存储主题分享(4)——RocksDB

KV&内存 情况	线程模型	存储引擎三变量			复制			事务
		是否有序	是否可变	是否缓冲	复制模式	一致性	快照原理	
内存+磁盘都包含KV，按冷热分层	多线程 无锁并发（一写多读）	有序	不可变	有	原生不提供复制，但可基于WAL和SST File编写异步、同步的复制，参考Rocksplitor	-	Manifest文件记录SST File情况，WAL恢复内存	ACID 支持两种隔离级别：Repeatable Read，ReadCommited

RocksDB是什么

- RocksDB 是一个KV的**存储引擎**，其中键和值都为**任意字节流**。它是一个 C++ 库，由 Facebook 基于 LevelDB 1.5版本开发，并提供向后兼容 LevelDB API 的支持。
- RocksDB 支持**各种存储硬件**，最初专注于快速闪存。它使用LSM树引擎进行存储，完全用 C++ 编写。（并有一个名为 RocksJava 的 Java 包装器。请参阅 RocksJava-Basics）
- RocksDB 可以适应各种生产环境，包括纯内存、闪存、硬盘或远程存储。在 RocksDB 无法自动适应的情况下，提供了**高度灵活的配置**，允许用户对其进行调整。它支持各种压缩算法，以及用于生产支持和调试的良好工具。

特性

- 针对希望在本地或远程存储系统上存储**TB级**数据的应用设计。
- 针对在**快速存储**（闪存设备或内存中）上存储**中小键值对**进行了优化。
- 在**多核**的处理器上运行良好。
- RocksDB 相比levelDB引入了数十项新的主要功能。请参阅 Features-Not-in-LevelDB
    - 备份恢复
    - 压缩
    - 多线程compaction、多线程memtable插入

- 列族(Column Family)
  - RocksDB 支持将数据库实例划分到多个列族。因此可以管理不同数据（用户、订单、互动）
  - 所有数据库都使用为 "default" 的列族创建，该列族用于未指定列族的操作。
  - 每个列族有一个独立的LSM树，并发写，提高性能
  - 独立配置、独立扩展
- Merge(Atomic Read-Modify-Write)

```
      // Gives the client a way to express the read -> modify -> write semantics
      // key:           (IN) The key that's associated with this merge operation.
      // existing_value:(IN) null indicates the key does not exist before this op
      // value:         (IN) the value to update/merge the existing_value with
      // 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.
      virtual bool Merge(const Slice& key,
                         const Slice* existing_value,
                         const Slice& value,
                         std::string* new_value,
                         Logger* logger) const = 0;
```

整体架构

- RocksDB按**顺序**组织所有数据（Memtable、SSTFile内部、L1-LN层的SSTFile之间），常见的API操作有：
  - Read: Get(key), NewIterator(), MultiGet(keys)
  - Write: Put(key, val), Merge(key, existing_value, value, new_value), Delete(key), and SingleDelete(key)
  - 其余API请参阅 Basic-Operations
- RocksDB的三个基本结构是memtable、sstfile和logfile。
  - memtable是内存中的数据结构——新的写操作会写入logfile并插入到memtable中

    - logfile是WAL(write ahead log)顺序写入存储。
    - 当memtable被填满时，数据会flush到存储的sstfile中，此时可以安全地删除对应logfile。sstfile中的数据按顺序存储，以快速查找。sst格式请参阅sstfile-format
- 内存顺序IO >> **内存随机IO** ≈ **磁盘顺序IO** >> 磁盘随机IO

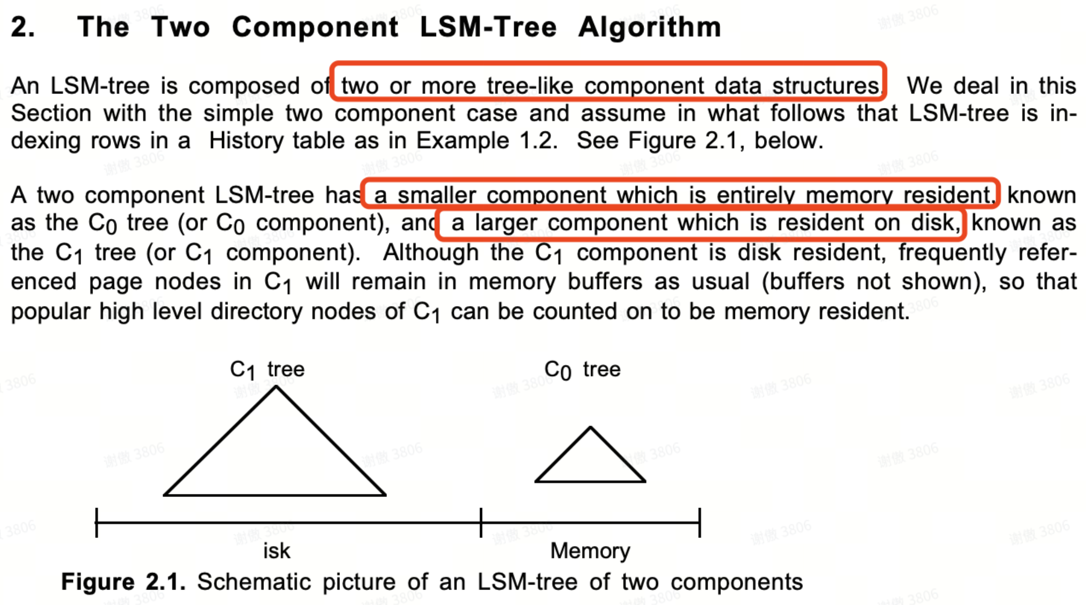
索引

LSM (Log Structured Merge) 树

📌 LSM树最早在1996年提出，是一种优化写多读少的数据结构

- 注意到有些文章认为LSM树并非一种数据结构，而是一种存储设计思想
    - 其实可以理解，因为rocksDB实现的LSM树与最早提出的LSM树与有了较大差别
    - 所以LSM树也可以认为是一种设计思想，不同产品有不同实现方式

- 论文地址：https://www.cs.umb.edu/~poneil/lsmtree.pdf
- LSM的简化理解：内存+磁盘的两部分构成



2. The Two Component LSM-Tree Algorithm

An LSM-tree is composed of two or more tree-like component data structures. We deal in this Section with the simple two component case and assume in what follows that LSM-tree is indexing rows in a History table as in Example 1.2. See Figure 2.1, below.

A two component LSM-tree has a smaller component which is entirely memory resident, known as the C_0 tree (or C_0 component), and a larger component which is resident on disk, known as the C_1 tree (or C_1 component). Although the C_1 component is disk resident, frequently referenced page nodes in C_1 will remain in memory buffers as usual (buffers not shown), so that popular high level directory nodes of C_1 can be counted on to be memory resident.

Figure 2.1. Schematic picture of an LSM-tree of two components

- 冷热数据分离，一层层merge

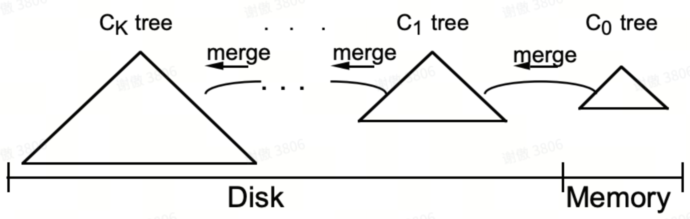


Figure 3.1. An LSM-tree of K+1 components

In general, an LSM-tree of K+1 components has components C_0, C_1, C_2, . . ., C_{K-1} and C_K, which are indexed tree structures of increasing size; the C_0 component tree is memory resident and all other components are disk resident (but with popular pages buffered in memory as with any disk resident access tree). Under pressure from inserts, there are asynchronous rolling merge processes in train between all component pairs (C_{i-1}, C_i), that move entries out from the smaller to the larger component each time the smaller component, C_{i-1}, exceeds its threshold

size. During the life of a long-lived entry inserted in an LSM-tree, it starts in the C_0 tree and eventually migrates out to the C_K, through a series of K asynchronous rolling merge steps.

