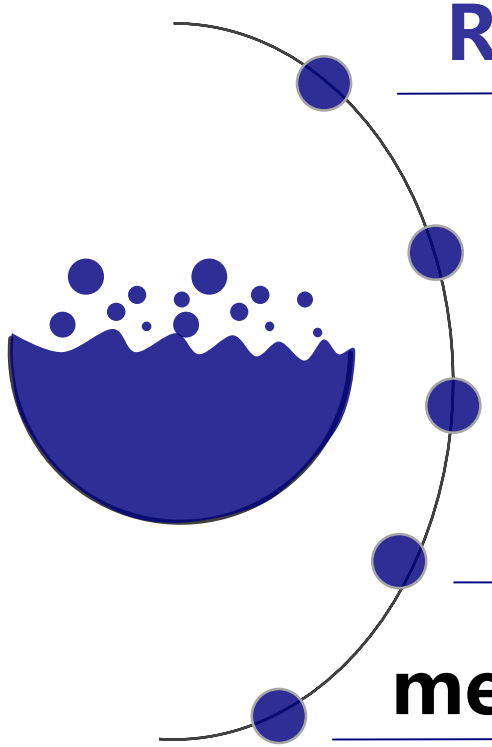
Alibaba Group
阿里巴巴集团

RocksDB/MyRocks源码学习—写

基础架构事业群-数据库技术-数据库内核
王德浩

目录 | Contents



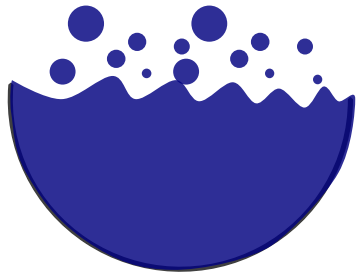
RocksDB写流程

✓总体来讲，RocksDB的写流程分为两个阶段完成

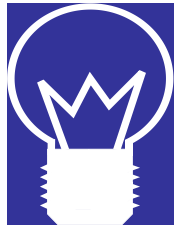
写入每个事务独有的WriteBatch

WriteBatch写入memtable

目录 | Contents



写WriteBatch



每一个线程发起的事务均有一个对应的writebatch。

Writebatch的作用在于：事务内的操作可以批量处理。

插入操作

插入操作比较好理解，直接写入writebatch就可以了。

