在重叠的 SSTable，假设为 a、b，加入 files_2 中，files_2=[a, b]；

⑥再次遍历 level-i 中的 SSTable，寻找是否有这样的 SSTable t，满足 t 加入 files 中后，重复步骤⑤没有新的 SSTable 加入 files_2。如果有这样的 SSTable t，则将 t 加入 files，files=[x, y, z, t1, t2, …];

⑦files 与 files_2 中所有的 SSTable 就是此次 compaction 需要合并的所有 SSTable。

2.5.3 compaction 具体实现

由于 SSTable 是按照 key 升序排序的，因此在 compaction 时，可以使用归并合并的方式，以两个 SSTable 的 compaction 为例，使用两个指针分别从头开始扫描两个 SSTable。如图 2.7-图 2.10 所示为归并合并的前几步示意图。若 t1 和 t2 指向的 k 相同，则选择最新版本的数据加入新 SSTable 中，并且 t1 和 t2 同时向前移动一格；若 t1 对应的 k1 小于 t2 对应的 k2，则 k1 加入新 SSTable 中，并且 t1 向前移动一格，否则 k2 加入新 SSTable 中，t2 向前移动一格。重复执行直到 SSTable1 和 SSTable2 都被扫描完毕。

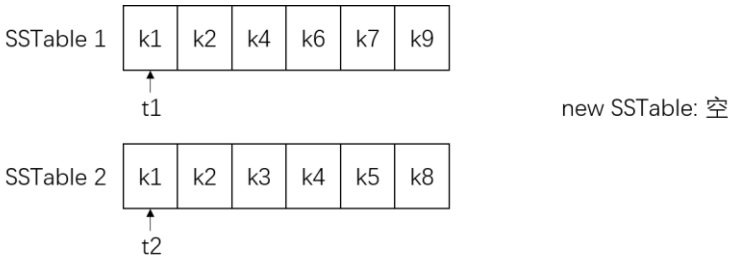


图 2.7 两个 SSTable 进行 compaction 的 step0

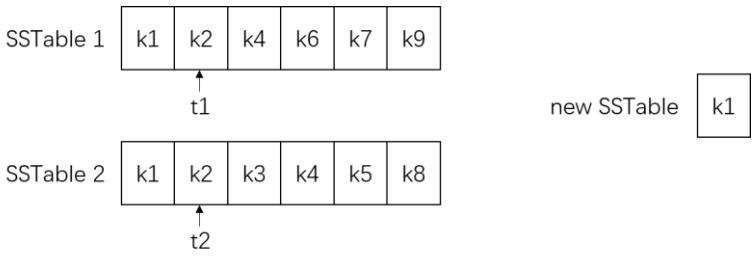


图 2.8 两个 SSTable 进行 compaction 的 step1

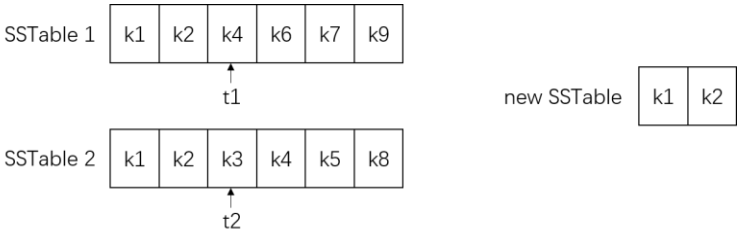


图 2.9 两个 SSTable 进行 compaction 的 step2

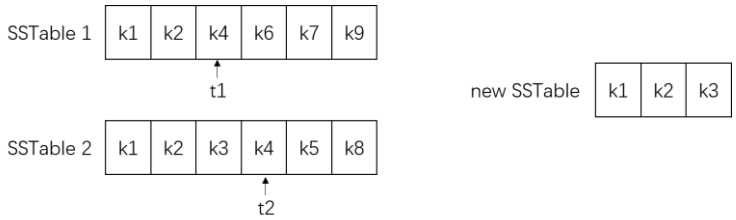


图 2.10 两个 SSTable 进行 compaction 的 step3

当然，实际情况可能不止两个 SSTable 进行合并，根据 2.5.2 节的选择算法，可能会有任意多个 SSTable 参与一次合并，整体执行思路与两个 SSTable 的合并思路一致，伪代码如下：

```
target    = [ ]  // 进行合并的 SSTables（int 型的后缀）
pointers = [ ]  // 指针，记录每个 SSTable 当前处理的进度（long 型的 offset）
ceilings   = [ ]  // 记录每个 SSTable 的最大 offset

while ( pointers 未达到 ceilings ) {
    找出 pointers 指向的最小的 Key（可能为多个）;
    将最小 key（若有多个则选择时间版本最新的）和对应的 value 写入新 SSTable;
    更新 pointers，最小 key 对应的 SSTable 的 pointer 往后移动;
}
新 SSTable 的 meta data 写入;
```

2.6 存储管理实现

如图 2.11 所示为 TMDB 存储层结构图。

在内存中，数据中保存在 Tuple List 中，元数据保存在 Class Table、Deputy Table、Object Table、Swiching Table、BiPointer Table 中，二者通过 Mem Manager 进行管理。此外，Mem Manager 提供插入数据的接口，同时实时监测 MemTable 的大小，当 MemTable 大小超过限制，则将 MemTable 转换成 immutable MemTable，然后触发 compaction 转化成 SSTable 并放置在 level-0。

在外存中，数据以 LSM-Tree 的形式管理。Level Manager 中记录每个 SSTable 属于哪个 level，同时记录一些关键的元信息，比如每个 SSTable 的最大 key、最小 key、大小等，同时提供调用 compaction 的接口，包括 2.5.2 中选择合适的 level 和 SSTable 来进行 compaction，以及 2.5.3 中具体地将若干个 SSTable 进行 compaction。

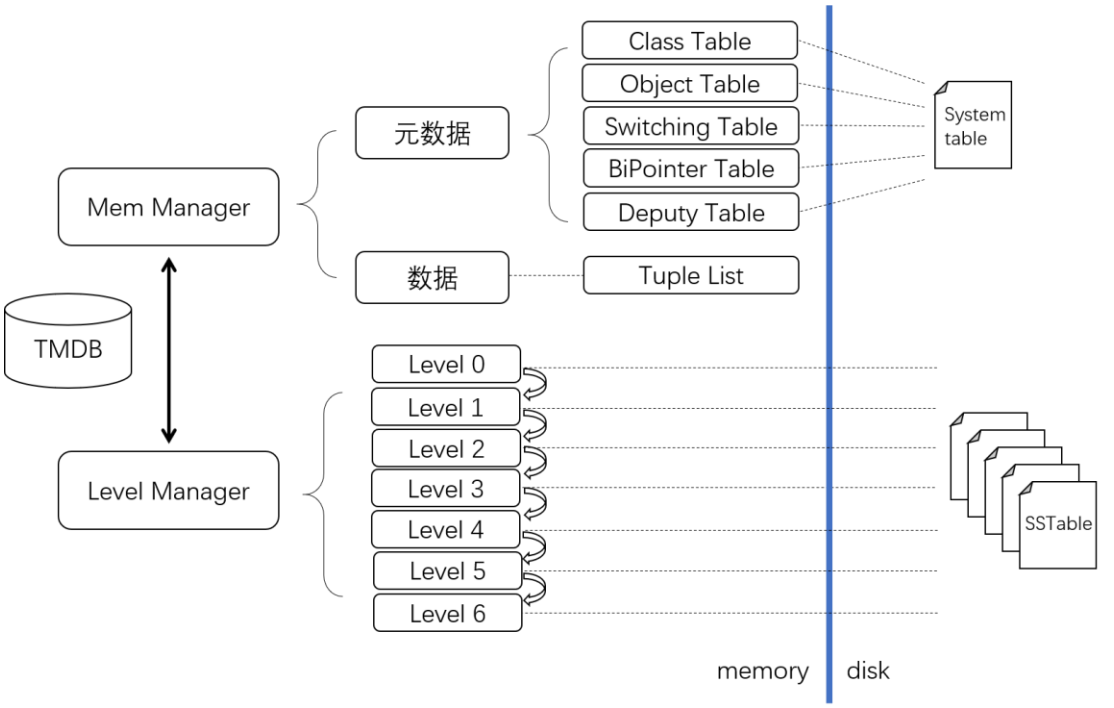


图 2.11 TMDB 存储层结构图

如图 2.12 为插入数据的时序图。首先 Transaction 接受请求，然后先写日志。写日志完成后，调用 Mem Manager 提供的 add()接口，往 MemTable 中添加数据。添加数据完成后需要判断 MemTable 是否达到大小上限，如果达到上限则需要调用 Mem Manager 提供的 writeInternal()接口进行 compaction 将 MemTable 转化成 SSTable，并通知 Level Manager 将新 SSTable 加入 level-0。此后，Level Manager 需要调用 compact()接口计算各 level 的得分，检查是否满足 compaction 的条件。如果满足，计算需要进行 compaction 的 level 和 SSTable，然后调用 Level Manager 提供的 writeToDisk()接口进行 compaction，同时记录日志。

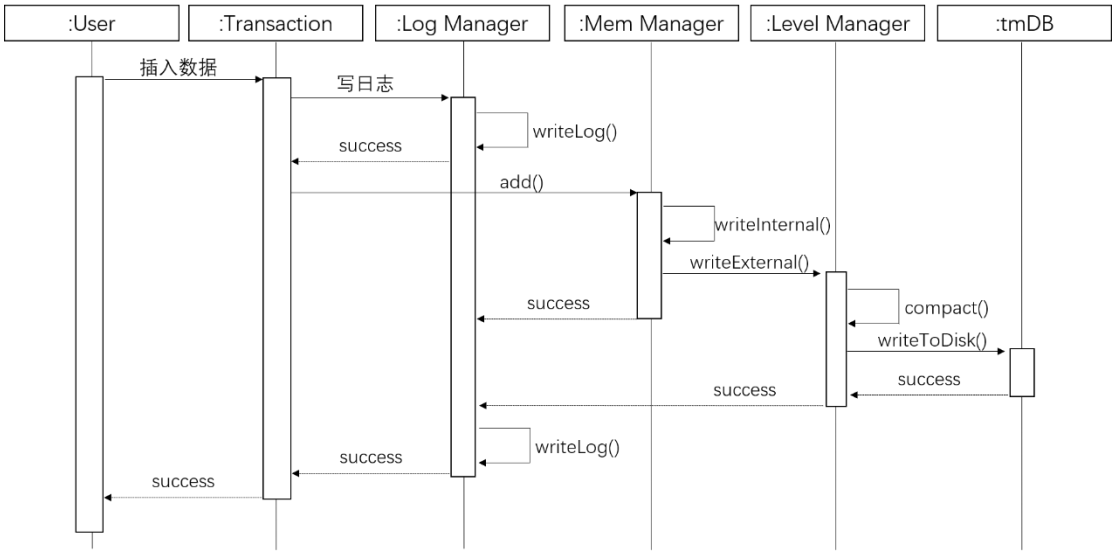


图 2.12 TMDB 的插入时序图

如图 2.13 为查询数据的时序图。相比于插入数据可能诱发的递归 compaction，查询流程则简单很多。首先 Transaction 接受请求，到 Mem Manager 管理的 MemTable 中遍历搜索目标 key。如果没有找到，则交由 Level Manager 中从 level-0 到 level-6 依次遍历每一层的所有 SSTable 搜索目标 key，具体的搜索单个 SSTable 的流程在 2.2 节中已经介绍。同时，

Level Manager 会记录在一个查询流程中目标 key 读 SSTable 的次数，如果超过一定阈值，则需要调用 compact()和 writeToDisk()进行 compaction。如果查询目标 key 读取多个 SSTable 就需要进行 compaction 的原因如下：在同一层中，理想情况下，每个 SSTable 是不存在重叠的，但是如果在同一层读取了多个 SSTable，则表示该层存在大量的数据重叠，已经造成了明显的读放大，因此需要调用 compaction，处理这些重复的数据，消除数据冗余。