- 尽量利用磁盘顺序IO，但parent node**有修改**

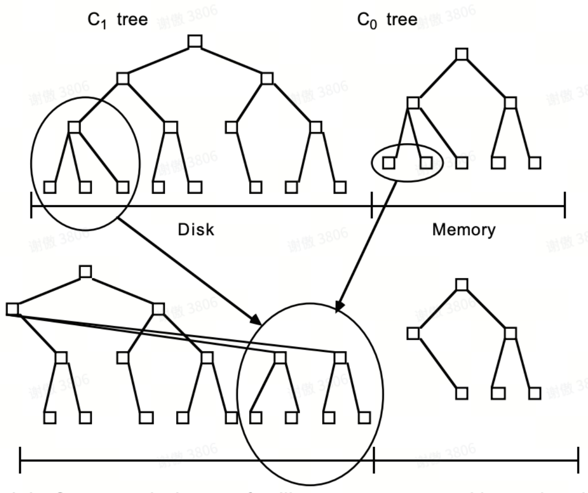
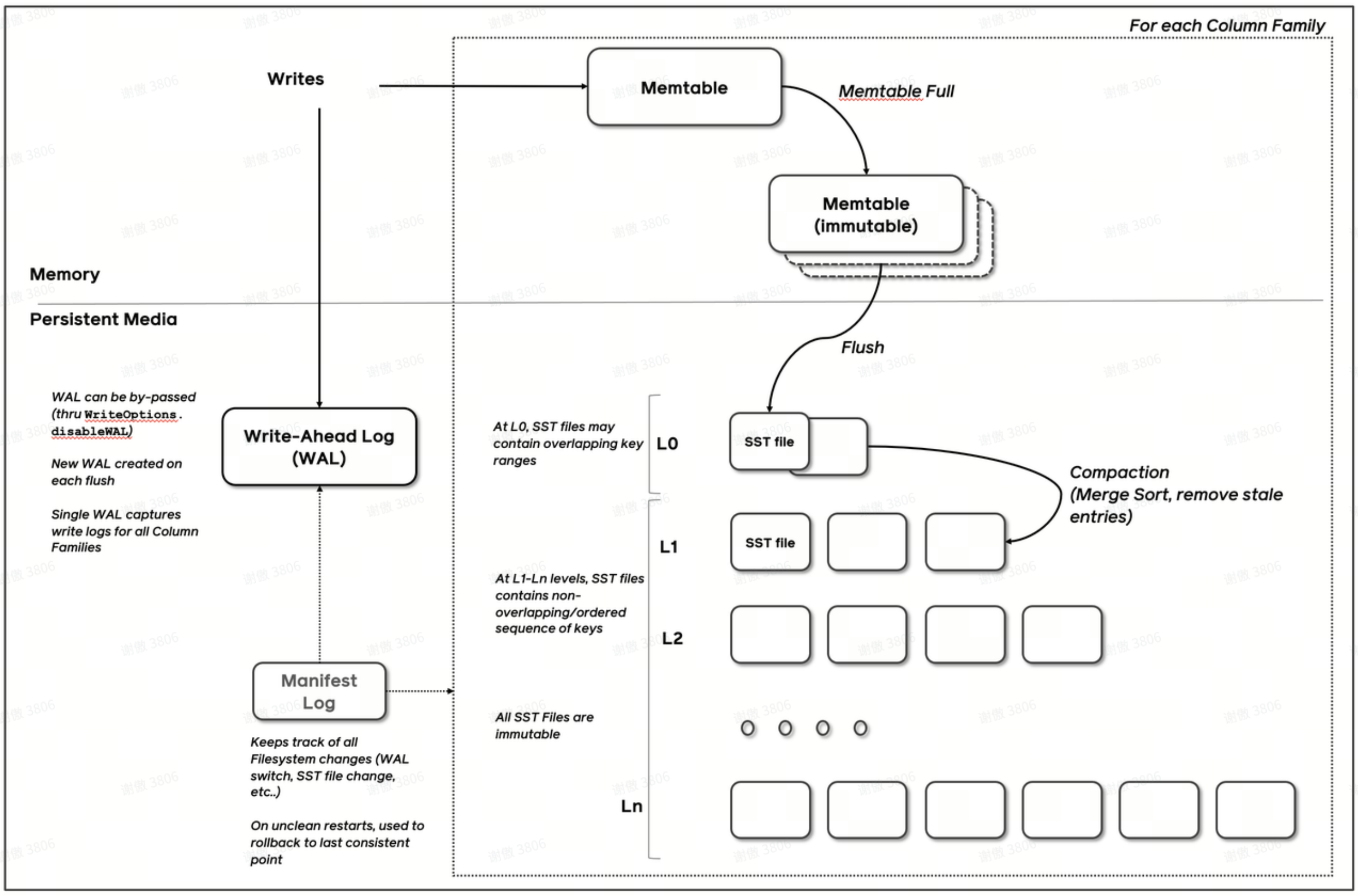


Figure 2.2. Conceptual picture of rolling merge steps, with result written back to disk

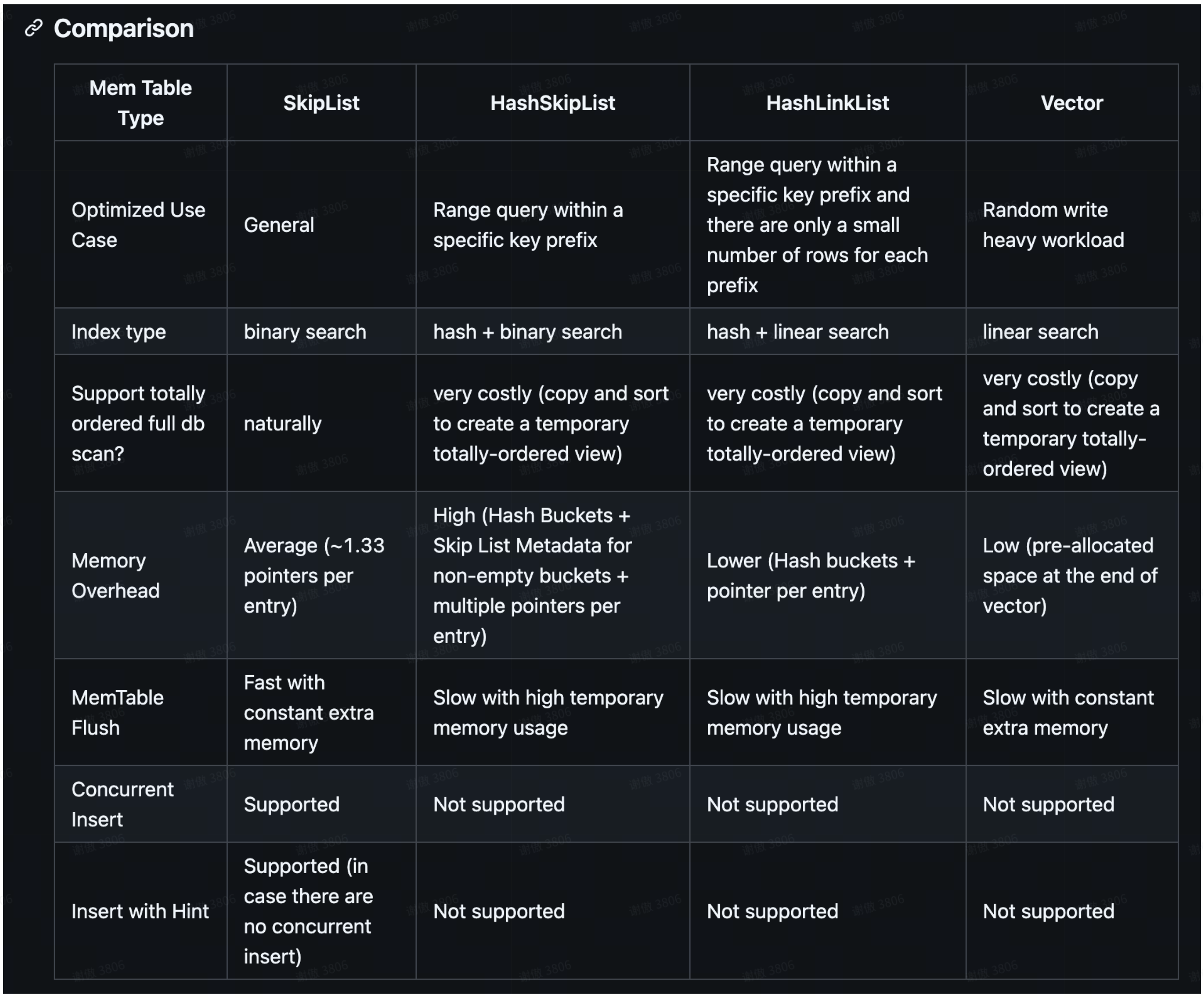
Newly merged blocks are written to new disk positions, so that the old blocks will not be overwritten and will be available for recovery in case of a crash. The parent directory nodes in C_1, also buffered in memory, are updated to reflect this new leaf structure, but usually remain in buffer for longer periods to minimize I/O; the old leaf nodes from the C_1 component are invalidated after the merge step is complete and are then deleted from the C_1 directory. In general, there will be leftover leaf-level entries for the merged C_1 component following each merge step, since a merge step is unlikely to result in a new node just as the old leaf node empties. The same consideration holds for multi-page blocks, since in general when the filling block has filled with newly merged nodes, there will be numerous nodes containing entries still in the shrinking block. These leftover entries, as well as updated directory node information, remain in block memory buffers for a time without being written to disk. Techniques to provide concurrency during the merge step and recovery from lost memory during a crash are covered in detail in Section 4. To reduce reconstruction time in recovery, checkpoints of the merge process are taken periodically, forcing all buffered information to disk.