删除操作

删除操作在writebatch里本质上也是写入，只不过是标记了待删除的key。

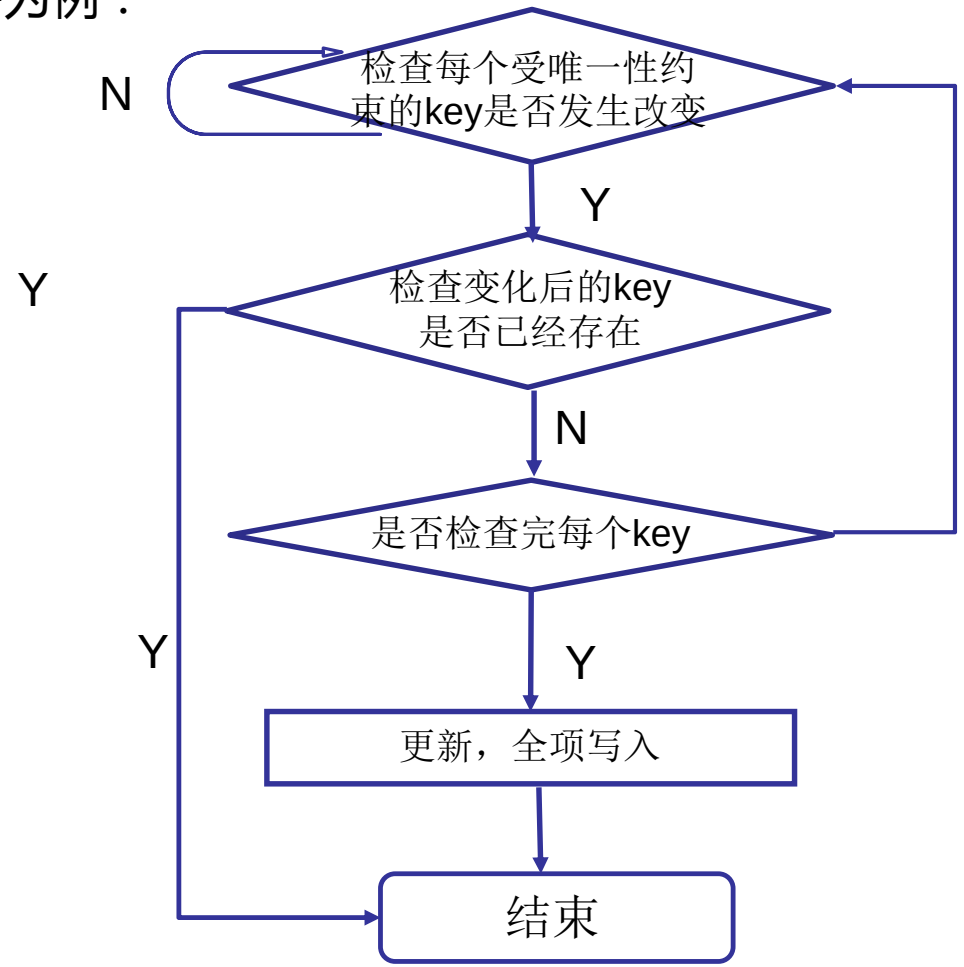
更新操作

如果更新主键或二级索引，会删除原有记录，插入新记录。其他情况下会新插入一条数据，而并不会删除原有的数据。

后台compaction的时候会真正的删除**重复的**和**标记删除**的数据。

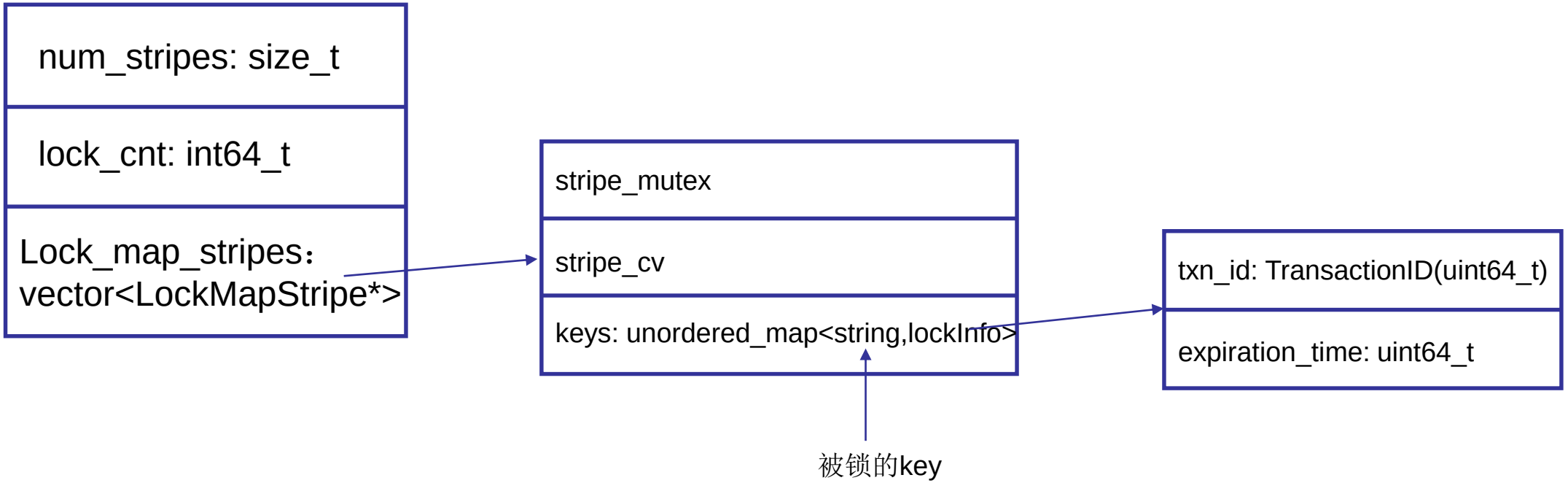
唯一性检查

- 会在写入前使用get_for_update(加锁，当前读)去检查需满足唯一性的key。
- 以一次update为例：



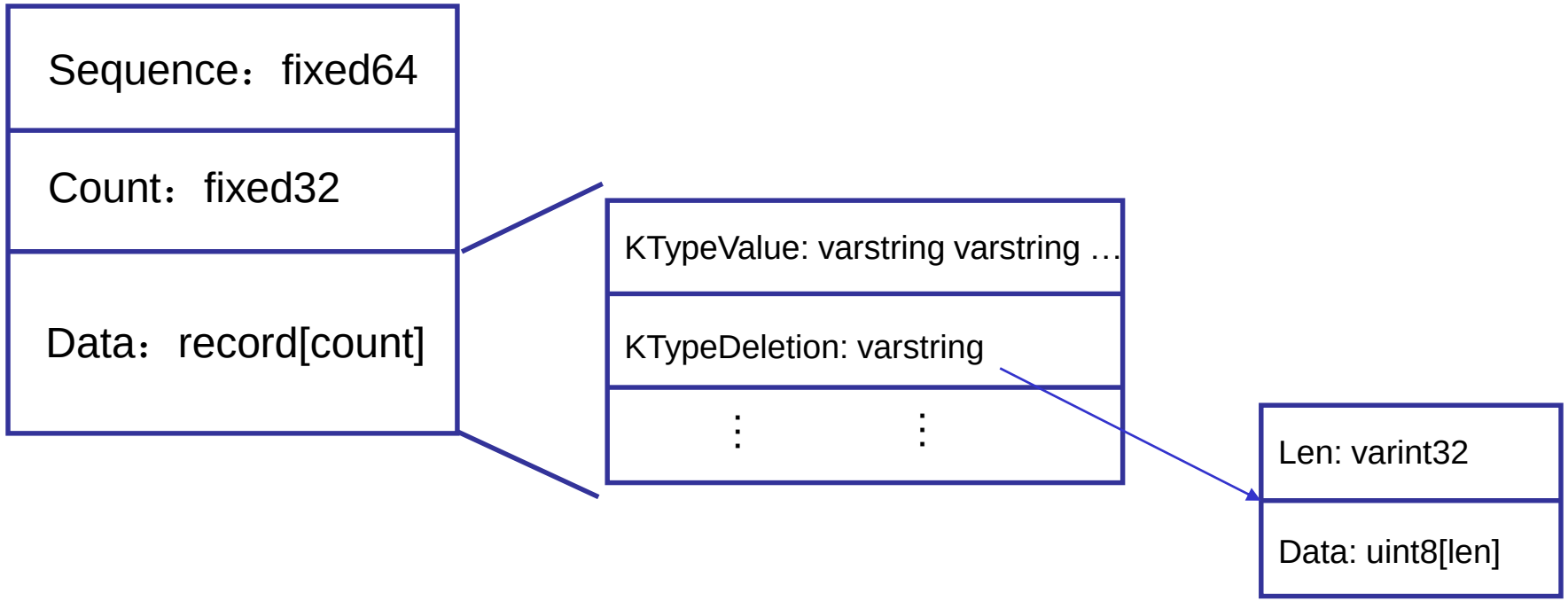
RocksDB写锁

✓**lockmap：**每一个Column family对应一个lockmap，用来存储上锁的信息:



