零声教育出品 Mark 老师 QQ : 2548898954

简介

> RocksDB 是 Facebook 的一个实验项目，目的是希望能开发一套能在服务器压力下，真正发挥高速存储硬件性能的高效数据库系统。这是一个C++库，允许存储任意长度二进制 KV 数据。支持原子读写操作。
>
> RocksDB 依靠大量灵活的配置，使之能针对不同的生产环境进行调优，包括直接使用内存，使用 Flash，使用硬盘或者 HDFS。支持使用不同的压缩算法，并且有一套完整的工具供生产和调试使用。
>
> RocksDB 大量复用了 **levedb** 的代码，并且还借鉴了许多 HBase 的设计理念。原始代码从 leveldb 1.5 上fork 出来。同时 RocksDB 也借用了一些 Facebook 之前就有的理念和代码。
>
> RocksDB 应用场景非常广泛；比如支持 redis 协议的 pika 数据库，采用 RocksDB 持久化 redis 支持的数据结构；MySQL 中支持可插拔的存储引擎，facebook 维护的 MySQL 分支中支持 RocksDB；

编译

rocksdb

```
git clone https://github.com/facebook/rocksdb.git
cd rocksdb
# 编译成调试模式
make
# 编译成发布模式
make static_lib
```

压缩库

ubuntu

```
# rocksdb 支持多种压缩模式
# gflags
sudo apt-get install libgflags-dev
# snappy
sudo apt-get install libsnappy-dev
# zlib
sudo apt-get install zlib1g-dev
# bzip2
sudo apt-get install libbz2-dev
# lz4
sudo apt-get install liblz4-dev
# zstandard
sudo apt-get install libzstd-dev
```

centos

```
# gflags
git clone https://github.com/gflags/gflags.git
cd gflags
git checkout v2.0
./configure && make && sudo make install
# snappy
sudo yum install snappy snappy-devel
# zlib
sudo yum install zlib zlib-devel
# bzip2
sudo yum install bzip2 bzip2-devel
# lz4
sudo yum install lz4-devel
# ASAN (optional for debugging)
sudo yum install libasan
# zstandard
sudo yum install libzstd-devel
```

基本接口

```
Status Open(const Options& options, const std::string& dbname, DB** dbptr);

Status Get(const ReadOptions& options, const Slice& key, std::string*
value);
Status Get(const ReadOptions& options,
                           ColumnFamilyHandle* column_family, const Slice&
key,
                           std::string* value);

Status Put(const WriteOptions& options, const Slice& key, const Slice&
value);
Status Put(const WriteOptions& options,
                     ColumnFamilyHandle* column_family, const Slice& key,
                     const Slice& value);

// fix read-modify-write 将 读取、修改、写入封装到一个接口中
Status Merge(const WriteOptions& options, const Slice& key, const Slice&
value);
Status Merge(const WriteOptions& options,
                       ColumnFamilyHandle* column_family, const Slice& key,
                       const Slice& value);

// 标记删除，具体在 compaction 中删除
Status Delete(const WriteOptions& options, const Slice& key);
Status Delete(const WriteOptions& options,
                        ColumnFamilyHandle* column_family, const Slice& key,
                        const Slice& ts);

// 针对从来不该写且已经存在的key; 这种情况比 delete 删除的快;
Status SingleDelete(const WriteOptions& options, const Slice& key);
Status SingleDelete(const WriteOptions& options,
                              ColumnFamilyHandle* column_family,
                              const Slice& key);