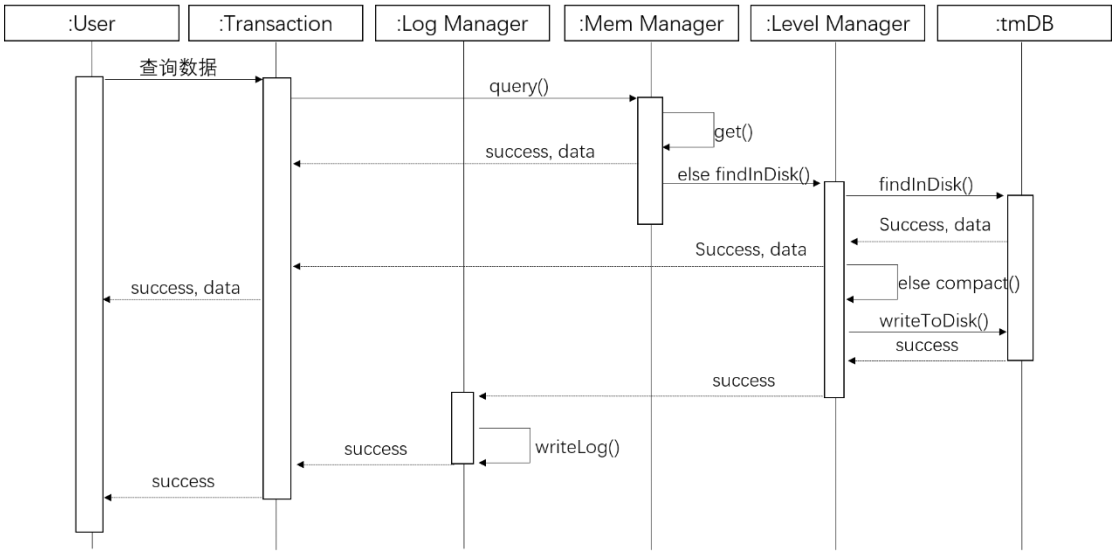


图 2.13 TMDB 的查询时序图

3 编译查询

3.1 查询分析

分析器部分的目的是对输入的查询字符串进行词法，语法和语义的检查。如果正确则生成一个查询树并交给计划器进行下一步的处理，否则返回一个错误。

这个阶段分为两个部分：首先是词法和语法分析过程。它将输入的查询字符串转换为语法元素的内部表现形式并以此为依据进行语法分析，检查命令是否符合 TOTEM 的 TOTEM-SQL 语法，这个部分的结果是一个与命令内容相应的分析树。其次是语义分析部分，或者称为转换阶段，它接受语法分析器传来的分析树，并检查该命令是否有不符合系统规定的地方，如果没有则最终转换为一个查询树返回。如下图所示。

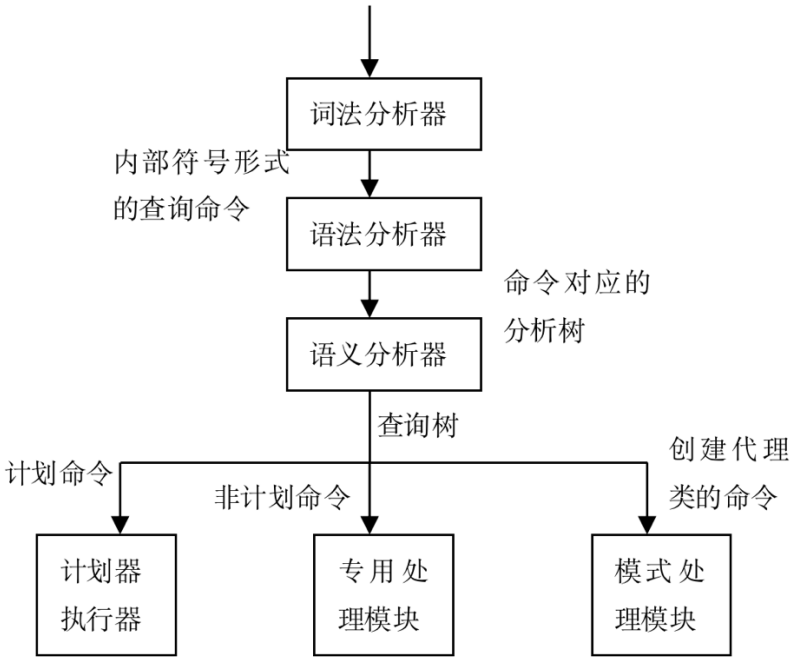


图 3.1 查询流程

所有的查询命令都会经过词法和语法分析过程。有一些复杂的查询命令（要产生计划的命令）会调用转换器部分，主要包括 SELECT, INSERT, DELETE 和 UPDATE 命令。它们的查询树会经过计划器转换为查询计划，然后由执行器根据计划来进行扫描工作。

其它许多较简单的非计划命令(如创建或删除基本类的命令)不需要进行类的扫描工作，只要在系统的模式数据上进行一些操作,因此在转换器部分只是进行了简单的逻辑错误的检查，而没有做任何实际的转换工作。它们的处理直接调用相应的模块就可以完成了。

创建类操作包括创建严格代理类和非严格代理类两部分。比较特殊的是创建严格代理类的命令。该命令首先要求创建代理类的模式信息，这部分相当于一个非计划命令；在此之后还要根据声明的代理规则对源类进行一次查询创建代理对象，并建立对象与代理对象的联

系，这部分又要和计划命令类似地调用计划器和执行器。为了程序接口的统一，在语义分析器中所有命令最后都是按统一的查询树形式输出的。

3.1.1 结构分析

TOTEM 移动端的词法、语法分析器是利用 JsqParser 工具和 javacc[4]制作的。词法分析器根据制定的 javacc 词法分析识别标识符、SQL 关键字和操作符等语法元素。语法分析器定义并且包含一套语法规则和触发这些规则时执行的动作，这些动作的代码用于建立分析树。因为命令的形式和用途各异，分析树采用了特定的分析树数据结构，每种数据结构对应一种命令形式。如果一个输入符合某个命令的格式，那么分析器将会分配一个相应的分析树结构，并且把命令中的各个语法成分填写到结构体的各个域中去。

为了更好地理解处理一个查询时 TOTEM 里使用的数据结构，我们用一个简单的例子演示在每个阶段数据结构所做的改变。

```
SELECT a->b->c
from table1
where c in (select d from table2)
and b=2;
```

该查询最开始是由用户输入的如上所示的一个字符串。词法分析器将会识别出下表所示的信息，并转换为相应的标号传递给语法分析器。

表 3.1 **词法分析结果示例**

关键字	SELECT, FROM , WHERE, IN, AND
标志符	a, b, c, d, table1, table2
数字	2
操作符	->, =

语法分析器接收到了例中的查询符号串后，会将其识别为一条 SELECT 查询命令并执行相应的动作，生成如下的分析树结构。

表 3.2 **分析树结构**

SelectStmt	
成员	**说明**
selectbody	select 的主体内容
withItemList	递归调用的 item list
useWithBrackets	是否使用 with 进行递归
selectBody	
deputySelect	是否使用了跨类查询
deputyChain	跨类查询的链表
distinct	哪些元素使用了 distinct 标志符
selectItem	查询的元素列表

intoTable	selectitem into 后面接的 table list
fromItem	from 后接的 table list
joins	参与 join 的表的 list
where	纪录整个 where clause 的内容
groupBy	分组的元素
orderByElements	输出时进行排序的参考元素
having	having 后接的元素
limit	限制输出多少

3.1.2 语义分析

语义分析（转换器）以分析器传递过来的分析树作为输入，然后递归地处理它。一般来说各种命令最后都会转换为一个查询（Query） 节点， 这个节点将是新数据结构的最顶端节点。以上面的查询为例，得到如下的结构：

表 3.3 语义分析成员

Query	
成员	**说明**
NodeTag type	标识位，用于标识该节点的具体属性
CmdType commandType	查询命令类型，本例中为 select，还有可能为 insert，update，delete 或 utility
Node utilityStmt	不需要计划的命令将被存放在该域中
int resultClass	如果是 SELECT INTO，这里对应目标类的编号
bool hasAggs	是否有聚集操作
bool hasSubLinks	是否有相关子查询
List rtable	所有查询中涉及到的类
FromExprjointree	查询中要进行连接的类，以及连接条件
List rowMarks	FOR UPDATE 列表
List targetList	查询选择的目标属性表达式列表
List groupClause	GROUP 子句
Node havingQual	HAVING 子句的表达式
List distinctClause	DISTINCT 子句
List sortClause	ORDER BY 子句
Node limitOffset	OFFSET 子句
Node limitCount	LIMIT 子句
Node setOperations	集合查询 UNION/INTERSECT/ EXCEPT 的树状结构
以下的域由计划器填写，在转换器中一直为 NULL	

形成查询节点后，转换器会检查 FROM 子句里面的类名是否存在。在类名全部被识别后，转换器检查查询所用的属性名是否包含在查询给出的类里。

非计划命令是那些不需要进行计划的查询命令。这类命令多是创建或删除一些数据库实

体（如基本类、函数、触发器等）。它们并不对用户类进行扫描，而主要在系统表中插入或删除一些信息。虽然非计划命令的工作相对简单，但是它们在上述的词法语法分析部分的处理过程与计划命令是完全一样的。以下面创建代理类的命令为例：

```
create selectdeputy deputy as
select a from table1;
```

该命令在词法和语法分析器处理之后会转换为如下分析树：

表 3.4 分析树结构

CreateDeputyClassStmt	
成员	**说明**
Table deputyClass	deputy class 的名字
selectStmt select	select statment 子节点，存储 select 语句
Boolean orReplace	是创建或者交换
String type	代理类的类型（group，join，union，select）

转换器接受到该分析树后，会检查创建的类的名字是否存在冲突，并检查各个属性即类型的合法性，以及属性和类上的约束。如果发现了错误即报错并中止查询，否则就将收集到的模式信息封装到 createDeputyClassstmt 域中去。

转换器会根据代理规则提取该代理类的模式信息，检查并存储到 createDeputyClassstmt 域中去，然后封装到 Query 节点中。当进入专门处理代理类创建的函数后，首先会在系统中填写该类的模式信息。需要注意的是，代理类的模式信息包括实属性和虚属性以及切换操作。

实现代理类的创建时，对于类的基本信息（如名字等）以及实属性和方法的处理过程与基本类的创建过程一致，不同的在于对代理规则的处理。下面详细说明：

a. 检查代理规则的格式是否正确

各种代理类型的代理规则都要符合一定的限制。例如 selection 型代理规则中不能有分组聚集操作和集合操作等，而且只能在一个源类上进行选择。在对代理规则进行处理之前，系统首先要检查代理规则在格式上是否满足对应代理类型的限制条件。

b. 提取虚属性的模式

代理规则中的各个目标表达式对应了代理类的虚属性。目标表达式的别名就是虚属性的属性名，而目标表达式的返回类型就是虚属性的数据类型。例如代理规则中的目标表达式“（age－18） AS grade”，别名为“grade”返回类型为整数型。因此，在类 undergraduate 的模式中有一个名为“grade”的整数型虚属性。虚属性在 pg_attribute 中占一行，并且在 isdeputy 域上为真。

c. 提取并存储 switching 表达式

目标表达式不仅确定了虚属性的模式，同时也定义了由源属性值生成虚属性值的计算过程，即 switching 操作。因此，需要把目标表达式以某种形式存储下来。但是，直接存储目标表达式是无意义的。这是因为目标表达式中的变量节点表示的是源类中的某个属性，它只在本查询命令中有效。

当进行虚属性的查询时，系统先扫描代理类的对象，然后根据代理对象的指针找到其源对象，最后取其属性值进行计算。要表示这种过程就需要前面介绍的路径表达式。因此我们在存储目标表达式之前，先要把其中的变量节点进行修改，以表示代理路径。这种修改过的表达式称为 switching 表达式。