Writebatch 结构

- Writebatch类的成员变量string rep_ 用来存放记录,它的格式如下：

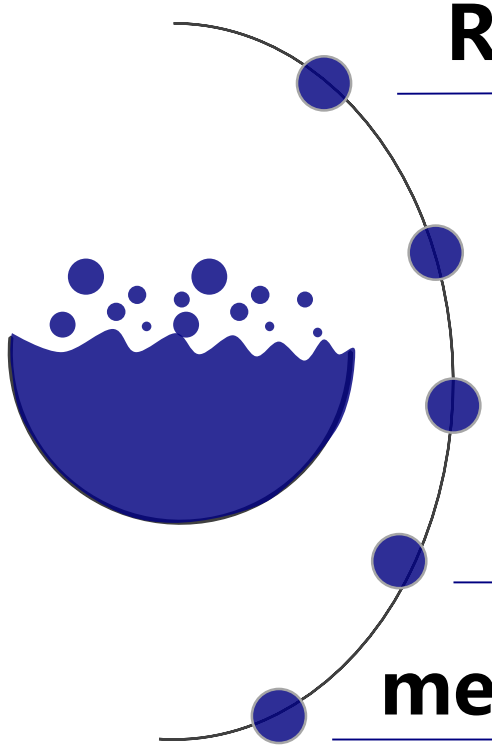


commit_in_the_middle

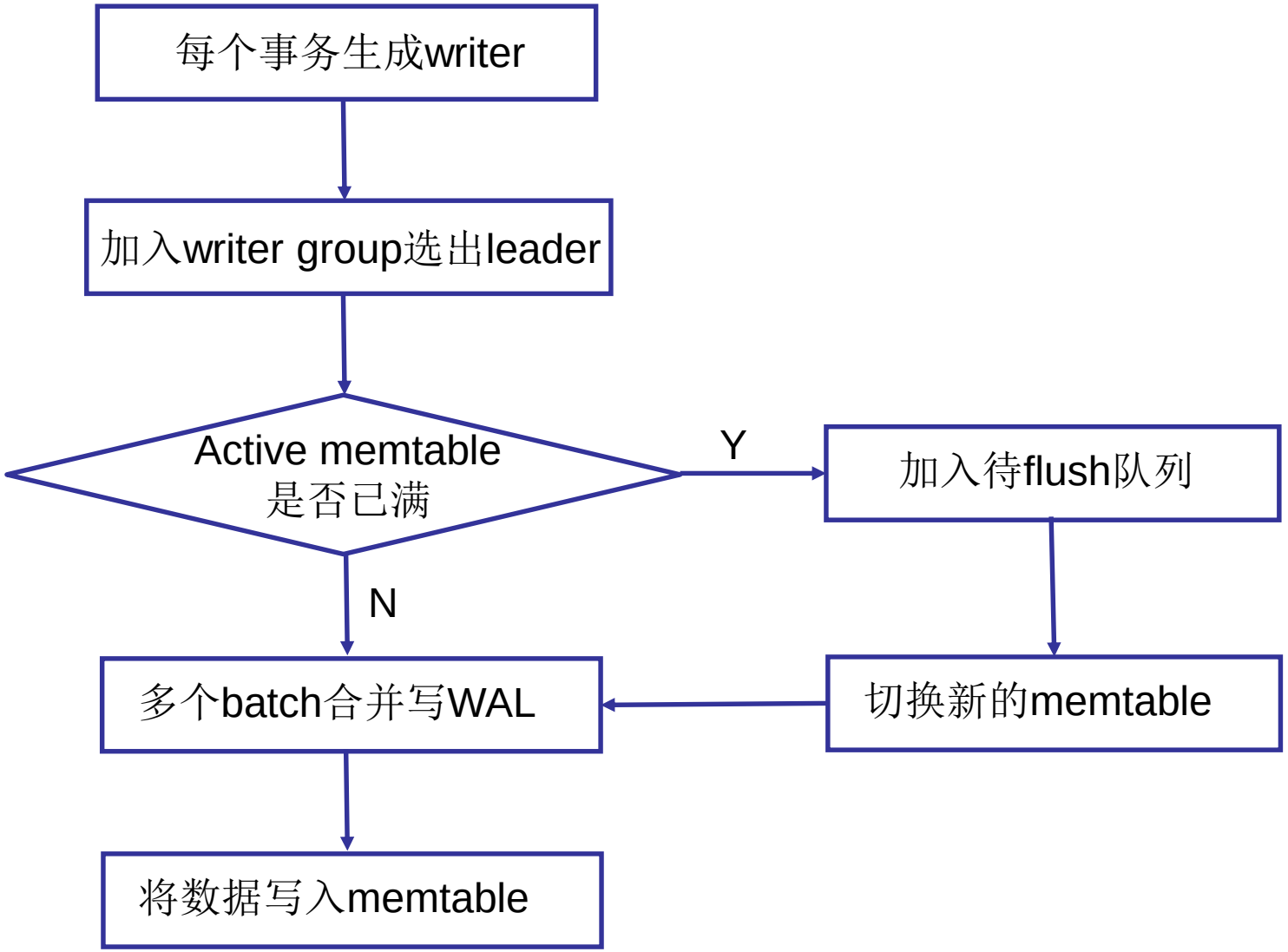
当commit_in_the_middle=1

- 当一个事务过大时，设置该参数可以执行事务阶段性提交，这就意味着不必等事务中所有语句执行完了再提交。
- 在一条语句写入writebatch后，判断如果设置了该参数为true，且目前writebatch中写入的记录数超过了bulk_load_size，就提交这一部分。
- bulk_load_size：default=1000，min=1，max=1024*1024*1024

目录 | Contents



写入memtable 流程

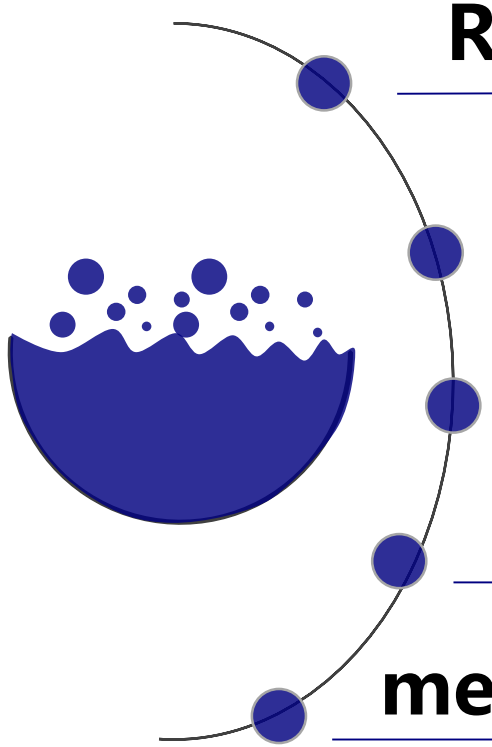


writer

首先，writer结构体包含一个batch写入的一些信息。每一个事务的线程运行到这里时，会生成相应的writer，同时到达的几个线程的writer会加入writer group，第一个到达的既是leader，负责执行最后的插入。

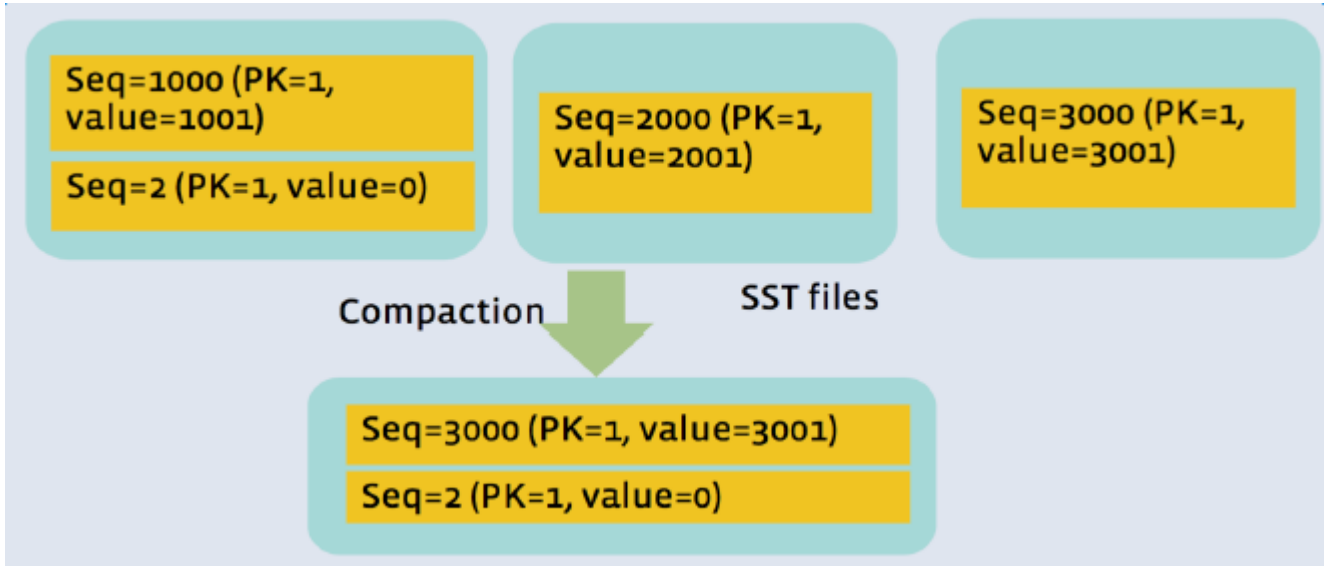
```
struct Writer {
   WriteBatch* batch;
   bool sync;
   bool disableWAL;
   bool disable_memtable;
   uint64_t log_used;  // log number that this batch was inserted into
   uint64_t log_ref;   // log number that memtable insert should reference
   bool in_batch_group;
   WriteCallback* callback;
   bool made_waitable;          // records lazy construction of mutex and cv
   std::atomic<uint8_t> state;  // write under StateMutex() or pre-link
   ParallelGroup* parallel_group;
   SequenceNumber sequence;  // the sequence number to use
   Status status;            // status of memtable inserter
   Status callback_status;   // status returned by callback->Callback()
   std::aligned_storage<sizeof(std::mutex)>::type state_mutex_bytes;
   std::aligned_storage<sizeof(std::condition_variable)>::type state_cv_bytes;
   Writer* link_older;  // read/write only before linking, or as leader
   Writer* link_newer;  // lazy, read/write only before linking, or as leader
}
```