```

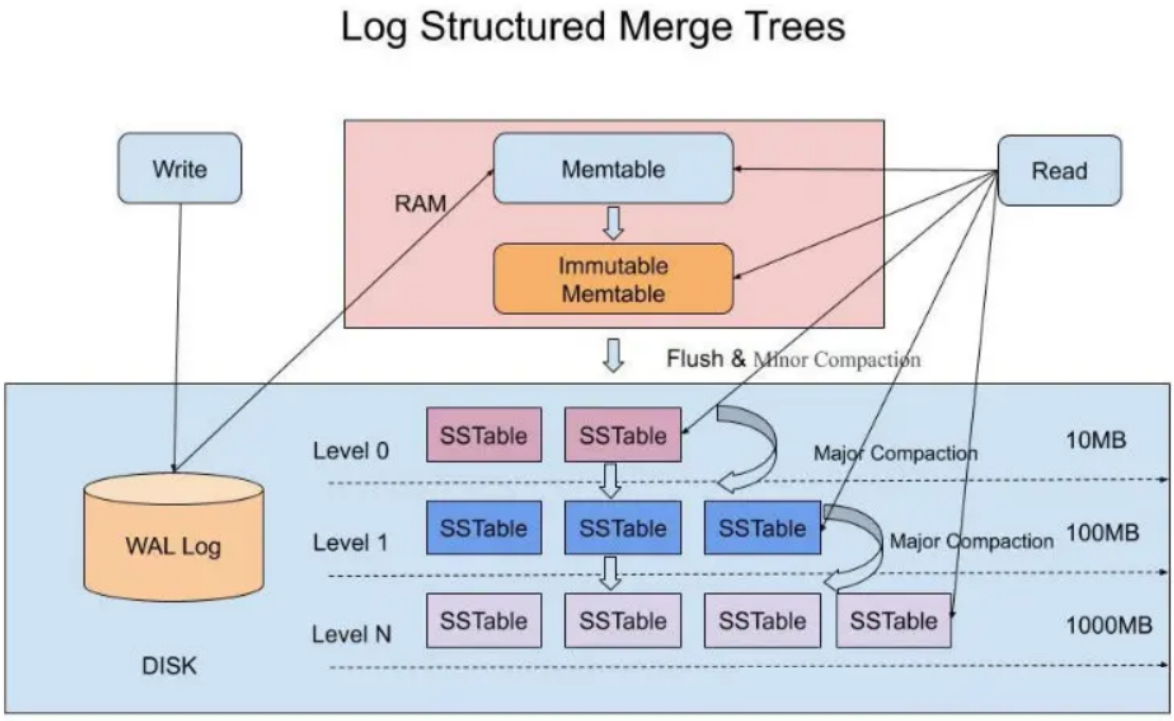
```
// 迭代器会阻止 compaction 清除数据，使用完后需要释放;
Iterator* NewIterator(const ReadOptions& options);
Iterator* NewIterator(const ReadOptions& options,
                      ColumnFamilyHandle* column_family)
```

高度分层架构

> RocksDB 是一种可以存储任意**二进制kv数据**的嵌入式存储。RocksDB 按顺序组织所有数据，他们的通用操作是 `Get(key)`，`NewIterator()`，`Put(key, value)`，`Delete(Key)` 以及 `SingleDelete(key)`。
>
> RocksDB 有三种基本的数据结构：memtable，sstfile以及logfile。memtable是一种内存数据结构——所有写入请求都会进入 memtable，然后选择性进入 logfile。logfile 是一个在存储上顺序写入的文件。当memtable 被填满的时候，他会被刷到 sstfile 文件并存储起来，然后相关的 logfile 会在之后被安全地删除。sstfile 内的数据都是排序好的，以便于根据 key 快速搜索。
>
> RocksDB 是基于 lsm-tree (log structured merge - tree) 实现;

LSM-Tree

> LSM-Tree 的核心思想是利用顺序写来提升写性能；LSM-Tree 不是一种树状数据结构，仅仅是一种存储结构；LSM-Tree 是为了**写多读少**（对读的性能实时性要求相对较低）的特定场景而提出的解决方案；如日志系统、海量数据存储、数据分析;