代理类的定义中，代理规则实际上只定义了本代理类和其直接父类之间的关系，而不关心这些父类是否也是代理类。因此很有可能代理规则中的目标表达式含有源类的虚属性的。在这种情况下，我们生成新的 switching 表达式时，首先要把目标表达式中虚属性的 Var 节点用其对应的 switching 表达式替换，然后再修改这个新的表达式树中的各个变量节点。

3.2 查询逻辑

1．创建源类

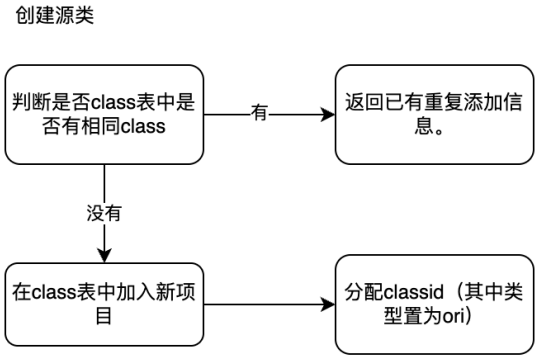


图 3.2 创建源类逻辑设计

2．删除对象

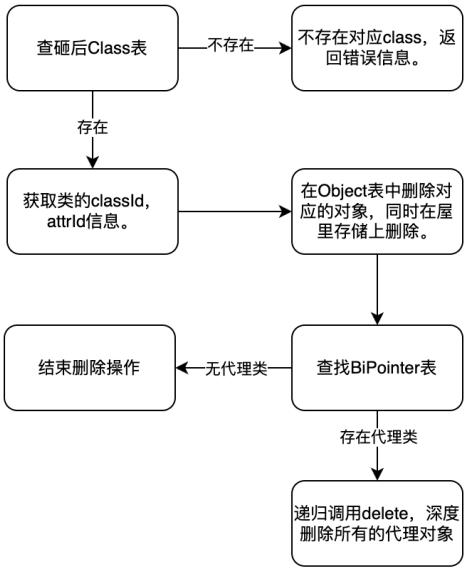


图 3.3 删除对象逻辑设计

3．查找

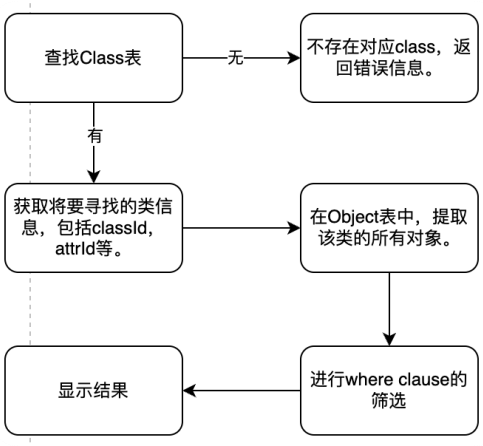


图 3.4 查找逻辑设计

4．插入

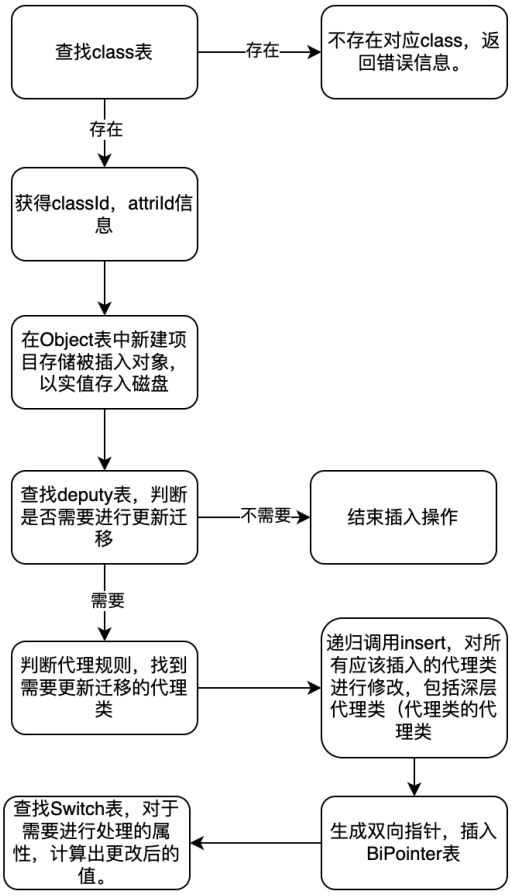


图 3.5 插入逻辑设计

5．删除类

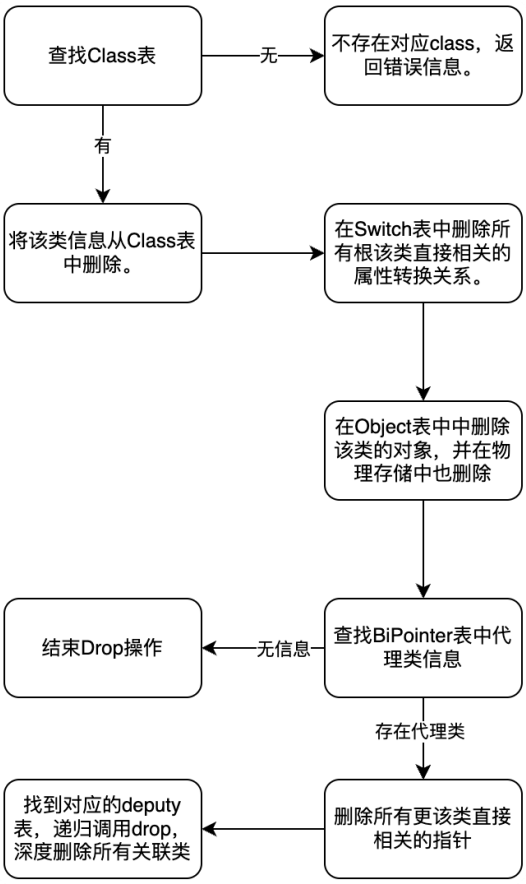


图 3.6 删除类逻辑设计

6．创建代理类

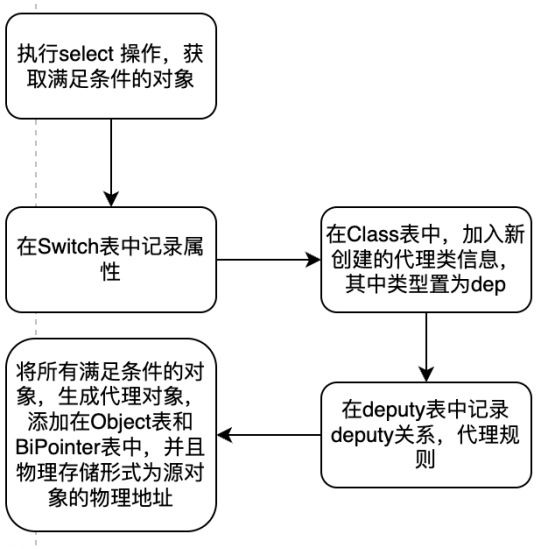


图 3.7 创建代理类逻辑设计

7．修改属性

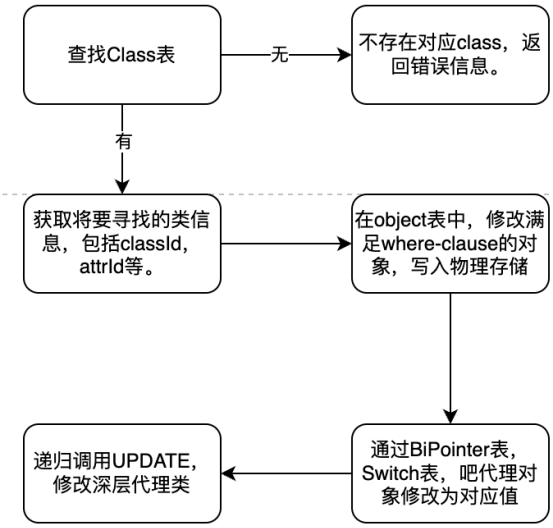


图 3.8 修改属性逻辑设计

8．跨类查找

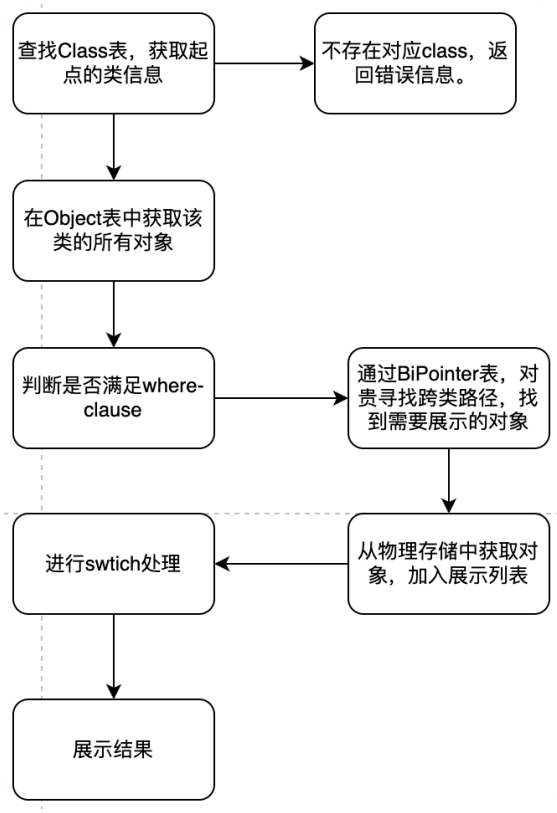


图 3.9 跨类查找逻辑设计

3.3 具体实现

3.3.1编译

编译部分拓展了基于 javacc 的 JSqlParser[5]实现。Java Compiler Compiler（JavaCC）是一种用于生成词法分析器和语法分析器的工具。它使用一种类似于 BNF 的语法，输入语言定义，并生成 Java 代码，用于执行词法分析和语法分析。JavaCC 使得生成语言分析器变得简单，支持各种语言特性，如自动生成状态机，语法制导翻译，词法优先级和语法优先级等。另外，JavaCC 还提供了诸如词法规则高亮显示，语法错误报告等高级功能。由于它生成的是 Java 代码，所以 JavaCC 生成的语言分析器具有跨平台性，高效性和可维护性。

JSqlParser 是一个 Java 库，可以解析 SQL 语句并将其转换为程序易于处理的表示形式。它可以从字符串或文件中解析 SQL 语句，并支持各种 SQL 命令和子句，包括 SELECT、INSERT、UPDATE、DELETE 和 CREATE 语句。解析后的 SQL 语句使用树状结构表示，程序可以轻松遍历和修改它。JSqlParser 旨在轻量级且易于使用，不需要任何外部依赖。它可用于构建各种应用程序，包括 SQL 查询生成器、数据库迁移工具和 SQL 查询分析和优化工具。