RocksDB改进的LSM树


For each Column Family
Writes
Memtable
Memtable Full
Memtable (immutable)
Memory
Persistent Media
Flush
WAL can be by-passed (thru WriteOptions.disableWAL)
New WAL created on each flush
Single WAL captures write logs for all Column Families
Write-Ahead Log (WAL)
At L0, SST files may contain overlapping key ranges
L0
SST file
Compaction (Merge Sort, remove stale entries)
L1
SST file
At L1-Ln levels, SST files contains non-overlapping/ordered sequence of keys
L2
Manifest Log
Keeps track of all Filesystem changes (WAL switch, SST file change, etc..)
On unclean restarts, used to rollback to last consistent point
All SST Files are immutable
Ln

Memtable

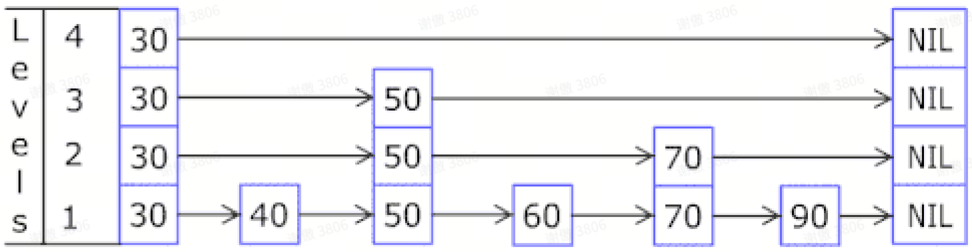
- Memtable
    - 读写请求都先从Memtable开始，Memtable必定保存最新值
    - 一个列族只有一个
    - 到达指定大小后变为Immutable Memtable，等待Flush到L0层的SST File
- Immutable Memtable
    - 只读
    - 一个列族有若干个，可以配置
    - Immutable Memtable Flush到L0层后，对应一个SST File
- 默认采用跳表skiplist(优点见下表)，可选其他数据结构
    - 跳表在读、写、flush、无锁并发等方面都表现良好



Comparison

Mem Table Type	SkipList	HashSkipList	HashLinkList	Vector
Optimized Use Case	General	Range query within a specific key prefix	Range query within a specific key prefix and there are only a small number of rows for each prefix	Random write heavy workload
Index type	binary search	hash + binary search	hash + linear search	linear search
Support totally ordered full db scan?	naturally	very costly (copy and sort to create a temporary totally-ordered view)	very costly (copy and sort to create a temporary totally-ordered view)	very costly (copy and sort to create a temporary totally-ordered view)
Memory Overhead	Average (~1.33 pointers per entry)	High (Hash Buckets + Skip List Metadata for non-empty buckets + multiple pointers per entry)	Lower (Hash buckets + pointer per entry)	Low (pre-allocated space at the end of vector)
MemTable Flush	Fast with constant extra memory	Slow with high temporary memory usage	Slow with high temporary memory usage	Slow with constant extra memory
Concurrent Insert	Supported	Not supported	Not supported	Not supported
Insert with Hint	Supported (in case there are no concurrent insert)	Not supported	Not supported	Not supported

skiplist



Log File

- 每次对RocksDB的更新都被写到两个地方:
  - 内存：Memtable
  - 磁盘：WAL
- WAL文件持续增长，直到超过单个文件最大配置，此时会创建新WAL文件

- 失败恢复时，用WAL恢复内存的Memtable、Immutable Memtable即可
- 当Immutable Memtable flush到磁盘以后，会更新列族的最大序列号
    - 所有列族共用一个WAL
    - 因此所有列族的最大序列号都超过一个WAL文件时，该WAL文件才会被删除

文件格式

日志文件包含一系列**变长**记录，变长记录按 `kBlockSize` (32k)聚合

```
      +-----+-------------+--+----+----------+------+-- ... ----+
File  | r0  |        r1   |P | r2 |    r3    |  r4  |           |
      +-----+-------------+--+----+----------+------+-- ... ----+
      <--- kBlockSize ------>|<-- kBlockSize ------>|

 rn = variable size records
 P = Padding
```

记录格式

有两种格式Legacy record format和Recyclable record format：

```
+---------+-----------+-----------+--- ... ---+
|CRC (4B) | Size (2B) | Type (1B) | Payload   |
+---------+-----------+-----------+--- ... ---+

CRC = 32bit hash computed over the payload using CRC
Size = Length of the payload data
Type = The type is used to group a bunch of records together to represent
       blocks that are larger than kBlockSize
  kZeroType = 0,
  kFullType = 1,
  kFirstType = 2,
  kMiddleType = 3,
  kLastType = 4,
Payload = Byte stream as long as specified by the payload size
```

```
+---------+-----------+-----------+----------------+--- ... ---+
|CRC (4B) | Size (2B) | Type (1B) | Log number (4B)| Payload   |
+---------+-----------+-----------+----------------+--- ... ---+
Same as above, with the addition of
Log number = 32bit log file number, so that we can distinguish between
```

```
records written by the most recent log writer vs a previous one.

Types:
  kRecyclableFullType = 5,
  kRecyclableFirstType = 6,
  kRecyclableMiddleType = 7,
  kRecyclableLastType = 8,
```

SST File (Sorted String Table)

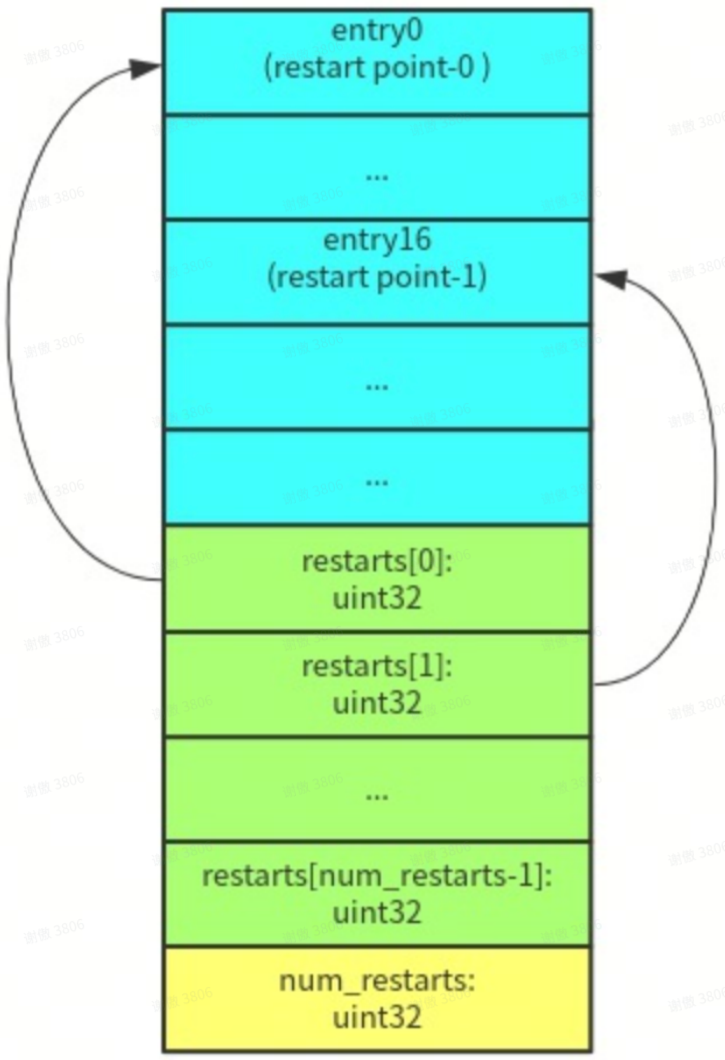
文件格式

```
<beginning_of_file>
[data block 1]
[data block 2]
...
[data block N]
[meta block 1: filter block]                  (see section: "filter" Meta Block)
[meta block 2: index block]
[meta block 3: compression dictionary block]  (see section: "compression dictionary" Meta Block)
[meta block 4: range deletion block]          (see section: "range deletion" Meta Block)
[meta block 5: stats block]                   (see section: "properties" Meta Block)
...
[meta block K: future extended block]  (we may add more meta blocks in the future)
[metaindex block]
[Footer]                               (fixed size; starts at file_size - sizeof(Footer))
<end_of_file>
```