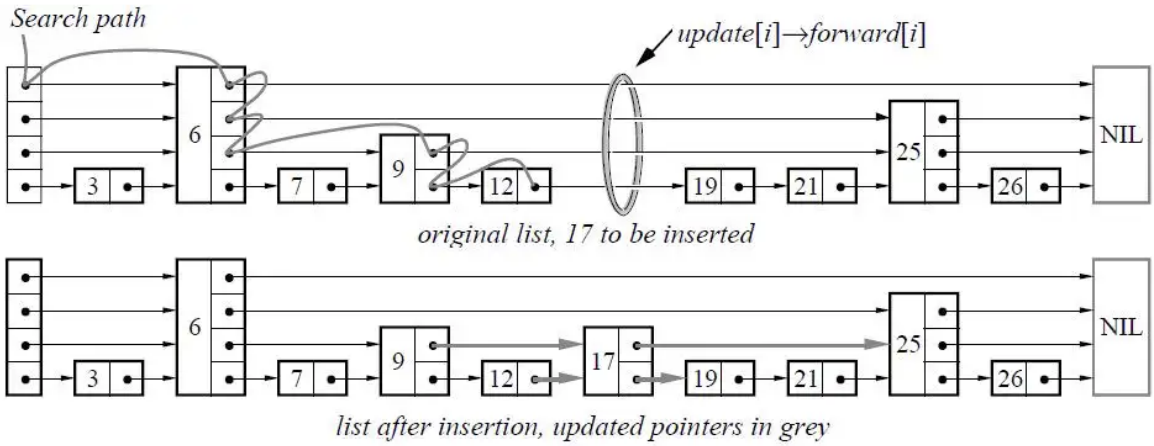
关于访问速度

- 磁盘访问时间：寻道时间 + 旋转时间 + 传输时间;
  - 寻道时间：8ms~12ms;
  - 旋转时间：7200转/min（半周 4ms）;
  - 传输时间：50M/s（约0.3ms）;
- 磁盘随机 IO ≪ 磁盘顺序 IO ≈ 内存随机 IO ≪ 内存顺序 IO;

MemTable

> MemTable是一个内存数据结构，他保存了落盘到 SST 文件前的数据。他同时服务于读和写——新的写入总是将数据插入到 memtable，读取在查询 SST 文件前总是要查询 memtable，因为 memtable 里面的数据总是更新的。一旦一个 memtable 被写满，他会变成不可修改的，并被一个新的 memtable 替换。一个后台线程会把这个 memtable 的内容落盘到一个 SST 文件，然后这个 memtable 就可以被销毁了。
>
> 默认的 memtable 实现是基于 skiplist 的。
>
> 影响 memtable 大小的选项：
>
> `write_buffer_size:` 一个 memtable 的大小；
>
> `db_write_buffer_size:` 多个列族的 memtable 的大小总和；这可以用来管理 memtable 使用的总内存数；
>
> `max_write_buffer_number:` 内存中可以拥有刷盘到 SST 文件前的最大 memtable 数；

Memtable类型	**SkipList**	**HashSkipList**	**HashLinkList**	**Vector**
最佳使用场景	通用	带特殊key前缀的范围查询	带特殊key前缀，并且每个前缀都只有很小数量的行	大量随机写压力
索引类型	二分搜索	哈希+二分搜索	哈希+线性搜索	线性搜索
是否支持全量db有序扫描?	天然支持	非常耗费资源（拷贝以及排序一生成一个临时视图）	同HashSkipList	同HashSkipList
额外内存	平均（每个节点有多个指针）	高（哈希桶+非空桶的skiplist元数据+每个节点多个指针）	稍低（哈希桶+每个节点的指针）	低（vector尾部预分配的内存）
Memtable落盘	快速，以及固定数量的额外内存	慢，并且大量临时内存使用	同HashSkipList	同HashSkipList
并发插入	支持	不支持	不支持	不支持
带Hint插入	支持（在没有并发插入的时候）	不支持	不支持	不支持



落盘策略

- Memtable 的大小在一次写入后超过 `write_buffer_size`。
- 所有列族中的 memtable 大小超过 `db_write_buffer_size` 了，或者 `write_buffer_manager` 要求落盘。在这种场景，最大的 memtable 会被落盘；
- WAL 文件的总大小超过 `max_total_wal_size`。在这个场景，有着最老数据的 memtable 会被落盘，这样才允许携带有跟这个 memtable 相关数据的 WAL 文件被删除。

WAL

> 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 fflush来保证一致性。
>
> WAL 创建时机：
>
> 1. db打开的时候；
> 2. 一个列族落盘数据的时候；（新的创建、旧的延迟删除）

