# 背景

RocksDB是FaceBook起初作为实验性质开发的，旨在充分实现快存上存储数据的服务能力。由Facebook的Dhruba Borthakur于2012年4月创建的LevelDB的分支，最初的目标是提高服务工作负载的性能，最大限度的发挥闪存和RAM的高度率读写性能。

Key和value是任意大小的字节流支持原子的读和写。除此外，RocksDB深度支持各种配置，可以在不同的生产环境（纯内存、Flash、hard disks or HDFS）中调优，RocksDB针对多核CPU、高效快速存储（SSD)、I/O bound workload做了优化，支持不同的数据压缩算法、和生产环境debug的完善工具。

RocksDB的主要设计点是在快存和高服务压力下性能表现优越，所以该db需要充分挖掘Flash和RAM的读写速率。 例如360开源的Pika， 是由360 DBA 和基础架构组联合开发的类 Redis 存储系统，完全支持 Redis 协议，用户不需要修改任何代码，就可以将服务迁移至 Pika，Pika底层存储引擎用的就是Rocksdb。

RocksDB需要支持高效的point lookup和range scan操作，需要支持配置各种参数在高压力的随机读、随机写或者二者流量都很大时性能调优，基于LSM树数据结构( log-structured merge-tree)，由C++编写并官方提供C、C++、Java(官方提供的称为RocksJava)三种语言的API，社区提供了不少第三方API，如python、go等。

尽管RocksDB不是一个SQL 数据库，但是有facebook有修改了代码的MyRocks存储引擎作为MySQL的存储引擎。和其他的NoSQL类似，RocksDB不提供关系型数据模型、不支持SQL查询，没有直接对辅助索引(secondary indexes)支持。

## 写多读少

## 原地写/追加写

在基于磁盘的 kv 型存储组件中，针对于写操作的实现方案包括原地写和追加写两种，其区别主要体现在更新数据的操作流程当中：

* 原地写：

倘若要针对一组 kv 数据执行更新操作，首先要找到 kv 老数据的所在位置，再在其基础之上执行进行更新。这个过程涉及到磁盘的随机IO，因此性能较差。

与之相对，在执行读操作时，可以根据 k 寻找到 kv 数据所在位置并直接拿到查询结果，因此读操作效率相对较高，且具有着不错的空间利用率。

* 追加写：

追加写类型的写操作中，无须区分本次写操作是插入还是更新，而是选择将 kv 对以追加的形式直接插入到文件的末尾位置。因此不涉及磁盘的随机 IO，只需要执行顺序 IO 操作，在写流程中的执行性能相较于原地写而言有较大的提升。

然而大家应该也注意到了，追加写策略在提升写操作效率的同时，所付出的代价是导致同一组 kv 对可能产生多份冗余数据，而除了最新记录外，此前的数据记录实际上都是无用的，因此会存在空间浪费的情况。

也正是因为这一原因，追加写策略下的读流程性能是比较差的。每次根据 k 查询数据时，都需要沿文件末尾向前反向遍历追溯，直到找到第一笔满足条件的 kv 对数据为止。在这种模式下，查询操作变成了线性时间复杂度，是无法接受的。

# 概述

参考：

<https://blog.csdn.net/weixin_43618070/article/details/102317769>

<https://www.cnblogs.com/xueqiuqiu/articles/9700198.html>

<https://www.jianshu.com/nb/48664459>

mongorocks引擎原理解析：<http://mysql.taobao.org/monthly/2018/04/02/>

WAL介绍：<http://mysql.taobao.org/monthly/2018/04/09/>

RocksDB原理：<https://zhuanlan.zhihu.com/p/616209332?utm_id=0>

## 简介

[RocksDB](https://github.com/facebook/rocksdb) 是由Facebook基于LevelDB开发的一款提供键值存储与读写功能的LSM-tree架构引擎（嵌入式存储引擎）。用户写入的键值对会先写入磁盘上的WAL (Write Ahead Log)，然后再写入内存中的跳表（SkipList，这部分结构又被称作MemTable）。LSM-tree引擎由于将用户的随机修改（插入）转化为了对WAL文件的顺序写，因此具有比B树类存储引擎更高的写吞吐。

内存中的数据达到一定阈值后，会刷到磁盘上生成SST文件（Sorted String Table），SST又分为多层（默认至多6层），每一层的数据达到一定阈值后会挑选一部分SST合并到下一层，每一层的数据是上一层的10倍（因此90%的数据存储在最后一层）。

Rocksdb目前已经运用在许多知名的项目中，例如TiKV，MyRocks，CrockRoach等。

说明：RocksDB作为高效发挥硬件存储性能的高性能嵌入式数据库，与其他数据库不同的是，不能提供网络服务，只负责存储服务。RocksDB有针对硬件的一些优化。

## 高性能/高可靠

RocksDB使用了许多技术来实现其高性能和可靠性，下面是一些主要的技术：

### LSM树

LSM树：LSM树（Log-Structured Merge Tree）是一种基于日志结构的数据结构，能够高效地存储和更新键值数据。它将数据分为多个层，每一层都是一个有序的键值存储文件，其中较旧的数据位于较低的层，较新的数据位于较高的层。当数据被写入时，它首先被写入到一个内存中的结构，称为内存表（MemTable），然后在后台异步地将内存表与磁盘上的某个层合并，最终生成新的文件。这种设计使得RocksDB能够高效地处理大量写入操作，并支持快速的范围查询。

### 压缩

压缩：RocksDB使用了多种压缩算法来压缩数据文件，减小了磁盘空间的占用，提高了存储效率。压缩算法包括LZ4、Snappy、Zlib等。

### 并发控制

并发控制：RocksDB使用多种技术来实现并发控制，以支持高并发读写操作。例如，它使用锁、读写锁、CAS等机制来保证多线程并发的正确性和一致性。

### 内存管理

内存管理：RocksDB使用了多种技术来管理内存，以保证高效的内存使用和低延迟的响应。例如，它使用了对象池、内存池等技术来减少内存分配和释放的开销，使用了缓存技术来缓存热点数据，使用了内存映射技术来快速加载数据文件等。

### 日志系统

日志系统：RocksDB使用了可插拔的日志系统，可以将日志输出到不同的目标，例如文件、控制台、网络等，以支持不同的日志需求。

### 文件格式

文件格式：RocksDB使用了一种自定义的文件格式，可以高效地存储键值数据，并支持快速的数据访问和查询。这种格式将数据划分为多个块，每个块包含多个键值对，每个块都有一个索引来支持快速的查找和范围查询。

说明：这里自定义的文件格式，包括内存memtable的分层和跳表数据结构设计，磁盘sstable的分层和数据结构设计。

## 功能特性

### ColumnFamily

RocksDB允许用户创建多个ColumnFamily ，这些ColumnFamily各自拥有独立的内存跳表以及SST文件，但是共享同一个WAL文件，这样的好处是可以根据应用特点为不同的ColumnFamily选择不同的配置，但是又没有增加对WAL的写次数。

写操作先写WAL，再写memtable，memtable达到一定阈值后切换为Immutable Memtable，只能读不能写。后台Flush线程负责按照时间顺序将Immu Memtable刷盘，生成level0层的有序文件（SST）。后台合并线程负责将上层的SST合并生成下层的SST。Manifest负责记录系统某个时刻SST文件的视图，Current文件记录当前最新的Manifest文件名。每个ColumnFamily有自己的Memtable，SST文件，所有ColumnFamily共享WAL、Current、Manifest文件，用户可以基于RocksDB构建自己的column families。很多应用程序把RocksDB当做库（libary），尽管他提供server或者CLI接口。

column family的具体使用场景，官方并没有给出非常好的说明，rocksdb支持跨column family的原子写操作，说明官方认为column family的数据之间还是有一定的关系的，而column family的数据文件是分割开的，包括sst文件和memtable都是不会共用的，所以官方还是希望不同column family的数据进行隔离。那么不同column family之间会怎样互相干涉呢？

Q：多column family的flush是怎么进行线程分布的？

A：可以共用一个线程池，所以写的多的会多占用线程。也可以分开配置线程池（指定不同的env），这样就会不互相干扰。

Q：多column family的内存是怎么分配的？

A：内存主要有两方面，第一方面是write buffer，第二方面是block cache。

write buffer是每个column family单独享有的，block cache可以配置成column family级别，也可以配置成整个db的column family共享一个block cache，也可以配置成多个db共享一个block cache。

参考：<https://www.cnblogs.com/xueqiuqiu/articles/10111070.html>

MANIFEST文件介绍：<http://mysql.taobao.org/monthly/2018/05/08/>

### Memtable

可插拔memtable，RocksDB的memtable的默认实现是一个skiplist。skiplist是一个有序集，当工作负载使用range-scans并且交织写入时，这是一个必要的结构。然而，一些应用程序不交织写入和扫描，而一些应用程序根本不执行范围扫描。对于这些应用程序，排序集可能无法提供最佳性能。因此，RocksDB支持可插拔的API，允许应用程序提供自己的memtable实现。

开发库提供了三个memtable：skiplist memtable，vector memtable和前缀散列（prefix-hash）memtable。Vector memtable适用于将数据批量加载到数据库中。每个写入在向量的末尾插入一个新元素; 当它是刷新memtable到存储的时候，向量中的元素被排序并写出到L0中的文件。前缀散列memtable允许对gets，puts和scans-within-a-key-prefix进行有效的处理。

### SSTFile/SSTTable

RocksDB在磁盘上的file结构sstfile由block作为基本单位组成，一个sstfile结构由多个data block和meta block组成，其中data block就是数据实体block，meta block为元数据block，其中data block就是数据实体block，meta block为元数据block。

sstfile组成的block有可能被压缩（compression），不同level也可能使用不同的compression方式。sstfile如果要遍历block，会逆序遍历，从footer开始。RocksDB是一个嵌入式的K-V（任意字节流）存储。所有的数据在引擎中是有序存储，可以支持Get(key)、Put（Key）、Delete（Key）和NewIterator()。RocksDB的基本组成是memtable、sstfile和logfile。memtable是一种内存数据结构，写请求会先将数据写到memtable中，然后可选地写入事务日志logfile。logfile是一个顺序写的文件。当内存表溢出的时候，数据会flush到sstfile中，然后这个memtable对应的logfile也会安全地被删除。sstfile中的数据也是有序存储以方便查找。

## 特点

<https://zhuanlan.zhihu.com/p/162052214>

### 优点

1. LevelDB是一个持久化存储的KV系统，和Redis这种内存型的KV系统不同，LevelDB不会像Redis一样狂吃内存，而是将大部分数据存储到磁盘上。
2. LevleDB在存储数据时，是根据记录的key值有序存储的，就是说相邻的key值在存储文件中是依次顺序存储的，而应用可以自定义key大小比较函数。
3. LevelDB支持数据快照（snapshot）功能，使得读取操作不受写操作影响，可以在读操作过程中始终看到一致的数据。
4. LevelDB还支持数据压缩等操作，这对于减小存储空间以及增快IO效率都有直接的帮助。

### 缺点

rocksdb是通过wal来保证数据的持久性的，当rocksdb出现问题当机后，可以通过重做wal来使rocksdb恢复到当机前的状态。但是这里存在两个问题：

1、当你为了读性能把memtable设置的足够大时，WAL也可能变得很大（Flush频率下降），此时如果发生当机，rocksdb需要足够长的时间来恢复。

2、如果机器硬盘出现损坏，wal被破坏，那么会出现数据损坏（这里没有double write机制？）。即使你做了Raid，也需要很长时间来恢复数据。

因此可用性是个大问题。

另外由于LSM架构，rocksdb的读性能存在问题。

怎么解决？

为了一致性，大部分的解决方案都是将一致性协议置于rocksdb之上，每份数据通过一致性协议提交到多个处于不同机器的rocksdb实例中，以保证数据的可靠性和服务的可用性。

## 对比/优化点

### RocksDB vs LevelDB

RocksDB与LevelDB对比：

1. 增加了column family，这样有利于多个不相关的数据集存储在同一个db中，因为不同column family的数据是存储在不同的sst和memtable中，所以一定程度上起到了隔离的作用。
2. 采用了多线程同时进行compaction的方法，优化了compact的速度。
3. 增加了merge operator，优化了modify的效率。
4. 将flush和compaction分开不同的线程池，能有效的加快flush，防止stall。
5. 增加了对write ahead log(WAL)的特殊管理机制，这样就能方便管理WAL文件，因为WAL是binlog文件。

### RocksDB vs MyRocks

# 基础概念

参考：

<https://alexstocks.github.io/html/rocksdb.html>

<https://github.com/facebook/rocksdb/wiki>

## LSM与WriteBatch

## 同步写与异步写

默认情况下，RocksDB 的写是异步的：仅仅把数据写进了操作系统的缓存区就返回了，而这些数据被写进磁盘是一个异步的过程。如果为了数据安全，可以用如下代码把写过程改为同步写：

rocksdb::WriteOptions write\_options;

write\_options.sync = true;

db->Put(write\_options, …);

这个选项会启用 Posix 系统的 fsync(...) or fdatasync(...) or msync(..., MS\_SYNC) 等函数。

异步写的吞吐率是同步写的一千多倍。异步写的缺点是机器或者操作系统崩溃时可能丢掉最近一批写请求发出的由操作系统缓存的数据，但是 RocksDB 自身崩溃并不会导致数据丢失。而机器或者操作系统崩溃的概率比较低，所以大部分情况下可以认为异步写是安全的。

RocksDB由于有WAL机制保证，所以即使崩溃，其重启后会进行写重放保证数据一致性。如果不在乎数据安全性，可以把write\_option.disableWAL 设置为 true，加快写吞吐率。

RocksDB调用Posix API fdatasync()对数据进行异步写。如果想用fsync()进行同步写，可以设置Options::use\_fsync为true。

## Column Family

CF提供了对DB进行逻辑划分开来的方法，用户可以通过CF同时对多个 CF 的KV进行并行读写的方法，提高了并行度。

## Wild Columns

参考：

<https://www.ctyun.cn/developer/article/493557684301893>

RocksDB 过去一直是纯键值存储，从存储引擎的角度来看，键和值是不透明且非结构化的字节数组。这对许多应用程序来说都很有效；但是，我们也有很多用例，其中数据具有一些内部结构。为了更好地服务于 RocksDB 中的此类用例，我们最近扩展了我们的数据模型以支持宽列。宽列模型在纯键值模型和关系模型之间提供了一个中间立场：多个命名列可以与一个键相关联，就像在关系数据库中一样，但是，与关系模型相反，列集可能因行而异。这些特性使宽列非常适合存储半结构化数据。

RocksDB从8.0开始推出了Wide Columns特性，支持半结构化的数据存储，不再只是单纯的key-value存储。wide-columns模型在kv存储和关系数据库之间起到了桥接作用。

WideColumn的特性具体来说就是把一个value拆分成多个属性记录，value为一组column的组合，在value中会同时记录column\_name和column\_value信息。在内部实现上，写入时会使用WideColumnSerialization::Serialize接口把一组WideColumns序列化为string落盘，读取时则使用WideColumnSerialization::Deserialize接口反序列化为WideColumns输出。

## BlobDB

BlobDB本质上是用于大值用例的RocksDB。WiscKey论文中提出的基本思想是键值分离：通过将大值存储在专用的blob文件中，并在LSM树中仅存储指向它们的小指针，我们避免在压缩期间一遍又一遍地复制值。这减少了写入放大，具有几个潜在好处，例如延长SSD寿命以及更好的写入和读取性能。另一方面，由于存在LSM树不再引用的blob，因此需要进行垃圾回收，因此这会带来一些空间放大的成本。

## User-defined Timestamp

应用程序可以为按键分配自定义格式的时间戳。

是否启用用户定义时间戳是列族设置。

在当前实现中，用户定义的时间戳存储在用户密钥和序列号之间，如下所示。

|user key|timestamp|seqno|type|

|<-------internal key-------->|

## Iterator

参考：

<https://blog.csdn.net/Z_Stand/article/details/111712716>

数据库中的所有数据都按逻辑顺序排列。应用程序可以指定键比较方法，该方法指定键的全序。迭代器 API 允许应用程序对数据库进行范围扫描。迭代器可以查找指定的键，然后应用程序可以从该点开始一次扫描一个键。迭代器 API 还可用于对数据库中的键进行反向迭代。创建迭代器时，会创建数据库的一致时间点视图。因此，通过迭代器返回的所有键都来自数据库的一致视图。

### Prefix Iterators

许多LSM引擎不支持高效的RangeScan操作，因为Range操作需要扫描所有的数据文件。一般情况下常规的技术手段是给key建立索引，只用遍历key就可以了。应用可以通过确认prefix\_extractor指定一个可以的前缀，RocksDB可以为这些key prefix建立Bloom索引，以加快查询速度。

### Prefix Seek

### Multi Column Family Iterator

### Tailing Iterator

### SeekForPrev

## Purging WAL files

一般情况下，RocksDB会删除一些过时的WAL文件，所谓过时就是WAL文件里面对应的一些key的数据已经被固化到磁盘了。但是RocksDB提供了两个选项以实让用户控制WAL何时删除：Options::WAL\_ttl\_seconds和Options::WAL\_size\_limit\_MB，这两个参数分别控制WAL文件的超时时间和最大文件 size。

如果这两个值都被设置为0，则log不会被固化到文件系统上。

如果Options::WAL\_ttl\_seconds为0而Options::WAL\_size\_limit\_MB不为0，RocksDB会每10分钟检测所有的WAL文件，如果其总体size超过 Options::WAL\_size\_limit\_MB，则RocksDB会删除最早的日志直到满足这个值位置。一切空文件都会被删除。

如果Options::WAL\_ttl\_seconds不为0而Options::WAL\_size\_limit\_MB为0，RocksDB会每Options::WAL\_ttl\_seconds/2检测一次WAL文件，所有TTL超过Options::WAL\_ttl\_seconds的WAL文件都会被删除。

如果两个值都不为0，RocksDB会每10分钟检测所有的WAL文件，所有不满足条件的WAL文件都会被删除，其中ttl参数优先。

## MemTable

### Slice & PinnableSlice

参考：

<https://gukaifeng.cn/posts/rocksdb-slice-lei/index.html>

RocksDB数据的读取：<http://mysql.taobao.org/monthly/2018/11/05/>

### MemTable and Table factories

RocksDB内存中的数据格式是skiplist，磁盘则是以table形式存储的SST文件格式。

table 格式有两种：继承自leveldb的文件格式和PlainTable格式。PlainTable 格式是针对低查询延迟或者低延迟存储媒介如SSD特别别优化的一种文件格式。

### Block size

RocksDB把相邻的key放到同一个block中，block是数据存储和传递的基本单元。默认Block的大小是4096B，数据未经压缩。

经常进行bulk scan操作的用户可能希望增大block size，而经常进行单key 读写的用户则可能希望减小其值，官方建议这个值减小不要低于1KB的下限，变大也不要超过a few megabytes。启用压缩也可以起到增大block size的好处。

修改Block size的方法是修改 Options::block\_size。

### Writer Buffer

Options::write\_buffer\_size指定了一个写内存buffer的大小，当这个buffer写满之后数据会被固化到磁盘上。这个值越大批量写入的性能越好。

RocksDB控制写内存buffer数目的参数是 Options::max\_write\_buffer\_number。这个值默认是2，当一个buffer的数据被 flush 到磁盘上的时候，RocksDB就用另一个 buffer作为数据读写缓冲区。

Options::minwritebuffernumberto\_merge设定了把写buffer的数据固化到磁盘上时对多少个buffer的数据进行合并然后再固化到磁盘上。这个值如果为1，则L0层文件只有一个，这会导致读放大，这个值太小会导致数据固化到磁盘上之前数据去重效果太差劲。

这两个值并不是越大越好，太大会延迟一个DB被重新打开时的数据加载时间。

### Key Layout

在前面提到“block是数据存储和传递的基本单元”，RocksDB的数据是一个range的key-value构成一个Region，根据局部性原理每次访问一个Region的key的时候，有很多概率会访问其相邻的key，每个Region的keys放在一个block里，多个Region的keys放在多个block里。

下面以文件系统作为类比，详细解释下RocksDB的文件系统：

​ filename -> permission-bits, length, list of fileblockids

​ fileblockid -> data

以多个维度组织key的时候，我们可能希望filename的前缀都是‘/‘，而 fileblockid的前缀都是‘0’，这样可以把他们分别放在不同的block里，以方便快速查询。

### Automic flush

### Compaction Filter

RocksDB 提供了一种在后台根据自定义逻辑删除或修改键/值对的方法。它对于实现自定义垃圾收集非常方便，例如根据 TTL 删除过期的键或在后台删除一定范围的键。它还可以更新现有键的值。

### Multi-Threaded Compactions

如果启用Level Style Compaction, L0存储着RocksDB最新的数据，Lmax存储着比较老的数据，L0里可能存着重复keys，但是其他层文件则不可能存在重复key。每个compaction任务都会选择Ln层的一个文件以及与其相邻的Ln+1层的多个文件进行合并，删除过期或者标记为删除 或者 重复的key，然后把合并后的文件放入Ln+1层。Compaction过程会导致写放大【如写qps是10MB/s，但是实际磁盘io是50MB/s】效应，但是可以节省空间并减少读放大。

如果启用Universal Style Compaction，则只压缩L0的所有文件，合并后再放入L0层里。

RocksDB的compaction任务线程不宜过多，过多容易导致写请求被hang住。

### Read-Modify-Write（Merge）Operator

### Thread Pool

RocksDB会创建一个thread pool与Env对象进行关联，线程池中线程的数目可以通过Env::SetBackgroundThreads()设定。通过这个线程池可以执行compaction与memtable flush任务。

当memtable flush和compaction两个任务同时执行的时候，会导致写请求被hang住。RocksDB建议创建两个线程池，分别指定HIGH和LOW两个优先级。默认情况下HIGH线程池执行memtable flush任务，LOW线程池执行compaction任务。

相关代码示例如下：

#include “rocksdb/env.h”

#include “rocksdb/db.h”

auto env = rocksdb::Env::Default();

env->SetBackgroundThreads(2, rocksdb::Env::LOW);

env->SetBackgroundThreads(1, rocksdb::Env::HIGH);

rocksdb::DB\* db;

rocksdb::Options options;

options.env = env;

options.max\_background\_compactions = 2;

options.max\_background\_flushes = 1;

rocksdb::Status status = rocksdb::DB::Open(options, “/tmp/testdb”, &db);

assert(status.ok());

还有其他一些参数。

## SST

### Creating and Ingesting SST files

### Bloom Filter

RocksDB的每个SST文件都包含一个Bloom filter。Bloom Filter只对特定的一组keys有效，所以只有新的SST文件创建的时候才会生成这个filter。当两个SST文件合并的时候，会生成新的filter数据。

当SST文件加载进内存的时候，filter也会被加载进内存，当关闭SST文件的时候，filter也会被关闭。如果想让filter常驻内存，可以用如下代码设置：

BlockBasedTableOptions::cache\_index\_and\_filter\_blocks=true

一般情况下不要修改filter相关参数。如果需要修改，相关设置上面已经说过，此处不再多谈。

## Write

### Single Delete

### Low Priority Write

### Time to Live/TTL

RocksDB在进行compact的时候，会删除被标记为删除的数据，会删除重复key的老版本的数据，也会删除过期的数据。数据过期时间由API DBWithTTL::Open(const Options& options, const std::string& name, StackableDB\*\* dbptr, int32\_t ttl = 0, bool read\_only = false)的ttl参数设定。

TTL的使用有以下注意事项：

1、TTL其值单位是秒，如果TTL值为0或者负数，则TTL值为infinity，即永不过期；

2、每个kv被插入数据文件中的时候会带上创建时的机器(int32\_t)Timestamp 时间值；

3、compaction时如果kv满足条件Timestamp+ttl<time\_now，则会被淘汰掉；

4、Get/Iterator的时候可能返回过期的kv（compact任务还未执行）；

5、不同的DBWithTTL::Open可能会带上不同的TTL值，此时kv以最大的TTL值为准；

6、如果DBWithTTL::Open的参数read\_only为true，则不会触发compact任务，不会有过期数据被删除。

### DeleteRange

## Transaction

### Transactions

当使用TransactionDB或者OptimisticTransactionDB的时候，可以使用RocksDB的BEGIN/COMMIT/ROLLBACK等事务API。RocksDB支持活锁或者死等两种事务。