目录 | Contents



Sequence Number

- 对于查询操作，会在查询前获取current sequence number。
- 对于写操作，会在事务提交写入Log前获得current sequence number，并将(current sequence number +1)写入Log。
- 在将数据写入memtable时，也会将(current sequence number +1)加入记录中事务提交成功后，才会为执行current sequence number =current sequence number +1
- 不改变主键的更新操作，不会立即删除原记录，但在compaction阶段，会根据sequence number的大小，保留sequence Number最大的记录，删除其它的。



RocksDB MVCC简述

✓**Sequence ：**事务开启时会获得一个sequence number，因此查询时遇到sequence number 比自己大的写入数据一律忽略。

✓**文件级别的多版本：**当sst文件发生compaction或memtable写入硬盘时，一个新的version产生，在源码中有一个VesionSet对象专门管理sequence和version，version保存的是sst文件的相关信息。在任意时刻，都有一个最新的当前version。当一个version不再被任何一个查询使用时，便会被删掉。

example

Iterator1正在查询

v1={f1, f2, f3} (current, used by iterator1)
files on disk: f1, f2, f3

新的memtable写入磁盘

v2={f1, f2, f3, f4} (current)
v1={f1, f2, f3} (used by iterator1)
files on disk: f1, f2, f3, f4

文件f2,f3,f4合并为f5

v3={f1, f5} (current)
v2={f1, f2, f3, f4}
v1={f1, f2, f3} (used by iterator1)
files on disk: f1, f2, f3, f4, f5

此时，v2既不是最新的，又没有被某个查询占用，删除。
同时，f4也被删除了。

v3={f1, f5} (current)
v1={f1, f2, f3} (used by iterator1)
files on disk: f1, f2, f3, f5

若此时Iterator1 被销毁，v1将被删除，同时，f2和f3也将被删除。

v3={f1, f5} (current)
files on disk: f1, f5

每个version和sst文件均保持着一个reference counts

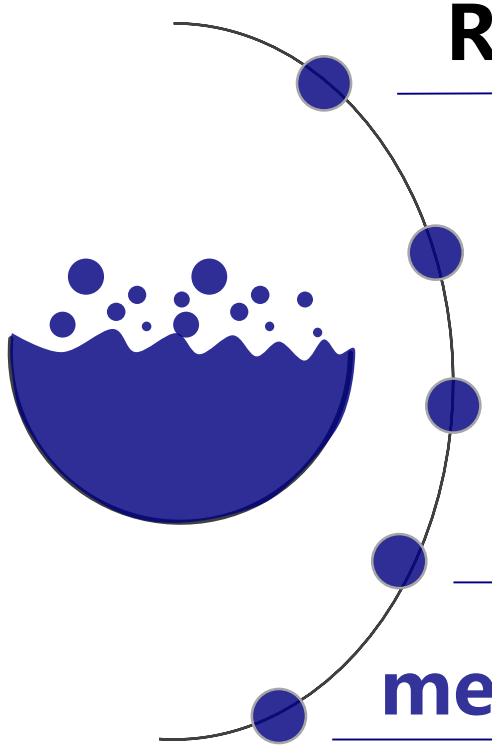
manifest文件

✓**VersionEdit：**这个类表示着version之间的变化，任何对version的修改（如文件增加删除，cf的修改等）都会先用versionedit表示，然后再去修改version：

Version0 +VersionEdit-->Version1

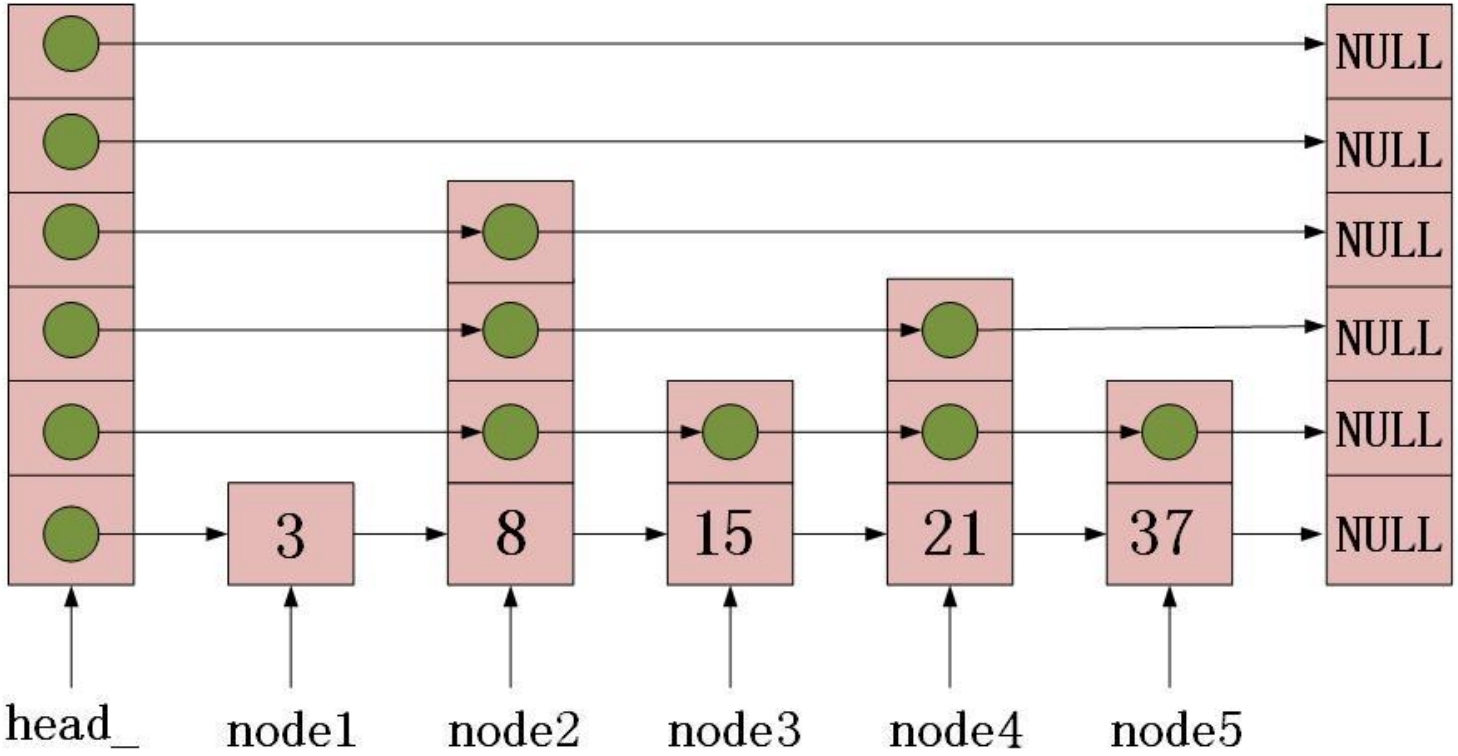
✓**manifest文件：**该文件保存的便是VersionEdit，即记录了状态转移，对应于current version，也有一个current manifest。因此该文件在开机时必不可少，恢复当前version。