Data Block与Entry

- Data Block内存放KV的基本单位是Entry，采用**前缀压缩**存放
    - 当存储一个 Key 时，会丢弃与前一个 Key 共享的前缀。
    - 为了方便查找，每隔 K 个 Key，会存储完整的 Key，称为 "restart point"。
    - 默认情况下，RocksDB 中数据块的 `block_restart_interval` 为 1。 这意味着每隔一个 Key 就会有一个 restart point。将 restart 间隔设置为 8 或 16 通常可以将 Index Block 的大小减半。

```
// 单个Entry格式如下
shared_bytes: varint32: 表示与前一个 Key 共享的前缀字节数
unshared_bytes: varint32: 表示与前一个 Key 不共享的字节数
value_length: varint32
key_delta: char[unshared_bytes]: 表示与前一个 Key 不共享的 Key 部分
value: char[value_length]

// Data Block尾部格式如下
restarts: uint32[num_restarts]
num_restarts: uint32
```



- Data Block内部的查找流程：
    - 在restarts数组上二分，定位key属于哪一个restart组
    - 从定位的restart组中，从前往后遍历

Meta Block

Bloom Filter

通过 `filter policy` 配置，当激活时，每个SST File都会创建filter block用于快速判定key是否存在于SST File。

如果**必不存在**就可以去更下一层的SST File中搜索。

```
 // 通常来讲，10bits per key准确率已经很高了
rocksdb::BlockBasedTableOptions table_options;
table_options.filter_policy.reset(rocksdb::NewBloomFilterPolicy(10, false));
my_cf_options.table_factory.reset(
  rocksdb::NewBlockBasedTableFactory(table_options));

1.5 bits per key (50% false positive rate) is 50% as effective as 100 bits per key
2.9 bits per key (25% false positive rate) is 75% as effective as 100 bits per key
4.9 bits per key (10% false positive rate) is 90% as effective as 100 bits per key
9.9 bits per key (1% false positive rate) is 99% as effective as 100 bits per key
15.5 bits per key (0.1% false positive rate) is 99.9% as effective as 100 bits per key
```

Index Block

通过offset和size指明每个Data Block的位置和大小

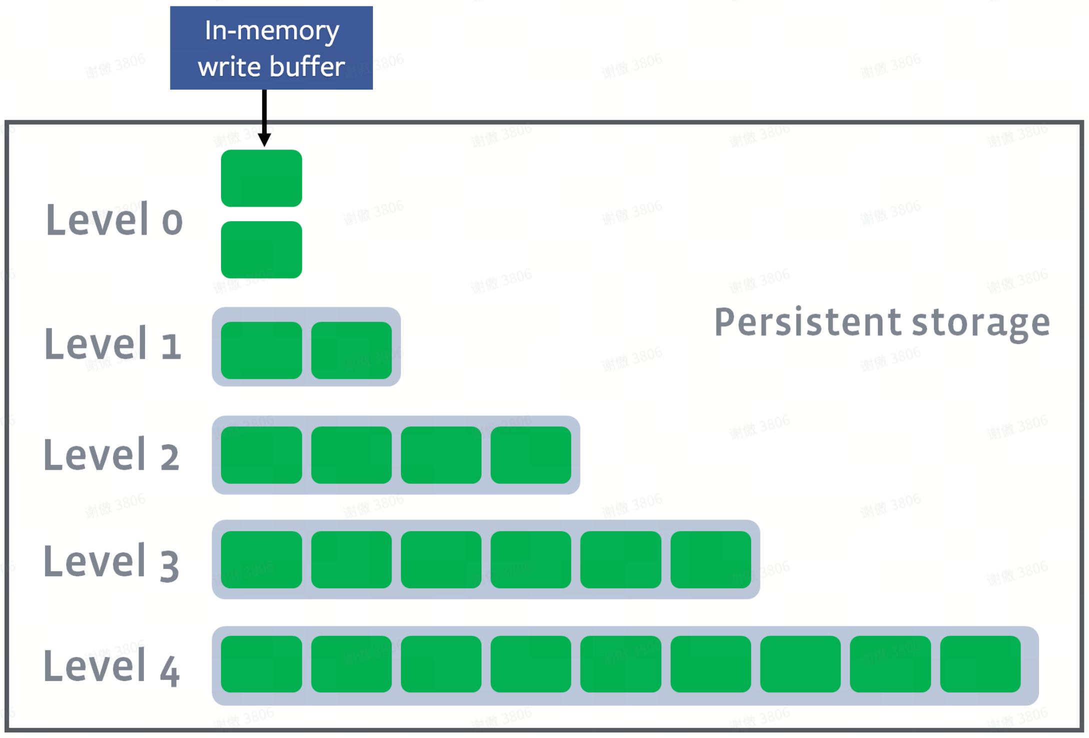
```
// 存放每个data block的BlockHandles，即offset和size
offset, size
offset, size
offset, size
offset, size
offset, size
offset, size
```

Compaction

📌 RocksDB介绍了Classic Leveled, Tiered, Tiered+Leveled, Leveled-N, FIFO等不同SST File的合并方式。

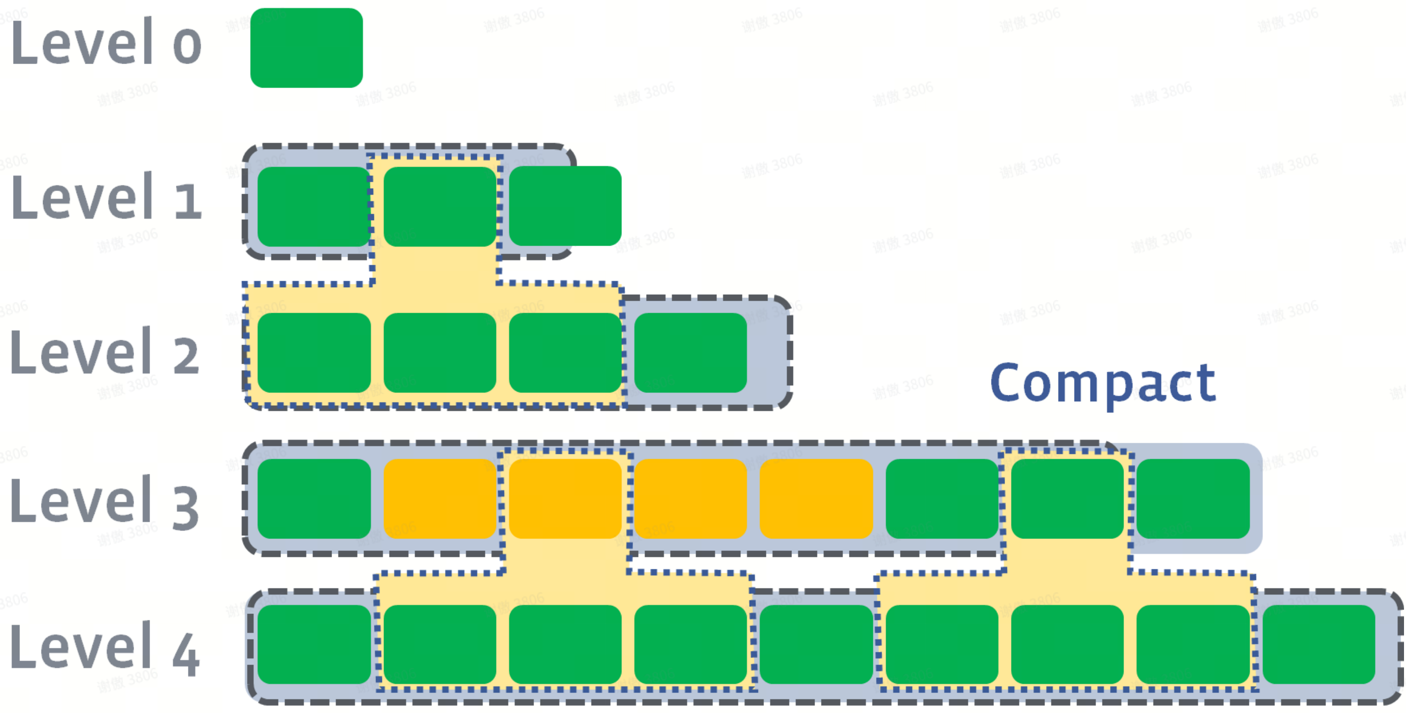
其中实现了Tiered+Leveled (Level Compaction in the code), Tiered (Universal in the code) 和 FIFO三种。详见：Compaction