WriteBatch默认使用了事务，确保批量写成功。

当打开一个TransactionDB的时候，如果RocksDB检测到某个key已经被别的事务锁住，则RocksDB会返回一个error。如果打开成功，则所有相关key都会被lock住，直到事务结束。TransactionDB的并发特性表现要比OptimisticTransactionDB好，但是TransactionDB的一个小问题就是不管写发生在事务里或者事务外，他都会进行写冲突检测。TransactionDB使用示例代码如下：

TransactionDB\* txn\_db;

Status s = TransactionDB::Open(options, path, &txn\_db);

Transaction\* txn = txn\_db->BeginTransaction(write\_options, txn\_options);

s = txn->Put(“key”, “value”);

s = txn->Delete(“key2”);

s = txn->Merge(“key3”, “value”);

s = txn->Commit();

delete txn;

OptimisticTransactionDB提供了一个更轻量的事务实现，它在进行写之前不会进行写冲突检测，当对写操作进行commit的时候如果发生了lock冲突导致写操作失败，则RocksDB会返回一个error。这种事务使用了活锁策略，适用于读多写少这种写冲突概率比较低的场景下，使用示例代码如下：

DB\* db;

OptimisticTransactionDB\* txn\_db;

Status s = OptimisticTransactionDB::Open(options, path, &txn\_db);

db = txn\_db->GetBaseDB();

OptimisticTransaction\* txn = txn\_db->BeginTransaction(write\_options, txn\_options);

txn->Put(“key”, “value”);

txn->Delete(“key2”);

txn->Merge(“key3”, “value”);

s = txn->Commit();

delete txn;

### Snapshots

RocksDB能够保存某个版本的所有数据（可称之为一个Snapshot）以方便读取操作，创建并读取Snapshot方法如下：

rocksdb::ReadOptions options;

options.snapshot = db->GetSnapshot();

… apply some updates to db ….

rocksdb::Iterator\* iter = db->NewIterator(options);

… read using iter to view the state when the snapshot was created ….

delete iter;

db->ReleaseSnapshot(options.snapshot);

如果ReadOptions::snapshot为null，则读取的snapshot为RocksDB当前版本数据的snapshot。

## Read-only and Secondary instances

## Approximate Sizes

GetApproximateSizes方法可以返回一个key range的磁盘占用空间大致使用量，示例代码如下：

rocksdb::Range ranges[2];

ranges[0] = rocksdb::Range(“a”, “c”);

ranges[1] = rocksdb::Range(“x”, “z”);

uint64\_t sizes[2];

rocksdb::Status s = db->GetApproximateSizes(ranges, 2, sizes);

上面的sizes[0]返回 [a..c) key range的磁盘使用量，而sizes[1]返回 [x..z) key range的磁盘使用量。

## Online Verification

## Checksums

Rocksdb对每个kv以及整体数据文件都分别计算了checksum，以进行数据正确性校验。下面有两个选项对checksum的行为进行控制。

ReadOptions::verify\_checksums强制对每次从磁盘读取的数据进行校验，这个选项默认为true。

Options::paranoid\_checks这个选项为true的时候，如果RocksDB打开一个数据检测到内部数据部分错乱，马上抛出一个错误。这个选择默认为false。

如果RocksDB的数据错乱，RocksDB会尽量把它隔离出来，保证大部分数据的可用性和正确性。

## Incremental Backups and Replication

RocksDB的API GetUpdatesSince可以让调用者从transaction log获知最近被更新的key（原文意为用tail方式读取transaction log），通过这个API可以进行数据的增量备份。

RocksDB在进行数据备份时候，可以调用API DisableFileDeletions停止删除文件操作，调用API GetLiveFiles/GetSortedWalFiles以检索活跃文件列表，然后进行数据备份。备份工作完成以后在调用API EnableFileDeletions让RocksDB再启动过期文件淘汰工作。

# 原理

<https://www.zhihu.com/question/270732348/answer/356254676>

<https://alexstocks.github.io/html/rocksdb.html>

## LSM设计思想

首先明确目标，我们需要解决的是写多读少的使用场景，在此之上提出的核心诉求包括：

* 由于写操作是最核心的使用场景，因此需要以提高写效率作为首要目标
* 读效率可以适当牺牲，但也需要保证在有限的容忍度范围以内

在此基础之上，我们很自然地想到使用追加写策略是较优的选择。于是我们需要考虑的问题就是，在传承顺序写所带来的优势的同时，如何尽可能去规避或者弱化其存在于读流程以及空间利用率方面的劣势。

### 追加写存在的问题

首先再梳理一下追加写（顺序写）策略所存在的问题：

* 数据冗余（空间浪费）

顺序写无视 k 此前的存在状况，简单粗暴地追加的方式完成数据的写入或者更新操作，因此不可避免地存在一组 kv 对对应多份冗余记录的情况。并且，除了最后一笔记录之外，此前多笔老数据都属于无用的冗余数据. 考虑一个最极端的场景，在对顺序写策略不执行任何优化改进的前提下，只需要一笔kv对数据，采用无限次写操作即可打满磁盘空间。

* 读性能低

顺序写模式下，执行一次查询操作需要反向追溯，直到找到第一笔满足条件的 kv 数据记录才能返回。因此，其最坏的时间复杂度是线性的 O(N)，显然无法满足使用诉求。

明确了存在的问题后，接下来我们就以这两项核心问题为主线展开推演和优化过程，尝试一步步构建出 lsm tree 的完整架构。

### 数据合并

首先考虑如何解决数据冗余的问题。

既然同一组 kv 对可能存在多份冗余数据，那我们自然能想到采用压缩合并的方式来解决这一问题。

于是，在读写主流程之外，我们可以异步启动一个负责压缩合并的线程，持续对重复的 kv 对数据进行合并，只保留最新记录，消除此前无用的老数据。

但是一旦这样做，新的问题也就随之产生了。这是因为针对同一文件的写入操作和压缩操作是需要互斥的，否则可能产生一系列严重的并发问题。

然而，我们一旦采取了互斥操作，那么在文件压缩合并期间，写入操作就会被阻塞，这样会严重影响到使用性能。

### 文件分块

在前面数据合并的基础之上，我们进一步采取的对策是对文件进行拆分。

倘若我们把一个大文件拆分为一系列的小文件（table），每次写操作只会追加到最新的new table中，而异步的压缩合并操作则只面向于老文件old table，这样两个流程之间就能够实现解耦，写操作不再有陷入阻塞的风险，同时压缩操作和写操作之间也能存在清晰明确的界限。

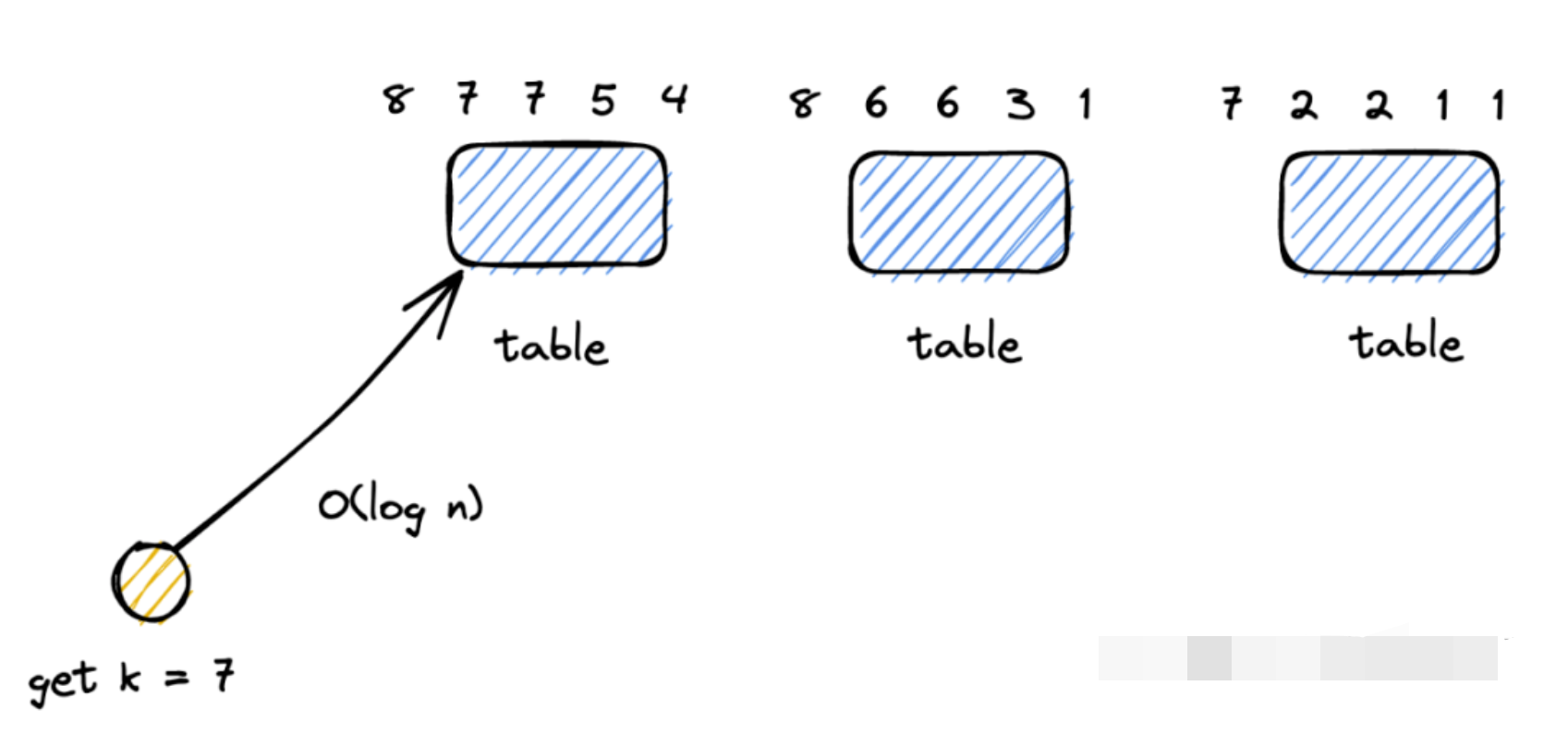
### 数据有序存储

数据冗余的优化探讨我们暂时先告一段落，接下来我们聊聊顺序写存在的另一个问题：很差的读性能。

目前现状是，每查询一个k，需要在全量数据范围内执行逆序遍历操作，直到找到满足条件的第一笔 kv 对记录位置。

要想降低读流程的时间复杂度，就需要在 kv 对的存储组织结构上做做文章。

这里我们可以采取的策略是，在组织每个 table 内的数据时，事先根据 k 进行数据排序，那么在查询环节中，我们就无需承受线性遍历的代价，而是能通过二分的方式在对数级的时间复杂度下获得我们想要的结果（即跳表的设计）。



但是大家需要注意，一旦 kv 数据在 table 中的存放顺序有了限制，这一点其实就和我们探讨这一问题的前提条件自相矛盾，已经打破了追加写策略的基础设定。

### 内存+磁盘

为了解决数据有序存储中遍历查找的问题，我们在磁盘文件disktable的基础之上，额外引入在内存中维护的memtable结构。

说明：提高查询的速率，肯定不是在磁盘上进行遍历查找，因此引入基于内存的数据结构memtable，并且这个结构需要是有序的，并且可以实现就地写，因为在内存中，可以容忍就地写带来的性能波动。

首先明确几个事项：

* memtable 在内存中缓存
* memtable 本身基于k进行有序存储
* memtable 中的写操作统一采用就地写
* memtable 是用户写操作的唯一入口
* memtable 数据量达到阈值后溢写到磁盘，成为disktable

重要的事情，我们再次强调一遍，**内存中的 memtable 采用就地写操作，而非追加写**。因为数据是维护在内存中的，因此哪怕写操作需要承受随机 IO 的代价，也处在可以接受的范围以内。

此外，当memtable中的数据量达到阈值后，再一次性flush到磁盘中成为 disktable。这个过程中实际上是以table为粒度，在磁盘中执行了我们所谓的“追加写”操作。

这样做还带来的一点好处是，由于所有的disk table都是由内部有序的memtable生成的，因此能做到disktable文件内部天然就是局部有序的。

然而，引入缓存之后，由此也引发出了三个新的问题：

* 内存是易失性存储，倘若 memtable 在溢写磁盘前就宕机，那么导致的数据丢失问题如何解决？

说明：预写（WAL）日志，将memtable的操作先持久化到磁盘。

* 在 memtable 溢写磁盘的过程中，外部的写操作需要阻塞，所带来的性能问题如何解决？

说明：通过memtable的分层实现，即active和inmutabletable实现。

* memtable 作为有序存储结构，内部采用什么样的数据结构进行实现？

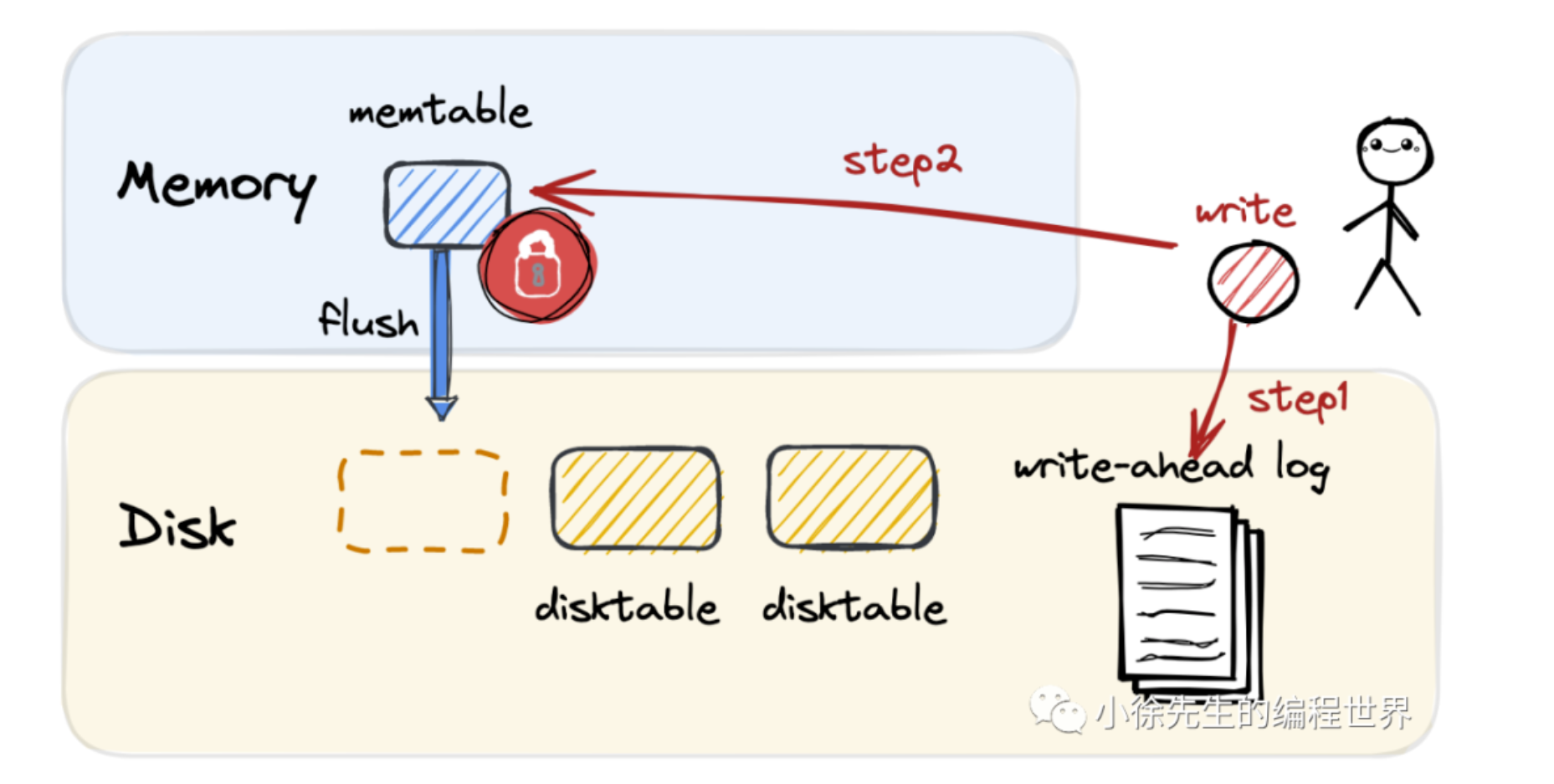
说明：采用跳表实现。

### 预写日志

首先面对第一个问题，如何避免因宕机导致memtable数据丢失。

解决这个问题的手段就是WAL（write-ahead log）预写日志技术：在将数据写入memtable，先通过追加写的方式，将操作记录到处于磁盘的WAL当中，这样哪怕宕机导致内存数据丢失，也能通过重放WAL的方式，重新恢复memtable 的数据。

此外，WAL和memtable可以建立对应关系，每当一个memtable被溢写到磁盘中成为disktable，其发生数据丢失问题的风险也就随之消除，因此对应的 WAL也就可以删除了。并且，由于WAL中也是追加写的操作，属于磁盘顺序IO，因此性能不会成为瓶颈。

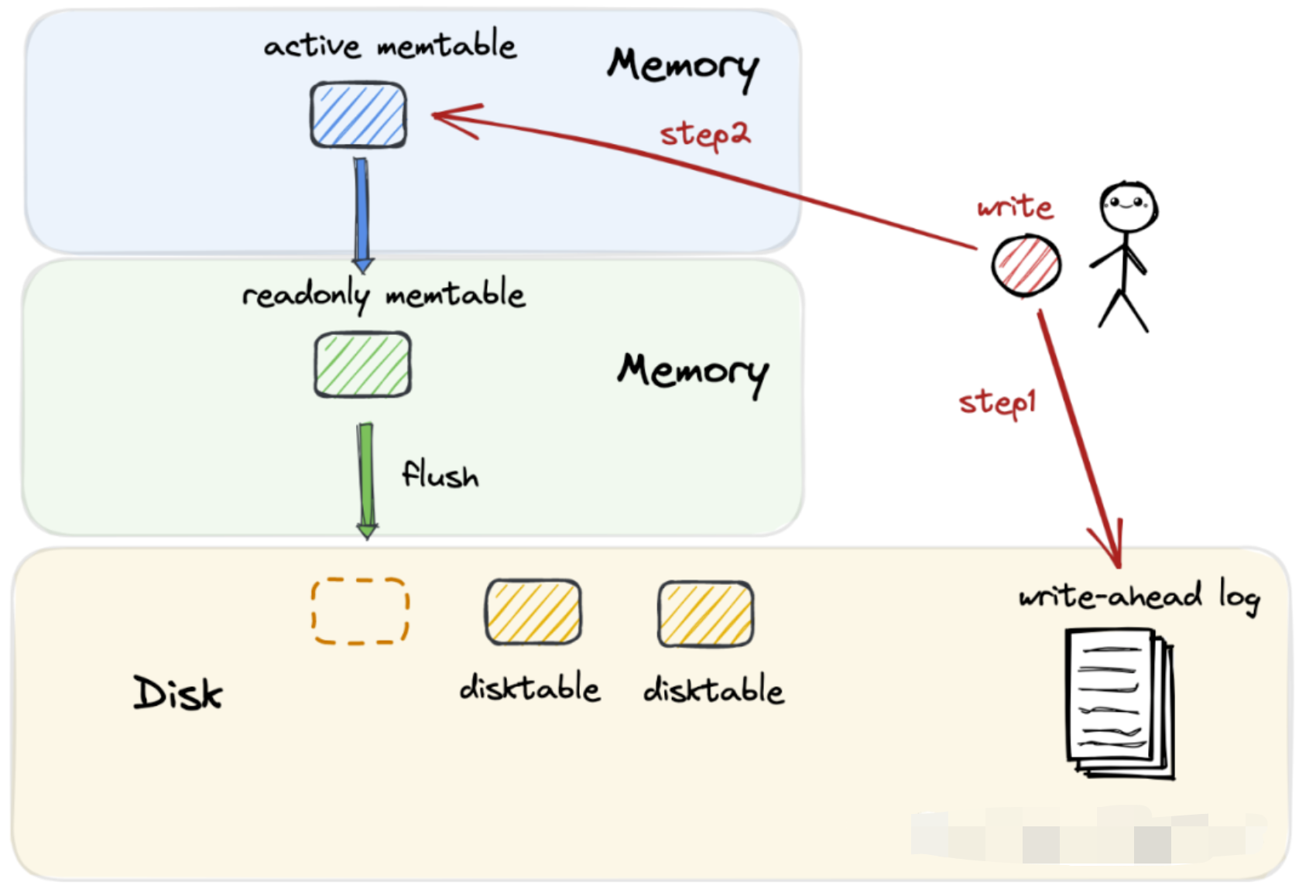


### 内存读写+只读分层

接下来面对第二个问题，在memtable 溢写磁盘期间，到来的写操作如何处理？

这里的解决思路是“金蝉脱壳”。每当 memtable 需要溢写时，就将其一分为二，将已有的旧数据归属到readonly memtable部分，成为一个只读的数据结构，专注于执行将其溢写到磁盘的流程；于此同时，建立一个全新空白的 active memtable，作为写操作的新入口，这样两个流程之间就实现了解耦，写操作不再需要阻塞。

由此可见，active memtable 持有的是最新的数据，readonly memtable 则次之，已经落到磁盘的 disktable 则再次。

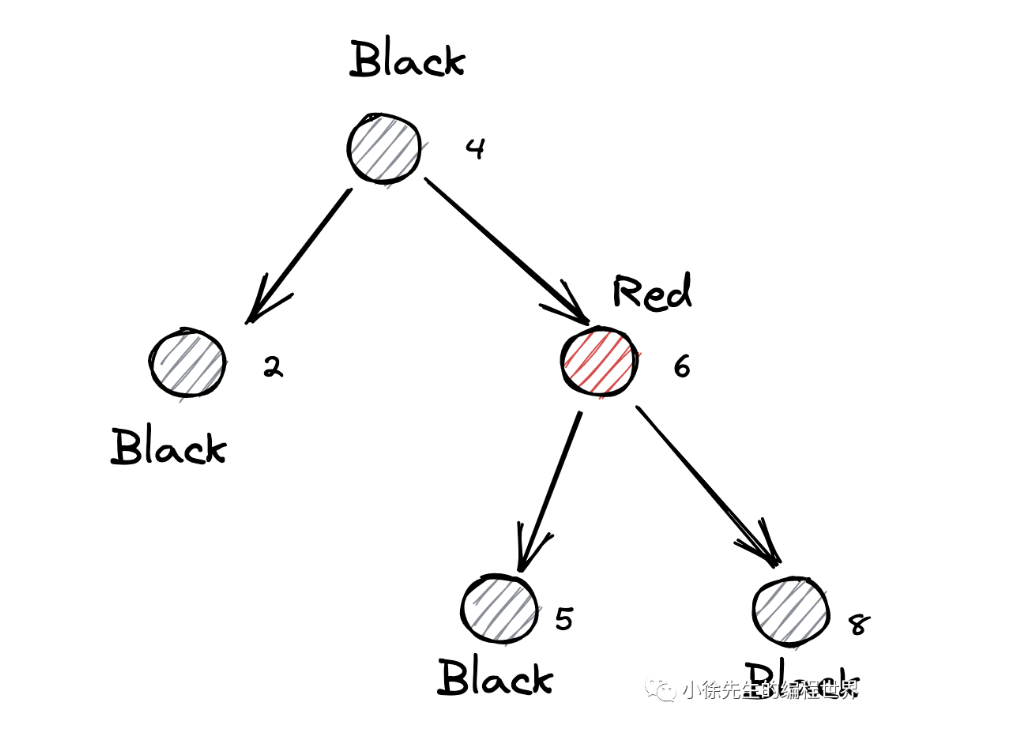


### 内存数据结构/memtable

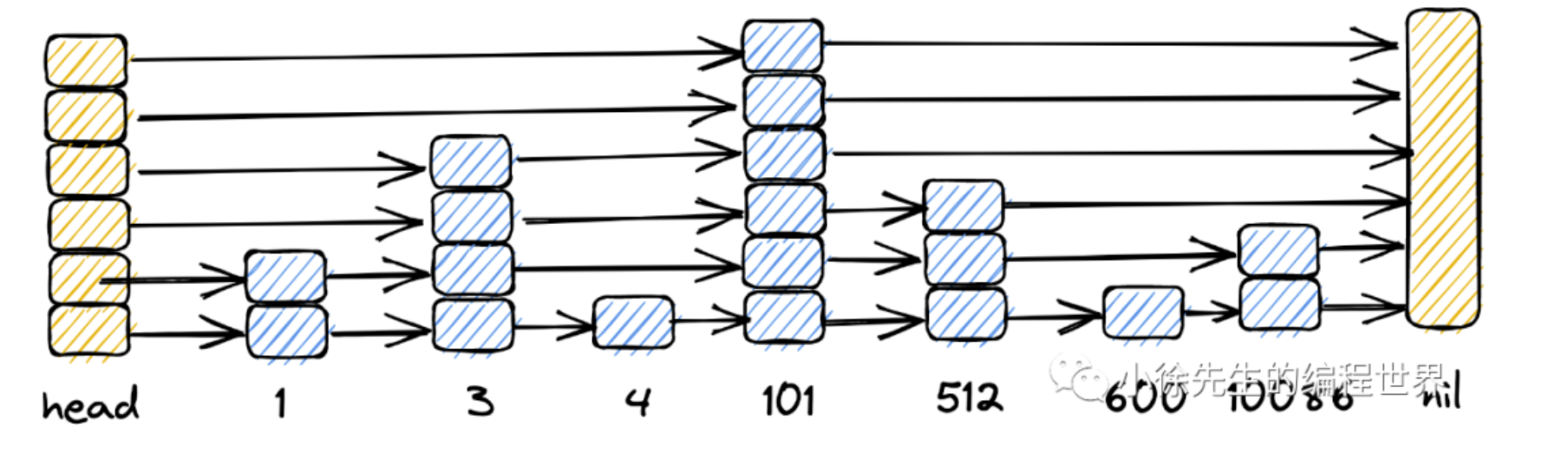
再聊第三个问题，内存中的 memtable 采用什么样的数据结构进行实现，其背后的核心要求是作为数据结构需要是一种有序表，能基于 k 进行数据的有序存储，从而保证所有的读操作和写操作都能在O(logN) 的时间复杂度之内拿下。