重要参数

- `DBOptions::max_total_wal_size`： 如果希望限制 WAL 的大小，RocksDB 使用 `DBOptions::max_total_wal_size` 作为列族落盘的触发器。一旦 WAL 超过这个大小，RocksDB 会开始强制列族落盘，以保证删除最老的 WAL 文件。这个配置在列族以不固定频率更新的时候非常有用。如果没有大小限制，如果这个WAL中有一些非常低频更新的列族的数据没有落盘，用户可能会需要保存非常老的 WAL 文件。
- `DBOptions::WAL_ttl_seconds`，`DBOptions::WAL_size_limit_MB`：这两个选项影响 WAL 文件删除的时间。非0参数表示时间和硬盘空间的阈值，超过这个阀值，会触发删除归档的WAL文件。

Immutable MemTable

> Immutable MemTable 也是存储在内存中的数据，不过是只读的内存数据；
>
> 当 MemTable 写满后，会被置为只读状态，变成 Immutable MemTable。然后会创建一块新的 MemTable 来提供写入操作；Immutable MemTable 将异步 flush 到 SST 中；

SST

> SST （Sorted String Table） 有序键值对集合；是 LSM-Tree 在磁盘中的数据结构；可以通过建立 key 的索引以及布隆过滤器来加快 key 的查询；LSM-Tree 会将所有的 DML 操作记录保存在内存中，继而**批量顺序**写到磁盘中；这与 B+ Tree 有很大不同，B+ Tree 的数据更新直接需要找到原数据所在页并修改对应值；而 SM-Tree 是直接 append 的方式写到磁盘；虽然后面会通过合并的方式去除冗余无效的数据；

文件格式

```
<beginning_of_file>
[data block 1]
[data block 2]
...
[data block N]
[meta block 1: filter块]
[meta block 2: stats块]
[meta block 3: 压缩字典块]
[meta block 4: 范围删除块]
...
[meta block K: 未来拓展块]        (我们以后可能会加入新的元数据块)
[metaindex block]
[index block]
[Footer]                                  (定长脚注，从file_size - sizeof(Footer)开始)
<end_of_file>
```

`BlockHandle`

```
offset:         varint64
size:           varint64
```

1. 键值对序列在文件中是以排序顺序排列的，并且切分成了一序列的数据块。这些块在文件头一个接一个地排列。每个数据块都根据 `block_builder.cc` 的代码进行编排（参考文件中的注释），然后选择性压缩（compress）；
2. 数据块之后，我们存出一系列的元数据块。支持的元数据块类型在下面描述。更多的数据块类型可能会在以后加入。同样的，每个元数据块都根据 `block_builder.cc` 的代码进行编排（参考文件中的注释），然后选择性压缩（compress）；
3. 一个 metaindex 块对每个元数据块都有一个对应的入口项，key 为meta块的名字，值是一个 BlockHandle ，指向具体的元数据块。
4. 一个索引块，对每个数据块有一个对应的入口项，key是一个string，该string >= 该数据块的最后一个key，并且小于下一个数据块的第一个key。值是数据块对应的BlockHandle。如果索引类型 IndexType 是kTwoLevelIndexSearch，这个索引块就是索引分片的第二层索引，例如，每个入口指向另一个索引块，该索引块包含每个数据块的索引；
5. 在文件的最后，有一个定长的脚注，包含metaindex以及索引块的BlockHandle，同时还有一个魔数：

```
    metaindex_handle: char[p];      // metaindex的Block handle
    index_handle:     char[q];      // 索引块的Block handle
    padding:          char[40-p-q]; // 填充0以达到固定大小
                                    // (40==2*BlockHandle::kMaxEncodedLength)
    magic:            fixed64;      // 0x88e241b785f4cff7 (小端)
```