- Classic Leveled
  - 前一层级的所有SST File与后一层级所有SST File合并
  - **每层严格有序**
  - 空间放大最小化，读写放大严重
- Tiered
  - 前一层级所有SST File合并后，生成后一层级单个SST File
  - 因此每层由**N个有序序列**组成，**序列内有序**
  - 最小化写放大，读、空间放大严重
- Tiered+Leveled
  - 层数低采用tiered，层数高采用leveled
  - Tiered
    - Memory(Memtable+Immutable Memtables)生成L0，L0层层内有N个有序序列
  - Leveled
    - RocksDB从all-to-all，优化为some-to-some，只合并key区间有重叠部分的SST File
    - L0->L1, ..., LN-1->LN都采取Leveled合并
      - 注意因为L0内SSTFile是重合的，所以尽管是some-to-some，但对于L0->L1来说**必然**是all-to-some



  - 写放大小于level，读放大小于tier

并发Compaction



写流程

- 写Memtable、WAL即返回
- Flush和Compaction交给后台线程

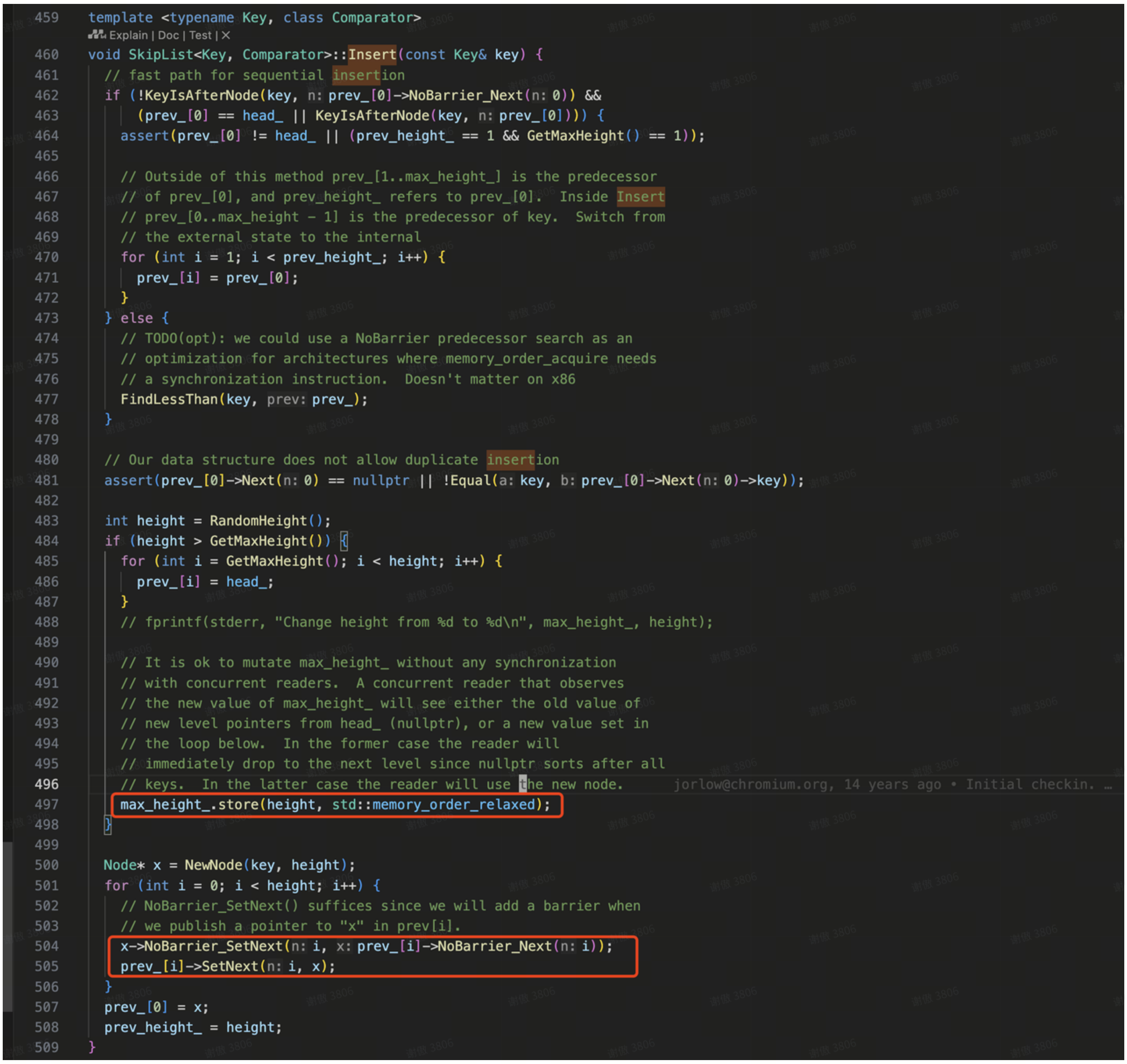
读流程

- 查Memtable以及按序查Immutable Memtable(s)
- 按层、按key的范围查询SST File
  - L0层要遍历所有SST File
  - L1-LN层根据key范围查询指定SST File
  - SST File有布隆过滤器用于判断SST File是否必不包含key
  - SST File内的Data Block在内存中有Block Cache，用LRU管理
  - SST File内查询分为**两个二分查找：**
    - 根据Index Block二分定位Data Block
    - 根据Data Block内restarts数组，定位restarts组
    - 最后从前往后遍历