在这些诉求之下，进入视野的候选项包括红黑树（red-black tree）和跳表（skip list）两种。

红黑树结构示意如下：



跳表结构示意如下



在读写性能上，跳表和红黑树的性能表现不分伯仲。但跳表相比于红黑树具有两大核心优势：

* 更简单的实现
* 更细的并发锁粒度

作为存储组件，memtable 中的有序数据不可避免地会被用户并发读写访问，因此需要加锁保证临界资源的安全性和一致性。

在锁粒度上，红黑树由于自身染色的机制，每次写操作时都需要对对整棵树的（全量数据）进行加锁，相对笨重；

而跳表则不同，在执行写操作时，可以只针对插入节点局部范围内以及对跳表全局最大高度进行加锁，拥有着更细的锁粒度，在很多场景中是可以做到并发写的。

综上，跳表无疑是更好的选择。同时，也是 rocksdb 中默认采用的 memtable 数据结构。

### 磁盘分层

磁盘文件需要要分块细化粒度，由此诞生出了一系列的disktable。

所有disktable是由内存中的memtable溢写得到的，这样disktable就天然具有两大优势：

* disktable内部不存在重复kv对数据，因为memtable执行的是就地写操作
* disktable内部的kv对数据是有序的，因为memtable数据本身就是有序的

然而，disktable间还存在一个局限：由于每个memtable视野有限，只能做到自身范围内的k去重和排序。因此，不同disktable之间可能存在重复冗余的 kv 对数据，且不同disktable之间的数据无法做到全局有序。

理清了现状，我们再在此基础之上，进一步引入磁盘disktable分层的概念。

* 首先，我们将磁盘整体分为level0-levelk共计k+1层
* 每个level层中的disktable数量保持一致
* level(i+1)中disktable 的容量大小固定为level(i)的T倍，T为常量，通常取值为10左右
* 数据流向是由浅入深，层层递进，即由level(i) -> level(i+1)
* memtable溢写的数据落到level0
* levelk作为兜底
* 当某个level内数据总量达到达到阈值时，会发起level(i) -> level(i+1)的归并操作
* 数据从level(i)流向level(i+1)过程中，通过归并操作进行去重和排序，保证level(i+1)中kv数据无重复且全局有序

结合上述设定，我们可以得出以下结论：

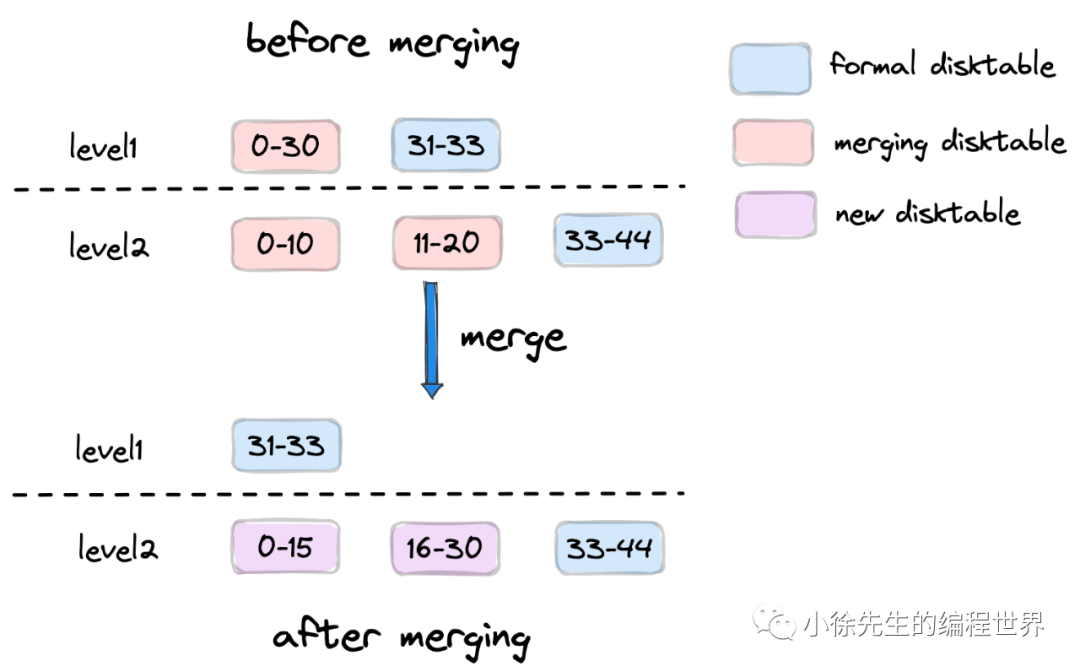
* level0 是特殊的，其中 disktable 之间可能存在冗余的 kv 对数据且不保证全局有序，因为其数据来自 memtable
* level1~levelk 中单层之内没有冗余的 kv 对数据，且保证全局有序
* 不同 level 层之间可能存在冗余的 kv 数据
* 较热（最近写入）的数据位于浅层，较冷（更早写入）的数据位于深层
* levelk 作为最深的一层，整体沉淀的数量达到全局的百分之九十左右

下面举一例，说明一下从 level1 到 level2 的数据归并过程。

假设此时 level1 层的数据总量已经达到阈值，接下来需要发起 level(1) -> level(2) 的归并操作：

* 从level1中随机选择一个 disktable，尝试将其合并到 level2. 由于数据是有序的，我们可以拿到其中 k 的取值范围. 假设其中最小的 key k\_min = 3，最大的 key k\_max = 30，记为[3,30]
* 假设 level2 中有 2 个 disktable 的 k 范围和待合并文件存在重叠，分别为 [0,16] 和 [17,32]
* 将 level1 的 [3,30] 与 level2 的 [0,10] 和 [11,20] 合并，这个过程本质上是个归并排序操作
* 新生成的 disktable [0,30] 不急于插入 level2 ，会根据 level2 中 disktable 的大小规模将其拆分为合适的数量
* 假设拆分得到的两个新的 disktable 分别为 [0,15] 和 [16,30]，将其插入到 level2
* 对应的老数据 level1 的[3,30] 以及 level2 的 [0,10] 和 [11,20] 都被被新数据替代，因此需要删除

值得一提的是，倘若因为这一合并操作，导致level2的数据容量又超出阈值，则会进一步引起level 2到level 3的数据合并操作，以此类推，层层递进。



### 磁盘数据结构/sstable

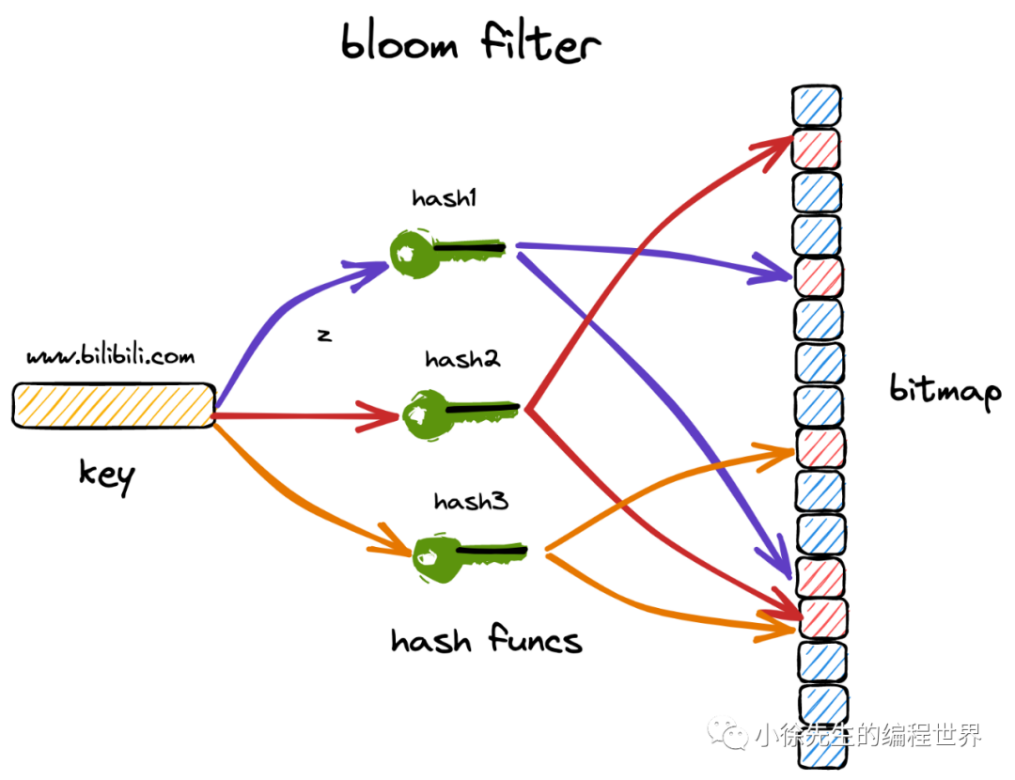
事实上，在 lsm tree的设定中，对于前文提到的每个磁盘文件块 disktable，设计了一类专门的数据结构sstable（sorted string table）。

每个 level 的 sstable 容量大约是上一层的10倍，因此一旦到了深层，sstable 的容量可能很大，对应展开的读操作会略显笨重。

sstable 在此基础上进行了优化：

* 首先 sstable 内部会进一步将 table 拆分为多个 block 块，其在逻辑意义上从属于同一个 sstable；
* 其次，sstable 中会额外维护一个索引信息，其中记录了每个 block 的 k\_min 和 k\_max 以及每个块中各行的 k\_max 和 k\_min，便于辅助的查询操作
* 此外，lsm tree 还维护着一个全局索引信息，记录着不同 level 中，每个 sstable 对应的k\_max 和 最小 k k\_min 的范围
* 最后，每个 sstable 还维护着一个布隆过滤器 bloomfilter，用于快速判断一个 k 是否存在于当前 sstable 中.

由于 bloom filter 具有假阳性的特点，因此判定不存在的 k 是必然不存在的，然而判定为存在的 k 也可能存在着一定的失误概率。

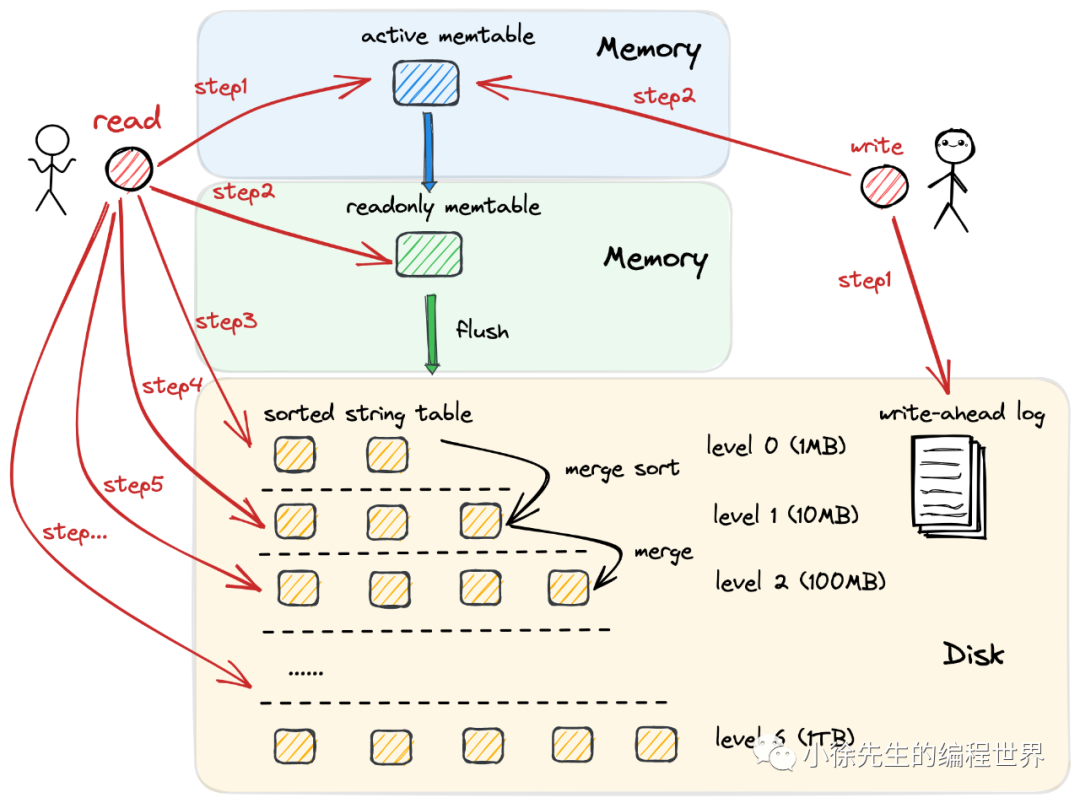


### 纵览lsm tree

到此为止，有关于 lsm tree 的拼图碎片已经集齐了，下面我们回过头来做个拼接和总览。

lsm tree全称 Log Structure Merge Tree，其核心设定如下：

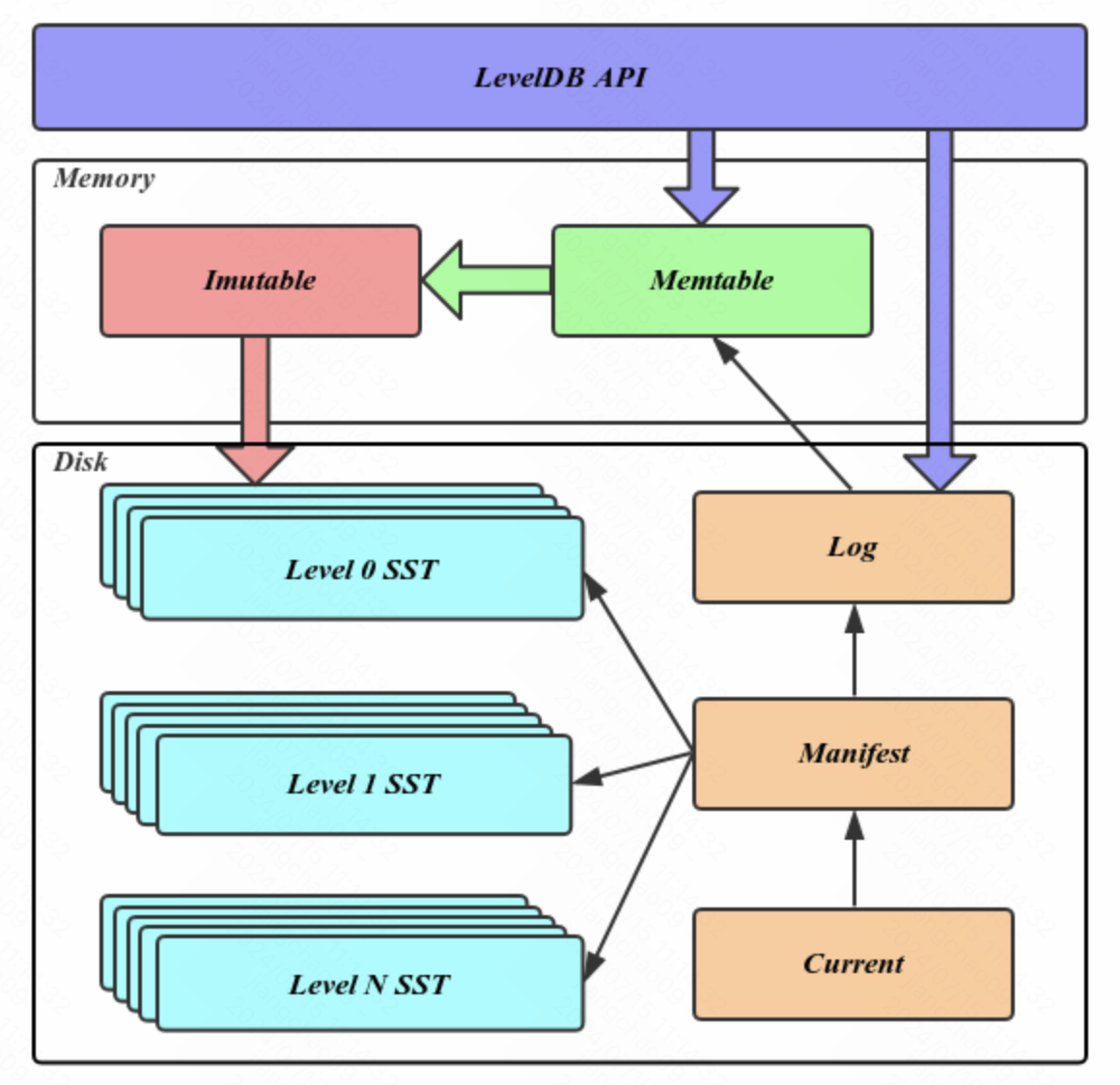
* 存储介质主要依赖磁盘（sstable），但上层也会借助内存的辅助（memtable）
* 内存（memtable）就地写，磁盘（sstable）顺序写
* 写入口为可读可写的 active memtable
* 达到阈值后 active memtable 转为只读的 readonly memtable
* memtable 保证有序，默认基于跳表实现
* 由于 sstable 来自 memtable，每个sstable 内部无冗余数据且有序
* 磁盘文件分层（level0~levelk），上层为近期写入的热数据，下层为较早写入的冷数据
* level(i+1) sstable 大小恒定为 level(i) 层 T 倍
* level0 sstable 之间存在冗余数据
* level1~levelk 单层内无冗余数据且全局有序
* 数据沿着 level 0 -> level k 的方向合并，自顶向下流动



## 数据库文件

主要有以下几种类型sst文件，CURRENT文件，manifest文件，log文件，LOG文件和LOCK文件。

sst文件存储的是落地的数据，CURRENT文件存储的是当前最新的是哪个manifest文件，manifest文件存储的是Version的变化，log文件是rocksdb的write ahead log，就是在写db之前写的数据日志文件，LOG文件是一些日志信息，是供调试用的，LOCK是打开db锁，只允许同时有一个进程打开db。



### log文件/WAL

log文件就是WAL。

顾名思义，就是在实际操作数据前先写日志，便于恢复。WAL在很多数据库中都存在。

RocksDB中的每个更新操作都会写到两个地方：

1、一个内存数据结构，名为memtable(后面会被刷盘到SST文件)

2、写到磁盘上的WAL日志。在出现崩溃的时候，WAL日志可以用于完整的恢复memtable中的数据，以保证数据库能恢复到原来的状态。在默认配置的情况下，RocksDB通过在每次写操作后对WAL调用flush来保证一致性。

### sst文件

数据文件。

RocksDB 的 Write-Ahead Log (WAL) 和 Manifest 文件在数据库的持久化和恢复机制中扮演了重要的角色。以下是它们的区别和联系：

**Write-Ahead Log (WAL)**

- 功能：WAL 记录所有尚未写入 SSTable 的写操作（如插入、更新和删除），确保在发生崩溃时可以恢复数据。

- 持久化：每个写操作在应用到内存中的 MemTable 之前，首先记录在 WAL 中。这样，如果数据库在写操作过程中崩溃，WAL 可以用来重放这些操作，恢复 MemTable 的状态。

- 格式：WAL 通常是顺序追加写入的日志文件。每个写操作以日志记录的形式存储，包含键值对数据及相关的元数据。

- 性能影响：WAL 的写入速度影响数据库的写操作性能，因为每次写操作需要同步到磁盘以确保数据安全。

**Manifest 文件**

- 功能：Manifest 文件记录了 SSTable 的元数据和数据库的整体状态。它保存了所有 SSTable 文件的元数据信息，以及数据库各个级别的 SSTable 文件布局。

- 持久化：每次生成新的 SSTable 或数据库状态发生变化时，Manifest 文件都会更新，以反映当前的数据库状态和 SSTable 文件的布局。

- 格式：Manifest 文件是一系列的元数据记录，描述了 SSTable 的生成、删除、合并等操作。每次写入 Manifest 文件都是追加操作，以保持完整的历史记录。

- 恢复：在数据库启动时，RocksDB 会读取 Manifest 文件来重建 SSTable 的元数据和整体结构。这使得数据库能够恢复到最后一次一致状态。

**联系**

- 数据持久化：两者都涉及数据库的持久化机制，但关注的层面不同。WAL 关注的是记录所有未持久化的写操作，而 Manifest 文件关注的是数据库的元数据和结构。

- 数据恢复：在数据库崩溃后，RocksDB 会首先使用 WAL 重放未持久化的写操作，以恢复 MemTable 的数据。然后，通过读取 Manifest 文件，恢复 SSTable 的元数据和整体结构，确保数据库状态的一致性。

- 顺序写入：两者都采用顺序写入的方式，WAL 是顺序记录写操作，Manifest 文件是顺序记录元数据变化。这种设计有助于提高写操作性能和数据的可靠性。

通过 WAL 和 Manifest 文件的配合，RocksDB 能够实现高效的数据写入、持久化和可靠的故障恢复机制。

### manifest文件

参考：

MANIFEST文件实现：<https://developer.aliyun.com/article/594728>

MANIFEST：

<https://github.com/johnzeng/rocksdb-doc-cn/blob/master/doc/MANIFEST.md>

MANIFEST恢复流程：<https://gocode.cc/project/13/article/190>

在RocksDB中MANIFEST保存了存储引擎的内部的一些状态元数据，简单来说当系统异常重启，或者程序异常被退出之后，RocksDB需要有一种机制能够恢复到一个一致性的状态，而这个一致性的状态就是靠MANIFEST来保证的。

MANIFEST在RocksDB中是一个单独的文件，而这个文件所保存的数据基本是来自于VersionEdit这个结构。

MANIFEST包含了两个文件，一个log文件一个包含最新MANIFEST文件名的文件，Manifest的log文件名是这样 MANIFEST-(seqnumber),这个seq会一直增长。只有当超过了指定的大小之后，MANIFEST会刷新一个新的文件，当新的文件刷新到磁盘(并且文件名更新)之后，老的文件会被删除掉。这里可以认为每一次MANIFEST的更新都代表一次snapshot。

### CURRENT文件

CURRENT 指最后的Manifest日志。

### LOG文件

统计日志。

### LOCK文件

无内容，open时创建，表示一个db在一个进程中只能被open一次，多线程共用此实例。

## Memtable

参考：

Memtable写入逻辑：<http://mysql.taobao.org/monthly/2018/08/08/>

解析memtable、跳表和内存池Arena：

<https://www.cnblogs.com/JayL-zxl/p/15025932.html>

Facebook Memtable：<https://github.com/facebook/rocksdb/wiki/MemTable>

相关代码：

memtable/\*

db/column\_faimily.cc // 创建MemTable

### 实现

在RocksDB中，每个ColumnFamily都有自己的Memtable，互不影响。而在RocksDB中Memtable有多种实现（SkipList/HashSkipList/HashLinkList/Vector），我们这次主要来分析默认的实现skiplist（只有skiplist是可以并发插入的）。

首先从创建Memtable开始，Memtable的创建（ColumnFamilyData::CreateNewMemtable）是在创建ColumnFamily(VersionSet::CreateColumnFamily)的时候创建的。这里就是创建memtable，然后设置到ColumnFamilyData的mem\_域中。

#### ColumnFamilyData::CreateNewMemtable

##### ColumnFamilyData::ConstructNewMemtable

###### MemTable::MemTable

上面所提及的，RocksDB有多种MemTable的实现，那么它是如何来做的呢，RocksDB通过memtable\_factory来根据用户的设置来创建不同的memtable。这里要注意的是核心的memtable实现是在MemTable这个类的table\_域中。

MemTable::MemTable:

table\_(ioptions.memtable\_factory->CreateMemTableRep(

comparator\_, &arena\_, ioptions.prefix\_extractor, ioptions.info\_log,

column\_family\_id)),

class MemTableRepFactory {

public:

virtual ~MemTableRepFactory() {}

virtual MemTableRep\* CreateMemTableRep(const MemTableRep::KeyComparator&,

Allocator\*, const SliceTransform\*,

Logger\* logger) = 0;

virtual MemTableRep\* CreateMemTableRep(

const MemTableRep::KeyComparator& key\_cmp, Allocator\* allocator,

const SliceTransform\* slice\_transform, Logger\* logger,

uint32\_t /\* column\_family\_id \*/) {

return CreateMemTableRep(key\_cmp, allocator, slice\_transform, logger);

}

........................