编译部分具体代码实现流程如下

1. 创建代理类

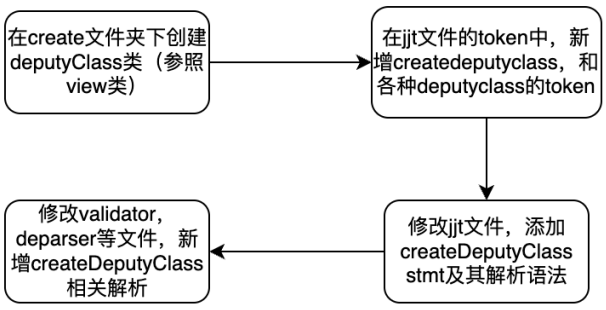


图 3.10 创建代理类编译实现流程

2. 跨类查询

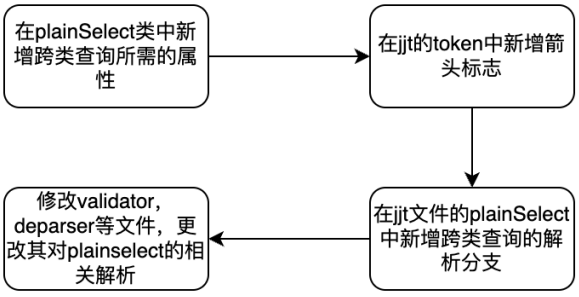


图 3.11 跨类查询编译实现流程

完成修改后的插件编译流程如下：

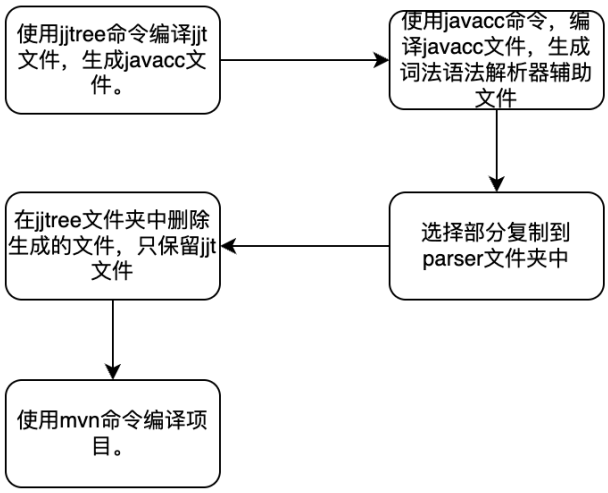


图 3.12 编译器编译过程

3.3.2查询

查询涉及的数据结构见第二章存储管理。

查询的具体流程如下：

1. 创建类

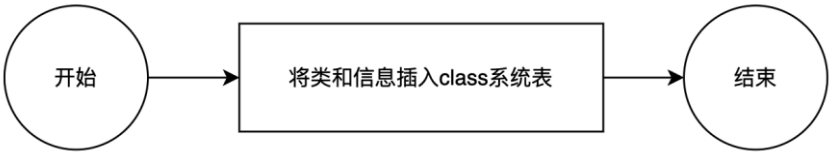
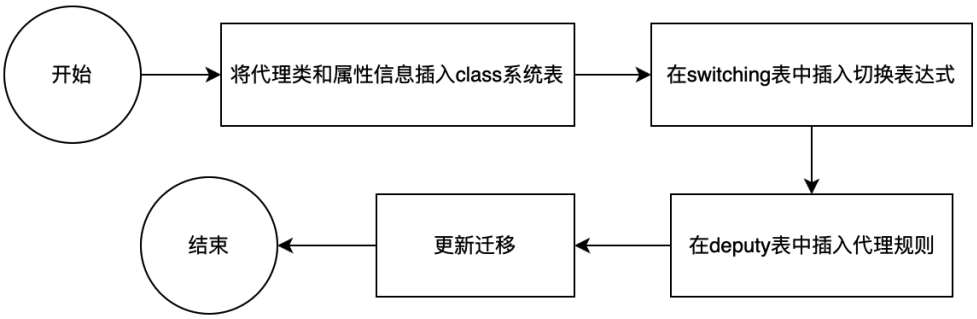