BlockCache

> 块缓存是 RocksDB 在内存中缓存数据以用于读取的地方。用户可以带上一个期望的空间大小，传一个 Cache对象给 RocksDB 实例。一个缓存对象可以在同一个进程的多个 RocksDB 实例之间共享，这允许用户控制总的缓存大小。块缓存存储未压缩过的块。用户也可以选择设置另一个块缓存，用来存储压缩后的块。读取的时候会先拉去未压缩的数据块的缓存，然后才拉取压缩数据块的缓存。在打开直接 IO 的时候压缩块缓存可以替代 OS 的页缓存。
>
> RocksDB 里面有两种实现方式，分别叫做 LRUCache 和 ClockCache。两个类型的缓存都通过分片来减轻锁冲突。容量会被平均的分配到每个分片，分片之间不共享空间。默认情况下，每个缓存会被分片到64个分片，每个分片至少有512kB空间。
>
> 用户可以选择将索引和过滤块缓存在 BlockCache 中；默认情况下，索引和过滤块都在 BlockCache 外面存储；
>
> ```
> 1 BlockBasedTableOptions table_options;
> 2 table_options.cache_index_and_filter_blocks = true;
> ```

LRU 缓存

> 默认情况下，RocksDB 会使用 LRU 块缓存实现，空间为 8MB。每个缓存分片都维护自己的 LRU 列表以及自己的查找哈希表。通过每个分片持有一个互斥锁来实现并发。不管是查找还是插入，都需要申请该分片的互斥锁。用户可以通过调用 `NewLRUCache` 创建一个 LRU 缓存；

Clock缓存

> ClockCache实现了CLOCK算法。每个clock缓存分片都维护一个缓存项的环形列表。一个clock指针遍历这个环形列表来找一个没有固定的项进行驱逐，同时，如果在上一个扫描中他被使用过了，那么给予这个项两次机会来留在缓存里。tbb::concurrent_hash_map 被用来查找数据。
>
> 与 LRU 缓存比较，clock 缓存有更好的锁粒度。在 LRU 缓存下面，每个分片的互斥锁在读取的时候都需要上锁，因为他需要更新他的 LRU 列表。在一个 clock 缓存上查找数据不需要申请该分片的互斥锁，只需要搜索并行的哈希表就行了，所以有更好锁粒度。只有在插入的时候需要每个分片的锁。用 clock 缓存，在一定环境下，我们能看到读性能的增长；

LSM-Tree 三大问题

读放大

> 数据分层存放，读取操作也需要分层依次查找，直到找到对应数据；这个过程可能涉及多次 IO；在 range query 的时候，影响更大；

空间放大

> 所有的写入操作都是顺序写，而不是就地更新，无效数据不会马上被清理掉；

写放大

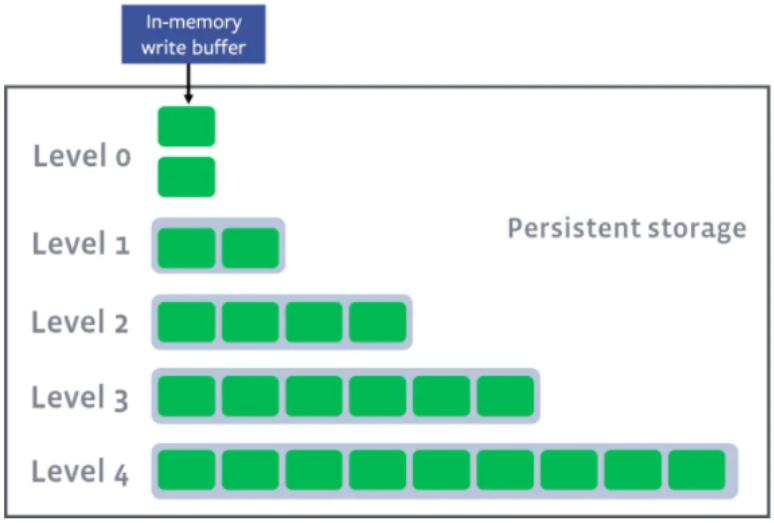
> 用来描述实际写入磁盘的数据大小和程序要求写入的数据大小之比；为了减小读放大和空间放大，RocksDB 采用后台线程合并数据的方式来解决；这些合并过程中会造成对同一条数据多次写入磁盘；