然后最后会调用对应的实现的CreateMemTableRep方法，这里我们就来看SkipList的实现。

CreateMemTableRep

class SkipListRep : public MemTableRep {

InlineSkipList<const MemTableRep::KeyComparator&> skip\_list\_;

...................................

public:

explicit SkipListRep(const MemTableRep::KeyComparator& compare,

Allocator\* allocator, const SliceTransform\* transform,

const size\_t lookahead)

: MemTableRep(allocator),

skip\_list\_(compare, allocator),

cmp\_(compare),

transform\_(transform),

lookahead\_(lookahead) {}

最终就是创建SkipListRep对象，在这个对象里面会创建SkipList(class InlineSkipList)。

这里SkipList就不分析实现了，如果对这个数据结构不了解的，可以去[WIKI](https://en.wikipedia.org/wiki/Skip_list)看一下。这里我们只需要知道最终所有的memtable数据都是保存在SkipList中就可以了。

HashSkipListRep

SkipListRep

#### MemTable::Add

在之前的分析中我们知道Memtable的插入是通过WriteBatch然后遍历ColumnFamily来插入的，而最终则是会调用MemTable::Add这个函数。

bool MemTable::Add(SequenceNumber s, ValueType type,

const Slice& key, /\* user key \*/

const Slice& value, bool allow\_concurrent,

MemTablePostProcessInfo\* post\_process\_info) {

bool res = table->InsertKeyConcurrently(handle);

if (UNLIKELY(!res)) {

return res;

}

..............................

}

最终会调用InlineSkipList来对数据进行插入。

##### InsertConcurrently

template <class Comparator>

bool InlineSkipList<Comparator>::InsertConcurrently(const char\* key) {

Node\* prev[kMaxPossibleHeight];

Node\* next[kMaxPossibleHeight];

Splice splice;

splice.prev\_ = prev;

splice.next\_ = next;

return Insert<true>(key, &splice, false);

}

看到这里或许会有疑问了，那就是skiplist里面只有key，而RocksDB是一个KV存储，那么这个KV是如何存储的呢，这里是这样的，RocksDB会将KV打包成一个key传递给SkipList，对应的KEY的结构是这样的。

// Format of an entry is concatenation of:

// key\_size : varint32 of internal\_key.size()

// key bytes : char[internal\_key.size()]

// value\_size : varint32 of value.size()

// value bytes : char[value.size()]

而数据的格式化就在之前的MemTable::Add中实现的。

uint32\_t key\_size = static\_cast<uint32\_t>(key.size());

uint32\_t val\_size = static\_cast<uint32\_t>(value.size());

uint32\_t internal\_key\_size = key\_size + 8;

const uint32\_t encoded\_len = VarintLength(internal\_key\_size) +

internal\_key\_size + VarintLength(val\_size) +

val\_size;

char\* buf = nullptr;

std::unique\_ptr<MemTableRep>& table =

type == kTypeRangeDeletion ? range\_del\_table\_ : table\_;

KeyHandle handle = table->Allocate(encoded\_len, &buf);

char\* p = EncodeVarint32(buf, internal\_key\_size);

memcpy(p, key.data(), key\_size);

Slice key\_slice(p, key\_size);

p += key\_size;

uint64\_t packed = PackSequenceAndType(s, type);

EncodeFixed64(p, packed);

p += 8;

p = EncodeVarint32(p, val\_size);

memcpy(p, value.data(), val\_size);

}

而对于真正的KEY的解析是在SkipList的Comparator中实现的（compare\_）。下面的代码片段可以看到会解析出来真正的key，然后再进行查找以及插入。

###### Insert

bool InlineSkipList<Comparator>::Insert(const char\* key, Splice\* splice,

bool allow\_partial\_splice\_fix) {

Node\* x = reinterpret\_cast<Node\*>(const\_cast<char\*>(key)) - 1;

const DecodedKey key\_decoded = compare\_.decode\_key(key);

...............................

}

### Column Family

参考：

ColumnFamily简介：<http://mysql.taobao.org/monthly/2018/06/09/>

#### 概述

在RocksDB 3.0中加入了Column Family特性，加入这个特性之后，每一个KV对都会关联一个Column Family，其中默认的Column Family是“default”。 Column Family主要是提供给RocksDB一个逻辑的分区。从实现上来看不同的Column Family共享WAL，而都有自己的Memtable和SST。这就意味着我们可以很快速方便的设置不同的属性给不同的Column Family以及快速删除对应的Column Family。

##### Option

RocksDB 的 Option 有 Options, ColumnFamilyOptions, DBOptions 三种。

ColumnFamilyOptions是table级的，而Options是DB级的，Options继承自 ColumnFamilyOptions和DBOptions，它一般影响只有一个CF的DB，如 “default”。

每个 CF 都有一个 Handle：ColumnFamilyHandle，在 DB 指针被 delete 前，应该先 delete ColumnFamilyHandle。如果 ColumnFamilyHandle 指向的 CF 被别的使用者通过 DropColumnFamily 删除掉，这个 CF 仍然可以被访问，因为其引用计数不为 0。

在以 Read/Write 方式打开一个 DB 的时候，需要指定一个由所有将要用到的 CF string name 构成的 ColumnFamilyDescriptor array。不管 “default” CF 使用与否，都必须被带上。

CF 存在的意义是所有 table 共享 WAL，但不共享 memtable 和 table 文件，通过 WAL 保证原子写，通过分离 table 可快读快写快删除。每次 flush 一个 CF 后，都会新建一个 WAL，都这并不意味着旧的 WAL 会被删除，因为别的 CF 数据可能还没有落盘，只有所有的 CF 数据都被 flush 且所有的 WAL 有关的 data 都落盘，相关的 WAL 才会被删除。RocksDB 会定时执行 CF flush 任务，可以通过 Options::max\_total\_wal\_size 查看已有多少旧的 CF 文件已经被 flush 了。

RocksDB 会在磁盘上依据 LSM 算法对多级磁盘文件进行 compaction，这会影响写性能，拖慢程序性能，可以通过 WriteOptions.low\_pri = true 降低 compaction 的优先级。

##### Set Up Option

RocksDB 有很多选项以专门的目的进行设置，但是大部分情况下不需要进行特殊的优化。这里只列出一个常用的优化选项。

cf\_options.write\_buffer\_size

CF 的 write buffer 的最大 size。最差情况下 RocksDB 使用的内存量会翻倍，所以一般情况下不要轻易修改其值。

Set block cache size

这个值一般设置为 RocksDB 想要使用的内存总量的 1/3，其余的留给 OS 的 page cache。

BlockBasedTableOptions table\_options;

… \\ set options in table\_options

options.table\_factory.reset(new

std::shared\_ptr<Cache> cache = NewLRUCache(<your\_cache\_size>);

table\_options.block\_cache = cache;

BlockBasedTableFactory(table\_options));

本进程的所有的 DB 所有的 CF 所有的 table\_options 都必须使用同一个 cahce 对象，或者让所有的 DB 所有的 CF 使用同一个 table\_options。

cf\_options.compression, cf\_options.bottonmost\_compression

选择压缩方法跟你的机器、CPU 能力以及内存磁盘空间大小有关，官方推荐 cf\_options.compression 使用 kLZ4Compression，cf\_options.bottonmost\_compression 使用 kZSTD，选用的时候要确认你的机器有这两个库，这两个选项也可以分别使用 Snappy 和 Zlib。

Bloom filter

官方真正建议修改的参数只有这个 filter 参数。如果大量使用迭代方法，不要修改这个参数，如果大量调用 Get() 接口，建议修改这个参数。修改方法如下：

table\_options.filter\_policy.reset(NewBloomFilterPolicy(10, false));

一个可能的优化设定如下：

cf\_options.level\_compaction\_dynamic\_level\_bytes = true;

options.max\_background\_compactions = 4;

options.max\_background\_flushes = 2;

options.bytes\_per\_sync = 1048576;

table\_options.block\_size = 16 \* 1024;

table\_options.cache\_index\_and\_filter\_blocks = true;

table\_options.pin\_l0\_filter\_and\_index\_blocks\_in\_cache = true;

上面只是罗列了一些优化选项，这些选项也只能在进程启动的时候设定。更多的选项请详细阅读参考文档1。

##### WriteOption & Persistence

RocksDB 每次接收写请求的时候，请求内容会先被写入 WAL transaction log，然后再把数据写入 memfile 里面。

Put 函数的参数 WriteOptions 里有一个选项可以指明是否需要把写请求的内容写入 WAL log 里面。

RocksDB 内部有一个 batch-commit 机制，通过一次 commit 批量地在一次 sync 操作里把所有 transactions log 写入磁盘。

#### 主要API

首先是创建Column Family，这里注意我们可以通过两种方式来创建Column Family，一种是在Open DB的时候通过传递需要创建的Column Family，一种是当DB创建并打开之后， 通过直接的CreateColumnFamily来创建Column Family。

DB::Open(const DBOptions& db\_options, const std::string& name,

const std::vector<ColumnFamilyDescriptor>& column\_families,

std::vector<ColumnFamilyHandle\*>\* handles, DB\*\* dbptr);

DB::CreateColumnFamily(const ColumnFamilyOptions& options,

const std::string& column\_family\_name,

ColumnFamilyHandle\*\* handle);

这里可以看到不管是哪一种方式最终都会返回一个ColumnFamilyHandle给调用者来使用。

然后就是删除Column Family的方式，这里很简单就是传递之前创建的ColumnFamilyHandle给RocksDB，然后用以删除。

DropColumnFamily(ColumnFamilyHandle\* column\_family);

#### 实现

所有的Column Family都是通过一个叫做ColumnFamilySet的结构来管理的，而每一个Column Family都是一个ColumnFamilyData.

先来看ColumnFamilySet，这里可以看到它有两个数据结构来管理Column Family，分别是map(column\_family\_data\_)以及一个双向链表(dummy\_cfd\_)。其中map用来保存Column Family名字和对应的id以及ColumnFamilyData的映射。这里要注意在RocksDB内部是将没一个ColumnFamily的名字表示为一个uint32类型的ID(max\_column\_family\_)。也就是这个ID是一个简单的递增的数值。

##### ColumnFamilySet

class ColumnFamilySet {

public:

// ColumnFamilySet supports iteration

public:

.................................

ColumnFamilyData\* CreateColumnFamily(const std::string& name, uint32\_t id,

Version\* dummy\_version,

const ColumnFamilyOptions& options);

iterator begin() { return iterator(dummy\_cfd\_->next\_); }

iterator end() { return iterator(dummy\_cfd\_); }

...............................

private:

friend class ColumnFamilyData;

// helper function that gets called from cfd destructor

// REQUIRES: DB mutex held

void RemoveColumnFamily(ColumnFamilyData\* cfd);

// column\_families\_ and column\_family\_data\_ need to be protected:

// \* when mutating both conditions have to be satisfied:

// 1. DB mutex locked

// 2. thread currently in single-threaded write thread

// \* when reading, at least one condition needs to be satisfied:

// 1. DB mutex locked

// 2. accessed from a single-threaded write thread

std::unordered\_map<std::string, uint32\_t> column\_families\_;

std::unordered\_map<uint32\_t, ColumnFamilyData\*> column\_family\_data\_;

uint32\_t max\_column\_family\_;

ColumnFamilyData\* dummy\_cfd\_;

// We don't hold the refcount here, since default column family always exists

// We are also not responsible for cleaning up default\_cfd\_cache\_. This is

// just a cache that makes common case (accessing default column family)

// faster

ColumnFamilyData\* default\_cfd\_cache\_;

..................................

};

然后来看ColumnFamilyData，这个数据结构就是用来表示一个ColumnFamily，保存了对应的信息，我们可以看到有ID/name以及当前ColumnFamily对应的所有的version(dummy\_versions\_)。其中这里的next\_/prev\_就是在ColumnFamilySet中用来表示所有ColumnFamily的双向链表。

##### ColumnFamilyData

class ColumnFamilyData {

public:

~ColumnFamilyData();

// thread-safe

uint32\_t GetID() const { return id\_; }

// thread-safe

const std::string& GetName() const { return name\_; }

// Ref() can only be called from a context where the caller can guarantee

// that ColumnFamilyData is alive (while holding a non-zero ref already,

// holding a DB mutex, or as the leader in a write batch group).

void Ref() { refs\_.fetch\_add(1, std::memory\_order\_relaxed); }

// Unref decreases the reference count, but does not handle deletion

// when the count goes to 0. If this method returns true then the

// caller should delete the instance immediately, or later, by calling

// FreeDeadColumnFamilies(). Unref() can only be called while holding

// a DB mutex, or during single-threaded recovery.

bool Unref() {

int old\_refs = refs\_.fetch\_sub(1, std::memory\_order\_relaxed);

assert(old\_refs > 0);

return old\_refs == 1;

}

..............................

private:

friend class ColumnFamilySet;

ColumnFamilyData(uint32\_t id, const std::string& name,

Version\* dummy\_versions, Cache\* table\_cache,

WriteBufferManager\* write\_buffer\_manager,

const ColumnFamilyOptions& options,

const ImmutableDBOptions& db\_options,

const EnvOptions& env\_options,

ColumnFamilySet\* column\_family\_set);

uint32\_t id\_;

const std::string name\_;

Version\* dummy\_versions\_; // Head of circular doubly-linked list of versions.

Version\* current\_; // == dummy\_versions->prev\_

......................................................

// Thread's local copy of SuperVersion pointer

// This needs to be destructed before mutex\_

std::unique\_ptr<ThreadLocalPtr> local\_sv\_;

// pointers for a circular linked list. we use it to support iterations over

// all column families that are alive (note: dropped column families can also

// be alive as long as client holds a reference)

ColumnFamilyData\* next\_;

ColumnFamilyData\* prev\_;

...................................

ColumnFamilySet\* column\_family\_set\_;

..................................

};

然后就是返回给调用者的ColumnFamilyHandleImpl结构，这个结构主要是封装了ColumnFamilyData。

##### ColumnFamilyHandleImpl

// ColumnFamilyHandleImpl is the class that clients use to access different

// column families. It has non-trivial destructor, which gets called when client

// is done using the column family

class ColumnFamilyHandleImpl : public ColumnFamilyHandle {

public:

// create while holding the mutex

ColumnFamilyHandleImpl(

ColumnFamilyData\* cfd, DBImpl\* db, InstrumentedMutex\* mutex);

// destroy without mutex

virtual ~ColumnFamilyHandleImpl();

virtual ColumnFamilyData\* cfd() const { return cfd\_; }

......................................

private:

ColumnFamilyData\* cfd\_;

DBImpl\* db\_;

InstrumentedMutex\* mutex\_;

};

接下来我们就来从ColumnFamily的创建以及删除来分析ColumnFamily的实现。我们从DBImpl::CreateColumnFamilyImpl开始，在这个函数中首先就是通过调用GetNextColumnFamilyID来得到当前创建的ColumnFamily对应的ID(自增)。然后再调用LogAndApply来对ColumnFamily 进行对应的操作，最后再返回封装好的ColumnFamilyHandle给调用者。

##### DBImpl::CreateColumnFamilyImpl

Status DBImpl::CreateColumnFamilyImpl(const ColumnFamilyOptions& cf\_options,

const std::string& column\_family\_name,

ColumnFamilyHandle\*\* handle) {

.......................................

{

...................................

VersionEdit edit;

edit.AddColumnFamily(column\_family\_name);

uint32\_t new\_id = versions\_->GetColumnFamilySet()->GetNextColumnFamilyID();

edit.SetColumnFamily(new\_id);

edit.SetLogNumber(logfile\_number\_);

edit.SetComparatorName(cf\_options.comparator->Name());

// LogAndApply will both write the creation in MANIFEST and create

// ColumnFamilyData object

{ // write thread

WriteThread::Writer w;

write\_thread\_.EnterUnbatched(&w, &mutex\_);

// LogAndApply will both write the creation in MANIFEST and create

// ColumnFamilyData object

s = versions\_->LogAndApply(nullptr, MutableCFOptions(cf\_options), &edit,

&mutex\_, directories\_.GetDbDir(), false,

&cf\_options);

write\_thread\_.ExitUnbatched(&w);

}

if (s.ok()) {

........................................

\*handle = new ColumnFamilyHandleImpl(cfd, this, &mutex\_);

ROCKS\_LOG\_INFO(immutable\_db\_options\_.info\_log,

"Created column family [%s] (ID %u)",

column\_family\_name.c\_str(), (unsigned)cfd->GetID());

}

.............................................

} // InstrumentedMutexLock l(&mutex\_)

.................................

return s;

}

最终会在LogAndApply调用ColumnFamilySet的CreateColumnFamily函数(通过VersionSet::CreateColumnFamily)，这个函数我们可看到主要做了下面三件事情。

1、创建ColumnFamilyData对象

2、将新的创建好的CFD加入到双向链表

3、对应的Map数据结构更新数据

###### ColumnFamilySet::CreateColumnFamily

// under a DB mutex AND write thread

ColumnFamilyData\* ColumnFamilySet::CreateColumnFamily(

const std::string& name, uint32\_t id, Version\* dummy\_versions,

const ColumnFamilyOptions& options) {

assert(column\_families\_.find(name) == column\_families\_.end());

ColumnFamilyData\* new\_cfd = new ColumnFamilyData(

id, name, dummy\_versions, table\_cache\_, write\_buffer\_manager\_, options,

\*db\_options\_, env\_options\_, this);

column\_families\_.insert({name, id});

column\_family\_data\_.insert({id, new\_cfd});

max\_column\_family\_ = std::max(max\_column\_family\_, id);

// add to linked list

new\_cfd->next\_ = dummy\_cfd\_;

auto prev = dummy\_cfd\_->prev\_;

new\_cfd->prev\_ = prev;

prev->next\_ = new\_cfd;

dummy\_cfd\_->prev\_ = new\_cfd;

if (id == 0) {

default\_cfd\_cache\_ = new\_cfd;

}

return new\_cfd;

}

##### ColumnFamilySet::RemoveColumnFamily

然后来看如何删除ColumnFamily，这里所有的删除最终都会调用ColumnFamilySet::RemoveColumnFamily函数，这个函数是是从两个Map中删除对应的ColumnFamily。这里或许我们要问了，为什么管理的双向链表不需要删除呢。这里原因是这样的，由于ColumnFamilyData是通过引用计数管理的，因此只有当所有的引用计数都清零之后， 才需要真正的函数ColumnFamilyData(也就是会从双向链表中删除数据)。

// under a DB mutex AND from a write thread

void ColumnFamilySet::RemoveColumnFamily(ColumnFamilyData\* cfd) {

auto cfd\_iter = column\_family\_data\_.find(cfd->GetID());

assert(cfd\_iter != column\_family\_data\_.end());

column\_family\_data\_.erase(cfd\_iter);

column\_families\_.erase(cfd->GetName());

}

因此我们来看ColumnFamilyData的析构函数。可以看到析构函数中会从双向链表中删除对应的数据,以及处理对应的Version(corrent\_)。

// DB mutex held

ColumnFamilyData::~ColumnFamilyData() {

assert(refs\_.load(std::memory\_order\_relaxed) == 0);

// remove from linked list

auto prev = prev\_;

auto next = next\_;

prev->next\_ = next;

next->prev\_ = prev;

if (!dropped\_ && column\_family\_set\_ != nullptr) {

// If it's dropped, it's already removed from column family set

// If column\_family\_set\_ == nullptr, this is dummy CFD and not in

// ColumnFamilySet

column\_family\_set\_->RemoveColumnFamily(this);

}

if (current\_ != nullptr) {

current\_->Unref();

}

..............................

}

最后我们来看一下在磁盘上ColumnFamily是如何保存的，首先需要明确的是ColumnFamily是保存在MANIFEST文件中的，信息的保存比较简单(之前的文章有介绍), 和MANIFEST中其他的信息没什么区别，因此这里我们主要来看数据的读取以及初始化，这里所有的操作都是包含在VersionSet::Recover中，我们来看这个函数。

函数主要的逻辑就是读取MANIFEST然后来再来将磁盘上读取的ColumnFamily的信息初始化(初始化ColumnFamilySet结构)，可以看到这里相当于将之前的create/drop 的操作全部回放一遍，也就是会调用CreateColumnFamily/DropColumnFamily来将磁盘的信息初始化到内存。

while (reader.ReadRecord(&record, &scratch) && s.ok()) {

VersionEdit edit;

s = edit.DecodeFrom(record);

if (!s.ok()) {

break;

}

// Not found means that user didn't supply that column

// family option AND we encountered column family add

// record. Once we encounter column family drop record,

// we will delete the column family from

// column\_families\_not\_found.

bool cf\_in\_not\_found =

column\_families\_not\_found.find(edit.column\_family\_) !=

column\_families\_not\_found.end();

// in builders means that user supplied that column family

// option AND that we encountered column family add record

bool cf\_in\_builders =

builders.find(edit.column\_family\_) != builders.end();

// they can't both be true

assert(!(cf\_in\_not\_found && cf\_in\_builders));

ColumnFamilyData\* cfd = nullptr;

if (edit.is\_column\_family\_add\_) {

if (cf\_in\_builders || cf\_in\_not\_found) {

s = Status::Corruption(

"Manifest adding the same column family twice");

break;

}

auto cf\_options = cf\_name\_to\_options.find(edit.column\_family\_name\_);

if (cf\_options == cf\_name\_to\_options.end()) {

column\_families\_not\_found.insert(

{edit.column\_family\_, edit.column\_family\_name\_});

} else {

cfd = CreateColumnFamily(cf\_options->second, &edit);

cfd->set\_initialized();

builders.insert(

{edit.column\_family\_, new BaseReferencedVersionBuilder(cfd)});

}

} else if (edit.is\_column\_family\_drop\_) {

if (cf\_in\_builders) {

auto builder = builders.find(edit.column\_family\_);

assert(builder != builders.end());

delete builder->second;

builders.erase(builder);

cfd = column\_family\_set\_->GetColumnFamily(edit.column\_family\_);

if (cfd->Unref()) {

delete cfd;

cfd = nullptr;

} else {

// who else can have reference to cfd!?

assert(false);

}

} else if (cf\_in\_not\_found) {

column\_families\_not\_found.erase(edit.column\_family\_);

} else {

s = Status::Corruption(

"Manifest - dropping non-existing column family");

break;

}

} else if (!cf\_in\_not\_found) {

if (!cf\_in\_builders) {

s = Status::Corruption(

"Manifest record referencing unknown column family");

break;

}

cfd = column\_family\_set\_->GetColumnFamily(edit.column\_family\_);

// this should never happen since cf\_in\_builders is true

assert(cfd != nullptr);

// if it is not column family add or column family drop,

// then it's a file add/delete, which should be forwarded

// to builder

auto builder = builders.find(edit.column\_family\_);

assert(builder != builders.end());

builder->second->version\_builder()->Apply(&edit);

}

if (cfd != nullptr) {

if (edit.has\_log\_number\_) {

if (cfd->GetLogNumber() > edit.log\_number\_) {

ROCKS\_LOG\_WARN(

db\_options\_->info\_log,

"MANIFEST corruption detected, but ignored - Log numbers in "

"records NOT monotonically increasing");

} else {

cfd->SetLogNumber(edit.log\_number\_);

have\_log\_number = true;

}

}

if (edit.has\_comparator\_ &&

edit.comparator\_ != cfd->user\_comparator()->Name()) {

s = Status::InvalidArgument(

cfd->user\_comparator()->Name(),

"does not match existing comparator " + edit.comparator\_);

break;

}

}

if (edit.has\_prev\_log\_number\_) {

previous\_log\_number = edit.prev\_log\_number\_;

have\_prev\_log\_number = true;

}

if (edit.has\_next\_file\_number\_) {

next\_file = edit.next\_file\_number\_;

have\_next\_file = true;

}

if (edit.has\_max\_column\_family\_) {

max\_column\_family = edit.max\_column\_family\_;

}

if (edit.has\_last\_sequence\_) {

last\_sequence = edit.last\_sequence\_;

have\_last\_sequence = true;

}

}

### SkipList/跳跃表

参考：

<https://kaiwudb.com/jishu/167.html>

最终所有的memtable数据都是保存在SkipList中。

### 分层存储/Tiered Storage

分层存储：

<https://github.com/facebook/rocksdb/wiki/Tiered-Storage-%28Experimental%29>

### Pipeline write

#### 概述

To enable pipelined write, simply set Options.enable\_pipelined\_write=true. db\_bench benchmark shows 20% write throughput improvement with concurrent writers and WAL enabled, when DB is stored in ramfs and compaction throughput is not the bottleneck.