图 3.13 创建类实现流程

2. 创建选择代理类



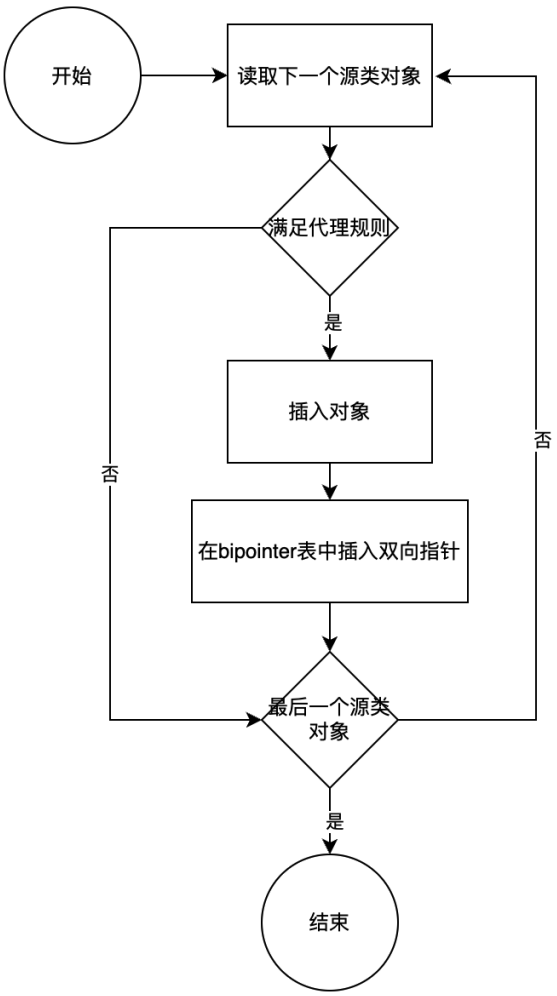


图 3.14 创建选择代理类的运行流程（下图是更新迁移）

3. 删除类

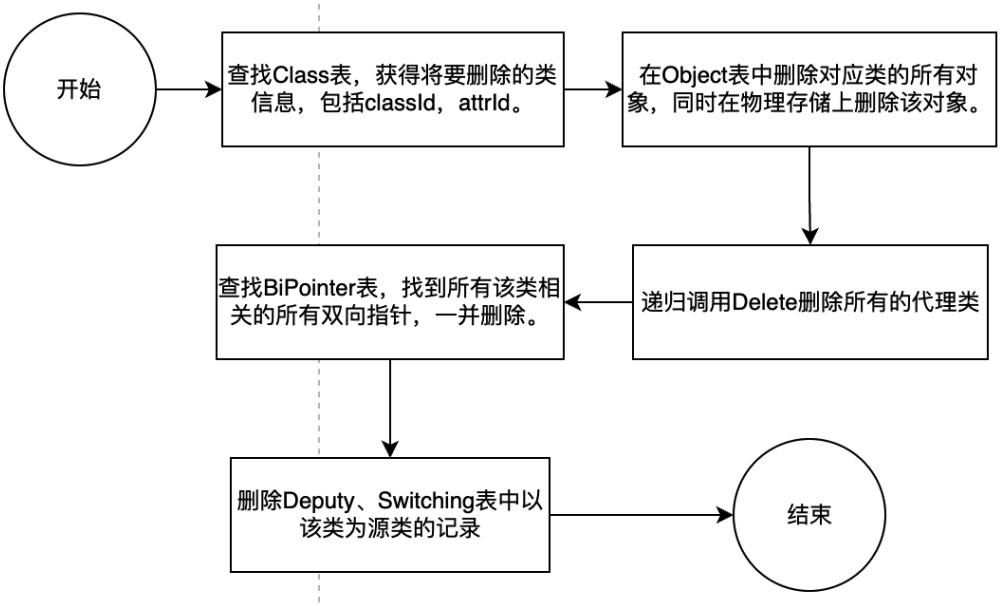


图 3.15 删除类的运行流程

4. 插入对象

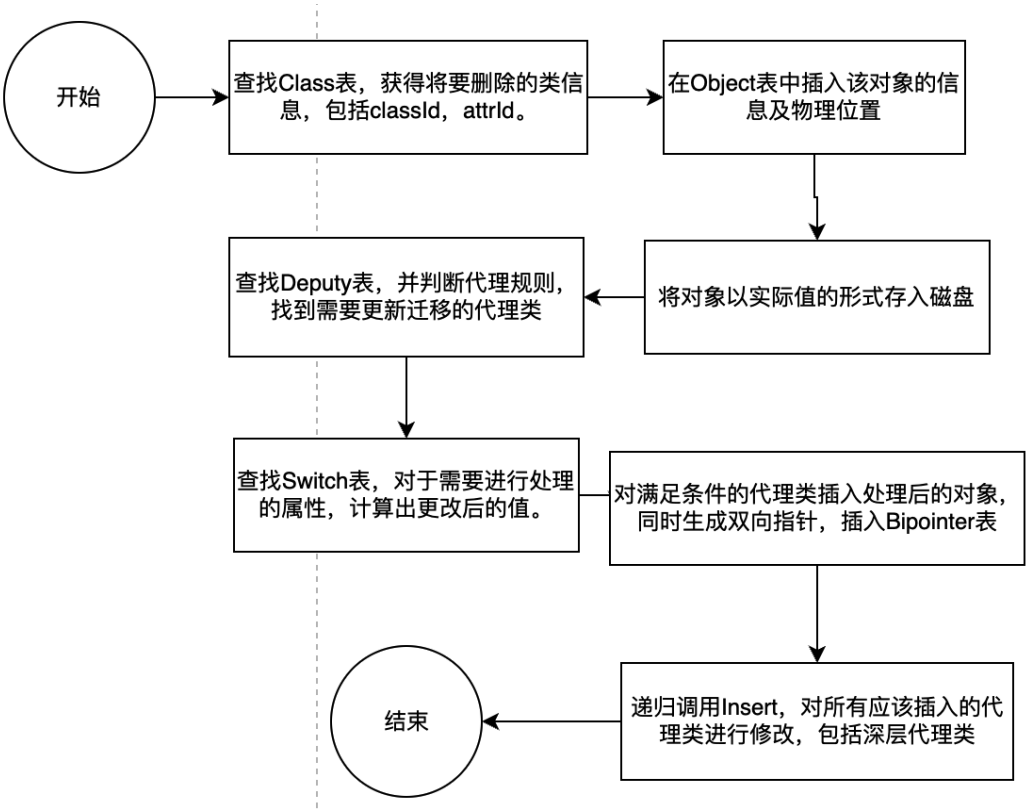


图 3.16 插入对象的运行流程

5. 删除对象

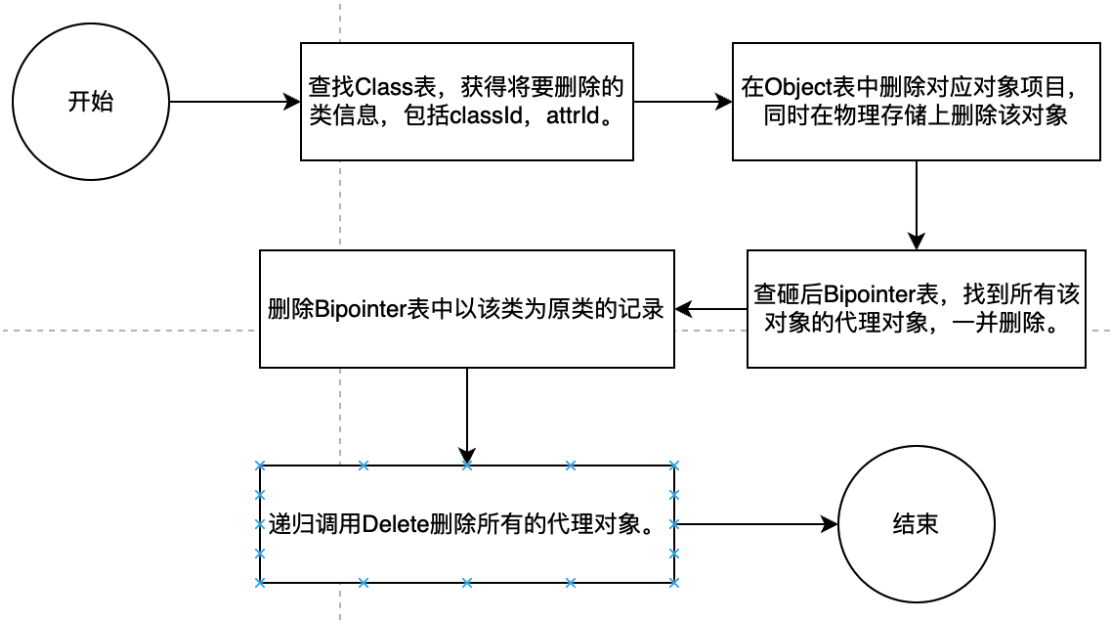


图 3.17 删除对象的运行流程

6. 修改对象

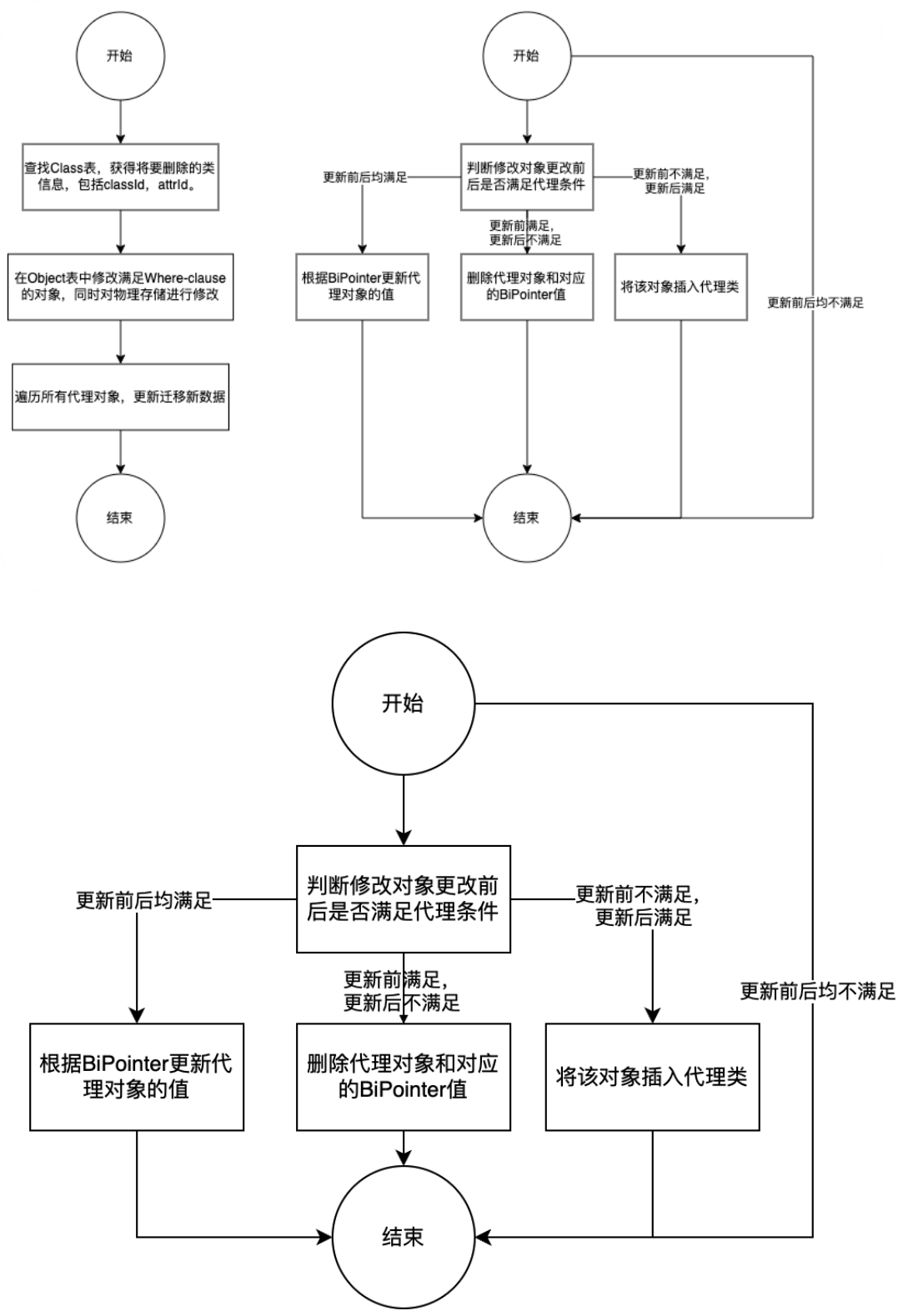


图 3.18 修改对象的运行流程（下图为更新迁移）

7. 查询对象

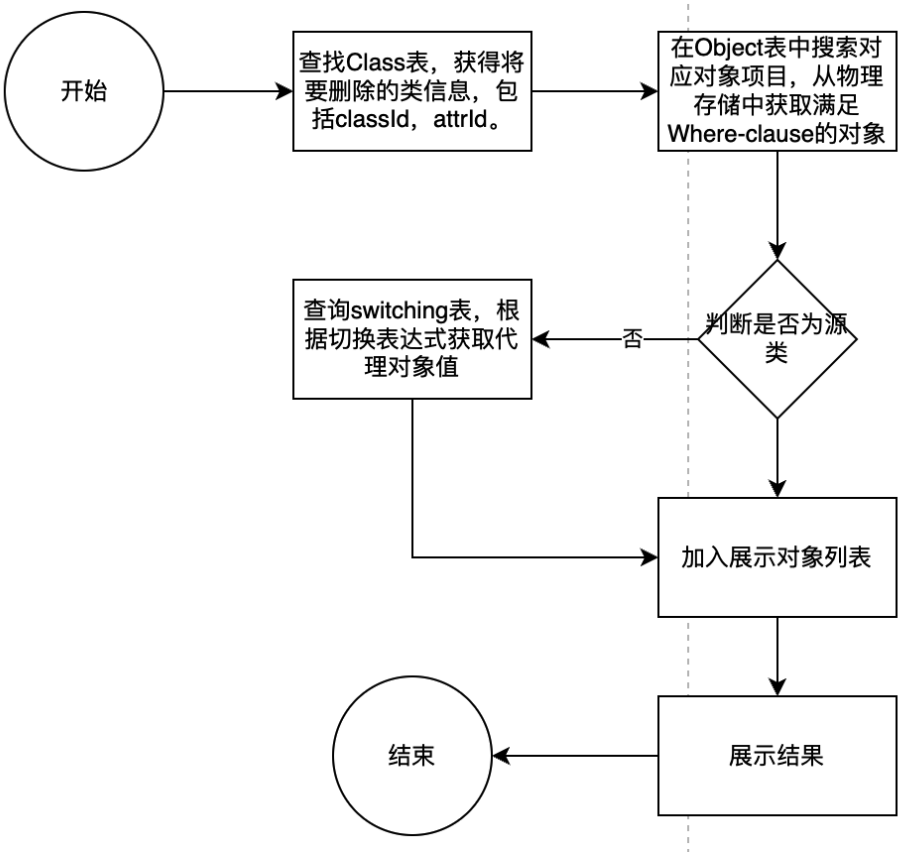


图 3.19 普通查询的运行流程

8. 跨类查询

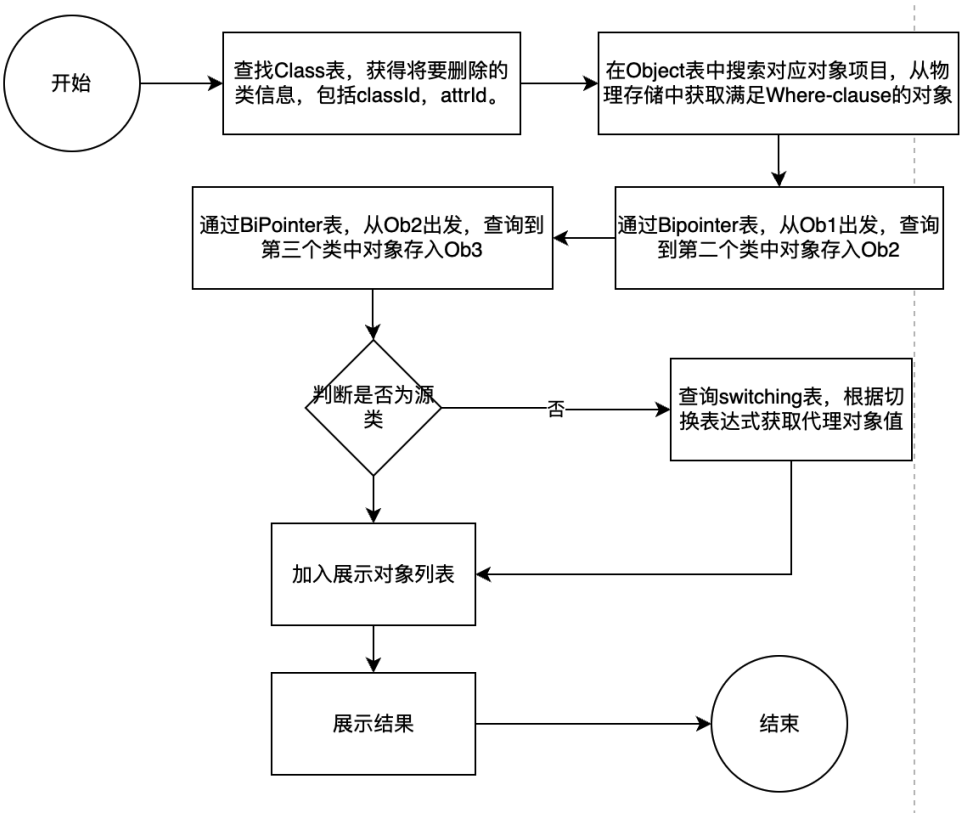


图 3.20 跨类查询的运行流程

4 日志管理

4.1 日志结构

- LogTableItem: 采用键值对存储形式。每个LogTableItem对应一个logid唯一标识该条日志记录，占四个字节；op为该条日志记录所记录的数据在数据库中的操作，0表示插入，1表示删除操作，占一字节；key代表日志记录的键，value为由key计算出来的对应值（即日志所记录的数据库中的对象）;offset为日志记录在文件中的偏移量，占八字节。
- 磁盘：日志以顺序写的方式存入磁盘，logid依照写的顺序递增。
- HashMap:HashMap的key是String类型,为日志记录的数据库中的对象；value是List类型，为一个logid列表，这些logid是记录key对应数据库中对象的所有LogTableItem的logid。
- 索引b树：每结点的第一行记录的是LogTableItem对应的logid，第二行记录的是该LogTableItem在日志文件中对应的偏移量，第三行记录的是该结点的孩子结点指针。