目录 | Contents



skiplist

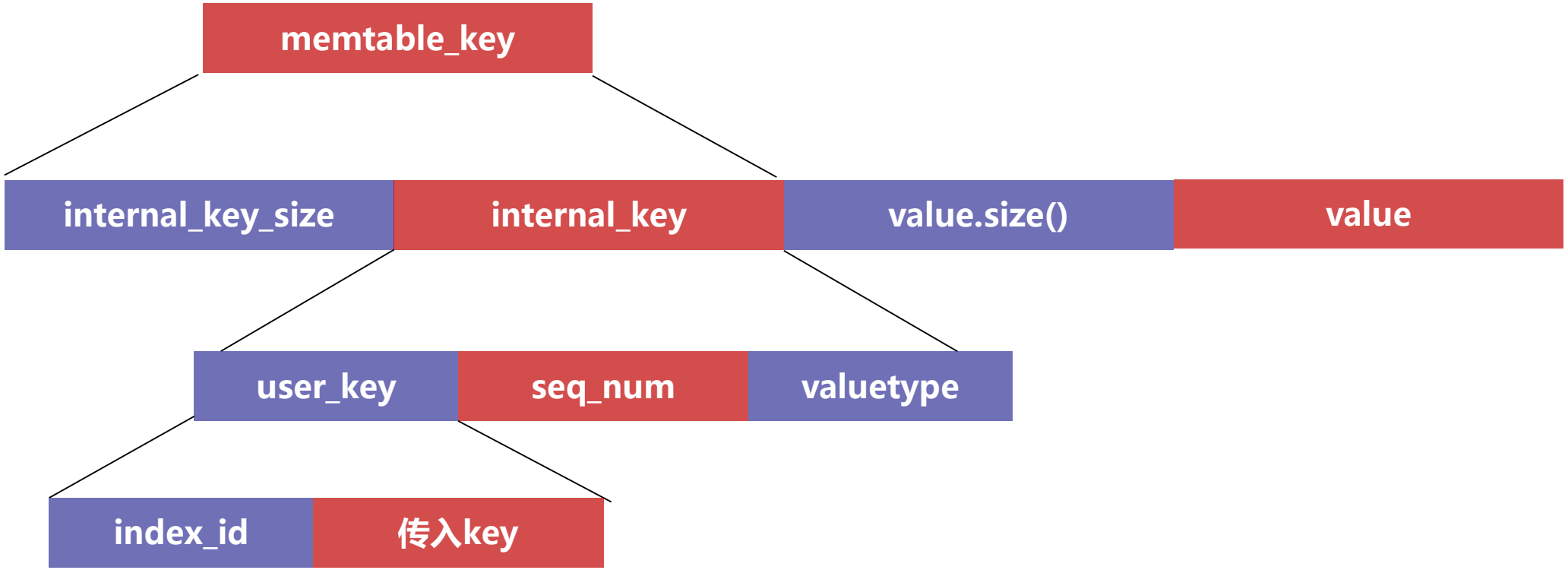
RocksDB的内部memtable由skiplist实现



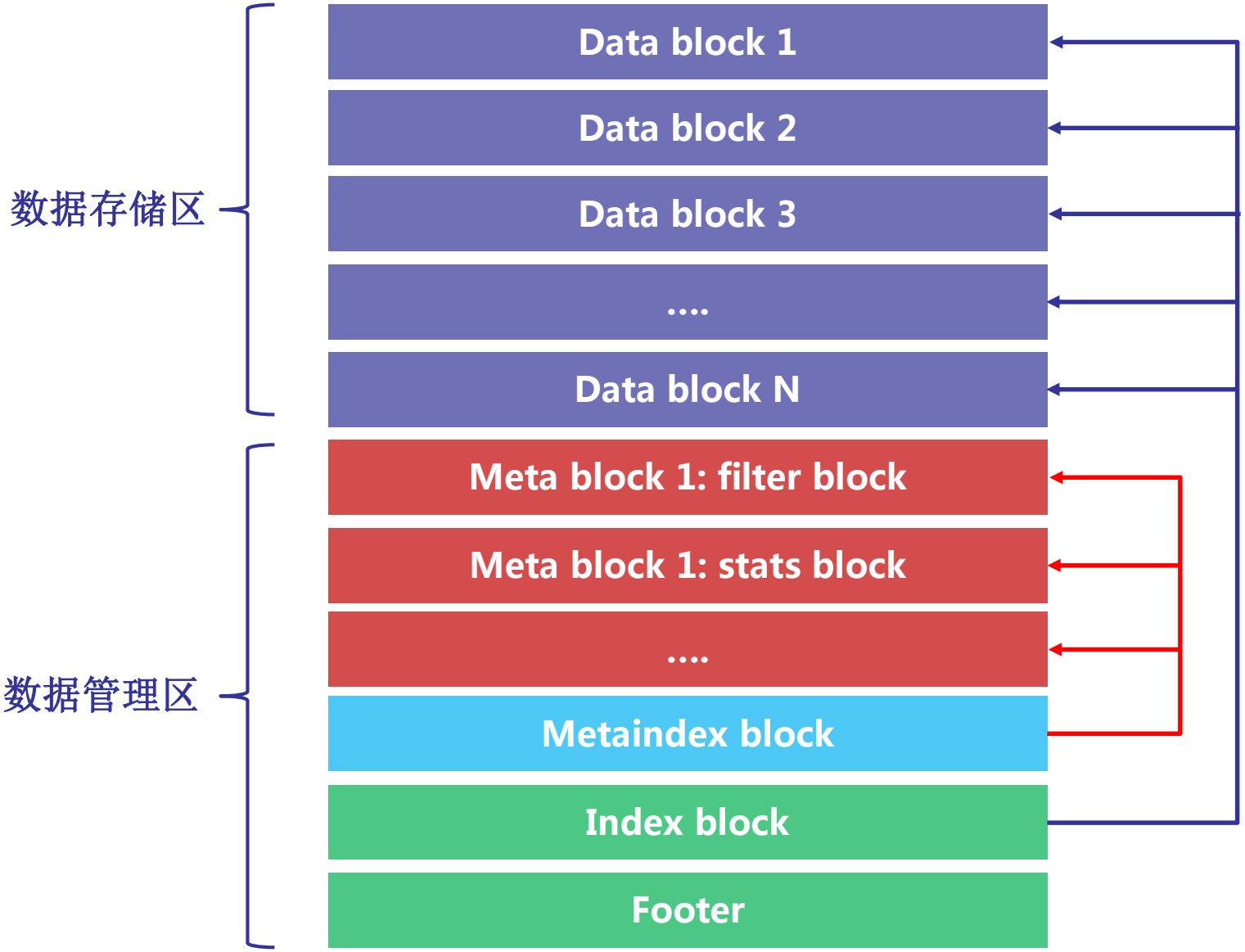
跳转表的实现：每个节点是一个数组，数组的大小代表了它的塔高。

跳转表的理论时间复杂度为O(logn)。

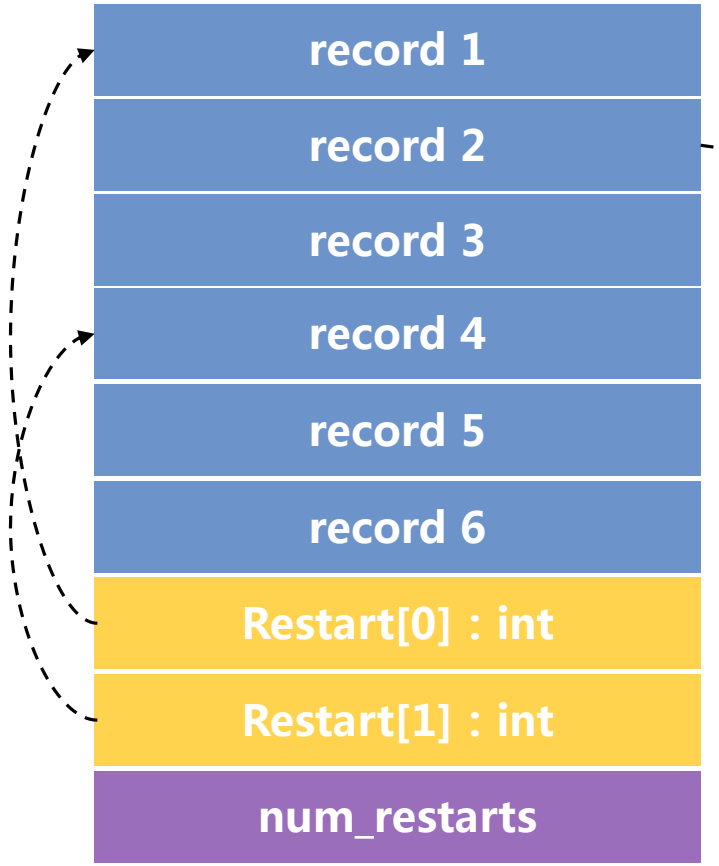
Memtable存储格式



SST 文件格式


数据存储区
Data block 1
Data block 2
Data block 3
....
Data block N
数据管理区
Meta block 1: filter block