要启用流水线写入，只需设置 Options.enable\_pipelined\_write = true。当DB存储在ramfs中并且压缩吞吐量不是瓶颈时，db\_benchbenchmark显示启用并发编写器和WAL可提高20%的写入吞吐量。

**pipeline write实现了哪些功能：**

流水线会负责把多个写请求打包成一个WriteGroup，把这些写请求的wal日志打包成连续的wal日志一起写下去，然后负责通过其它线程，把各自的数据插入到metmable当中。

流水线本身也实现了switch metmable与 metmable insert两种请求的barrier本身也兼顾了stalling write机制。

图示

描述已自动生成

这里pipeline的主要意思，就是上一轮的Group写入wal结束以后，下一轮新的Group又可以开始写wal。由不同的线程（Group）像流水线一样接替写wal日志。

图表, 箱线图

描述已自动生成

而pipeline的主要逻辑就是实现上述功能，如何在并发的写请求里面挑出来GroupLeader，GroupFollower，如何实现多个Group之间流水线一样的写入wal日志。

同时pipeline还要兼容 switchmemtable与写入metmable的并发。

#### 关键对象

WriteBatch 每一个写请求，都会带自己的WriteBatch，它里面记录了事务写入的key value数据。

WriteThread::Writer： 代码一个写请求，在并发纬度，也代表当前写请求所在的线程。多个Writer会形成一个双向链表。

WriteThread 每一个DB只有一个object，多线程（多个写请求）并发过程中如何实现pipeline写入的。

WriteThread ::WriteGroup 多个顺序的 WriteThread::Writer 的打包，顺序按照 WriteThread::Writer 调用 WriteThread::JoinBatchGroup 的先后顺序（严格意义上应该是指 WriteThread::Writer 链表的顺序）

#### 样例

下图展示了一个打包wal的样例， Writer1 把Writer1 Writer2 Writer3 打包成一个WriteGroup。

1、初使状态，当前的DB还没有任务一个写请求进来。

2、第一个写请求(Writer1)进来，因为 newest\_writer\_ 为空，Writer1（它所在的线程）被委派成了 STATE\_GROUP\_LEADER 角色，负责把多个writer的wal日志进行打包，并写入。 而 newest\_writer\_ 指向了 Writer1。

3、第二个写请求进来(Writer2)，直接通过link\_older指针挂到Writer1的后面，因为当前已经有STATE\_GROUP\_LEADER ，所以它不会成STATE\_GROUP\_LEADER ,等待被委派其它角色。

4、第三个写请求进来(Writer3)，直接通过link\_older指针挂到Writer1的后面，因为当前已经有STATE\_GROUP\_LEADER ，所以它不会成STATE\_GROUP\_LEADER ,等待被委派其它角色。

5、Wrtier1 所在EnterAsBatchGroupLeader，把Writer1 Writer2 Write3 要写入的数据打包成一个WriteGroup，叫做wal\_write\_group, WriteGroup的leader 指向 Writer1, WriteGroup的last\_writer\_ 指向Writer3。

6、Writer1 所在的线程，把wal\_write\_group写入wal 日志（rocksdb的流程，tdsql 3.0 不需要）

7、Writer1 ExitAsBatchGroupLeader

case1: 如果这个过程中，没有新来的write请求，也就是说 7.1 先于7.2 发生，那么 newest\_writer\_会被置为nullptr，Writer4 在linkOne的时候，发现newest\_writer\_为nullptr，也会像Writer1 一样，被委派成 STATE\_GROUP\_LEADER 角色

case2: 如果 7.2 先于 7.1发生，ExitAsBatchGroupLeader会把Writer4 置为 STATE\_GROUP\_LEADER

图形用户界面

中度可信度描述已自动生成

#### writer角色的转换

WriteThread::Writer 通过 WriteThread::JoinBatchGroup 后，会被赋予各种角色，下面是同一个writer在各个角色之间的转换

图示

描述已自动生成

#### 关键代码分析

##### EnterAsBatchGroupLeader

该函数是一个write抢到了GroupLeader的角色，开始设置Group的边界，把哪几个Write打包成一起。

这里有两个条件会限制group的大小：

1. 这个group包含所有的WriteBatch的大小
2. 找到了最后一个writer了

图示

描述已自动生成

##### WriteThread::LinkOne

尝试把当前writer的old\_linker 连接到 newest\_writer\_ 如果中间没有其它并发linkone，那么年上一次 newest\_writer\_ 是不是nullptr，如果是的话，返回true : link as leader

图示

描述已自动生成

/\*

\* 尝试把当前writer的old\_linker 连接到 newest\_writer\_

\* 如果中间没有其它并发linkone，那么年上一次 newest\_writer\_ 是不是nullptr，如果是的话，返回true : link as leader

如果它链接到已经有了 leader了，它就简单的连接到 newest\_writer\_ 的后面。

\*/

bool WriteThread::LinkOne(Writer\* w, std::atomic<Writer\*>\* newest\_writer) {

assert(newest\_writer != nullptr);

assert(w->state == STATE\_INIT);

Writer\* writers = newest\_writer->load(std::memory\_order\_relaxed);

// w（当前的writer） 来抢 newest\_writer 这个位置，如果抢占成功了。而且上一次 newest\_writer 是nullptr，

// 那么当就是 leader writer, return true

// 如果 newest\_writer 上个值有人了，那说明它没有抢到leder

while (true) {

// If write stall in effect, and w->no\_slowdown is not true,

// block here until stall is cleared. If its true, then return

// immediately

if (writers == &write\_stall\_dummy\_) {

if (w->no\_slowdown) {

w->status = Status::Incomplete("Write stall");

SetState(w, STATE\_COMPLETED);

return false;

}

// Since no\_slowdown is false, wait here to be notified of the write

// stall clearing

{

MutexLock lock(&stall\_mu\_);

writers = newest\_writer->load(std::memory\_order\_relaxed);

if (writers == &write\_stall\_dummy\_) {

stall\_cv\_.Wait();

// Load newest\_writers\_ again since it may have changed

writers = newest\_writer->load(std::memory\_order\_relaxed);

continue;

}

}

}

w->link\_older = writers;

if (newest\_writer->compare\_exchange\_weak(writers, w)) {

return (writers == nullptr);

}

}

}

#### memtable writer 与 memtable switch的barrier

写memtable 与 memtable switch是不能并发的，这里pipeline构造了它们的barrier。

case1: 如果在pipeline write中调用了switch memtable，那么只有GroupLeader可以调用switch memtable，然后再调用 WriteThread::WaitForMemTableWriters 函数，等上一个Group所有的memtable writer都退出，然后再switch memtable，作为当前的groupLeader，当前是不会写memtable，只要保证上一个Group的memtable writer都退出，就可以保证当前没有并发任务写memtable

waitForMemTableWriters 函数分析

图示

描述已自动生成

case2: 不在pipeline write调用switch metmable，通过WriteThread::EnterUnbatched 抢一个空的pipeline的GroupLeader，然后再等上一个Group的memtable writer退出，然后再switch memtable，然后调用WriteThread::ExitUnbatched 让出自己GroupLeader的位置。

static WriteThread::AdaptationContext eu\_ctx("EnterUnbatched");

void WriteThread::EnterUnbatched(Writer\* w, InstrumentedMutex\* mu) {

assert(w != nullptr && w->batch == nullptr);

mu->Unlock();

bool linked\_as\_leader = LinkOne(w, &newest\_writer\_);// 尝试去抢一个GroupLeader

if (!linked\_as\_leader) {//

STORAGE\_LOG\_INFO("linked\_as\_follower WriteThread::EnterUnbatched::Wait enable\_pipelined\_write:%d",

enable\_pipelined\_write\_);

TEST\_SYNC\_POINT("WriteThread::EnterUnbatched:Wait");

// Last leader will not pick us as a follower since our batch is nullptr

AwaitState(w, STATE\_GROUP\_LEADER, &eu\_ctx);//没有抢成功，就等自己被设置为GroupLeader

} else { // 抢GroupLeader 成功

STORAGE\_LOG\_INFO("linked\_as\_leader WriteThread::EnterUnbatched not wait enable\_pipelined\_write:%d",

enable\_pipelined\_write\_);

}

if (enable\_pipelined\_write\_) {

WaitForMemTableWriters();//等上一个Group的memtbale writer都退出

}

mu->Lock();

}

### Write Buffer Manager

参考：

Facebook Write Buffer Manager：

<https://github.com/facebook/rocksdb/wiki/Write-Buffer-Manager>

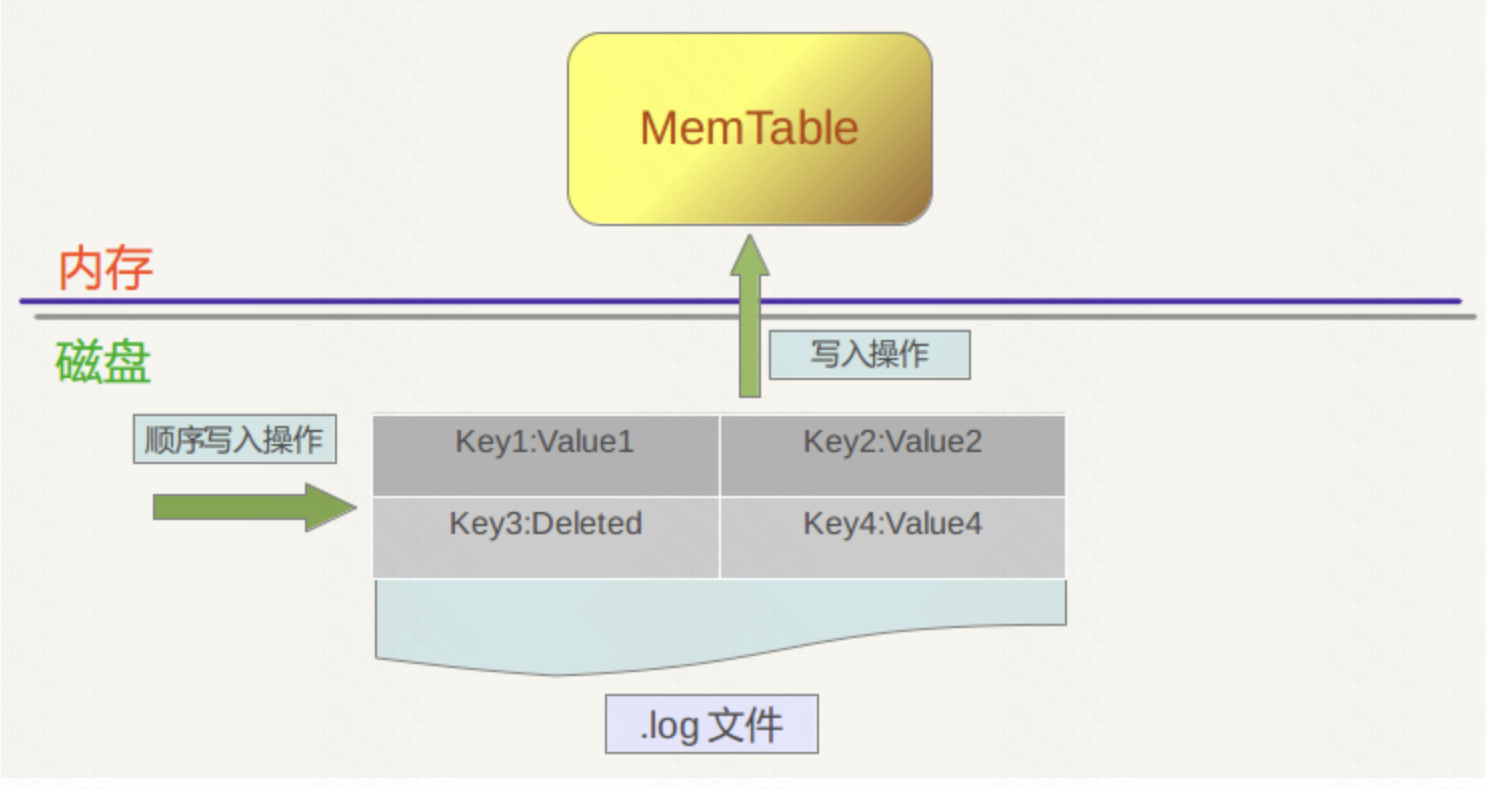
### Flush/minor compaction

参考：

Memtable flush分析：<http://mysql.taobao.org/monthly/2018/09/04/>

RocksDB的内存数据在memtable中存着，有active-memtable和 immutable-memtable两种。active-memtable是当前被写操作使用的memtable，当 active-memtable空间写满之后(Options.writebuffersize 控制其内存空间大小)这个空间会被标记为readonly成为immutable-memtable。memtable实质上是一种有序SkipList，所以写过程其实是写WAL日志和数据插入SkipList的过程。

RocksDB 的数据删除过程跟写过程相同，只不过 插入的数据是 “key:删除标记”。



immutable-memtable被写入L0的过程被称为flush或者minor compaction。flush的触发条件是immutable memtable数量超过 minwritebuffernumberto\_merge。flush过程以column family为单位，一个column family会使用一个或者多个immutable-memtable，flush会一次把所有这些文件合并后写入磁盘的L0 sstfile中。

在compaction过程中如果某个被标记为删除的 key 在某个 snapshot 中存在，则会被一直保留，直到snapshot不存在才会被删除。

RocksDB的compaction策略分为Universal Compaction和Leveled Compaction两种。两种策略分别有不同的使用场景。综述就是Leveled Compaction有利于减小空间放大却会增加读放大，Universal Compaction有利于减少读放大却会增大空间放大。

Flush是指将memtable的数据导入到sst中，变成持久化存储，就不怕数据丢失了。

1、首先在memtable的add的时候，会检测是否memtable的大小达到了max write buffer，如果是就将should\_flush\_置为true，并会在WriteBatch的Handler里面调用CheckMemtableFull，将当前column family加入flush\_scheduler。

2、在Write的时候，调用ScheduleFlushes，将需要flush的column family的memtable切换一个新的，同时将原来的memtable加入cfd的imm中，如果这个column family data的imm数量大于min\_write\_buffer\_number\_to\_merge，并启动一个新的线程调用BGWorkFlush。

由于真正的Flush过程是在另一个线程完成的，所以这个地方并不会block写过程。

另外，如果total\_log\_size大于max\_wal\_log\_size并且不是只有一个column family，也会触发flush，因为flush能将memtable持久化到磁盘上，同时对应的wal就可以删除了。

### Compaction/major compaction/压缩

参考：

<https://github.com/facebook/rocksdb/wiki/Compaction>

<https://www.cnblogs.com/ljx-null/p/16154773.html>

#### Leveled Compaction/分级压缩

#### Universal compaction style/通用压缩

#### FIFO compaction /FIFO压缩

#### Manual Compaction/手动压缩

#### Subcompaction

#### 选择压缩文件级别

#### 管理磁盘空间利用率

#### Trivial Move Compaction/琐碎移动压缩

#### Remote Compaction/远程压缩

### Merge

RocksDB自身之提供了Put/Delete/Get等接口，若需要在现有值上进行修改操作【或者成为增量更新】，可以借助这三个操作进行以下操作实现之：

调用Get接口然后获取其值；

在内存中修改这个值；

调用Put接口写回RocksDB。

如果希望整个过程是原子操作，就需要借助RocksDB的Merge接口了。[参考文档](https://www.jianshu.com/p/e13338a3f161)给出了RocksDB Merge接口定义如下：

1、封装了read - modify - write语义，对外统一提供简单的抽象接口；

2、减少用户重复触发Get操作引入的性能损耗；

3、通过决定合并操作的时间和方式，来优化后端性能，并达到并不改变底层更新的语义；

4、渐进式的更新，来均摊更新带来带来的性能损耗，以得到渐进式的性能提升。

RocksDB提供了一个MergeOperator作为Merge接口，其中一个子类AssociativeMergeOperator可在大部分场景下使用，其定义如下：

// The simpler, associative merge operator.

class AssociativeMergeOperator : public MergeOperator {

public:

virtual ~AssociativeMergeOperator() {}

// Gives the client a way to express the read -> modify -> write semantics

// key: (IN) 操作对象 KV 的 key

// existing\_value:(IN) 操作对象 KV 的 value，如果为 null 则意味着 KV 不存在

// value: (IN) 新值，用于替换/更新 @existing\_value

// new\_value: (OUT) 客户端负责把 merge 后的新值填入这个变量

// logger: (IN) Client could use this to log errors during merge.

//

// Return true on success.

// All values passed in will be client-specific values. So if this method

// returns false, it is because client specified bad data or there was

// internal corruption. The client should assume that this will be treated

// as an error by the library.

virtual bool Merge(const Slice& key,

const Slice\* existing\_value,

const Slice& value,

std::string\* new\_value,

Logger\* logger) const = 0;

private:

// Default implementations of the MergeOperator functions

virtual bool FullMergeV2(const MergeOperationInput& merge\_in,

MergeOperationOutput\* merge\_out) const override;

virtual bool PartialMerge(const Slice& key,

const Slice& left\_operand,

const Slice& right\_operand,

std::string\* new\_value,

Logger\* logger) const override;

};

RocksDB AssociativeMergeOperator被称为关联性Merge Operator，[参考文档](https://www.jianshu.com/p/e13338a3f161)给出了关联性的定义：

调用Put接口写入RocksDB的数据的格式和Merge接口是相同的

用用户自定义的merge操作，可以将多个merge操作数合并成一个

MergeOperator还可以用于非关联型数据类型的更新。例如，在RocksDB中保存json字符串，即Put接口写入data的格式为合法的json字符串。而Merge接口只希望更新json中的某个字段。所以代码可能是这样：

// Put/store the json string into to the database

db\_->Put(put\_option\_, "json\_obj\_key",

"{ employees: [ {first\_name: john, last\_name: doe}, {first\_name: adam, last\_name: smith}] }");

// Use a pre-defined "merge operator" to incrementally update the value of the json string

db\_->Merge(merge\_option\_, "json\_obj\_key", "employees[1].first\_name = lucy");

db\_->Merge(merge\_option\_, "json\_obj\_key", "employees[0].last\_name = dow");

`AssociativeMergeOperator无法处理这种场景，因为它假设Put和Merge的数据格式是关联的。我们需要区分Put和Merge的数据格式，也无法把多个merge操作数合并成一个。这时候就需要Generic MergeOperator。`

// The Merge Operator

//

// Essentially, a MergeOperator specifies the SEMANTICS of a merge, which only

// client knows. It could be numeric addition, list append, string

// concatenation, edit data structure, ... , anything.

// The library, on the other hand, is concerned with the exercise of this

// interface, at the right time (during get, iteration, compaction...)

class MergeOperator {

public:

virtual ~MergeOperator() {}

// Gives the client a way to express the read -> modify -> write semantics

// key: (IN) The key that's associated with this merge operation.

// existing: (IN) null indicates that the key does not exist before this op

// operand\_list:(IN) the sequence of merge operations to apply, front() first.

// new\_value: (OUT) Client is responsible for filling the merge result here

// logger: (IN) Client could use this to log errors during merge.

//

// Return true on success. Return false failure / error / corruption.

// 用于对已有的值做Put或Delete操作

virtual bool FullMerge(const Slice& key,

const Slice\* existing\_value,

const std::deque<std::string>& operand\_list,

std::string\* new\_value,

Logger\* logger) const = 0;

// This function performs merge(left\_op, right\_op)

// when both the operands are themselves merge operation types.

// Save the result in \*new\_value and return true. If it is impossible

// or infeasible to combine the two operations, return false instead.

// 如果连续多次对一个 key 进行操作，则可以可以借助 PartialMerge 将两个操作数合并.

virtual bool PartialMerge(const Slice& key,

const Slice& left\_operand,

const Slice& right\_operand,

std::string\* new\_value,

Logger\* logger) const = 0;

// The name of the MergeOperator. Used to check for MergeOperator

// mismatches (i.e., a DB created with one MergeOperator is

// accessed using a different MergeOperator)

virtual const char\* Name() const = 0;

};

**工作原理**

当调用DB::Put()和DB:Merge()接口时，并不需要立刻计算最后的结果。RocksDB将计算的动作延后触发，例如在下一次用户调用Get，或者RocksDB决定做Compaction时。所以，当merge的动作真正开始做的时候，可能积压(stack)了多个操作数需要处理。这种情况就需要MergeOperator::FullMerge来对existing\_value和一个操作数序列进行计算，得到最终的值。

**PartialMerge和Stacking**

有时候, 在调用FullMerge之前, 可以先对某些merge操作数进行合并处理, 而不是将它们保存起来, 这就是PartialMerge的作用: 将两个操作数合并为一个, 减少FullMerge的工作量.

当遇到两个merge操作数时, RocksDB总是先会尝试调用用户的PartialMerge方法来做合并, 如果PartialMerge返回false才会保存操作数. 当遇到Put/Delete操作, 就会调用FullMerge将已存在的值和操作数序列传入, 计算出最终的值.

**使用Associative Merge的场景**

merge操作数的格式和Put相同

多个顺序的merge操作数可以合并成一个

**使用Generic Merge的场景**

merge操作数的格式和Put不同

当多个merge操作数可以合并时，PartialMerge()方法返回true

## Cache

### Block cache

Block Cache是RocksDB的数据的缓存，这个缓存可以在多个RocksDB的实例下缓存。一般默认的Block Cache中存储的值是未压缩的，而用户可以再指定一个Block Cache，里面的数据可以是压缩的。用户访问数据先访问默认的Block Cache，待无法获取后再访问用户Cache，用户Cache的数据可以直接存入page cache中。

Cache有两种：LRUCache和BlockCache。Block分为很多Shard，以减小竞争，所以shard大小均匀一致相等，默认Cache最多有64个shards，每个shard的最小size为512k，总大小是8M，类别是LRU。

std::shared\_ptr<Cache> cache = NewLRUCache(capacity);

BlockedBasedTableOptions table\_options;

table\_options.block\_cache = cache;

Options options;

options.table\_factory.reset(new BlockedBasedTableFactory(table\_options));

这个Cache是不压缩数据的，用户可以设置压缩数据BlockCache，方法如下：

table\_options.block\_cache\_compressed = cache;

如果Cache为nullptr，则RocksDB会创建一个，如果想禁用Cache，可以设置如下Option：

table\_options.no\_block\_cache = true;

默认情况下RocksDB用的是LRUCache，大小是8MB，每个shard单独维护自己的LRU list和独立的hash table，以及自己的Mutex。

RocksDB还提供了一个ClockCache，每个shard有自己的一个circular list，有一个clock handle会轮询这个circular list，寻找过时的kv，如果entry中的kv已经被访问过则可以继续存留，相对于LRU好处是无mutex lock，circular list本质是tbb::concurrenthashmap，从benchmark来看，二者命中率相似，但吞吐率Clock比LRU稍高。

Block Cache初始化之时相关参数：

capacity总的内存使用量

num\_shards\_bits把key的前n bits作为shard id，则总shard的数目为2^ num\_shards\_bits；

strict\_capacity\_limit在一些极端情况下block cache的总体使用量可能超过 capacity，如在对block进行读或者迭代读取的时候可能有插入数据的操作，此时可能因为加锁导致有些数据无法及时淘汰，使得总体capacity超标。如果这个选项设置为true，则此时插入操作是被允许的，但有可能导致进程OOM。如果设置为false，则插入操作会被refuse，同时读取以及遍历操作有可能失败。这个选项对每个shard都有效，这就意味着有的shard可能内存已满，别的shard却有很多空闲。

high\_pri\_pool\_ratio block中为高优先级的block保留多少比例的空间，这个选项只有LRU Cache有。

默认情况下index和filter block与block cache是独立的，用户不能设定二者的内存空间使用量，但为了控制RocksDB的内存空间使用量，可以用如下代码把index和filter也放在block cache中：

BlockBasedTableOptions table\_options;

table\_options.cache\_index\_and\_filter\_blocks = true;

index与filter一般访问频次比data高，所以把他们放到一起会导致内存空间与cpu资源竞争，进而导致cache性能抖动厉害。有如下两个参数需要注意：cacheindexfilterblockswithhighpriority和highpripoolratio一样，这个参数只对LRU Cache有效，两者须同时生效。这个选项会把LRU Cache划分为高prio和低prio区，data放在low区，index和filter放在high区，如果高区占用的内存空间超过了capacity \* highpripoolratio，则会侵占low区的尾部数据空间。

pin\_l0\_filter\_and\_index\_blocks\_in\_cache把level0的index以及filter block放到Block Cache中，因为l0访问频次最高，一般内存容量不大，占用不了多大内存空间。

SimCache用于评测Cache的命中率，它封装了一个真正的Cache，然后用给定的capacity进行LRU测算，代码如下:

​ c++

​ // This cache is the actual cache use by the DB.