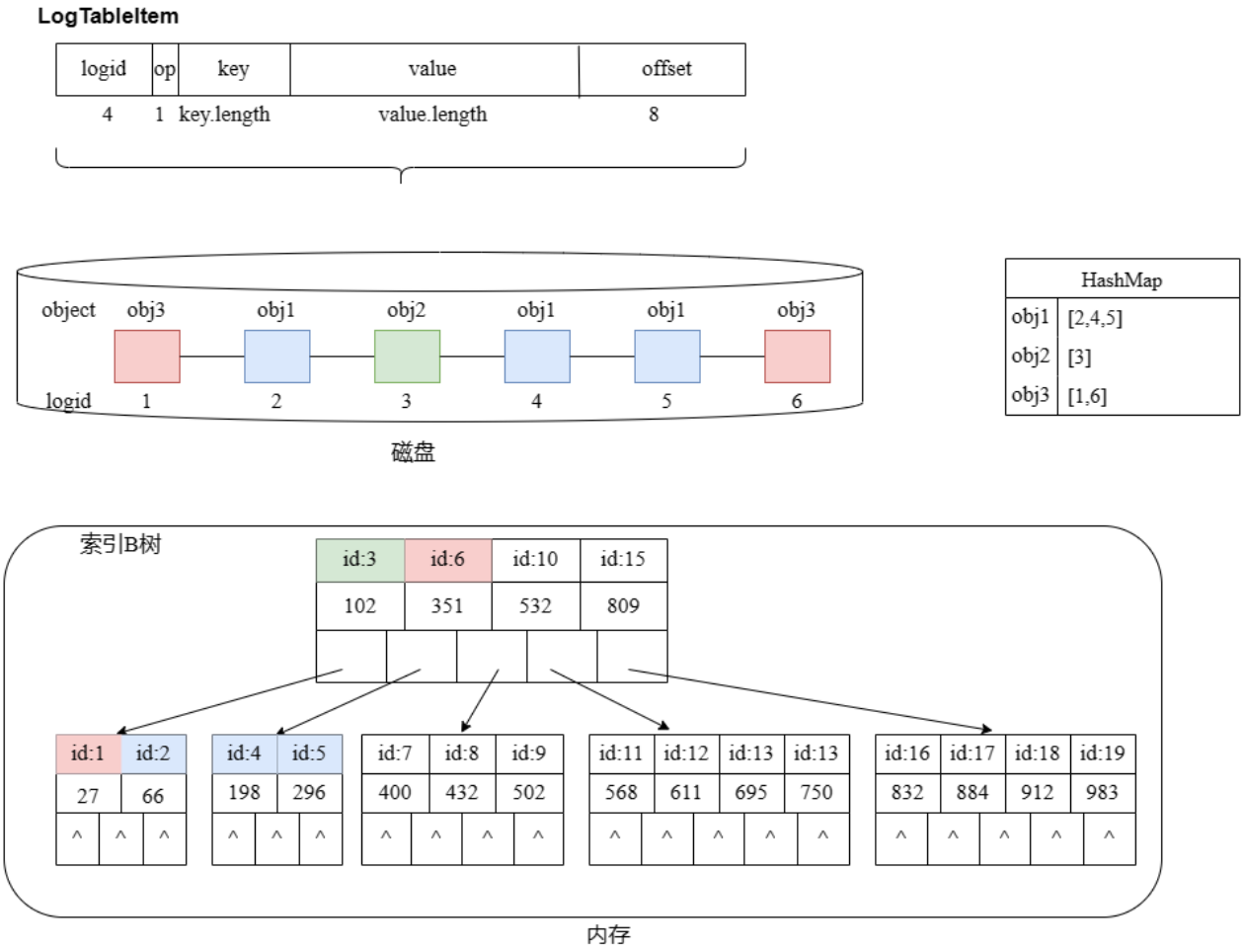


图 4.1 日志整体结构

LogTableItem 类：

```
public class LogTableItem {
    public int logid;//日志记录 id
    public Byte op;//0 表示插入，1 表示删除操作
    public String key;//键
    public String value;//值
    protected long offset;//日志记录在文件中的偏移量

    public LogTableItem(int logid,Byte op,String key,String value){
        this.logid=logid;
        this.op=op;
```

```
        this.key=key;
        this.value=value;
    }
    public LogTableItem(){};
}
```

4.2 日志基本操作

日志基本包括以下功能：对新日志文件与新索引 b 树文件的初始化、删除日志目录下的旧文件、给定参数 key、op、value 将日志持久化到磁盘、加载并显示目前日志文件中的所有日志记录，这些功能函数均在文件 LogManager.java 中实现。

4.2.1日志管理类和日志管理类的初始化

日志管理器的公共字段包含 logFile、fileTree、logIOAccess、checkpoint、 check_off、limitedSize、writeB_size、currentOffset、currentId。它们的具体信息如下：

表 4.1 日志管理类成员

成员	说明
File logFile	日志文件
File fileTree	存 B 树文件
RandomAccessFile logIOAccess	LogFile 的 RandomAccessFile 指针
int checkpoint	日志的检查点
long check_off	日志检查点在日志文件中的偏移量
long limitedSize	日志文件限制大小
long writeB_size	将索引 B 树持久化到磁盘的临界日志文件大小
long currentOffset	当前写 LogTableItem 起点在日志文件中的偏移量
int currentId	当前所写 LogTableItem 的 logid
BufferedOutputStream bTreeWriteAccess	存 B 树文件的输出字节缓冲流

在日志管理中，还需要声明如下数据结构：

（1）声明索引 b 树，BTree_Indexer 类将在后文介绍。声明语句如下：

```
public BTree_Indexer<String, Long> bTree_indexer;
```

（2）声明 hashmap，将日志记录按对象分类，key 为 String 类型（存的是对象），value 为 List 类型（由 LogTableItem 的 logid 组成的列表）。声明语句如下：

```
public static Map< String, List<Integer>> Map;
```

（3）声明遍历 hashMap 的 keySet。声明语句如下：

```
public Iterator<Map.Entry<String, List<Integer>>> iterator;
```

在 Totem 移动端数据库中，实现日志管理器的初始化函数是 LogManager() 构造函数。建立日志目录，固定路径为"/data/data/drz.tmdb/log/"，接着建立日志文件和存放 b 树的文件，初始化上述日志管理类成员，其中 checkpoint、check_off 初始为-1，表示开始日志中没有设置检查点。并且若当时存 b 树的文件为空，则初始化新的索引 b 树，否则将 b 树从磁盘加载到内存。

```
  //构造方法
public LogManager() throws IOException {
    File dir = new File(Constants.LOG_BASE_DIR);
    if(!dir.exists()){
        dir.mkdirs();
    }
    logFile = new File(Constants.LOG_BASE_DIR + "TMDBlog");
    fileTree = new File(Constants.LOG_BASE_DIR + "log_btree");
    if(!logFile.exists())
        logFile.createNewFile();
    if(!fileTree.exists())
        fileTree.createNewFile();
    bTreeWriteAccess = new BufferedOutputStream(new
FileOutputStream(fileTree));
    logIOAccess = new RandomAccessFile(logFile, "rw");
    Map= new HashMap < String, List<Integer> > ();
    checkpoint=-1;
    check_off=-1;
```

```
    currentOffset = 0;
    currentId = 0;
    //app 启动时将 b 树从磁盘加载到内存
    if(fileTree.length()==0){
        bTree_indexer = new BTree_Indexer<>();
    }else{
        bTree_indexer=new BTree_Indexer<>("log_btree",0);
    }
}
```

4.2.2写日志

在数据写入 Memtable 之前，根据 WAL，需要将记录写入日志持久化到磁盘。WriteLog(String k,Byte op,String v)方法用于根据传入的参数——准备记录的 LogTableItem 对应的键和值以及相应的数据库操作，将写好的日志记录 LogTableItem 持久化到磁盘。具体步骤如下。

首先，新建 LogTableItem 对象，利用构造函数 LogTableItem(int logid,Byte op,String **key**,String **value**)，其中 logid 为 currentId，其余均为方法所传入的参数。这个 LogTableItem 对象的属性 offset 置为 currentOffset。将 currentId 自增 1.调用方法 writeLogItemToSSTable(LogTableItem log)将日志记录写入文件中（该方法将在后文中介绍）。然后，将该日志记录的 logid 按对象分类。遍历 hashmap，若在 hashmap 的 key 中已经存在该 LogTableItem 的属性 **value** 值，则表示记录同样的对象的日志记录在之前出现过，将本次所写的 LogTableItem 对应的 logid 加入到 hashmap 中该 key 对应的 value 列表中；否则新建一个列表，将本次所写的 LogTableItem 对应的 logid 加入，再在 hashmap 中加入这个键值对。最后，调用 BTree_indexer 类的 insert()方法将记录该条 LogTableItem 的 offset 的节点插入 b 树中。若日志文件达到将索引 B 树持久化到磁盘的临界日志文件大小，则调用 BTree_indexer 类的 write()方法将索引 B 树持久化到磁盘，以防突然断电索引 B 树没有保存。

注意：加粗的 key 和 value 表示 LogTableItem 对象的属性，否则表示 hashmap 的键值对。

下图是写日志的流程。

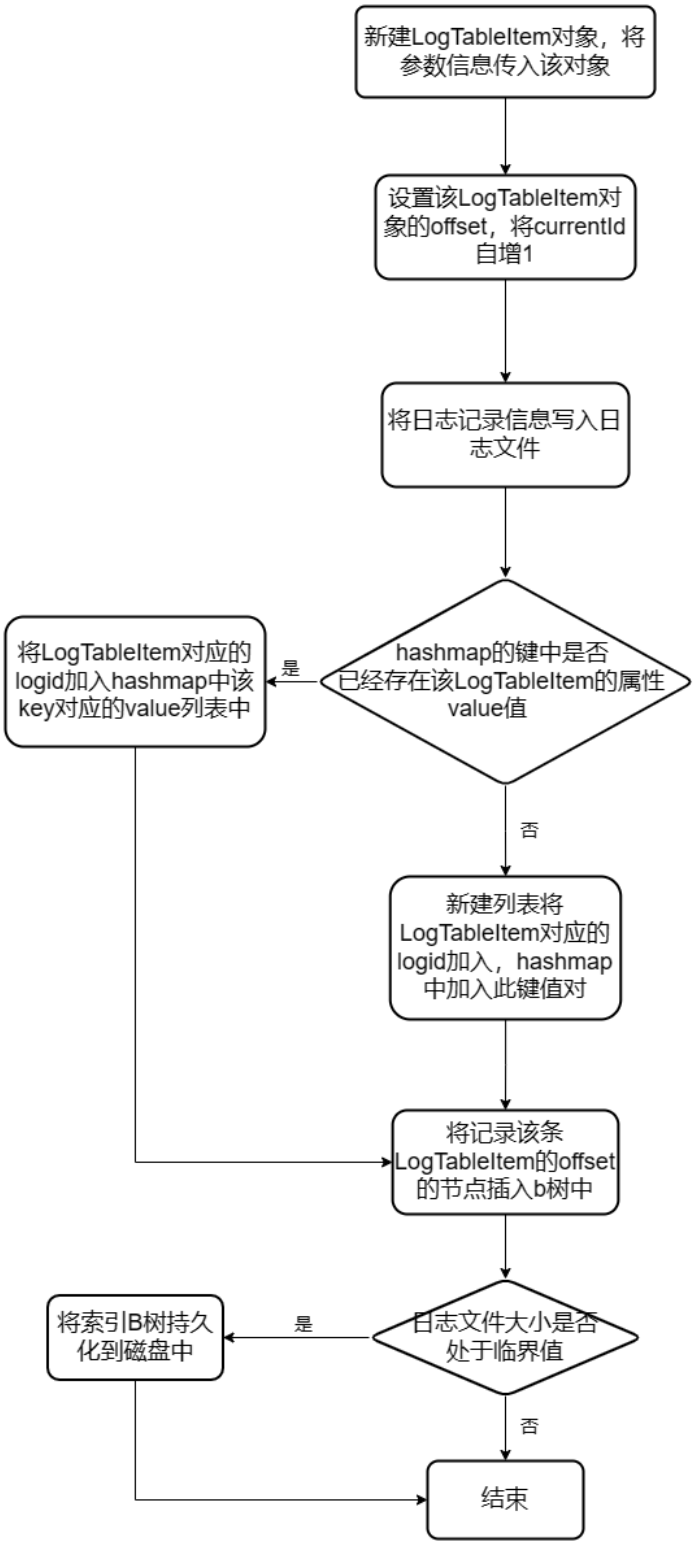


图 4.2 写日志的流程

4.2.3将 LogTableItem 信息写入日志文件

该部分由函数 writeLogItemToSSTable(LogTableItem log)实现，其中参数为要写入日志文件的 LogTableItem 对象。用日志文件 RandomAccessFile 指针找到即将写入日志文件的起始位置，然后依次写入 LogTableItem 对象的属性 logid、op、key、value、offset。利用 RandomAccessFile 指针方法确定写完后指针的位置，设置为 currentOffset。

