非关系数据库hw4

10205101530-赵晗瑜

1. **RocksDB**是一种用于快速存储的嵌入式持久存储，请查阅官方网站或相关文献，完成以下两项工作：

RocksDB简介：

RocksDB是Facebook公司基于Google的LevelDB代码库创建的高性能、持久键值的嵌入式单机存储引擎。它针对固态存储驱动器（SSD）的某些特性做了优化，主要面向大规模（分布式）应用程序，并被设计为嵌入在更高级别应用程序中的组件。因此，每个RocksDB实例只管理单个服务器节点的存储设备上的数据；它不处理任何跨应用间操作，例如数据复制和负载均衡，也不执行高级操作，例如快照，检查点，它将这些操作的实现留给上层应用程序，但提供适当的备份还原、故障恢复等支持，以便它们可以更高效的执行和有效的控制。

(1)理解并介绍**RocksDB**的数据模型和**RocksDB**的存储引擎（因为发现数据模型和存储引擎的关系比较紧密，所以就放在一起写了）

RocksDB按照Key-Value形式存储数据，数据在内部根据Key进行排序

1. **RocksDB**按顺序组织所有数据，通用操作包括
    - Get(key)
    - Put(key)
    - Delete(key)
    - NewIterator(key)
2. **RocksDB**的三种数据结构
    - memtable
        i. 内存数据结构
        ii. 所有写入的请求都会进入memtable，选择性进入logfile
    - ssfile

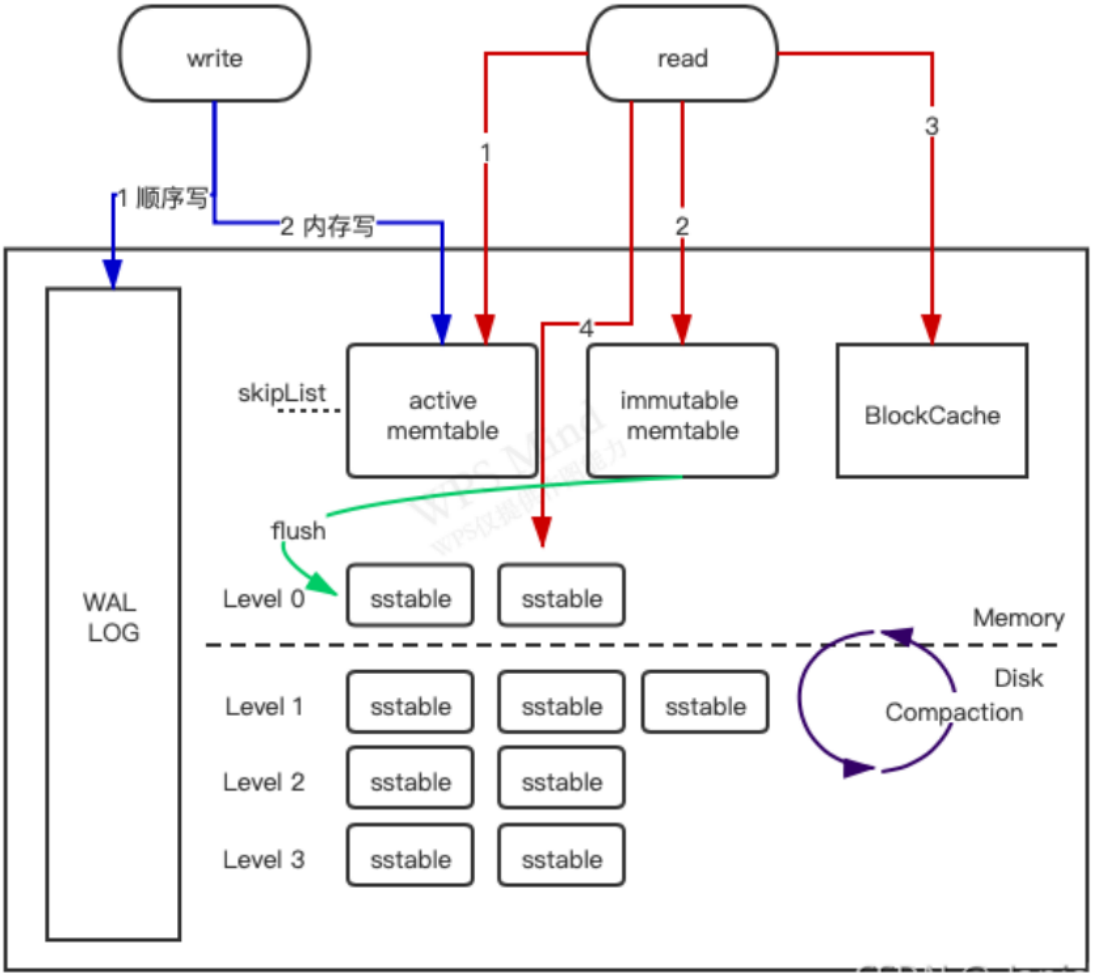
i. 当memtable被填满时，他会被刷到ssfile文件并存储起来，然后相关的logfile会在之后被安全地删除