线程模型

无锁并发

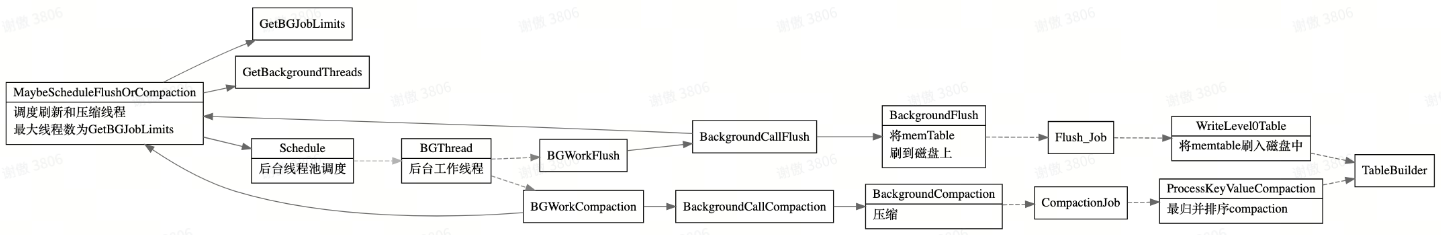
- 只需要锁max_height，和log个前驱节点。共计 height+1 个atomic变量
- Tricks: 保证读请求进行时不销毁skiplist，保证所有节点不删除（直到skiplist销毁时才删）
- 读写并发，但不支持写并发（一写多读）
  - 写请求不影响链表
    - 读请求如果不读写请求的数据，即访问原先存在数据，则必定能访问到
    - 读请求如果读读写请求的数据，则遍历链表有可能访问到，也有可能读不到，但不影响最终一致
  - 写请求影响高度
    - 读请求会在查找时正确地下降，因为所有节点最后都是nil（可视为无限大）



```
template <typename Key, class Comparator>
Explain | Doc | Test | ×
void SkipList<Key, Comparator>::Insert(const Key& key) {
  // fast path for sequential insertion
  if (!KeyIsAfterNode(key, n: prev_[0]->NoBarrier_Next(n: 0)) &&
      (prev_[0] == head_ || KeyIsAfterNode(key, n: prev_[0]))) {
    assert(prev_[0] != head_ || (prev_height_ == 1 && GetMaxHeight() == 1));

    // Outside of this method prev_[1..max_height_] is the predecessor
    // of prev_[0], and prev_height_ refers to prev_[0].  Inside Insert
    // prev_[0..max_height - 1] is the predecessor of key.  Switch from
    // the external state to the internal
    for (int i = 1; i < prev_height_; i++) {
      prev_[i] = prev_[0];
    }
  } else {
    // TODO(opt): we could use a NoBarrier predecessor search as an
    // optimization for architectures where memory_order_acquire needs
    // a synchronization instruction.  Doesn't matter on x86
    FindLessThan(key, prev: prev_);
  }

  // Our data structure does not allow duplicate insertion
  assert(prev_[0]->Next(n: 0) == nullptr || !Equal(a: key, b: prev_[0]->Next(n: 0)->key));

  int height = RandomHeight();
  if (height > GetMaxHeight()) {
    for (int i = GetMaxHeight(); i < height; i++) {
      prev_[i] = head_;
    }
    // fprintf(stderr, "Change height from %d to %d\n", max_height_, height);

    // It is ok to mutate max_height_ without any synchronization
    // with concurrent readers.  A concurrent reader that observes
    // the new value of max_height_ will see either the old value of
    // new level pointers from head_ (nullptr), or a new value set in
    // the loop below.  In the former case the reader will
    // immediately drop to the next level since nullptr sorts after all
    // keys.  In the latter case the reader will use the new node.      jorlow@chromium.org, 14 years ago • Initial checkin. …
    max_height_.store(height, std::memory_order_relaxed);
  }

  Node* x = NewNode(key, height);
  for (int i = 0; i < height; i++) {
    // NoBarrier_SetNext() suffices since we will add a barrier when
    // we publish a pointer to "x" in prev[i].
    x->NoBarrier_SetNext(n: i, x: prev_[i]->NoBarrier_Next(n: i));
    prev_[i]->SetNext(n: i, x);
  }
  prev_[0] = x;
  prev_height_ = height;
}
```

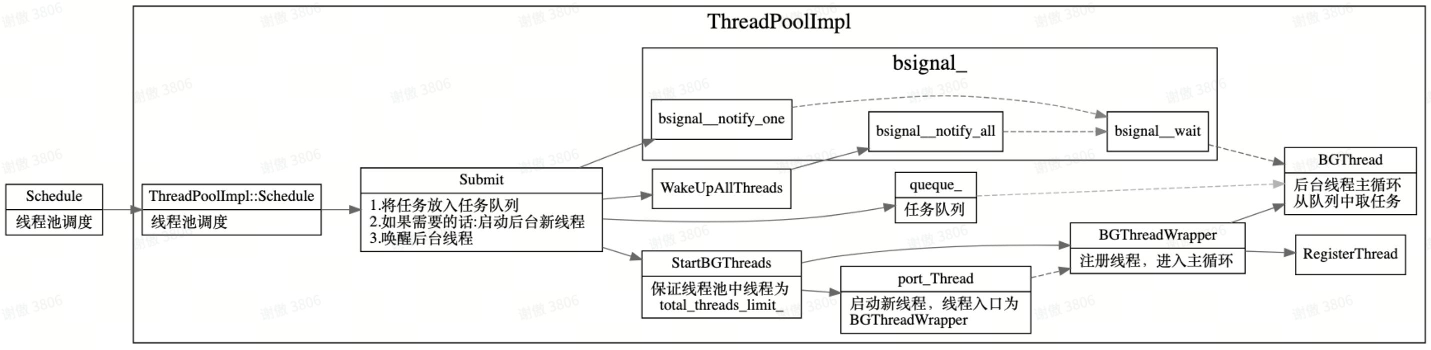
后台线程

- Flush时机：https://github.com/facebook/rocksdb/wiki/MemTable
- Compaction时机：https://github.com/facebook/rocksdb/wiki/Leveled-Compaction
- 函数入口：MaybeScheduleFlushOrCompaction()



线程池实现

- 实现入口：ThreadPoolImpl



事务

📌 RocksDB 实现了 **ReadCommited** 和 **RepeatableRead** 隔离级别

- ReadUncommited 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
- **ReadCommited** 读取已提交内容，事务只能看到其他已提交事务的更新内容，多次读的时候可能读到其他事务更新的内容
- **RepeatableRead** 可重复读，确保事务读取数据时，多次操作会看到同样的数据行(使用快照隔离来实现)。
- Serializability 可串行化，强制事务之间的执行是有序的，不会互相冲突。

- OptimisticTransactionDB对应ReadCommited
  - 事务准备时不拿任何锁
  - 提交时有冲突则报错不修改
- TransactionDB对应RepeatableRead

    - 事务准备时会锁所有更改的key
    - 没锁到即报错
    - 所有写操作key锁都拿到后，commit保证必定成功（只要数据库不挂）

RocksDB优缺点

优点

- 写多读少的业务场景

缺点

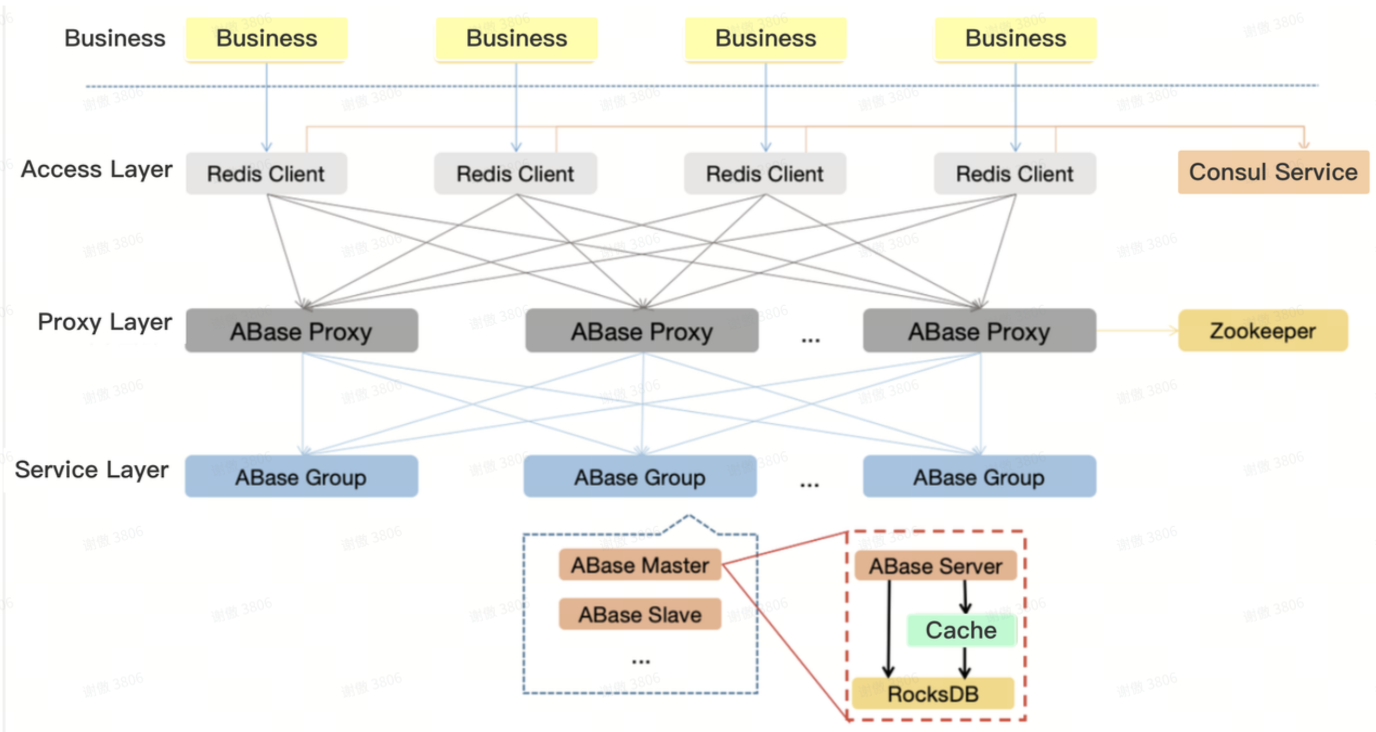
- 读放大：冷热数据分离导致数据存放位置无法确定，最差会从Memtable、Immutable Memtable、L0、...、LN一路找下去
- 写放大：初始只需写入WAL + Memtable，但会有多次compaction（且热数据会在L0中的多个SST File以及L1-LN的某几层的单个SST File中出现）
    - 常见原因及解法见https://cloud.tencent.com/developer/article/1517019
- 空间冗余：追加写带来的，同一数据可能会在L0-LN多层出现
- Write Stall（写停滞）：磁盘I/O速度跟不上Memtable写入速度，导致Flush和Compaction阻塞了写入
    - 详细原因及解法见上面链接