RocksDB compaction

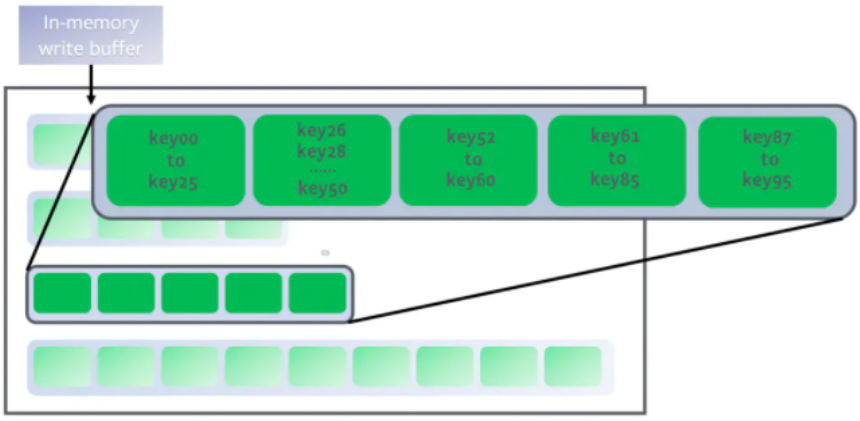
> 为了在读放大、空间放大、以及写放大之间进行取舍，以此适应不同的业务场景；所以需要选择不同的合并算法；rocksdb 默认采用 leveled compaction；