​ std::shared\_ptr<Cache> cache = NewLRUCache(capacity);

​ // This is the simulated cache.

​ std::shared\_ptr<Cache> sim\_cache = NewSimCache(cache, sim\_capacity, sim\_num\_shard\_bits);

​ BlockBasedTableOptions table\_options;

​ table\_options.block\_cache = sim\_cache;

大概只有容量的 2% 会被用于测算。

### SecondaryCache

参考：

<https://github.com/facebook/rocksdb/wiki/SecondaryCache-%28Experimental%29>

## WAL

参考：

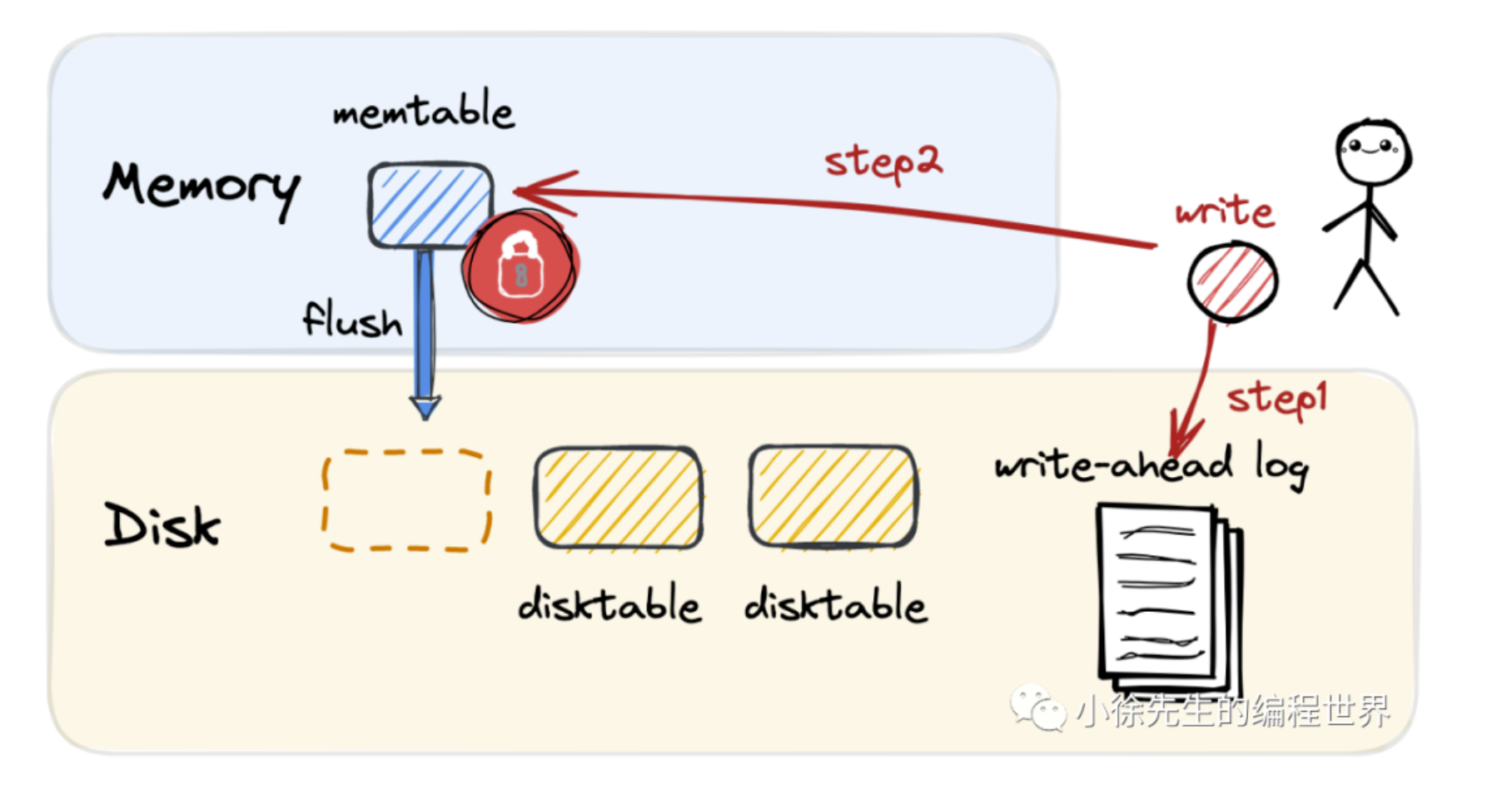
<https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-%28WAL%29>

### 概述

如何避免因宕机导致 memtable 数据丢失。

解决这个问题的手段就是WAL（write-ahead log）预写日志技术：在将数据写入memtable，先通过追加写的方式，将操作记录到处于磁盘的WAL当中，这样哪怕宕机导致内存数据丢失，也能通过重放 WAL 的方式，重新恢复 memtable 的数据。

此外，WAL和memtable可以建立对应关系，每当一个memtable被溢写到磁盘中成为disktable，其发生数据丢失问题的风险也就随之消除，因此对应的 WAL 也就可以删除了。并且，由于WAL中也是追加写的操作，属于磁盘顺序IO，因此性能不会成为瓶颈。



### 原理

1. 每次写操作，rocksdb会先写write ahead log，然后才会写db。
2. write ahead log可以配置到单独的空间，并且可以配置WAL文件的单独的删除机制。这种原因是为了保存WAL文件，达到特殊的目的，比如，其他sst文件放在不可靠存储里面，而WAL放到可靠存储里面。

#### 预写日志文件格式

#### WAL恢复模式

#### WAL性能

#### WAL压缩

## SSTable

### 版本管理

### 文件格式

#### Block-based Table Format/基于块的表格格式

#### PlainTable Format/PlainTable格式

#### CuckooTable Format/CuckooTable格式

#### Index Block Format/索引块格式

#### Bloom Filter/布隆过滤器

#### Data Block Hash Index/数据块哈希索引

### Compaction

参考：

Level Compact 分析：<http://mysql.taobao.org/monthly/2018/10/08/>

#### Rocksdb CompactionIterator实现分析

#### Rocksdb compaction files的选取逻辑

##### 概述

RocksDB的Compaction流程可以保证不同sst文件之间key-value之间的有序性，数据的压缩存储，清理过时数据。以下内容主要介绍的是Compaction过程对于Compaction FileMetaData的文件选取逻辑。

下图所列出来的是Compaction的整体流程，包括三个阶段：Prepare key， Process key 以及 Write key.

图形用户界面

描述已自动生成

Prepare key： 主要是计算每一层的Score，确定需要参与Compaction的file， 然后将参与Compaction的文件分解为多个SubCompaction（SubCompaction主要是针对L0向其他层进行compaction时，也能够并行化）。

Process key:将SubCompaction中的file构建一个MergingIterator，MergingIterator实现逻辑是小顶堆，遍历小顶堆，根据不同的key\_type, 对key value 数据进行合并以及排序操作。

Write key: 将key-value数据写入对应的block 数据之中。

##### 实现

以下部分主要内容为Prepare key实现

图示

描述已自动生成

上图为Compaction prepare key 较详细的实现，主要内容为以下几部分

根据Function Score 计算每一层的Score， 这一步的计算方式因level\_compaction\_dynamic\_level\_bytes的值而不同。

遍历每一层level，取出该层参与Compaction的文件 （Score>1），根据Cleancut取出input\_level 以及 output\_level 需要参与此次Compaction 的FileMetaData.

将以上取出来的FileMetaData的key边界取出来，边界排序去重，计算SubCompaction的个数，确定最终Subcompaction key的边界范围。

为每一个SubCompaction创建一个线程，进入下一个阶段（Process key）

###### Compaction触发条件

Rocksdb的compaction的触发条件有两类：

1、某一个level的数据太多

VersionStorageInfo的compaction\_score\_的计算方法是level0的是当前文件数目/level0\_file\_num\_compaction\_trigger，其他层是该层当前文件大小总和/该层的配置的允许文件总和最大值。

基于level的存储的compaction总的来说，就是一次挑选某一个level的一个文件，然后将该文件和高level的多个相交文件merge，最后生成多个高level的文件。具体的细节是：每次会挑选compaction score最高的一个level，并在这个level中找到一个文件大小最大，并且上一个level的相交文件没有在compaction的一个文件。

2、seek太多

疑问：

如果option里面soft\_rate\_limit设置的为0.0和1之间，compaction并不会触发，但是会触发write delay，这是为什么？

以下是判断是否发生Compaction的主体函数：

1. 是否存在超时的sst
2. files\_marked\_for\_periodic\_compaction\_ 、return bottommost\_files\_marked\_for\_compaction\_以及files\_marked\_for\_compaction\_ 文件是否为空。
3. 遍历每一层，判断其Score 是否大于1

###### Compaction Score的计算

###### Comapaction每一层level大小的计算过程

###### 挑选参与Compaction的FileMetaData文件

###### Compaction job根据获取到数据分配compaction线程

#### Rocksdb compaction流程示意图

图示

描述已自动生成

### 写stall

在DBImpl也就是db的实例里面有一个WriteController，同时在ColumnFamilySet里面也有这个WriteController的指针，这个数据结构会控制db的写stall行为。

在ColumnFamily进行SuperVersion变更的时候（增加新memtable，flush增加sst，compaction）都会查看需不需要stall Write，stall的条件是：

1）imm的数量大于等于option允许的最大数目

2）level0的文件的数量大于option允许的数目

如果没有满足上面两项，但是compaction score比较大会delay写

## MANIFEST

参考：

<https://github.com/facebook/rocksdb/wiki/MANIFEST>

### 概述

### Track WAL in MANIFEST

## 输入输出/IO

### Rate Limiter/速率限制器

### SST File Manager/ SST文件管理器

### Direct I/O

## 读写放大

### 写放大

RocksDB的写会写WAL（Write Ahead Log），如果sync的话，会写一次磁盘，然后会写memtable，写rocksdb的时候有可能会卡住。

说明：即sync刷盘可能会拖慢写操作，导致写放大。

### 读放大

RocksDB的读，会首先读memtable，如果memtable没有找到的话，会读下面level的数据，由于level0的多个sst会有交叠，所以每个sst都会通过filemeta判断在不在最小和最大的范围内，如果在就需要读这个sst的文件内容，来查看，其他level的sst文件不会有数据交叠的情况，所以只会有一个文件可能含有这个数据。

可以看出来读放大还是比较严重的。rocksdb为了减少读放大，增加了cache。

#### cache

**读cache：**

rocksdb的读cache分为两部分：table cache和block cache。这两个都是LRUCache。

block cache存储的block，包括index block和filter block（通过options可以配置）。

table cache存储的是table，是整个文件的meta信息和Foot信息。table\_cache\_size的消耗内存的大小是有Options里面的max\_open\_files决定的。

#### Bloomfilter

**Bloomfilter：**

bloomfilter的增加并不能减少写放大，因为bloomfilter是table范围的或者block范围的，而且bloomfilter是存储在文件中的，那么必须把这些从文件里面读出来后才能起到作用。

## 操作接口

参考：

RocksDB读写流程：<https://zhuanlan.zhihu.com/p/49966056>

### 写流程

参考：

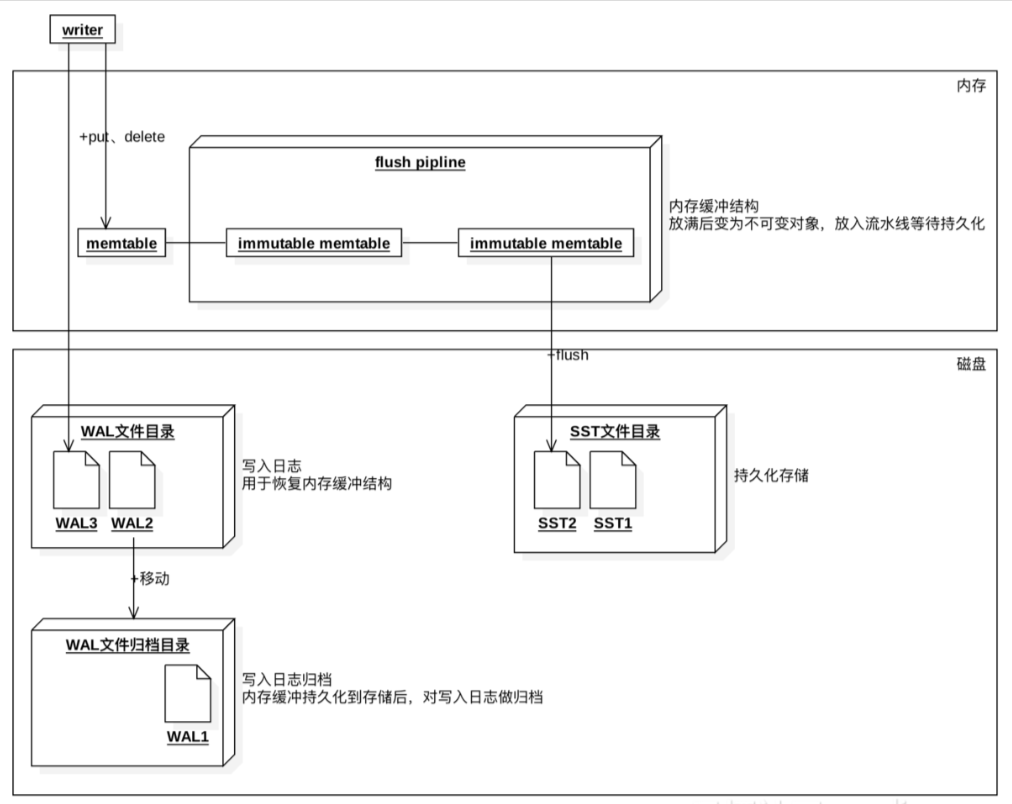
写入逻辑的实现：<http://mysql.taobao.org/monthly/2018/07/04/>

Write Prepared Policy：<http://mysql.taobao.org/monthly/2018/08/02/>

RocksDB Write Batch分析：<https://youjiali1995.github.io/rocksdb/write-batch/>

RocksDB写入流程详解：<https://zhuanlan.zhihu.com/p/33389807>

写入流程如下：

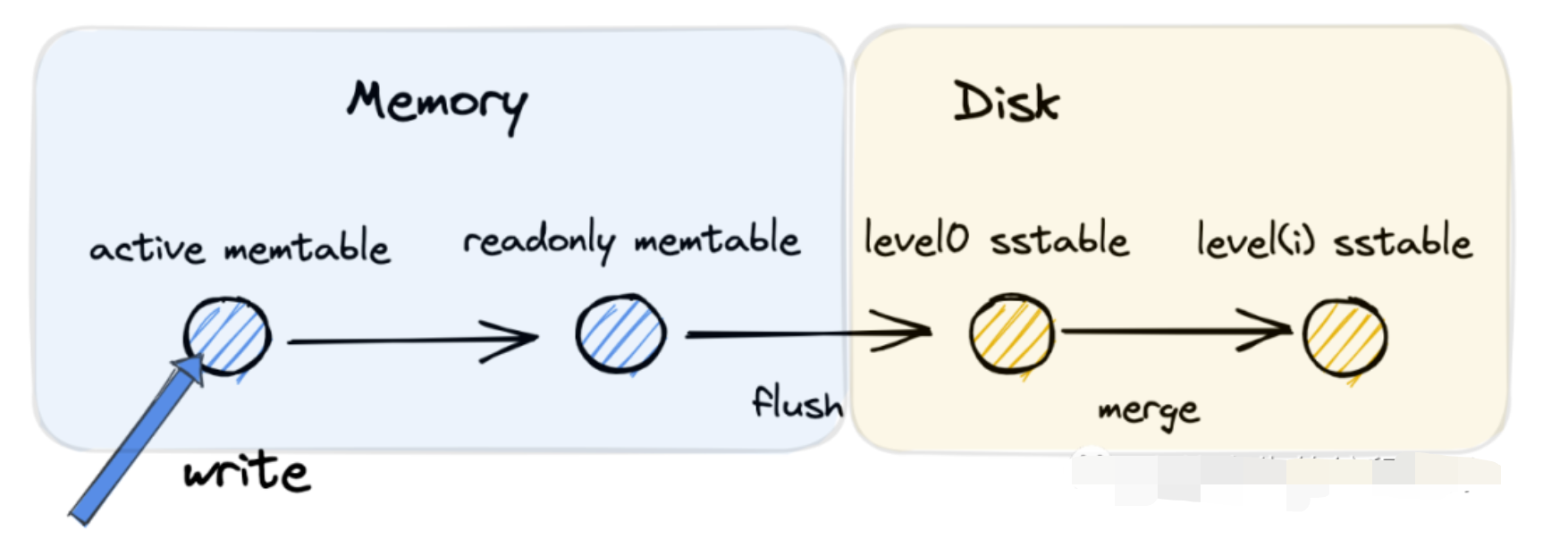


可以看到主要的三个组成部分，内存结构memtable，类似事务日志角色的WAL文件，持久化的SST文件。

数据会放到内存结构memtable，一定条件下触发写到到SST文件。写入WAL文件是可选的，用来恢复未写入到磁盘的memtable。

RocksDB写入时，直接以append方式写到log文件以及memtable，随即返回，因此非常快速。

memtable/immute memtable触发阈值后，flush到Level0 SST，Level0 SST触发阈值后，经合并操作（compaction）生成level 1 SST，level1 SST合并操作生成level 2 SST，以此类推，生成level n SST。



* 基于就地写模式（这里是优化为就地写而非顺序写，但是这里是基于内存而非磁盘的就地写，因此性能损耗非常小），写入内存中的active memtable
* active memtable达到阈值后转为只读的readonly memtable
* readonly memtable会flush到磁盘，成为level0的sstable
* level(i)层数据容量达到后，会基于归并的方式合并到level(i+1)，以此类推

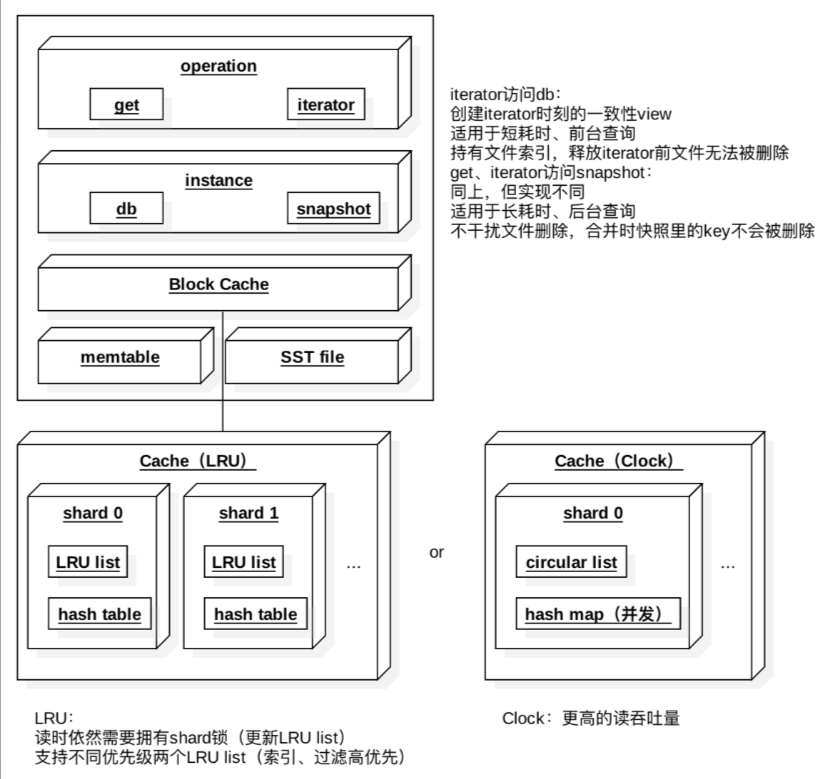
### 读流程

参考：

数据的读取：<http://mysql.taobao.org/monthly/2018/11/05/>

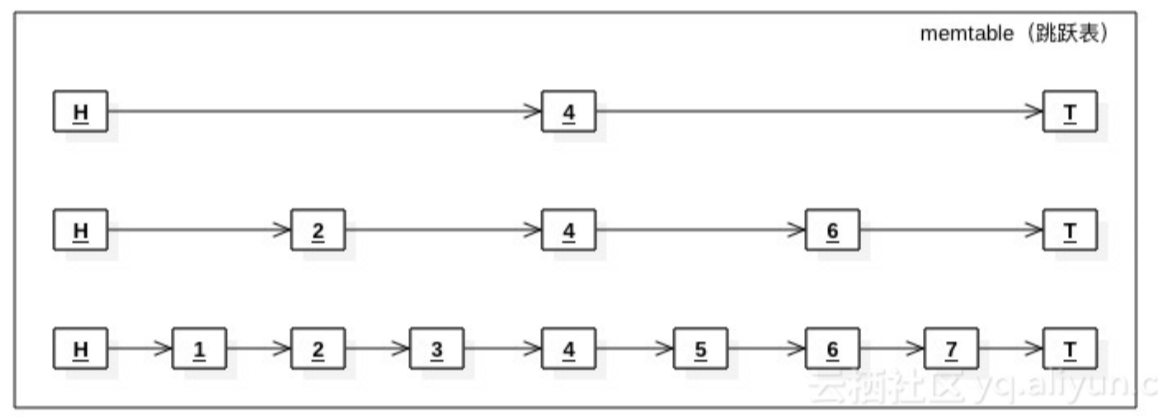
<http://mysql.taobao.org/monthly/2018/12/08/>