RocksDB的应用

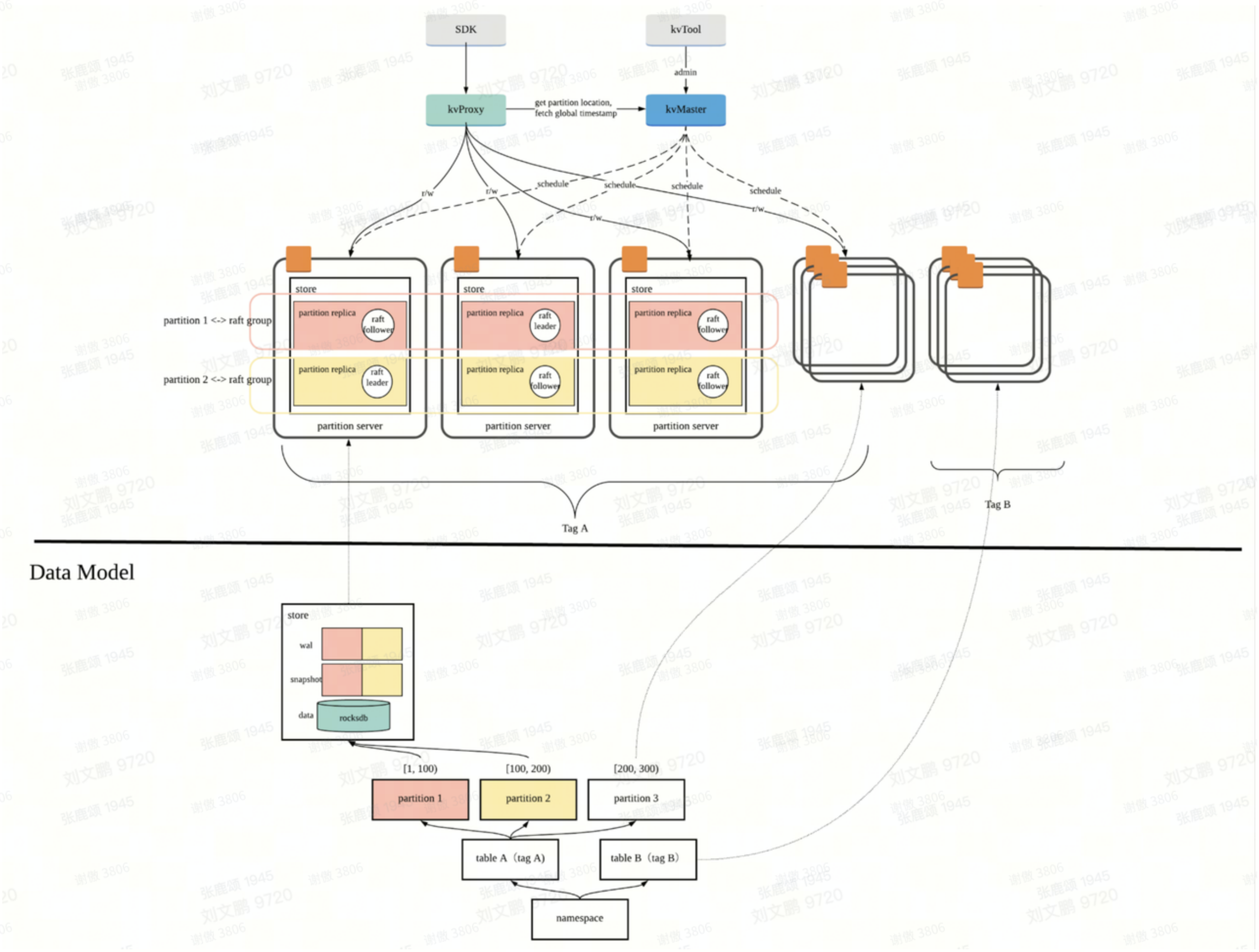
📌 作为高效的存储引擎，RocksDB应用于以下几个数据库：

公司内

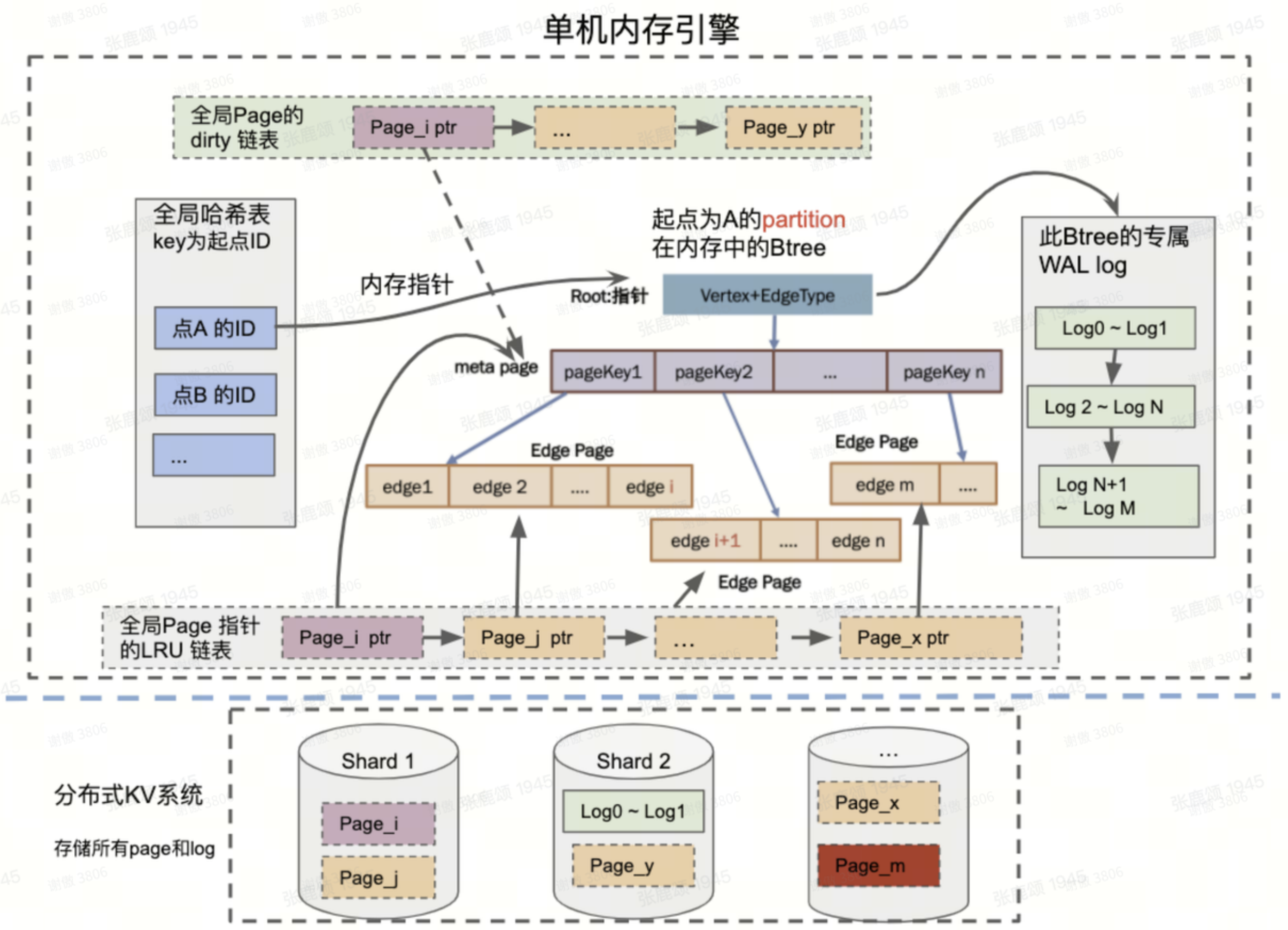
- Abase An Introduction to Abase/Redis：rocksDB作为存储引擎，套了Access Layer, Proxy Layer, Service Layer一层壳