ii. ssfile中的数据都是排序好的，以便根据key进行快速搜索

- logfile

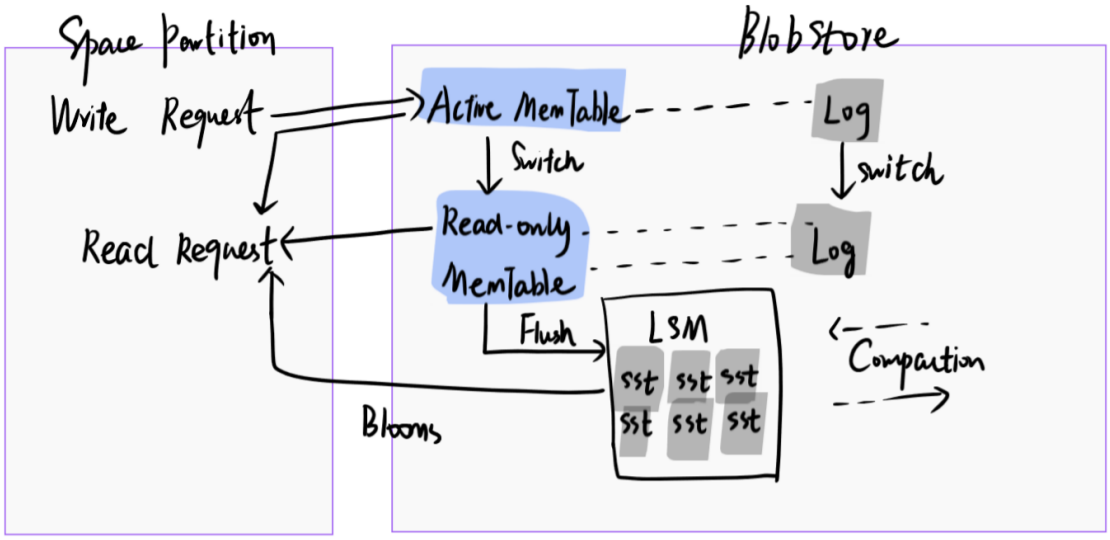
i. 有序写数据结构

3. **RocksDB**采用 **Log-Structured Merge（LSM） trees**作为基本的数据结构，整体数据模型如下图所示：



- 为了保证数据的有序性，插入、搜索的高效性，MemTable基于 跳表实现
- WAL用于故障发生时的数据恢复，可选择关闭
- BlockCache：RocksDB 在内存中缓存数据以供读取的地方。一个Cache对象可以被同一个进程中的多个RocksDB实例共享，用户可以控制整体的缓存容量。有LRUCache和ClockCache两类
- 除了第0层的SST文件，其余层的SST文件之间都是有序的
- 将数据写入sst则是通过异步的flush和compaction 后台线程进行的