读取流程如下：

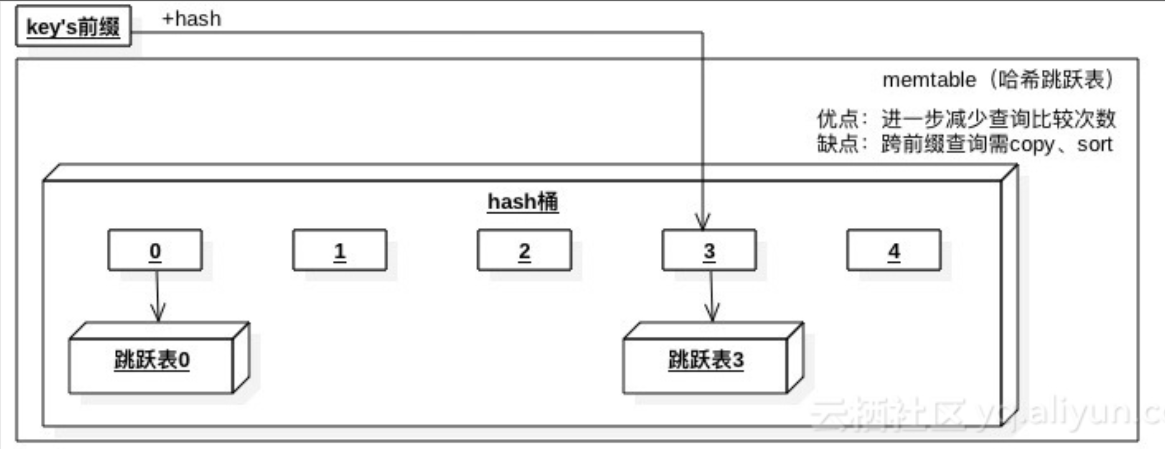


memtable和SST文件组成数据的全集。之上是缓存层，缓存为提升查询性能做了分片，底层都采用hash查询，不同缓存结构的区别在于热点数据的替换逻辑。访问数据库时，都是访问的打开时间点的view（我猜测一个key有不同时间戳的多条记录）。除了直接查询db，还提供了查询快照的机制。直接访问db时，会持有文件句柄，这样多个SST文件合并时，已经被合并但被访问的文件就不能被删除。而快照机制保证了访问过程中文件能被删除（我并未想明白如何做到的），不过打开期间被删除的key的记录还会在新合并的文件里存在。

memtable的结构有几种可选，本质都是排序的结构（为了支持范围查询）

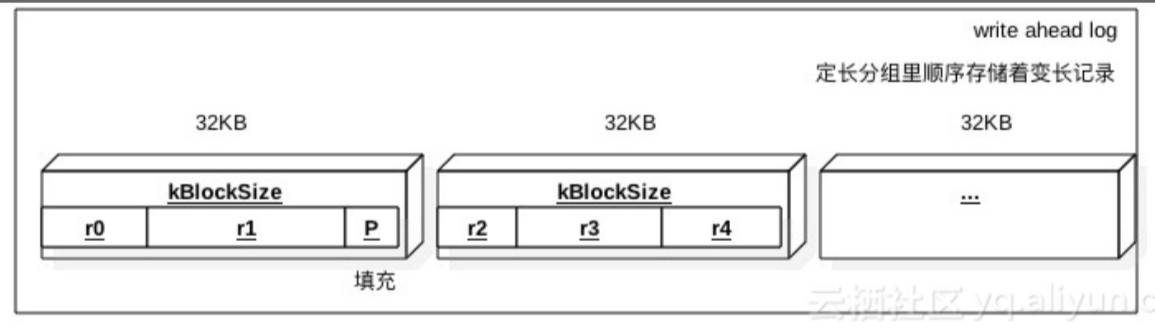


其中之一是上图的跳跃表，不了解跳跃表机制的读者可以简单理解为有序支持近似二分查找的时间复杂度为log2(N)的结构

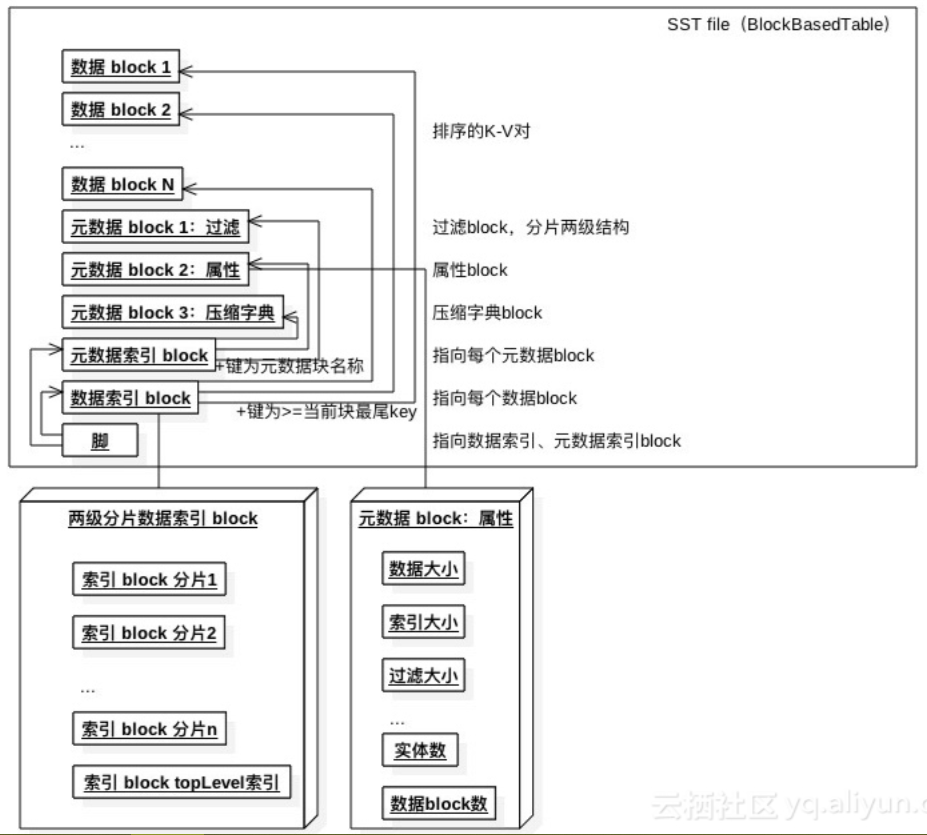


另外一种是hash结合跳跃表，是按照key的前缀做hash，单独访问一个key时性能更好，范围查询性能会差些。

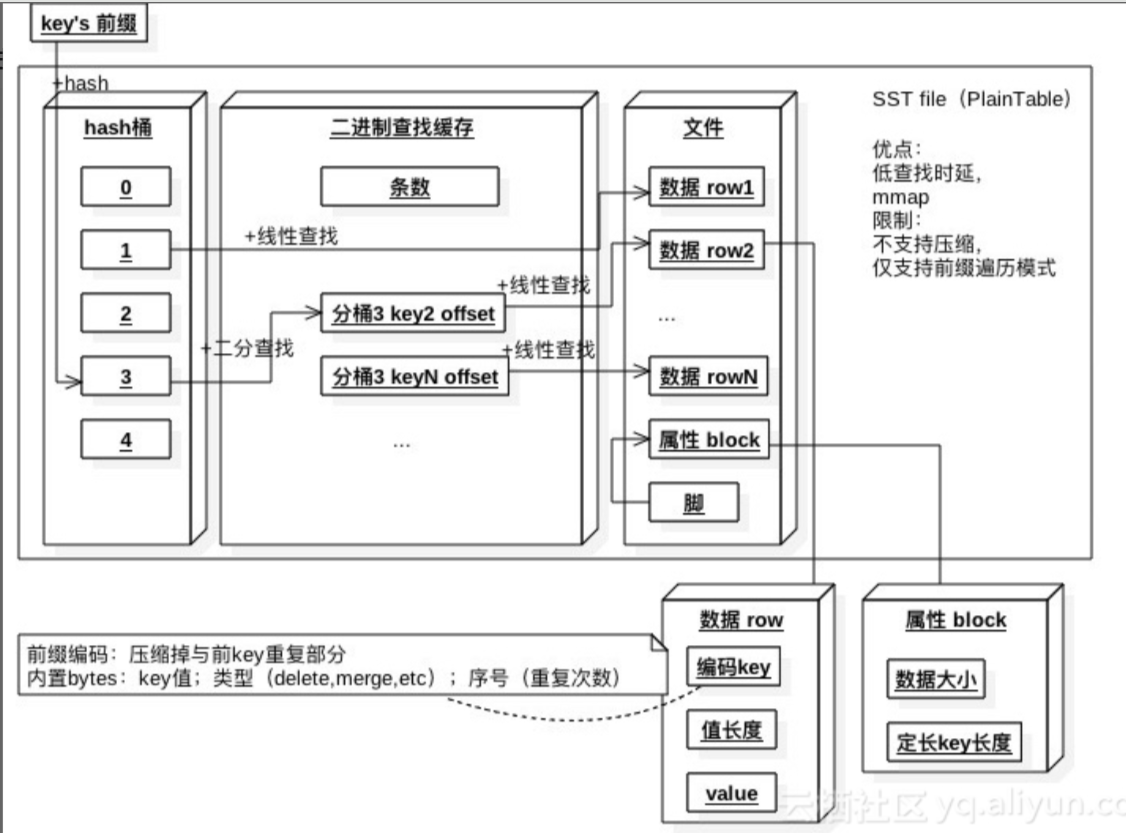
WAL文件结构如下图，按照写入的顺序来存储变长的K-V，按照固定长度来分组存储（可能一个K-V跨多个分组）的目的是便于读取



支持几种SST文件结构



上图为按照多块来存储的结构。每块的K-V都是有序的，而多块也是有序的。文件中包含元数据相关的信息，包括数据压缩字典、过滤器等。会按照数据块所属的K-V范围来创建索引，为提升查询性能会给索引分片。



另外一种结构是每个K-V来存储。它的索引比较特殊，由hash结构和二进制查找缓存两部分组成。依然按照key的前缀做hash，如果桶对应的K-V记录很少，则直接指向第一个key（有多个key属于该桶）的记录位置。如果属于桶的K-V记录多于16条，或者包含多于一个前缀的记录，则先指向二进制查找缓存（先二分查找），而后指向第一个key的记录位置。

随着K-V的写入，会生成很多的SST文件，这部分文件需要被合并到一起。从而降低打开文件数量，并且移除已经不存在的记录。通常可以配置两种方式，通用合并（下图左侧）与level合并（右侧）。



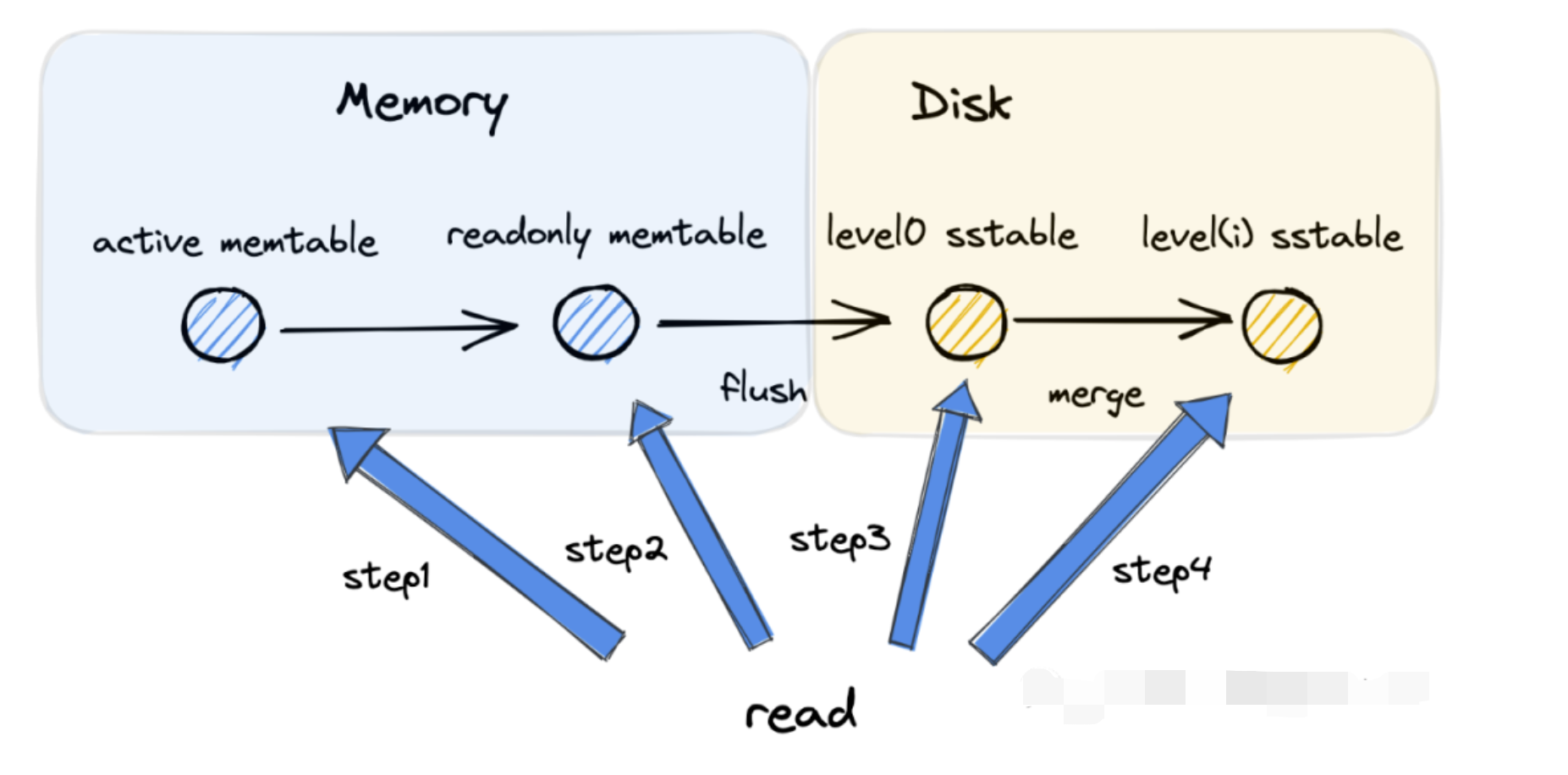
其中一个概念是level，可以简单理解成越老的数据在越高的level（也就是数据最初写入到最低的level，level0就是memtable）。

我将通用合并简单理解为一种简单粗暴的合并，可以尽量降低写磁盘的压力，会增大读取的压力，临时空间占用大。

一般多采用level合并的方式。每个level都有max大小，超出后会触发本level与下一level的文件合并到一起。不同level的合并是可以并发执行的。

对rocksdb做个总结。所有记录在业务上是有序的，对key的查询其实会执行类似二分查找。持久化是通过写入有序文件来实现的。高性能的写入是通过先写入内存结构来保证的（写满的内存结构刷到持久化文件）。提供了level机制对数据做分层，优先查询最新写入的level来优化查询性能。

按照memtable --> Level 0 SST–> Level 1 SST --> … -> Level n SST的顺序读取数据。这和记录的新旧顺序是一的。因此只要在当前级别找到记录，就可以返回。



* 尝试读 active memtable
* 尝试读 readonly memtable
* 尝试读 level0，需要按照溢写顺序进行倒序，依次读 level0 中的每个 sstable（level0 sstable间数据可能冗余）
* 根据全局的索引文件，依次读level1~levelk，每个level最多只需要读一个sstable
* 读一个sstable时，借助内部的bloom filter和索引，加速查询流程

综上所述，在lsm tree架构下，一次读操作可以在常数级别的IO次数下完成，同时每次 IO 操作中则需要承受对应 table 内数据量对数级别的查询时间复杂度.

# 日志和监控

参考：

<https://github.com/facebook/rocksdb/wiki/Logger>

## Logger/记录器

## Statistics/统计数据

## Compaction Stats and DB Status/Compaction统计和DB状态

## Perf Context and IO Stats Context/性能和IO统计上下文

## EventListener/事件监听器

# 事务

参考：

TransactionDB 介绍：<http://mysql.taobao.org/monthly/2018/10/09/>

RocksDB TransactionDB事务实现分析：<https://yq.aliyun.com/articles/272166/>

RocksDB事务分析：<https://zhuanlan.zhihu.com/p/35195328>

## 基本概念

### LSN (log sequence number)

Q1：此处的LSN和时间戳的关系？在分布式系统下，一个事务的多个节点上的LSN之间应该是什么关系？

A1：TDStore已经修改LSN的获取方式为从MC来全局获取，事务开始时获取一次，事务提交时获取一次。

RocksDB中的每一条记录(KeyValue)都有一个LogSequenceNumber(后面统称lsn)，从最初的0开始，每次写入加1。该值为逻辑量，区别于InnoDB的lsn为redo log物理写入字节量。

这个lsn在RocksDB内部的memtable中是单调递增的，在WriteAheadLog(WAL)中以WriteBatch为单位递增(count(batch.records)为单位)。

Q2：WriteBatch，是否类似于OCC的写集？如果是，则读集是否应该维护一个类似WriteBatch的东东？

但后面谈到锁时，表明WriteBatch不等同于OCC的写集，因为写写并发提前被封锁机制禁止了。

WriteBatch是一次RocksDB::Put()的原子操作集合，不同的WriteBatch间是遵循ACID特性(要么完全成功要么完全失败，并且相互隔离)，结构如下：

WriteBatch :=

sequence: fixed64

count: fixed32

data: record[count]

从RocksDB外部能看到的LSN是按WriteBatch递增的(LeaderWriter(或LastWriter)最后一次性更新)，所以进行snapshot读时，使用的就是此lsn。

Q3：考虑RDB的snapshot和MVCC中的snapshot的异同点。

同时考虑，对于分布式事务中同一个事务的不同的节点上，子节点的快照的建立，依据的是谁（是子节点自己的LSN—不是，计划使用HLC）？

A3：分布式事务开始前，SQLEngine从mc获取开始时间戳，各个子节点以该开始时间戳作为snapshot

注意: 在WAL中每条WriteBatch的lsn并不严格满足以下公式(比如2pc情况下):

lsn(WriteBatch[n]) < lsn(WriteBatch[n+1])，可能相等

Q4：考虑为什么会出现此处说的“可能相等的情况”？

一个事务，是否可以使用多个WriteBatch，以实现对于大事务的支持？

A4：数据只有写入到memtable中，lsn才会加一，而并不是所有的writebatch都会向memtable里写数据，比如事务2pc中prepare阶段的prepare writebatch，只会写wal，而不会写memtable（写memtable的时机为commit阶段）。很少情况使用多个WriteBatch，对大事务的支持，在同一region上使用多个WriteBatch，未必有效果哈。从资源消耗上来讲，所有WriteBatch都是存储在内存中的，分成多个WriteBatch并不会降低内存的消耗。从实现方式上来讲，原生RocksDB通过WriteBatch保证事务的原子性，如果大事务使用多个WriteBatch，则这些WriteBatch之间还需要使用额外的手段保证原子性，会加大事务提交的开销。

Q4.1 回复还需要细致一点儿。

1、如一个WriteBatch最大支持多大的写事务量？即单个事务的更新量的极限是多少？

2、“很少情况使用多个WriteBatch”—是一定没有还是个别情况有？如果有，特别需要明确说明。

3、可以深入说明LSN对于数据可见性的价值。标志了一个事务的提交，是写入数据到memtable还是LSN加一，还是如图1中有commit的一个标志然后才写入数据到memtable？

`

### Snapshot

Snapshot是RocksDB的快照，实际存储的就是一个lsn.

class SnapshotImpl {

public:

// 当前的lsn

SequenceNumber number\_;

private:

SnapshotImpl\* prev\_;

SnapshotImpl\* next\_;

SnapshotList\* list\_;

// unix时间戳

int64\_t unix\_time\_;

// 是否属于Transaction(用于写冲突)

bool is\_write\_conflict\_boundary\_;

};

查询时如果设置了snapshot为某个lsn, 那么对于此snapshot的读来说，只能看到lsn(key)<=lsn(snapshot)的key，大于该lsn的key是不可见的。

Q5：为什么是一个双向列表？这样做有什么好处？

一个snapshot不是隶属于一个事务？

A5：维护snapshot双向列表的目的是，方便定位最旧的snapshot，以避免进行compaction操作时，将被该snapshot可见的中间版本清除掉，造成数据不一致

Q5.1：如果链表中间有个SI对应的事务已经提交，则compaction是否清理该SI对应的数据版本？

Q5.2： 一定需要双向链表，才可以找到最旧的SI？ 且一定需要该双向链表吗？

Q5.3：该双向链表是RDB的原生设计？

snapshot的创建和删除都需要由一个全局的DoubleLinkList (DBImpl::SnapshotList)管理，天然的根据创建时间(同样也是lsn大小)的关系排序，使用之后需要通过DBImpl::ReleaseSnapshot释放。snapshot还用于在RocksDB事务中实现不同的隔离级别。

### 隔离级别

Q6：隔离级别是如何实现的（这里没有具体说明）？实现效率可进行比较。优先级，中

A6：TDStore使用rocksdb提供的乐观事务，未提交事务所写入的数据会存储在WriteBatch中，对其它事务不可见，保证了Read Committed。在此基础上，RocksDB提供了SI隔离级别，事务开始时可以通过为事务设置snapshot，保证Repeatable Read。

为了实现事务下的一致性非锁定读(读可以并发)，不同的数据库(引擎)实现了不同的读隔离级别。SQL规范标准中定义了如下四种：

ReadUncommited ReadCommited RepeatableRead Serializable

Oracle No Yes No Yes

MySQL Yes Yes Yes Yes

RocksDB No Yes Yes No

ReadUncommitted 读取未提交内容，所有事务都可以看到其他未提交事务的执行结果。存在脏读。

ReadCommitted读取已提交内容，事务只能看见其他已经提交事务所做的改变，多次读取同一个记录可能包含其他事务已提交的更新。

RepeatableRead 可重读，确保事务读取数据时，多次操作会看到同样的数据行(InnoDB通过NextKeyLocking对btree索引加锁解决了幻读)。

Serializable串行化，强制事务之间进行排序，不会互相冲突。

大部分数据库(如MySQL InnoDB、RocksDB)，通过MVCC都可以实现上述的在非排它锁锁定情况下的多版本并发读。

## RocksDB Transaction

简单的例子：

// 基本配置,事务相关操作需要TransactionDB句柄

Options options;

options.create\_if\_missing = true;

TransactionDBOptions txn\_db\_options;

TransactionDB\* txn\_db;

// 用支持事务的方式opendb

TransactionDB::Open(options, txn\_db\_options, kDBPath, &txn\_db);

// 创建一个事务上下文, 类似MySQL的start transaction

Transaction\* txn = txn\_db->BeginTransaction(write\_options);

// 直接写入新数据

txn->Put("abc", "def");

// ForUpdate写，类似MySQL的select ... for update

s = txn->GetForUpdate(read\_options, "abc", &value);

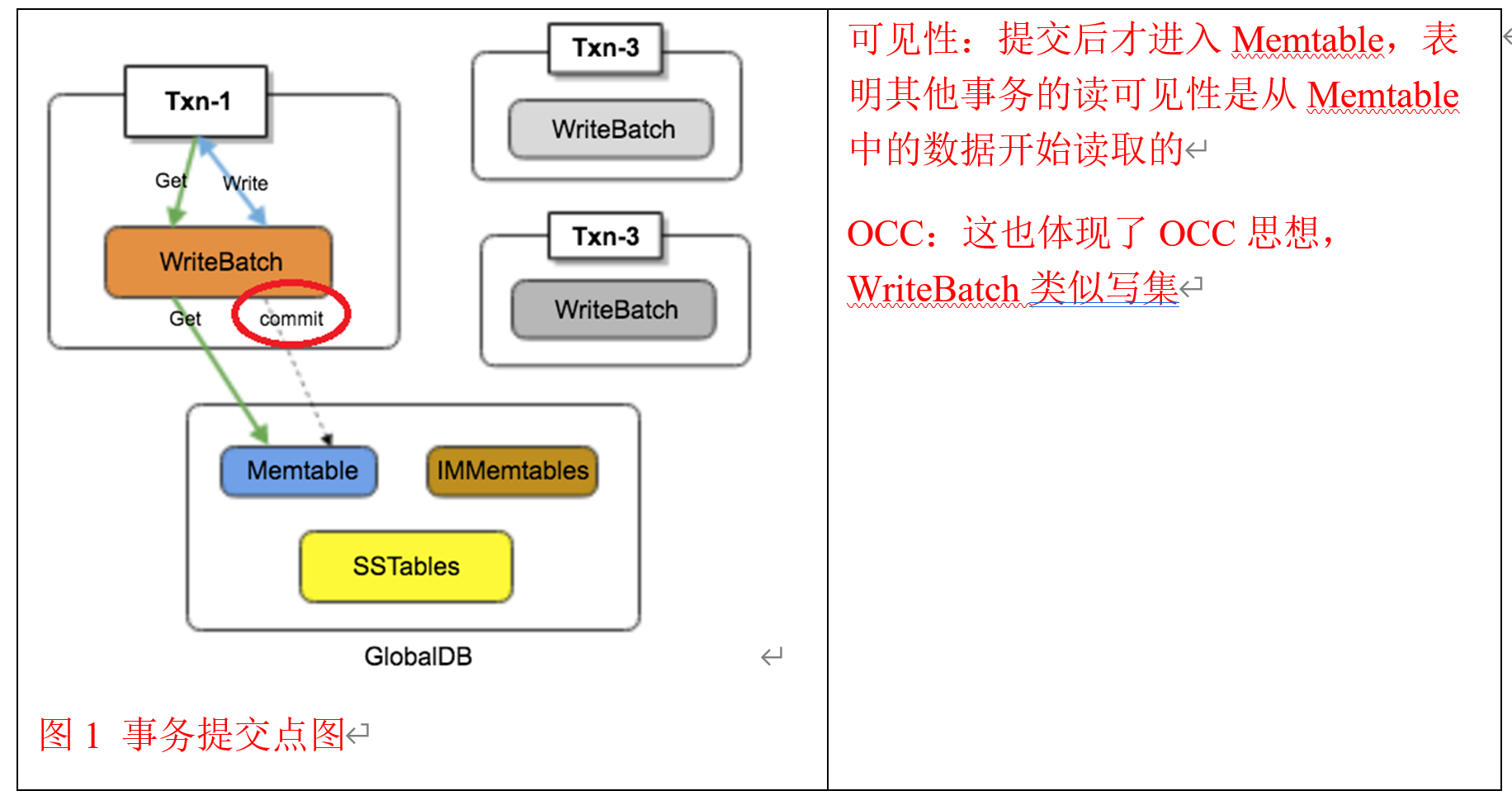
txn->Commit(); // or txn->Rollback();

如图1，RocksDB的一个事物操作，是通过事物内部申请一个WriteBatch实现的，所有commit之前的读都优先读该WriteBatch(保证了同一个事务内可以看到该事务之前的写操作)，写都直接写入该事务独有的WriteBatch中，提交时在依次写入WAL和memtable，依赖WriteBatch的原子性和隔离性实现了ACID。

Q7：分布式事务，父事务和子事务之间，需要传递什么信息以维系父子事务之间的关系？数据结构应该怎么修改？

A7：读写阶段，SQLEngine维护当前分布式事务涉及到的region列表；2PC阶段，SQLEngine将region列表同步给协调者，协调者再把region列表同步给各个参与者。同一个分布式事务中，各个region上的子事务trans id相同

Q7.1“协调者再把region列表同步给各个参与者”—为什么这么做？参与者什么情况下需要知道region列表中的所有内容?