Meta block 1: stats block
....
Metaindex block
Index block
Footer

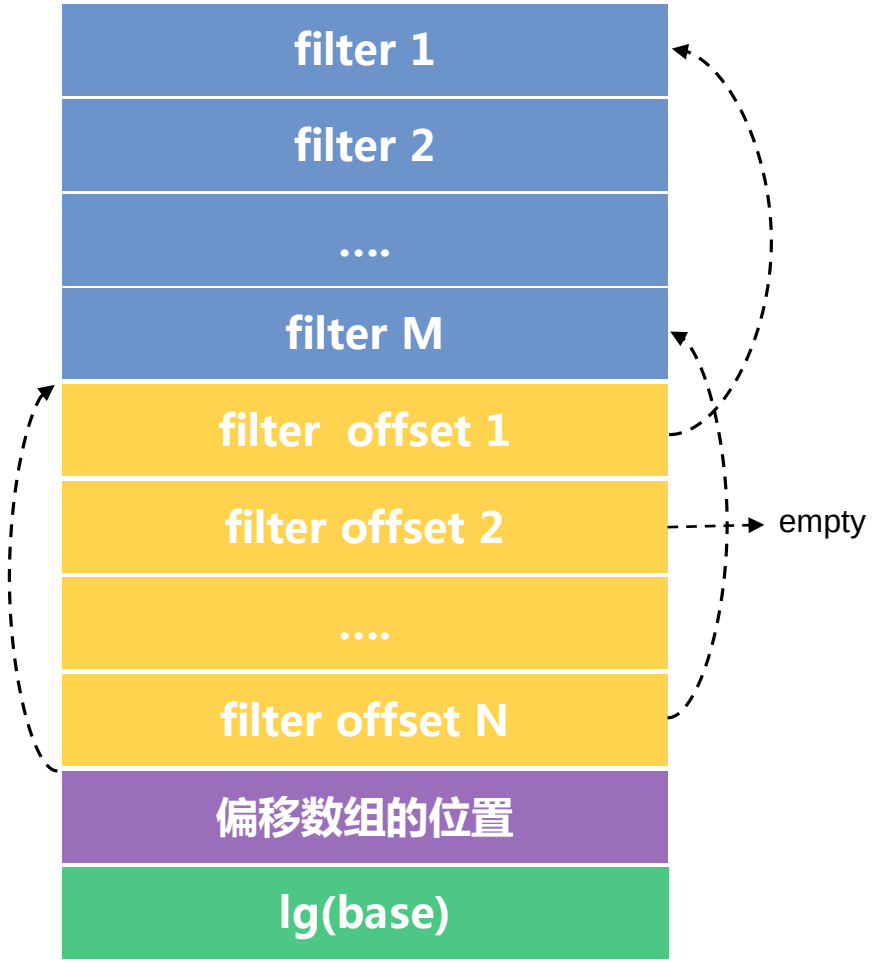
Data Block



data block中的key是有序存储的，相邻的key之间可能有重复，因此存储时采用前缀压缩，后一个key只存储与前一个key不同的部分。那些重启点指出的位置就表示该key不按前缀压缩，而是完整存储该key。

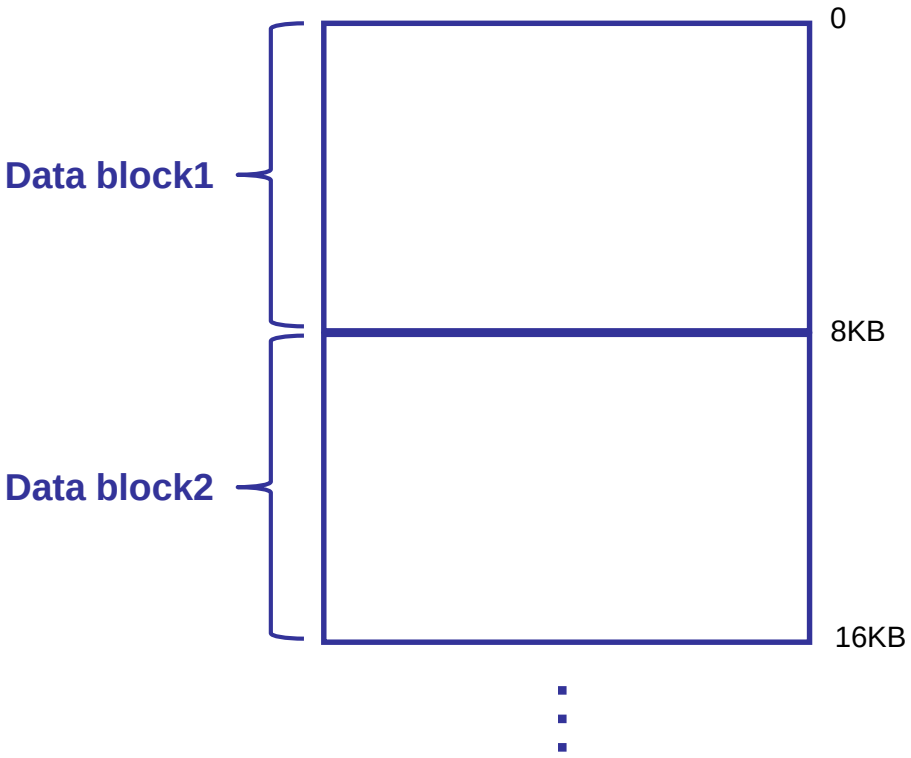
Restart[n]是一个32位的整型，而且一个restart点指向的共享前缀的组是**定长的**，如图中每3条记录共享key前缀。RocksDB中则默认16条记录共享前缀。