4. **RocksDB**的**LSM**基本操作



- 写数据

  当发起一个写请求时，RocksDB首先会进行一个预写的操作，将数据首先以日志的形式写入到持久化存储中（主要是为了防止数据的不一致性），也就是说，为了数据的一致性，同时，也可以为后续故障恢复做一个记录。

  预写入时，采用的是连续写入的方式，这比随机写入的效率要高，尤其是针对持久化存储。接下来，RocksDB会将数据写入到内存中，RocksDB采用内存中一个叫做MemTable的数据结构。

  RocksDB采用跳表的数据结构主要是为了提高读写性能，当内存表写满之后，RocksDB会将Memtable内存表转化成只读状态（read-only），同时新建一个内存表保证后续的写入，只读内存表会通过flush的操作刷到磁盘上，也就意味着每一个只读内存表都会完完整整地写入到磁盘文件中，在RocksDB中文件的名称叫做SST。

- 读数据

  当外界发起一个读请求时，RocksDB会同时从内存表、只读内存表和持久化存储上同时去读取这些数据，主要是因为它查询的数据可能在任何一个地方，并且可能有多条记录

  那么所以**RocksDB**读到相同数据的时候，如何保证哪一个是最新的呢？即哪一个是用户想要的呢？

  这就涉及到RocksDB的一个重要概念——版本机制，RocksDB会给每一个内存表记录一个递增的版本号，那么通过简单的对比我们就可以知道版本信息，即可得出哪一个是最新的

  与此同时，为了加速持久化存储数据结构的效率，SST的每一个文件都自带一个布隆过滤器，特点是节省空间，但是偶尔也会有一些误判，

- 压缩

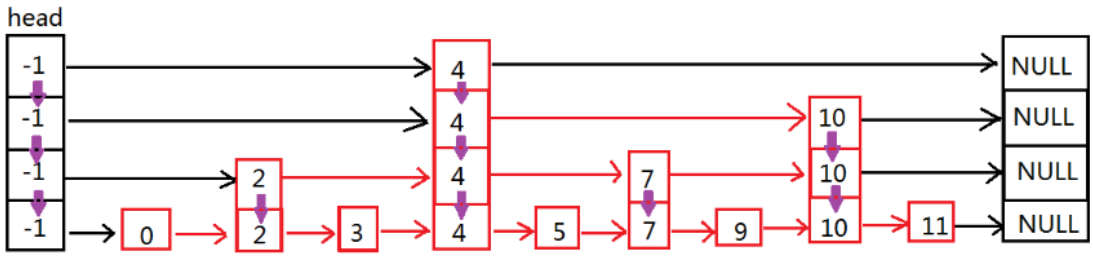
  虽然LSM tree的顺序写入保证了写性能，但是其本身的存储结构却牺牲了读性能，所以需要通过compaction这样的机制随着IO的持续写入来不断调整数据存储系统的结构，来降低对读的影响。

  Rocksdb有多种compaction策略：

  - size-tiered compaction：每层允许的SST文件最大数量都有个相同的阈值，随着memtable不断flush成SST，某层的SST数达到阈值时，就把该层所有**SST**全部合并成一个大的新**SST**，并放到较高一层去。
  - leveled compaction：对于L1层及以上的数据，将size-tiered compaction中原本的大SST拆开，成为多个key互不相交的小SST的序列（L0层是从memtable flush过来的新SST，该层各个SST的key是可以相交的），并且其数量阈值单独控制。可见，leveled compaction与size-tiered compaction相比，每次做compaction时不必再选取一层内所有的数据，并且每层中SST的key区间都是不相交的，重复key减少了。