有些单独写操作也可以通过TransactionDB直接写

txn\_db->Put(write\_options, "abc", "value");

txn\_db->Get(read\_options, "abc", &value);

用TransactionDB::Put()，内部会直接生成一个auto transaction，将这个单独的操作封装成一个transaction，并自动commit。所以在TransactionDB中，所有的入口内部都会转化成trasaction(所以显示的transaction是可以马上读取到了外面TransactionDB::Put()的数据，注意这不属于脏读)这个和MySQL的形式是类似的，默认每个SQL都是个auto transaction。但这种transaction是不会触发写冲突检测。

Q8: 讨论“auto transaction”的价值。

“用TransactionDB::Put()，内部会直接生成一个auto transaction，将这个单独的操作封装成一个transaction，并自动commit。”，这说明，一个put算是一个事务，那我们在一个事务中写入多条KV，则不能使用put？而是有另外的接口？

A8：这里我们用TransactionDB::BeginTransaction创建一个新的Transaction，再调用Transaction::Put，在该Transaction提交之前，写入的数据都在内存的WriteBatch里，对其它事务不可见，在提交阶段再将数据刷入到memtable中，以此保证事务多次写入的原子性。

### GetForUpdate

类似MySQL的select ... for update，RocksDB提供了GetForUpdate接口。区别于Get接口，GetForUpdate对读记录加独占写锁，保证后续对该记录的写操作是排他的。所以一般GetForUpdate会配合snapshot和SetSnapshotOnNextOperation()进行读，保证多个事务的GetForUpdate都可以成功锁定，而不是一个GetForUpdatech成功其他的失败。尤其是在一些大量基于索引更新的场景上。

### 事务并发

不同的并发事务之间，如果存在数据冲突，会有如下情况：

* 事务都是读事务，无论操作的记录间是否有交集，都不会锁定。
* 事务包含读、写事务：

所有的读事务不会锁定，读到的数据取决于snapshot设置。

写事务之间如果不存在记录交集，不会锁定。

写事务之间如果存在记录交集，此时如果未设置snapshot，则交集部分的记录是可以串行提交的。如果设置了snapshot，则第一个写事务(写锁队列的head)会成功，其他写事务会失败(之前的事务修改了该记录的情况下)。

### 独占写锁和写冲突

RocksDB事务写锁是基于Key Locking行锁的(实现上锁力度会粗一些)，所以在多个Transaction同时更新一条记录，会触发独占写锁定。如果还设置了snapshot的情况下，会触发写冲突分析。每个写操作(Put/Delete/Merge/GetForUpdate)开始之前，会进行写锁定，见TransactionLockMgr代码。如果存在记录有交集，写锁定会锁住一片key保证只有一个事物会独占写。

|  |  |
| --- | --- |
| image.png  图2 封锁机制 | Q9：数据经过TransactionLockMgr代码才会进入WriteBatch，即对于写写冲突会进行封锁，这违背了OCC的原则。  思考是否需要改进？怎么改进？  A9：目前RocksDB同时提供了乐观事务和悲观事务，TDStore是在乐观事务基础上改的，而TransactionLockMgr是悲观事务用到的。  Q9.1 这个回答是想表明图2中的机制，在TDStore的实现中，是被跳过去不采纳了，是不？ |

内部实现还是比较精炼的，全局有个LockMaps结构，里面按照ColumnFamily级别和num\_strips(默认16)级别做了shard进一步降低冲突(此处RocksDB还针对每个LockMap做了ThreadLocal优化)。最底层是一个ColumnFamily下某一个strip的LockMapStripe结构

Q10：如下内容，如果是所有key共用OS锁，则需要改进，如按range分配锁，或固定X个锁，自动划分等。优先级中。

struct LockMapStripe {

// 当下所有keys共用的os锁

std::shared\_ptr<TransactionDBMutex> stripe\_mutex;

std::shared\_ptr<TransactionDBCondVar> stripe\_cv;

// key -> 记录key， value -> 每个key对应的LockInfo结构

// map中所有的key共享上述os锁，作者这里提到了未来会有更细粒度的锁

// TODO(agiardullo): Explore performance of other data structures.

std::unordered\_map<std::string, LockInfo> keys;

};

struct LockInfo {

// 是否是独占锁(也可以是共享锁)

bool exclusive;

// 等待这个key的所有事务链表

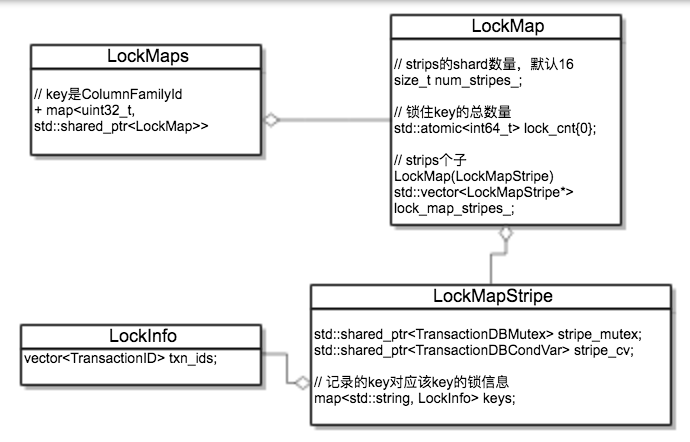
autovector<TransactionID> txn\_ids;

// 锁超时时间

uint64\_t expiration\_time;

};

关系图



针对每一个LockMapStripe里所有的key，有一个LockInfo(包含是否是排它锁，这个key挂的事务ID列表,超时时间)的map，所有落在这个map里的key如果存在并发写的情况，则会等待写锁释放。这里有个粒度问题，两个不相关的key如果落在同一个map里，也会等写锁。不如InnoDB的页锁冲突小，RocksDB作者在注释里提到之后会有更好的方案

加锁代码：

Status TransactionImpl::TryLock(ColumnFamilyHandle\* column\_family,

const Slice& key, bool read\_only,

bool exclusive, bool untracked) {

// tracked\_keys\_cf记录着当前事务中所有操作的key(涉及所有ColumnFamily)

auto iter = tracked\_keys\_cf->second.find(key\_str);

if (iter == tracked\_keys\_cf->second.end()) {

// 没找该key说明之前该事务之前一定没有独占锁定这个key

previously\_locked = false;

} else {

if (!iter->second.exclusive && exclusive) {

// 如果之前是共享锁，现在申请独占锁，则进行锁升级

lock\_upgrade = true;

}

previously\_locked = true;

current\_seqno = iter->second.seq;

}

if (!previously\_locked || lock\_upgrade) {

// 通过全局的LockMgr独占锁定该key(内部使用os锁)，如果没有其他事务操作该key(也可

// 能不同的key命中同一个LockMapStrip)，则TryLock理解返回并持有该key独占写锁。否则，

// TryLock需要等待其他事务释放该key的独占写锁，或者等待其他事务锁超时

s = txn\_db\_impl\_->TryLock(this, cfh\_id, key\_str, exclusive);

}

......

// 如果没有设置snapshot方式(可以通过创建事务的TransactionOptions指定snapshot或者

// 调用Transaction的SetSnapshot()方法)，则直接获取最新的lsn

if (untracked || snapshot\_ == nullptr) {

......

} else {

// 如果设置了snapshot，需要通过ValidateSnapshot判断是否有其他事务对该key进行了

// 更改(如该事务等待TryLock独占写锁时，其他获得了该锁的事务更新了该key)。具体实现

// 就是是在memtable，immemtable以及sst中取得该key最大的lsn对应的记录(通过

// DBImpl::GetLatestSequenceForKey)，看该lsn是否大于当前snapshot的lsn，

// 大于则写冲突。

if (s.ok()) {

s = ValidateSnapshot(column\_family, key, current\_seqno, &new\_seqno);

........

}

}

if (s.ok()) {

// 将当前key写入tracked\_keys\_cf

TrackKey(cfh\_id, key\_str, new\_seqno, read\_only, exclusive);

}

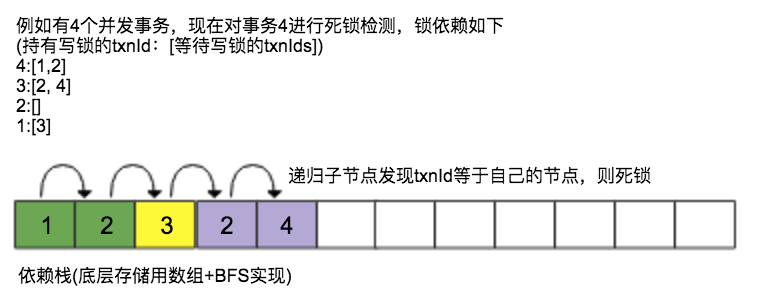
return s;

}

Q11：上述代码，优先级中。需要确认过程

### 死锁检测/超时

创建事务时 TransactionOptions.deadlock\_detect 选项可以支持死锁检测(默认不开启，性能影响较大，尤其是热点记录场景下。依赖timeout机制解决死锁)。如果多个事务之间发生死锁，则当前检测到死锁的事物失败(可以回滚)。死锁检测是通过刚才提到的LockInfo中全局事物ID列表以和当前事务ID进行环检测实现，通过广度优先递归遍历当前事务ID依赖的事务ID，判断其是否指向自己，如果能递归的找到自己的ID则说明有环，发生死锁。deadlock\_detect\_depth参数可以指定检测的深度，防止过深的依赖。



### Optimistic Transaction

相较于悲观锁，RocksDB也实现了一套乐观锁机制的OptimisticTransaction，接口上和Transaction是一致的。不过在写操作(Put/Delete/Merge/GetForUpdate)时，不会触发独占写锁和写冲突检测，而是在事务commit时("乐观"锁)，写入WAL时判断是否存在写冲突，而commit失败。这种方式的好处时，更新操作或者GetForUpdate()时，不用加独占写锁，省去了加锁的代价，乐观的认为没有写冲突，推迟到事务提交时一次性提交所有写入的key进行判断。

Q12：即与Q9不同，此是TDStore目前采用的方式，即没有经过TransactionLockMgr代码的代码。其流程，是否可以按前面锁相关的，写出来？

A12：乐观事务流程：

- 事务开始时，会传入一个开始时间戳start\_ts

-当事务写数据时，会将数据写入到内存中的WriteBatch里，WriteBatch保证事务之间的隔离性，写入到WriteBatch里的数据不会被其它事务读取

-当事务读取数据时：

 假如事务写过该数据项，则从WriteBatch中读取之前写过的版本

 否则，会选取commit\_ts＜当前事务start\_ts的事务进行读取。假如该数据项上，有正在提交的事务加了锁，那么需要等待锁释放之后再读取。

- 当事务（参与者）进入到2pc的prepare阶段时：

 先以key为单位，对所有写过的数据项加锁，假如某个数据项已经被其它事务加了锁，那么认为出现写写冲突，回滚当前事务 ---- Q12.1 确认此处是排序后加锁吧？如不是，有死锁可能。

 若加锁成功，则获取该数据项上的最新版本commit\_ts，若最新版本commit\_ts ＞ 当前事务start\_ts，也认为出现了写写冲突，回滚当前事务 ---- Q12.2 数据项上的commit\_ts记录在哪里？

 若未出现写写冲突，则将WriteBatch中的数据编码成日志，同步到raft中（数据被编码成redo log，同时后面还会附加上prepare log，prepare log相当于一个tag，表示全部数据都已经被编码为redo log）

* 当事务（参与者）进入到2pc的commit阶段时：

 将commit log同步到raft中

 将数据落盘到memtable中 ---- Q12.3 为什么是“落盘”到memtable中？是把一个事务的WriteBatch中的数据按key逐条按序拷贝到memtable中（有个排序merge的过程）？

 释放prepare阶段中对数据项加的锁

- 当事务（参与者）进入到2pc的clear阶段时（TDStore为实现无状态协调者所引入的新阶段）：

 将clear log同步到raft中 ---- Q12.4 有什么作用？

 释放事务上下文

## MVCC

RocksDB实现的ReadCommited和RepeatableRead隔离级别，类似其他数据库引擎，都使用MVCC机制。例如MySQL的InnoDB，通过undo page实现了行记录的多版本，这样可以在不同的隔离级别下，看到不同时刻的行记录内容。不过undo需要undo页的存储空间以及redo日志的保护(redo写undo)，这跟其btree的in-place update有关，而RocksDB依靠其天然的AppendOnly，所有的写操作都是后期merge，自然地就是key的多版本(不同版本可能位于memtable,immemtable,sst)，所以RocksDB首先MVCC是很容易的，只需要通过snapshot(lsn)稍加限制即可实现。

例如需要读取比某个lsn小的历史版本，只需要在读取时指定一个带有这个lsn的snapshot，即可读到历史版本。所以，在需要一致性非锁定读读取操作时，默认ReadCommited只需要按照当前系统中最大的lsn读取(这个也是默认DB::Get()的行为)，即可读到已经提交的最新记录(提交到memtable后的记录一定是已经commit的记录，未commit之前记录保存在transaction的临时buffer里)。在RepeatableRead下读数据是，需要指定该事务的读上界(即创建事务时的snapshot(lsn)或通过SetSnapshot指定的当时的lsn)，已提交的数据一定大于该snapshot(lsn)，即可实现可重复读。

txn = txn\_db->BeginTransaction(write\_options);

// ReadCommited (default)

txn->Get(read\_options, "abc", &value);

txn = txn\_db->BeginTransaction(write\_options, txn\_options);

txn\_options.set\_snapshot = true;

// RepeatableRead

read\_options.snapshot = txn->GetSnapshot();

s = txn->Get(read\_options, "abc", &value);

可见snapshot对于MVCC有着很重要的意义：

1、snapshot可以实现不同隔离级别的非锁定读

2、snapshot可以用于写冲突检测

3、snapshot由全局的snapshot链表进行管理，在compaction时，会保留该链表中snapshot不被回收

Q13：为什么“snapshot由全局的snapshot链表进行管理”？类似Q5，但这里在问：全局的snapshot链表存在的价值在哪里？

A13：Snapshot的价值体现在compaction过程中，中间版本的数据不会被回收。这里举个例子：

- memtable中某key上目前有两个版本，分别为1时刻写的“a”、3时刻写的“b”。

- 在4时刻，某读事务T1获取Snapshot

- 在5时刻，某写事务T2写“c”，并提交

- 随后，rocksdb对memtable中的数据进行compaction。

Compaction操作会将数据项上的多个版本合并成一个，下沉到immutable memtable或sst中。这里，假如compaction将中间版本"b"会回收，只留下最新版本“c”，那么事务T1将无法读取到正确的版本。

所以，为了避免这种情况，rocksdb在对某数据项进行compaction时，会依靠snapshot链表，找到时间戳最早的snapshot，仅对该snapshot可见的版本之前的历史版本进行合并。在该例中，rocksdb仅会将1时刻写的“a”回收，不会回收“b”，保证事务T1读数据的正确性。

Q13.1如果目的只是为了找到最旧的SI，则记录最旧的SI即可，为什么要用双向的链表数据结构？ 双向的链表是有序的吧？最旧且已经完成的事务的SI，才可被从双向的链表中淘汰吧？能否一次淘汰最旧的n个SI？

## DQL、DML操作

### 查询操作

前述了数据可见性判断的方式，不再单独表述。

### 更新操作

### 插入操作

### 删除操作

CRDB实现方式：Tombstone：一种特殊的数据版本，Value为空，事务读到Tombstone版本则意味着当前记录此时已被删除

## 2PC两阶段提交

Q14：TDStore的2PC与原生2PC的差异在哪里？

A14：差异主要体现在：

- TDStore将原生的WAL去掉了，改成同步Raft Log，以支持多副本

- 原生2PC只提供了参与者的Prepare和Commit阶段逻辑，而TDStore在此基础上添加了协调者的逻辑：

 实现了协调者和参与者之间的通信

 TDStore的协调者是无状态的，为了实现协调者宕机恢复，TDStore为每个参与者添加了Clear阶段，保证了协调者的状态是可恢复的

 实现了异步提交机制

RocksDB除了实现了基本类型的事务，还实现了2pc(https://github.com/facebook/rocksdb/wiki/Two-Phase-Commit-Implementation。某种程度上看，需求来自于MySQL的MyRocks引擎，binlog和引擎日志(redolog、wal)有一个XA的约束，防止出现写一个日志成功，另一个失败的情况。所以需要引擎日志实现2pc来支持binlog和引擎日志的原子提交。

详细文档可参见：

<https://github.com/facebook/rocksdb/wiki/Two-Phase-Commit-Implementation>

两阶段提交在原有的Transaction基础之上，在写记录和commit之间增加了一个Prepare操作：

BeginTransaction;

Put()

Delete()

.....

Prepare(xid)

Commit(xid) // or Rollback(xid)

### 2PC实现原理

前面几个步骤和普通的Transaction基本都是一直的，主要是后面Prepare和Commit有所区别。首先，2pc的事务有一个全局的事务表，所有2pc的事务都要有一个name，在设置name的同时，将该事务注册到全局事务表里：

Status TransactionImpl::SetName(const TransactionName& name) {

if (txn\_state\_ == STARTED) {

......

// 向事务管理器注册事务

txn\_db\_impl\_->RegisterTransaction(this);

......

}

* prepare阶段

// 设置事务状态为开始PREPARE

txn\_state\_.store(AWAITING\_PREPARE);

// PREPARE之后不允许事务超时, 可能会遇到2pc的通病????

expiration\_time\_ = 0;

WriteOptions write\_options = write\_options\_;

write\_options.disableWAL = false;

// MarkEndPrepare会将当前batch开头和结尾写入PREPARE标记

// 正常的WriteBatch格式一般是:

// Sequence(0);NumRecords(2);Put(a,1);Delete(b);

// MarkEndPrepare之后:

// Sequence(0);NumRecords(4);BeginPrepare();Put(a,1);Delete(b);EndPrepare(transaction\_id);

// 对WriteBatch开始和结束分别加入Begin/End，标识是个PREPARE

WriteBatchInternal::MarkEndPrepare(GetWriteBatch()->GetWriteBatch(), name\_);

// 将更改之后的WriteBatch写入db，这里只写WAL，不写memtable

s = db\_impl\_->WriteImpl(write\_options, GetWriteBatch()->GetWriteBatch(),

/\*callback\*/ nullptr, &log\_number\_, /\*log ref\*/ 0,

/\* disable\_memtable\*/ true);

if (s.ok()) {

.......

txn\_state\_.store(PREPARED);

}

整个过程将修正后的prepared writebatch只是写入WAL日志，并不会更新memtable，这样保证了其他的普通事务和2pc事务是不能访问到该2pc事务的记录(memtable不可见)，保证了隔离性。这里有个点需要注意，大部分RocksDB的写操作都是一定写memtable和WAL(可以disable)的，所以全局的LSN就会递增。但prepare步骤是不写入memtable的，所以LSN不会增加，这就解释了文章开头说的WAL中LSN并不一定满足lsn(WriteBatch(n)) < lsn(WriteBatch(n+1))。

* commit阶段

Q15：为什么“这也就是说2pc是多个WriteBatch”？ 为什么2pc会有多个WriteBatch？不能就使用OCC操作阶段的WriteBatch作为提交时的WriteBatch吗？

A15：OCC操作阶段的WriteBatch中包含事务所写入的数据，以及prepare标识，这些数据在prepare阶段写入WAL。而这里说的“临时”WriteBatch，是在commit阶段，又向WAL写入的commit log，表示事务的状态已经变更为“提交”。事务提交过程中，写WAL是分为多次进行的，所以用到多个WriteBatch Q15.1 WriteBatch既可以放用户数据，也可以放WAL，只是一块私有内存，是不是？

// 设置事务状态为准备commit

txn\_state\_.store(AWAITING\_COMMIT);

// 获取临时的一个WriteBatch buffer，区别于prepare之前的操作的WriteBatch

// 所以commit的WriteBatch和prepare的WriteBatch是单独分开的，这也就是说2pc

// 是多个WriteBatch所以需要额外保证原子性。

WriteBatch\* working\_batch = GetCommitTimeWriteBatch();

// 写入commit标识和事务ID

WriteBatchInternal::MarkCommit(working\_batch, name\_);

// WAL终止点(暂没想到更好的叫法)，后续写入的数据，WAL会全部忽略

working\_batch->MarkWalTerminationPoint();

// 将包含prepare的全部数据追加到WriteBatch里，这些数据是供memtable写入用的

WriteBatchInternal::Append(working\_batch, GetWriteBatch()->GetWriteBatch());

// 数据写入memtable(包含prepare)，并将commit事件写入WAL

s = db\_impl\_->WriteImpl(write\_options\_, working\_batch, nullptr, nullptr,

log\_number\_);

if (!s.ok()) {

return s;

}

// 从全局事务表里删除该事务

txn\_db\_impl\_->UnregisterTransaction(this);

commit阶段主要做两件事：

1、将commit标识写入WAL

2、将数据写入memtable(让其他事务可以访问到)---特别注意：事务的数据从此可见了。相当于事务的提交标志

整体回顾整个2pc提交的流程，prepare阶段生成BeginPrepare/EndPrepare相关的WAL记录，并写入WAL持久化(这里可以防止crash时，仍旧可以构建出来该事务)，但为了保证隔离性，不会写入memtable。commit阶段将Commit的WAL记录写入WAL，并写入memtable，让其他事务可见。这里用了多个WriteBatch，打破了RocksDB默认的单WriteBatch原子性的保证，所以需要在WAL记录中增加额外标识，并在crash时，重建内存2pc事务状态。

### 2PC Recovery

RocksDB的2pc是跨WriteBatch实现的prepare和commit，所以可能存在中间态，比如prepare之后commit之前crash了。这时候系统启动时要重建所有的正在执行的事务(仅2pc事务，普通事务通过单个WriteBatch已经保证了原子性)。MemtableInserter作为处理WriteBatch中每一条记录，在遇到BeginPrepare/EndPrepare时，会在内存中重建事务的上下文，具体可见MemtableInserter代码本文不赘述。

## MyRocks

RocksDB的TransactionDB支持了大部分MySQL对事务的规范，整体接口形式和行为基本一致，有些细节比如online ddl、gap locking的支持、需要binglog开启row模式等有差别。

具体可见 <https://github.com/facebook/mysql-5.6/wiki>

# 源码

## 主要流程

### RocksDB的初始化

### create table

### select

### insert

### explain

### commit

### rollback

# 应用场景

参考：

<https://www.jianshu.com/p/3302be5542c7>

RocksDB的典型场景（低延时访问）:

1. 需要存储用户的查阅历史记录和网站用户的应用
2. 需要快速访问数据的垃圾检测应用
3. 需要实时scan数据集的图搜索query
4. 需要实时请求Hadoop的应用
5. 支持大量写和删除操作的消息队列