- ByteKV ByteKV 介绍：RocksDB改动底层存储引擎，改动优化较多（RocksDB自身问题，ByteKV需求与RocksDB实现不匹配的问题。见 ByteKV 单机存储引擎 - BlockDB 概要设计）
  - Namespace->Table->Partition->Store
  - 每个Store对应一个RocksDB实例



- BG2 ByteGraph—自研万亿级图数据库 ByteGraph-self-developed trillion-level graph database
  - 查询层、存储层为B+树，page和WAL放在abase或者byteKV等分布式KV里



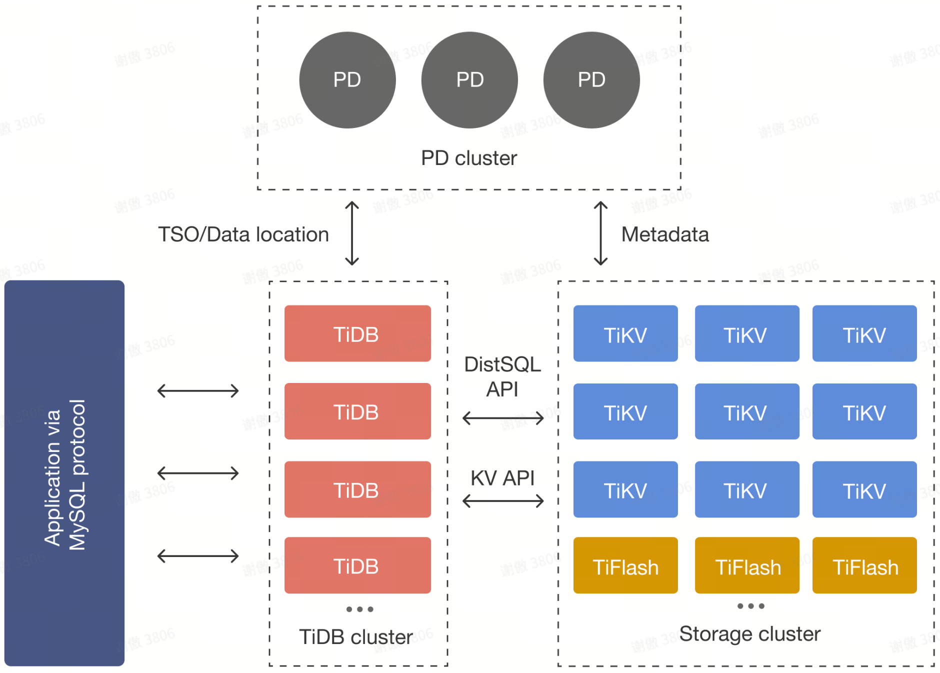
业界

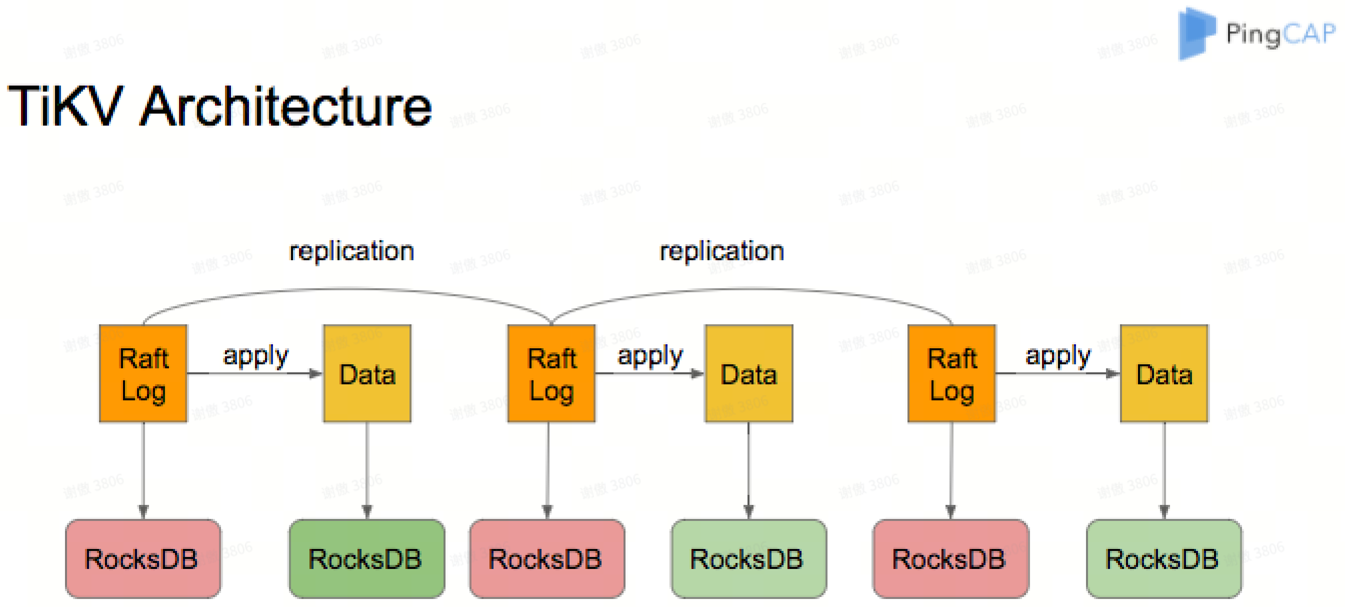
- Pika：用rocksDB做为redis的持久化

> Pika 是一个以 RocksDB 为存储引擎的的大容量、高性能、多租户、数据可持久化的弹性 KV 数据存储系统，完全兼容 Redis 协议，支持其常用的数据结构，如 string/hash/list/zset/set/geo/hyperloglog/pubsub/bitmap/stream 等 Redis 接口。
>
> Redis 的内存使用量超过一定阈值【如 16GiB 】时，会面临内存容量有限、单线程阻塞、启动恢复时间长、内存硬件成本贵、缓冲区容易写满、一主多从故障时切换代价大等问题。Pika 的出现并不是为了替代 Redis, 而是 Redis 补充。Pika 力求在完全兼容Redis 协议、继承 Redis 便捷运维设计的前提下，通过持久化存储的方式解决了 Redis 一旦存储数据量巨大就会出现内存容量不足的瓶颈问题，并且可以像 Redis 一样，支持使用 slaveof 命令实现主从模式，还支持数据的全量同步和增量同步。
>
> 还可以通过 twemproxy or Codis 以静态数据分片方式实现 Pika 集群。

- MyRocks：mysql + rocks：用RocksDB替换InnoDB，以使用replication、sql layer等mysql特性
- TiDB：TiKV是TiDB的存储引擎，RocksDB是TiKV的存储引擎

> RocksDB 作为 TiKV 的核心存储引擎，用于存储 Raft 日志以及用户数据。每个 TiKV 实例中有两个 RocksDB 实例，一个用于存储 Raft 日志（通常被称为 raftdb），另一个用于存储用户数据以及 MVCC 信息（通常被称为 kvdb）。kvdb 中有四个 ColumnFamily：raft、lock、default 和 write：





参考资料

- https://stackshare.io/stackups/leveldb-vs-rocksdb
- rocksDB github官方文档
- https://www.bilibili.com/video/BV1Jr4y1W7Wn/?spm_id_from=333.337.search-card.all.click&vd_source=297e801803d56855d9dce0af1fb27b3c
- https://zhuanlan.zhihu.com/p/45652076
- RocksDB 学习分享
- https://vinodhinic.medium.com/lets-rock-3a73fbc6ea79
- https://cloud.tencent.com/developer/article/1517019
- https://blog.csdn.net/Z_Stand/article/details/107447152?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216404987751678035721 5077%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fblog.%2522%257D&request_id=164049877516780357215077&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~blog~first_rank_ecpm_v1~rank_v31_ecpm-3-107447152.nonecase&utm_term=transaction&spm=1018.2226.3001.4450

相关会议纪要

📌 01-02 | 核心服务存储主题分享 2025年1月2日 - 智能纪要

12-19 | 核心服务存储主题分享 2024年12月19日 - 智能纪要

12-05 | 核心服务存储主题分享 2024年12月5日 - 智能纪要