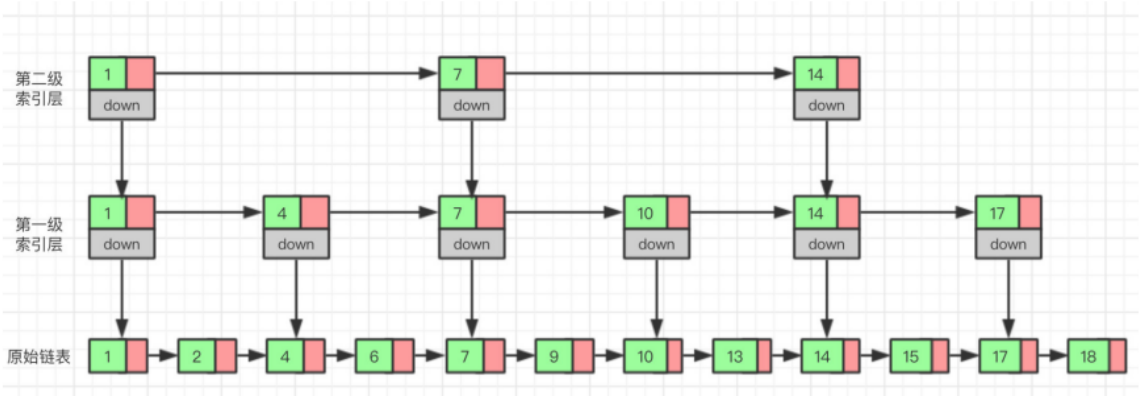
5. 跳表（**MemTable**）：

跳表（Skiplist）是一个特殊的链表，相比一般的链表，有更高的查找效率。平均期望的查找、插入、删除时间复杂度都是**O(logn)**。Redis中的有序集合zset；LevelDB、RocksDB、HBase中Memtable都采用跳表实现。



6. 演化思路：

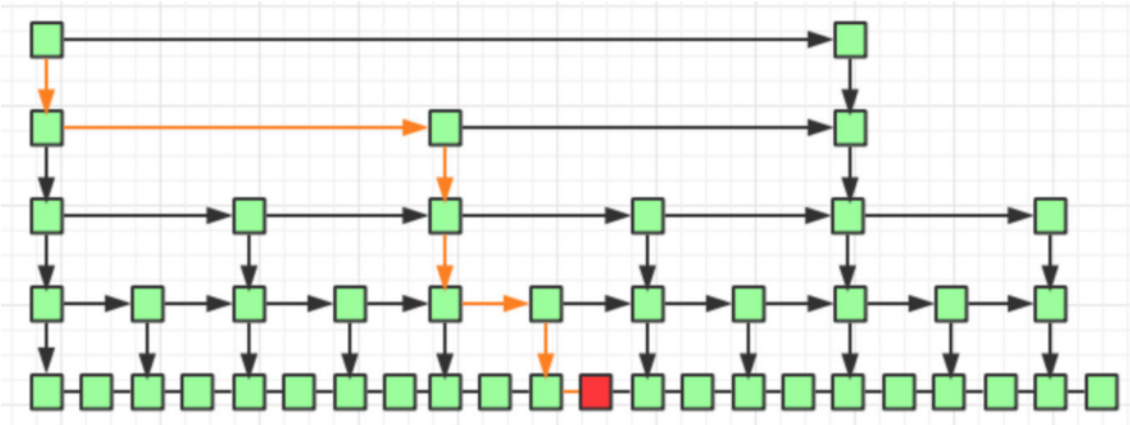
对于单链表来说，即使数据是已经排好序的，想要查询其中的一个数据，只能从头开始遍历链表，这样效率很低，时间复杂度很高，是 O(n)。为了提高查询的效率，可以为链表建立一个“索引”。（查找15的过程：1->7->14->14->14->15）



每两个节点建立一个索引，跳表高度：log(n)。若在查询跳表的时，每一层都需要遍历 k 个结点，最终的时间复杂度为 O(k*log(n))。

可以得到k=2，原因如下：

最高一级索引只有两个结点，每下一层索引在上一层索引两个结点之间增加了一个结点，也就是上一层索引两结点的中值。因此在搜索过程中，搜索指针pointer从顶层头结点开始，进行类似于二分查找的判断，并确定新的头结点的索引列。这样可以得到，在每一行最多遍历2个节点。