Meta Block：filter



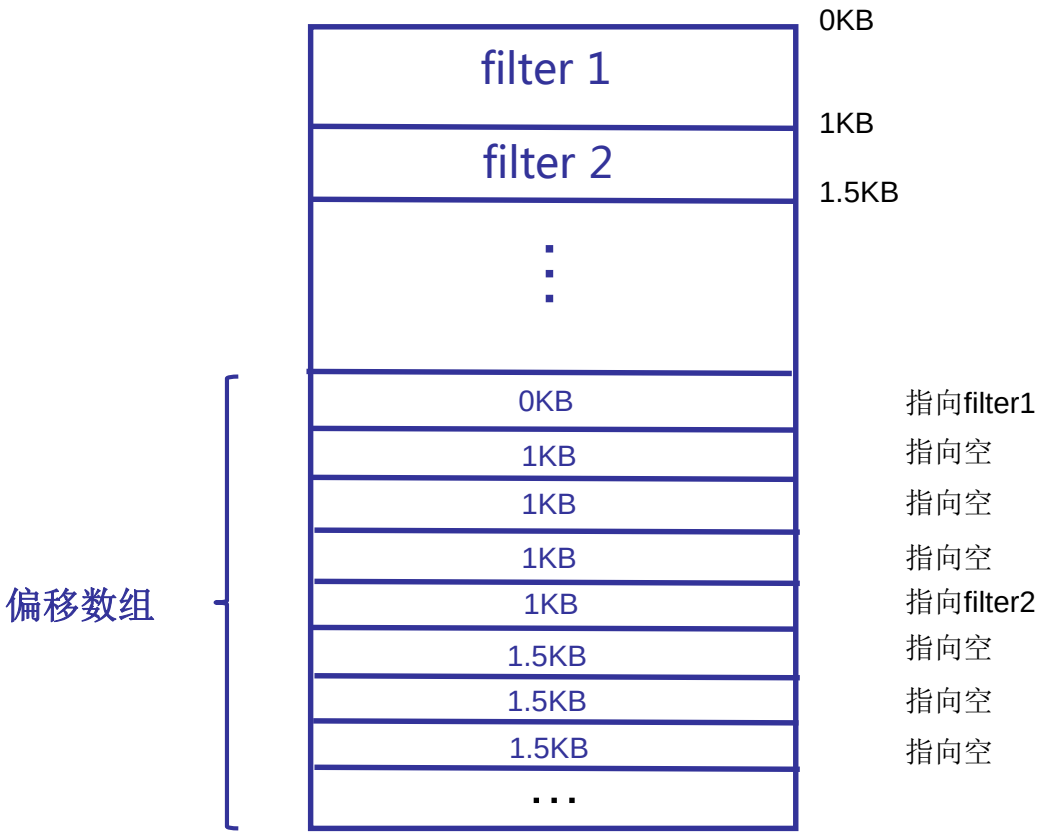
- base是一个数值，大小为2KB。它是以对数（lg）的方式进行存储的，因此lg(base)约等于11，存储只占1字节。
- 在数据存储区的[i*base,(i+1)*base)这一部分的数据（data block），会根据filter offset i 去找相对应的filter j。注意这里的i和j并不一一对应的，由于block的大小可能大于2KB，会导致一些filter offset i指向空。
- filter j 存储的是bloom filter的内容，程序会首先利用它去判断key是否match。不匹配的话直接忽略相应的data block

Example



在某sst文件内，假设一个块大小为8KB

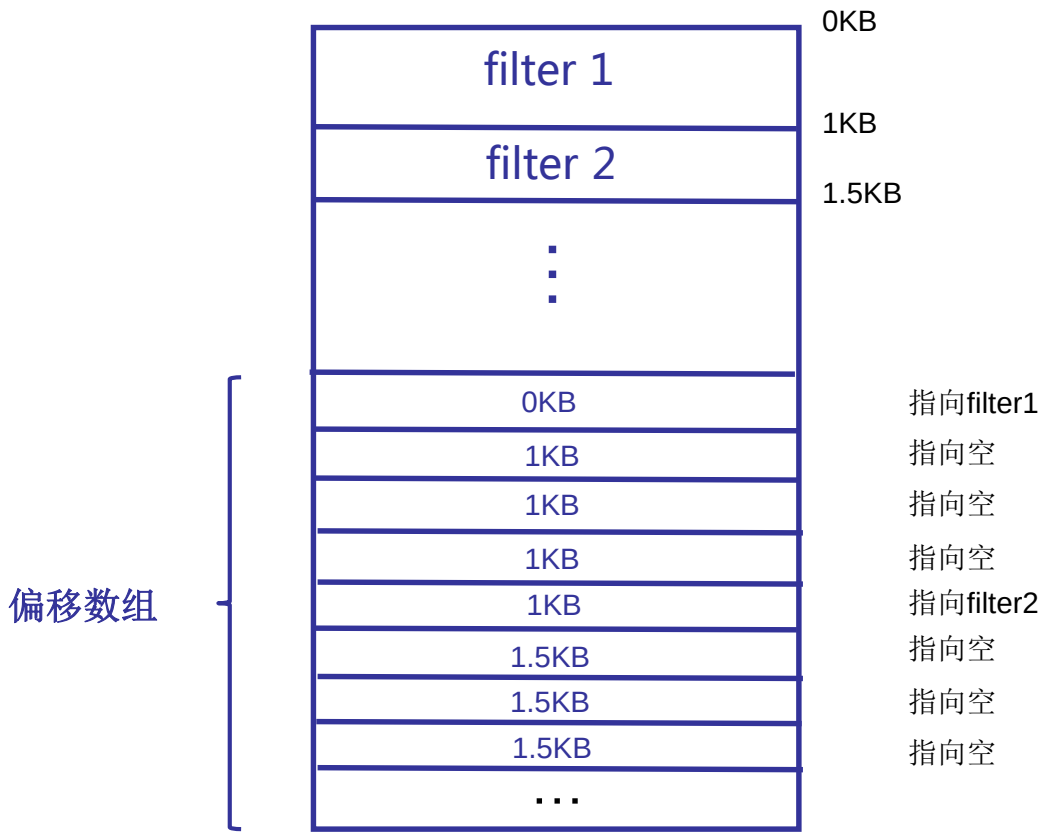
假设data block1作为一个整体flush时生成了1KB大小的filter 1，data block2则生成了0.5KB 的filter 2。Filter block如下：



Example

Bloom filter读取流程，假设获取了data block2的offset为8KB

- 计算出 index=8KB/base , 即index= 4 .
- 获取offset_array[index]（offset_array为偏移数组），为1KB，说明该data block对应的filter的偏移开始处为1KB
- 获得offset[index+1]，为1.5KB，说明data block2对应的filter的的范围便是1KB-1.5KB。