leveled compaction

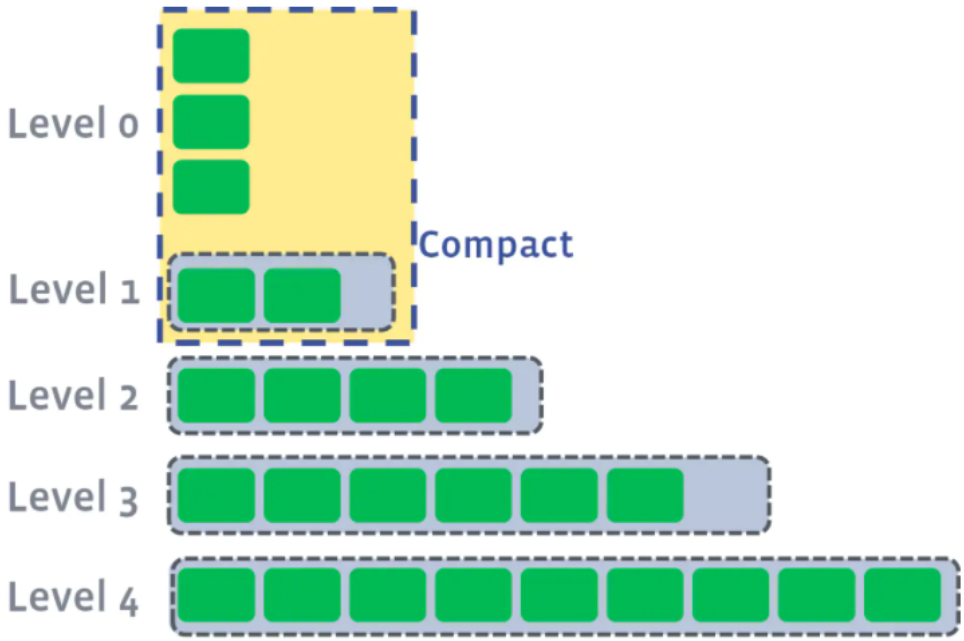
sst 组织方式：



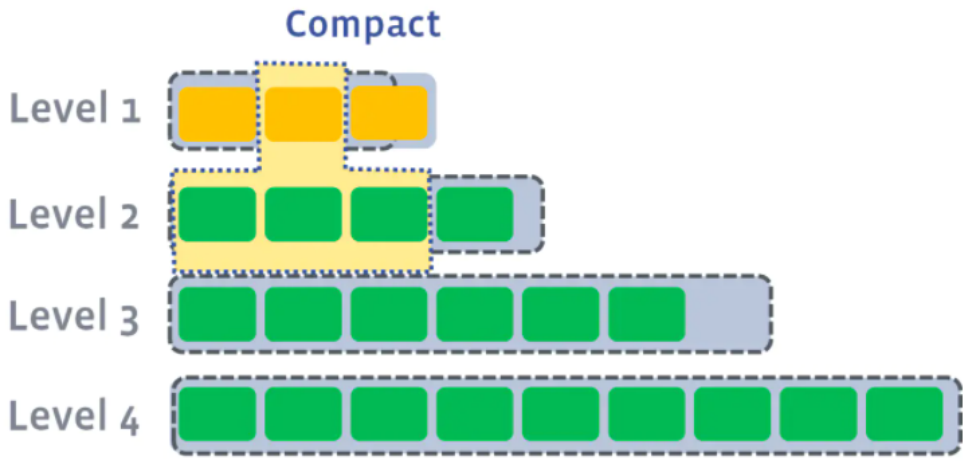
`L1 -> Ln` **按序分片，保存不同文件：**



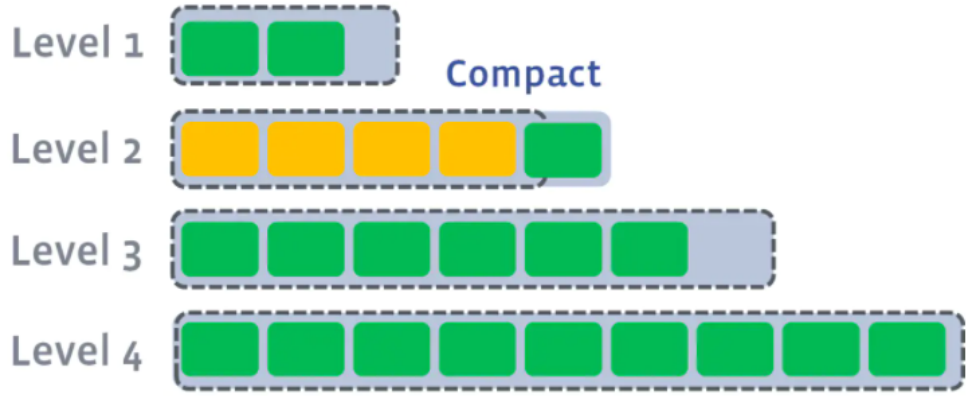
`L0 -> L1` **compaction：**



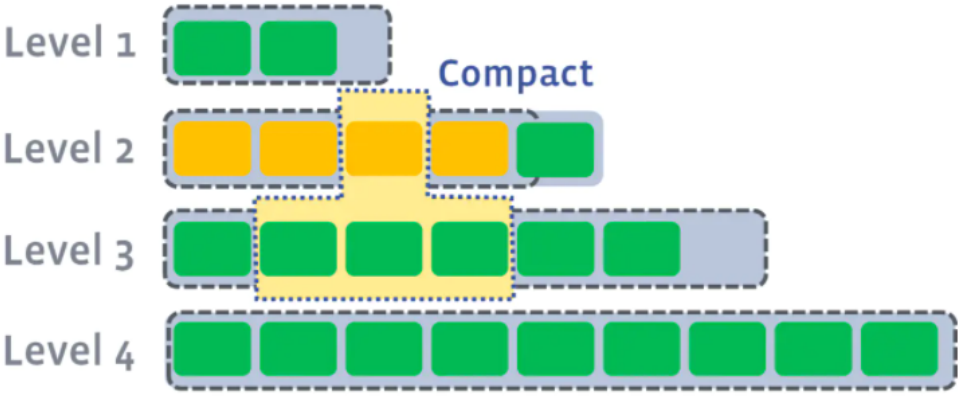
`L1 -> L2 compaction:`



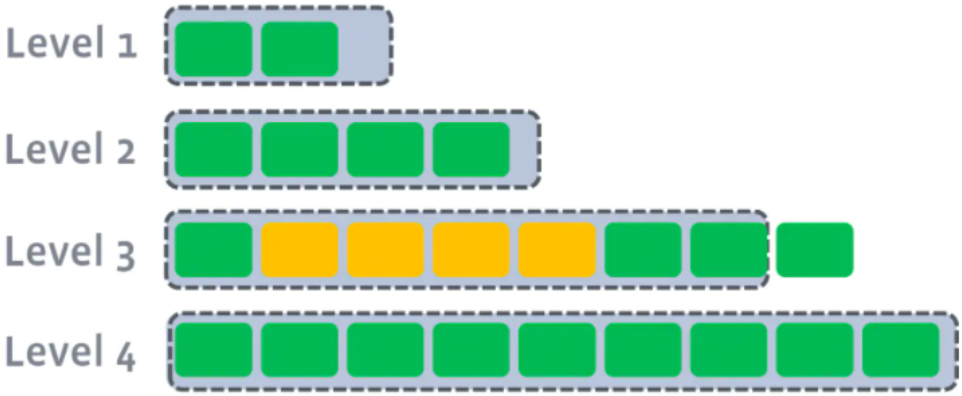