可以看出，跳表的效率比链表高了，但是跳表需要额外存储多级索引，所以需要的更多的内存空间（空间换时间）。

7. 插入、删除

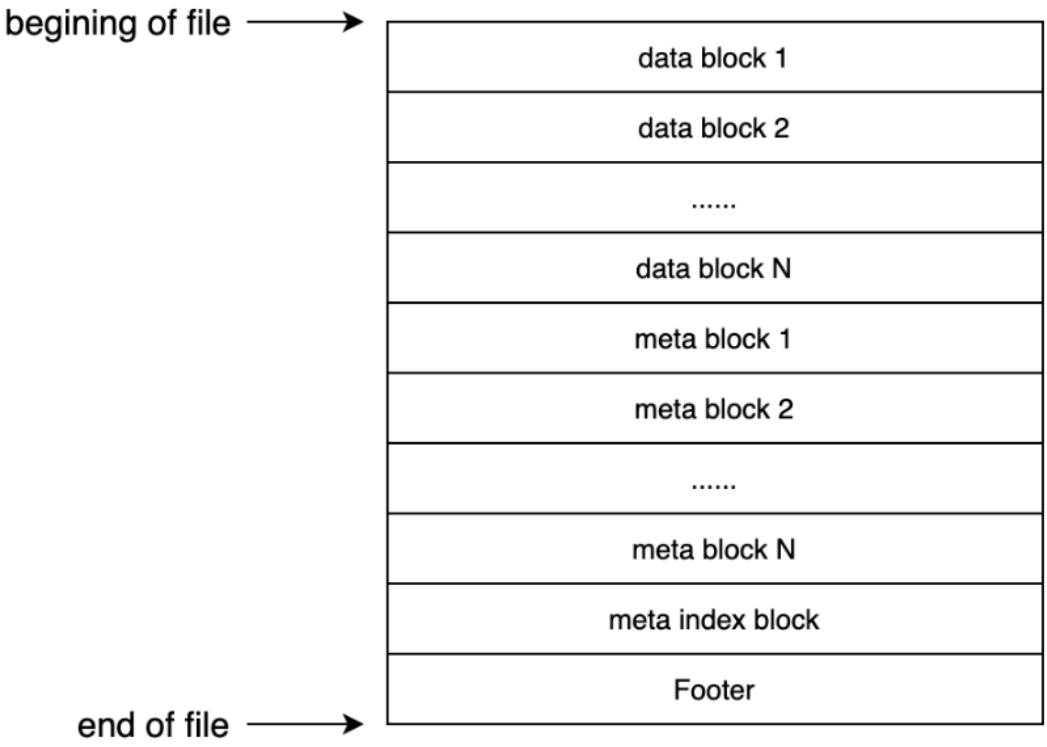
跳表的查询的时间复杂度为 O(log(n)），因为找到位置之后插入和删除的时间复杂度很低，为 O(1)，所以最终插入和删除的时间复杂度也为 O(log(n))。

一些注意的点：删除时，可能需要将索引中的节点一并删除；不停插入之后，可能造成索引之间的结点过多，因此需要维护。

8. **SST**

LSM tree保证了数据是有序写入 (memtable – skiplist)，提高了写性能，但是因为其本身的分层结构，牺牲了读性能（一个key若存储在了低级别的level，从上到下每一层都要进行查找，代价极大）。所以，针对读的性能提升有了很多的优化：bloom filter(高效判断一个key是否不存在)，index-filter （二分查找，消耗低内存的情况下）索引key-value数据。这一些数据都需要存储在SST文件之中，用来进行k-v数据的有序管理。

SST结构可以抽象为下图：



- Footer：主要是用来索引 meta index block 和 index block
- meta index block：主要是为了索引列出的多个meta block
- index block：是属于一种meta block，它是用来索引data block
- metablock：index block、filter block、range_del block，compression block，properties block

    i. filter block：用来保存一些bloom filter用来加速查找；
    ii. range_del block是保存客户端针对key有DeleteRange的操作而标记的一批key；
    iii. compression block保存了通过字典压缩的key的前缀数据，也是为了加速读；
    iv. properties block保存了当前SST文件内部的属性数据，像有多少个datablock，多少个index block，整个SST文件有多大等各维度的数据。