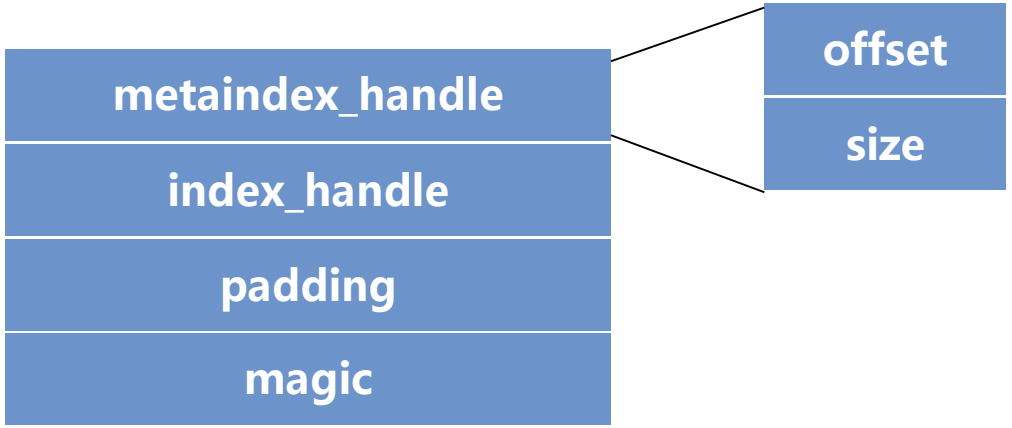


其它

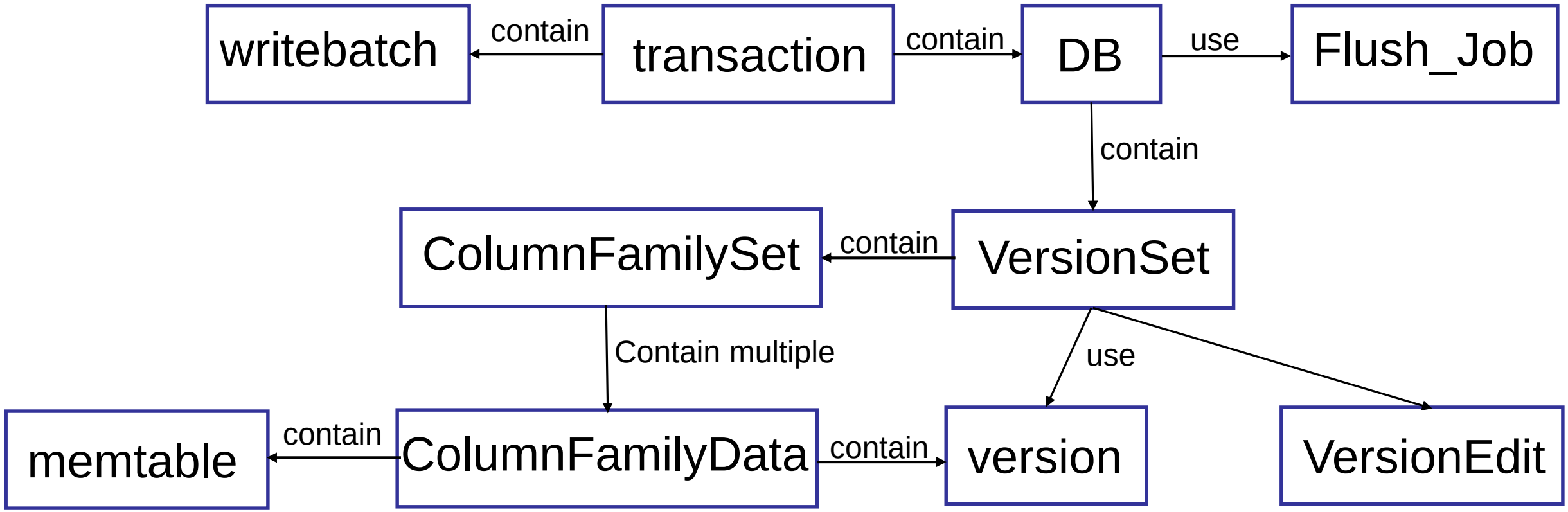
✓meta block: stats

- data size
- index size
- filter size
- raw key size
- raw value size
- number of entries
- number of data blocks

✓footer



RocksDB简化版类关系图



Options文件

- Options文件一般包括四个部分：Version section, DBOptions section, CFOptions section, TableOptions section。其中后两个section每个cf都有自己单独的。

- 它们记录的信息有的在my.cnf中指定了，但options文件的作用在于，在数据库运行的过程中，如果需要修改某些options，例如增加/删除column family等，修改后的参数会自动同步到options文件里去。

- 由于RocksDB在打开某个DB时，会要求传递options作为参数：
  Status DB::Open(const DBOptions& db_options, const std::string& dbname,
  const std::vector<ColumnFamilyDescriptor>& column_families,
  std::vector<ColumnFamilyHandle*>* handles, DB** dbptr)

- 因此没有options文件的话，开发者每次都得自己记下options（rocksdb4.2版本前是这样的），但有了options文件，每次打开DB时加载一下该文件就行了。

example

```
[Version]
  rocksdb_version=4.3.0
  options_file_version=1.1

[DBOptions]
  stats_dump_period_sec=600
  max_manifest_file_size=18446744073709551615
  bytes_per_sync=8388608
  delayed_write_rate=2097152
  WAL_ttl_seconds=0
  WAL_size_limit_MB=0
  max_subcompactions=1
  wal_dir=
  wal_bytes_per_sync=0
  db_write_buffer_size=0
  keep_log_file_num=1000
  table_cache_numshardbits=4
  max_file_opening_threads=1
  writable_file_max_buffer_size=1048576
  random_access_max_buffer_size=1048576
  use_fsync=false
  max_total_wal_size=0
```

```
[CFOptions "default"]
  compaction_style=kCompactionStyleLevel
  compaction_filter=nullptr
  num_levels=6
  table_factory=BlockBasedTable
  comparator=leveldb.BytewiseComparator
  max_sequential_skip_in_iterations=8
  soft_rate_limit=0.000000
  max_bytes_for_level_base=1073741824
  memtable_prefix_bloom_probes=6
  memtable_prefix_bloom_bits=0
  memtable_prefix_bloom_huge_page_tlb_size=0
  max_successive_merges=0
  arena_block_size=16777216
  min_write_buffer_number_to_merge=1
  target_file_size_multiplier=1
  source_compaction_factor=1
  max_bytes_for_level_multiplier=8
  compaction_filter_factory=nullptr
  max_write_buffer_number=8
```

```
[TableOptions/BlockBasedTable "default"]
  format_version=2
  whole_key_filtering=true
  skip_table_builder_flush=false
  no_block_cache=false
  checksum=kCRC32c
  filter_policy=rocksdb.BuiltinBloomFilter
  block_size_deviation=10
  block_size=8192
  block_restart_interval=16
  cache_index_and_filter_blocks=false
  pin_l0_filter_and_index_blocks_in_cache=false
  index_type=kBinarySearch
  hash_index_allow_collision=true
  flush_block_policy_factory=FlushBlockBySizePolicyFactory
```

致谢

谢谢大家！