合并后:



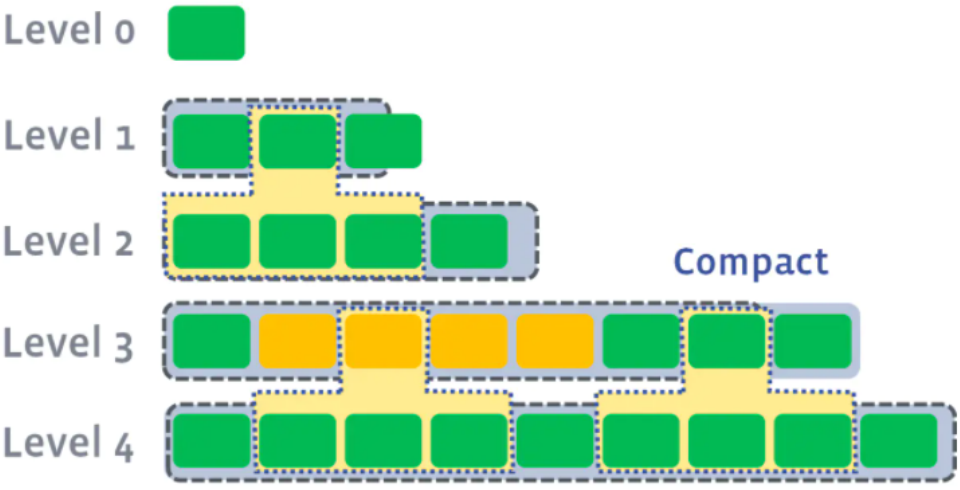
`L2 -> L3 compaction:`



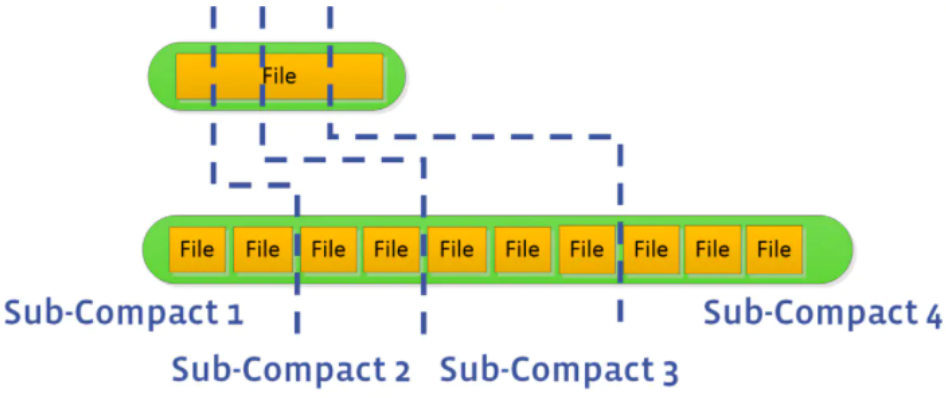
合并后：



并行 compaction：



`L0 subcompaction:`



> L0 向 L1 的 compaction 不可以与其他 level 进行并行 compaction。这将成为整体 compaction 的瓶颈；可以通过设置 `max_subcompactions` 来加速 L0 到 L1 的 compaction；