9. **WAL**

WAL主要作用是用来恢复节点断电，死机时 memtable中的未commited中的数据。 所以WAL 的写入需要优先于memtable，且每一次写入都需要flush ，这也是write head的由来。

2. 阅读文献:Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung. The Google file system. In Proceedings of the nineteenth ACM symposium on Operating systems principles October 2003.理解并介绍GFS的数据一致性策略。

GFS的一致性模型

概念定义

1. 客户端读取不同的 Replica 时可能会读取到不同的内容，那这部分文件是**inconsistent**的

2. 所有客户端无论读取哪个 Replica 都会读取到相同的内容，那这部分文件就是**consistent**

3. 所有客户端都能看到上一次修改的所有完整内容，且这部分文件是一致的，那么我们说这部分文件是**defined**

保证：

1. 文件命名空间的修改（例如，文件创建）是原子性的。

2. 文件数据修改：

    - 如果一次写入操作成功且没有与其他并发的写入操作发生重叠，那这部分的文件是defined
    - 如果有若干个写入操作并发地执行成功，那么这部分文件会是consistent但会是undefined。在这种情况下，客户端所能看到的数据通常不能直接体现出其中的任何一次修改
    - 失败的写入操作会让文件进入inconsistent的状态

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Table 1: File Region State After Mutation

3. 随机写与追加写

- 随机写

  当写入时，指定的数据会被直接写入到客户端指定的偏移位置中，覆盖原有的数据。GFS 并未为该操作提供太多的一致性保证：如果不同的客户端并发地写入同一块文件区域，操作完成后这块区域的数据可能由各次写入的数据碎片所组成，即进入不确定的状态。

- 追加写

  所有的追加写入都会成功，但是有可能被执行了多次，而且每次追加的文件偏移量由GFS自己计算。

  GFS返回给客户端一个偏移量，表示了包含了写入记录的、已定义的region的起点。

4. **Leases**

每个mutation都要在一个chunk的所有replica上执行成功才能应答客户端，因此，从所有replica中选择一个作为**primary**，负责给mutation定序，其他replica按相同顺序执行mutation。

primary由master选择，然后给primary授予lease（初始为60秒）。**primary**定期请求**master**延长**lease**（heartbeat请求中携带），一旦lease过期，master可以认为primary故障，选其他replica作为新的primary。

5. **GFS**支持一个宽松的一致性模型

首先，文件命名空间的修改，如文件创建是原子性的，仅由master控制：命名空间锁提供了原子性和正确性、master操作日志定义了这些操作在全局的顺序。

对于文件的数据修改，文件状态会进入以下三种状态之一：

- 一致的（consistent）：对于一个chunk，所有client看到的副本内容都是一样的
- 不一致的**(Inconsistent)**：客户端读取不同的 Replica 时可能会读取到不同的内容，那这部分文件是不一致的
- 定义的 **(defined)**：数据修改后是一致的，且 **client** 可以看到写入操作的全部内容（ 换句话说，可以看到每步操作修改后的内容 ）

一个文件的当前状态将取决于此次修改以及修改是否成功

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

- 当一个数据写操作成功执行，且没有并发写入，那么影响的 **region** 就是 **defined**：所有 **client** 都能看到写入的内容。（ 隐含了 **consistent** ）
- 当并行修改写完成之后，**region** 处于 **consistent but undefined** 状态：所有 **client** 看到同样的数据，但是无法读到任何一次写入操作写入的数据 （ 因为可能有并行写操作覆盖了同一区域 ）。
- 失败的写操作导致 **region** 处于 **inconsistent** 状态（ 同时也是 **undifined** 的 ）：不同 **client** 在不同时间会看到不同的数据。
- 当对文件进行追加操作，若追加操作成功，那么 **region** 处于 **defined and consistent** 状态；若某次追加操作失败，**client** 重新请求后会导致数据填充和重复数据的情况，此时 **region** 处于 **defined but inconsistent** 状态。