下图是将 LogTableItem 信息写入日志文件的流程。

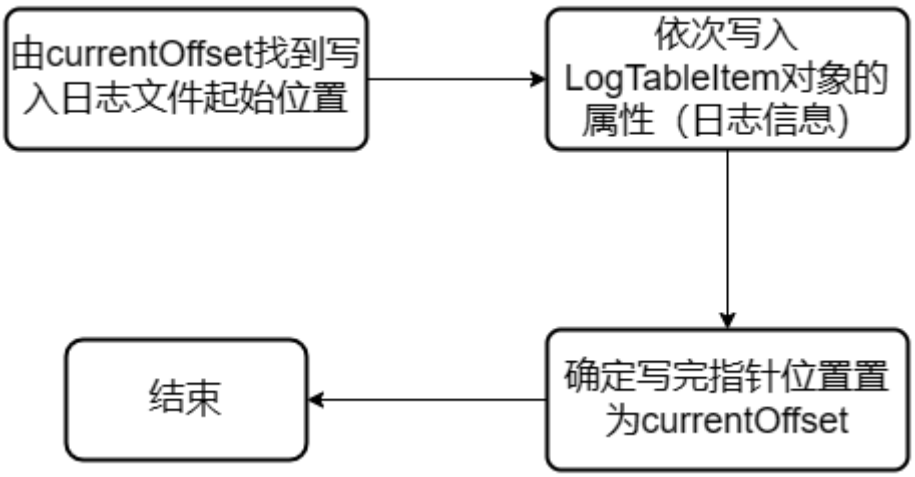


图 4.3 将 LogTableItem 加信息写入日志文件流程

4.2.4加载日志并显示

此功能的实现在 loadlog()方法中。首先由 RandomAccessFile 类指针找到日志文件的起始位置，然后循环读每条 LogTableItem 的所有属性信息并打印出来，直到循环到 currentId 之后退出循环。

下图是加载日志并显示的流程。

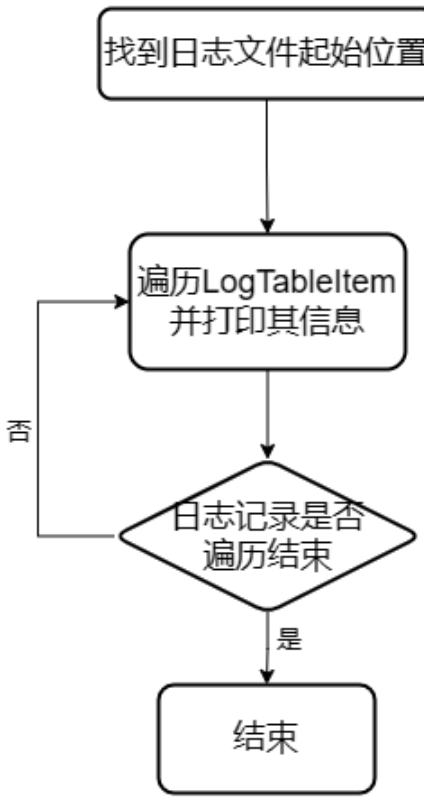


图 4.4 加载日志并显示流程

4.2.5删除旧日志相关文件并初始化新文件

在方法 deleteDirectory(String dir)中删除 dir 目录及下面的文件，具体做法如下：首先检查如果 dir 对应的文件不存在，或者不是一个目录，则退出。然后，设置一个 File 类型的列表，用于存给定目录路径下的所有文件。遍历目录下子文件调用 deleteFile(String fileName)方法，如果文件路径所对应的文件存在，并且是一个文件，则直接删除。最后删除当前目录，若全部删除成功则方法返回 true。

在方法 init()中，首先删除 Map 中所有键值对，调用 deleteDirectory(String dir)，传入参数为日志目录，若日志目录及下面的日志文件和存 B 树文件删除成功，则初始化新的日志目录、日志文件、存 B 树文件，初始化索引 B 树，初始化日志管理成员。

下图是删除旧日志相关文件并初始化新文件的流程。

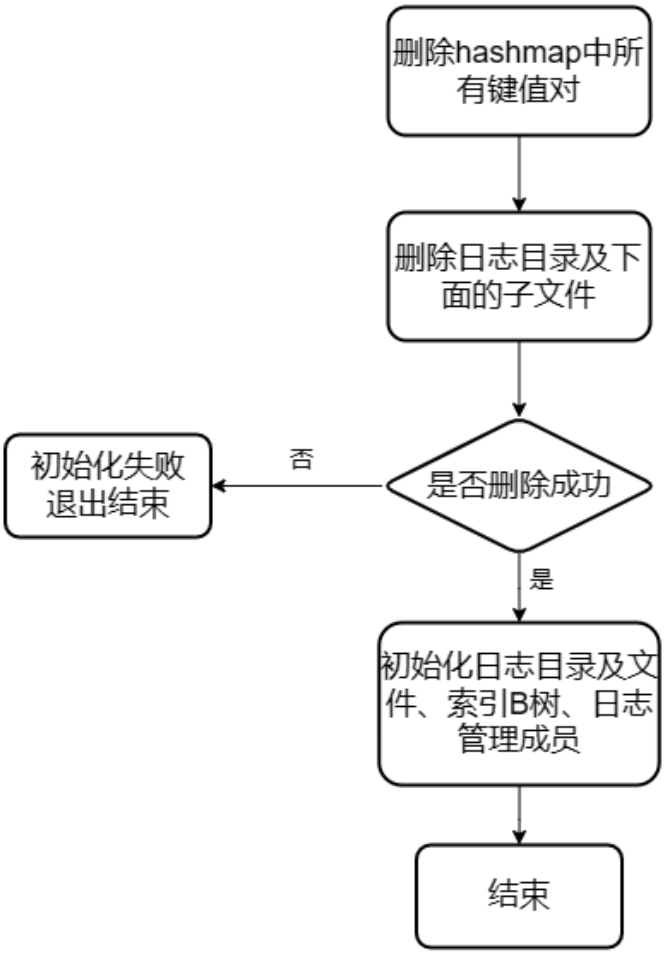


图 4.5 加载日志并显示流程

4.3 利用 B 树实现索引

4.3.1B 树介绍

为了减少磁盘 I/O 以及提高查询日志记录的效率，我们设计了 B 树来实现索引的功能。

B 树是自平衡的树形结构，所以在保存的时候可以按照有序的规则去储存索引信息，这个树形结构每一个节点可以保存多个关键字也就是相当于可以存储多个索引。 这种数据结构能够让查找数据、顺序访问、插入数据及删除的动作，都在对数时间内完成，为系统大块数据的读写操作做了优化。B 树通过减少定位记录时所经历的中间过程，从而加快存取速度。B 树有如下几个特征：

- 根结点最少可以只有 1 个关键字。
- 每个结点最多有 m-1 个关键字。
- 非根结点至少有 Math.ceil(m/2)-1 个关键字。
- 每个结点中的关键字都按照从小到大的顺序排列，每个关键字的左子树中的所有关键字都小于它，而右子树中的所有关键字都大于它。
- 根结点到每个叶子结点的长度都相同。

4.3.2索引 B 树结构设计

索引 B 树类在 BTree_Indexer 文件中。依据日志特征，我们设计的索引 B 树结构如下图所示。

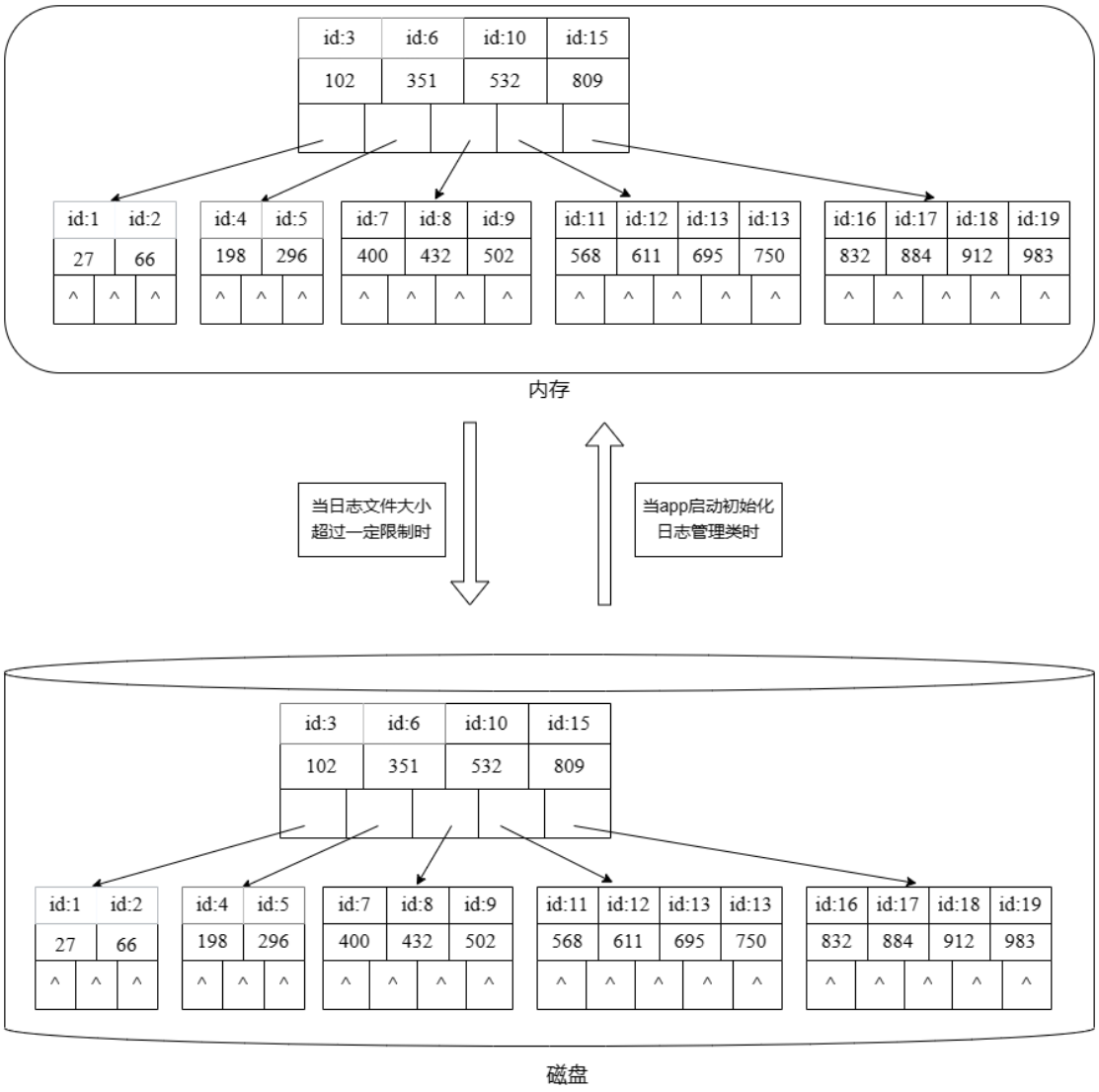


图 4.6 索引 B 树结构

图中的索引 B 树每结点的第一行记录的是日志记录 LogTableItem 最大的 logid，第二行记录的是该 LogTableItem 在日志文件中对应的偏移量 offset，第三行记录的是该结点的孩子结点指针。在此例中 k1-k19 是递增的。索引 B 树首先在内存中初始化，当日志文件大小超过一定限制时，为了防止突然掉电或系统崩溃，将 B 树持久化到磁盘上，并且在 app 每次启动、进行日志管理类初始化时，将 B 树从磁盘重新加载到内存。

4.3.3索引 B 树相关操作

- 初始化索引 B 树：

```
bTree_indexer = new BTree_Indexer<>();
```

- 将 B 树从磁盘加载到内存：

```
//从文件名为 Filename 的文件的 offset 偏移量处开始读 B 树
bTree_indexer=new BTree_Indexer<>(String Filename,long offset);
```

- 将记录 offset 的节点插入 B 树中：

```
//在 B 树中插入键为 key，值为 value 的节点
bTree_indexer.insert(String key,String value);
```

- 将 B 树持久化到磁盘：

```
// BTNodes 的持久化保存, 写到 fileName 偏移为 offset 处，返回值为该节点的长度
bTree_indexer. write(BufferedOutputStream outputStream, long offset)
```

- 利用 B 树进行日志记录的查询：

```
//在 B 树中搜索给定键 key 对应的值
bTree_indexer. search(K key)
```

在方法 LogTableItem searchLog(int logID)中实现寻找指定 logid 的日志在文件中的位置并返回该日志对象，首先调用 B 树查找方法 search 得到 logID 对应日志记录 LogTableItem 在日志文件中的偏移量 offset。将日志文件的 RandomAccess 指针移到该偏移量处，依次读出该 LogTableItem 的 logid、op、key、value、offset 并存入 LogTableItem 对象中，得到并返回该对象。

4.4 崩溃恢复

4.4.1检查点（checkpoint）

1、原理

写日志和写数据文件是数据库中 I/O 消耗最大的两种操作。在这两种操作中写数据文件属于随机写，而写日志文件是顺序写，因此为了保证数据库的性能，通常数据库都是需要在提交完成之前要先保证日志都被写入到日志文件中，而脏数据块先保存在数据缓存（buffer cache）中再不定期的分批写入到数据文件中。

由以上机制来看可以知道，日志写入和提交操作是同步的，而数据写入和提交操作是不同步的。这样就存在一个问题，当一个数据库崩溃的时候并不能保证缓存里面的脏数据全部写入到数据文件中，这样在实例启动的时候就要使用日志文件进行恢复操作，将数据库恢复到崩溃之前的状态，保证数据的一致性。所以，检查点是这个过程中的重要机制，可以通过它来确定，在系统崩溃恢复时哪些重做日志应该被扫描并应用于恢复。

检查点的作用主要是在基于 WAL 日志恢复时避免读取所有的日志记录，同时允许删除一些不必要的日志内容。在检查点位置之前的所有日志中记录过的数据变化都已经全部从缓存区中刷入磁盘，反映到了磁盘文件中，从而在宕机恢复时只需从检查点标记之后读取日志并进行恢复，检查点标记之前的日志已经没有意义，可以删除这些日志磁盘文件。

检查点原理如下图。

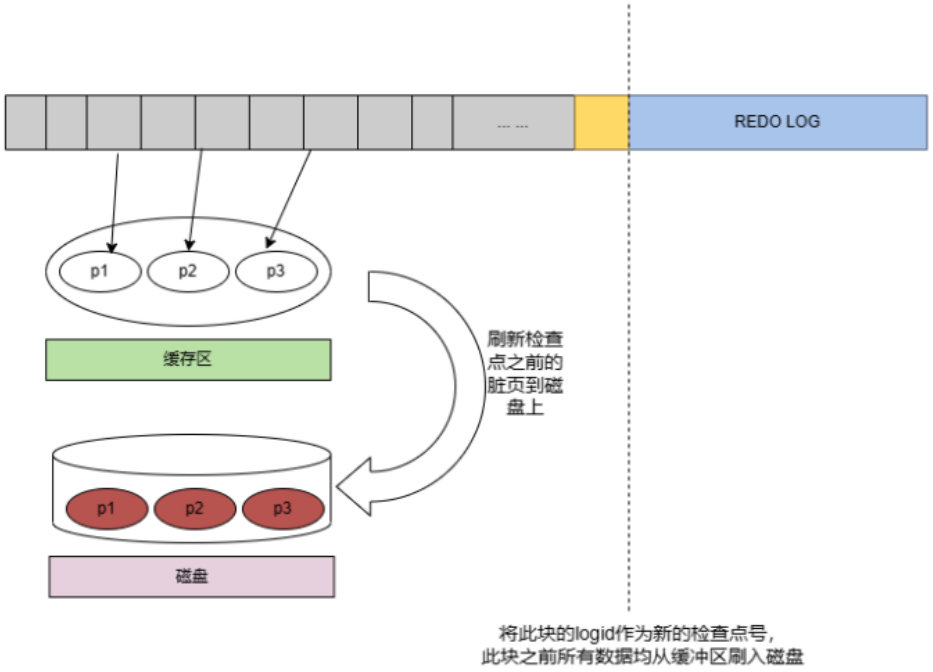


图 4.7 检查点原理

2、实现

当存储模块将内存中的数据全部持久化保存并将内存清空时，调用设置检查点方法。在方法 setCheckpoint()实现设置检查点。

为避免日志文件写满，在每次设置检查点时检查日志文件大小，若超过一定限制则将目前快写满的日志文件删除，因为数据已经持久化到磁盘不需要在此时检查点之前的日志记录了，再调用 init()方法重新初始化新的日志文件及索引 B 树。

若日志文件没有到达限制大小，则设置当前日志记录的 logid 为检查点 checkpoint，设置当前的 offset 为检查点的偏移量即 check_off。

设置检查点流程如下图所示。

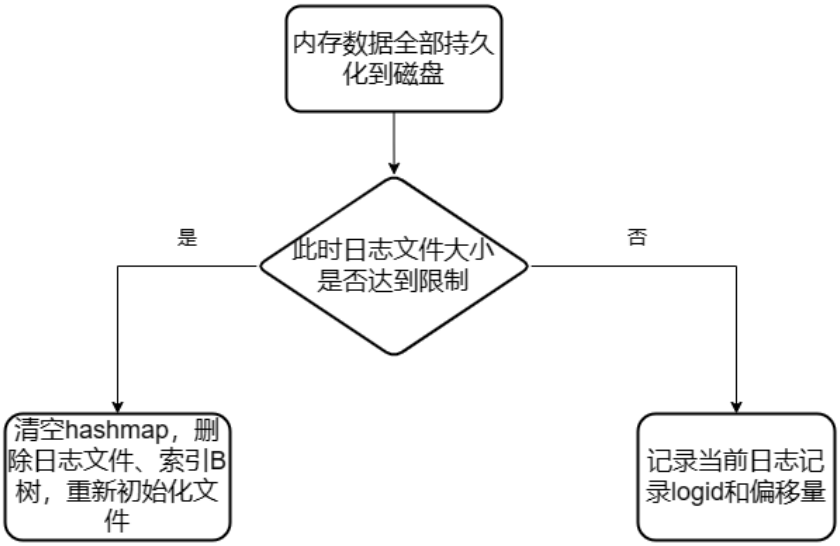


图 4.8 设置检查点

4.4.2Redo

当系统遇到问题而崩溃时，需要查看日志记录进行恢复，把数据库中数据恢复到系统掉电前的那一个时刻，从而使得数据库能正常的启动起来。日志记录了数据库中哪个对象个做了什么操作，当系统崩溃时，虽然脏页数据没有持久化，但是日志记录已经持久化，接着数据库重启后，可以根据日志中的内容进行 redo，将所有数据恢复到最新的状态。结合之前对检查点的描述，可以知道我们需要从检查点之后（即 checkpoint+1）位置的日志块开始读取，并根据读取内容对数据库进行恢复。

首先，方法 LogTableItem[] readRedo()用来加载需要重做的 LogTableItem，新建 LogTableItem 对象数组用于存放需要重做的 LogTableItem。若此时数据库还没有设置检查点，则则从日志文件偏移量为 0 的位置开始读，即从头开始读，需要重做的日志记录数量为 currentId+1，即日志文件中全部日志记录数量；若此时已经设置了检查点，则从检查点的偏移量即 check_off 的位置开始读日志文件，需要重做的日志记录数量为 currentId-checkpoint，即检查点后日志记录数量。根据上述分类依次读文件中的日志记录信息，并存入日志对象数组中。最后返回该数组。

然后，在 redo()方法中根据日志记录重做未保存的数据库操作。调用上述方法 readRedo()得到需要重做的日志记录，遍历该数组，通过读数组元素 LogTableItem 对象的 value 值得到 LogTableItem 记录的数据库中的对象，转化为 JSONObject 形式，然后调用存

储模块的 add()方法重新将该对象加入到数据库中，从而实现数据的恢复。

如下图所示为 redo 的整体流程图。

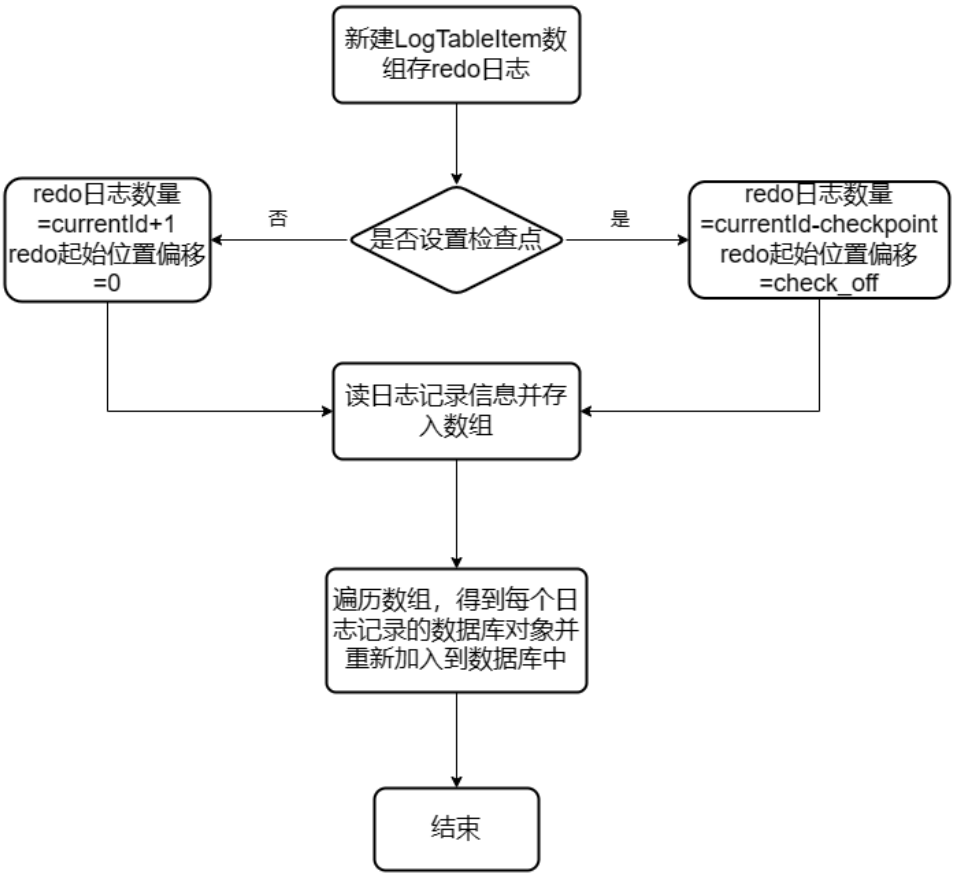


图 4.9 redo 流程图

参考文献

[1] Luo, Chen, and Michael J. Carey. "LSM-based storage techniques: a survey." The VLDB Journal 29.1 (2020): 393-418.

[2] Qader, Mohiuddin Abdul, Shiwen Cheng, and Vagelis Hristidis. "A comparative study of secondary indexing techniques in LSM-based NoSQL databases." Proceedings of the 2018 International Conference on Management of Data. 2018.

[3] Option of Compaction Priority. http://rocksdb.org/blog/2016/01/29/compaction_pri.html, January 29, 2016.

[4] V. Kodaganallur, "Incorporating language processing into Java applications: a JavaCC tutorial," in IEEE Software, vol. 21, no. 4, pp. 70-77, July-Aug. 2004, doi: 10.1109/MS.2004.16.

[5] GitHub. “Home · JSQLParser/JSqlParser Wiki.” https://github.com/JSQLParser/JSqlParser.

[6] B. Zhai, Y. Shi and Z. Peng, "Object Deputy Database Language," Fourth International Conference on Creating, Connecting and Collaborating through Computing (C5'06), Berkeley, CA, USA, 2006, pp. 88-95, doi: 10.1109/C5.2006.27.

[7] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD '20). Association for Computing Machinery, New York, NY, USA, 877–892. https://doi.org/10.1145/3318464.3389716

[8] Youjip Won, Sundoo Kim, Juseong Yun, Dam Quang Tuan, and Jiwon Seo. 2019. DASH: database shadowing for mobile DBMS. Proc. VLDB Endow. 12, 7 (March 2019), 793–806. https://doi.org/10.14778/3317315.3317321

[9] Leon Lee, Siphrey Xie, Yunus Ma, and Shimin Chen. 2022. Index checkpoints for instant recovery in in-memory database systems. Proc. VLDB Endow. 15, 8 (April 2022), 1671–1683. https://doi.org/10.14778/3529337.3529350