1、复习/学习并介绍LSM（ Log-Structured Merge Tree ）树数据结构

（1）（10分）当数据从内存flush到外存时，为什么采用append方式？

（2）（10分）LSM树结构中compaction操作的目的是什么？

（3）（10分）应用于column-family数据库时，Buffer中的数据如何组织？

2、解释Bloom Filter的原理。

①原理：

②**Redis**中的布隆过滤器：

（**1**）应用场景：

具体例子：

（**2**）工作原理

1) 工作流程-添加元素

2) 工作流程-判定元素是否存在

3) 为什么是可能“存在”

（**3**）基本命令：

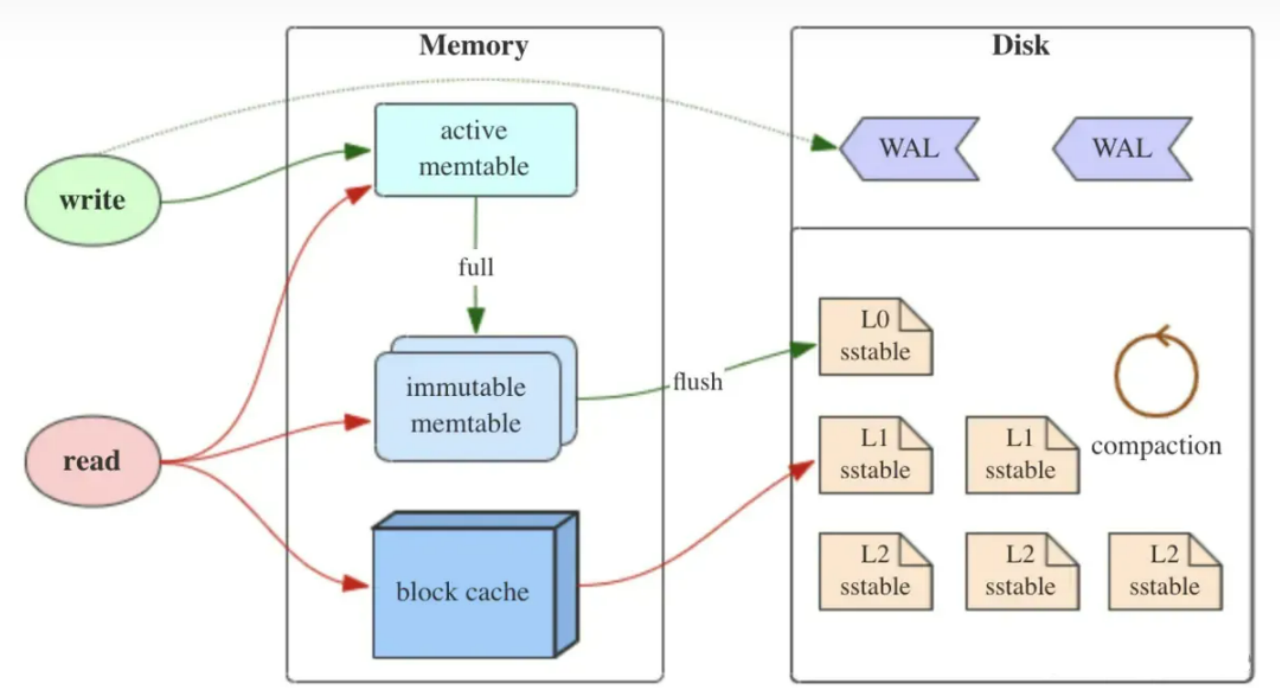
③布隆过滤器的缺点

④布隆过滤器解决的问题：

1、复习/学习并介绍**LSM**（ **Log-Structured Merge Tree** ）树数据结构

Log-Structured的思想最早由 Rosenblum和Ousterhout于1992年在研究日志结构的文件系统时提出。他们将整个磁盘看做是一个日志，在日志中存放永久性数据及其索引，每次都添加到日志的末尾；通过将很多小文件的存取转换为连续的大批量传输，使得对于文件系统的大多数存取都是顺序性的，从而提高磁盘带宽利用率，故障恢复速度快。 O'Neil等人受到这种思想的启发，借鉴了Log不断追加（而不是修改）的特点，结合B-tree的数据结构，提出了一种延迟更新，批量写入硬盘的数据结构LSM-tree及其算法。LSM-tree努力地在读和写两方面寻找一个平衡点以最小化系统的存取性能的开销，特别适用于会产生大量插入操作的应用环境。

LSM树的核心特点是利用顺序写来提高写性能，但因为分层(此处分层是指的分为内存和文件两部分)的设计会稍微降低读性能，但是通过牺牲小部分读性能换来高性能写，使得LSM树成为非常流行的存储结构。



如上图所示，LSM树有以下三个重要组成部分：

- **MemTable**

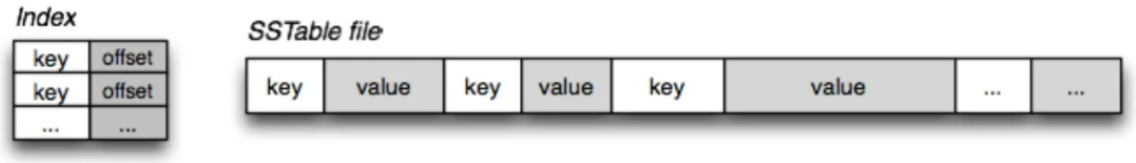
  MemTable是在内存中的数据结构，用于保存最近更新的数据，会按照**Key**有序地组织这些数据，LSM树对于具体如何组织有序地组织数据并没有明确的数据结构定义，例如Hbase使跳跃表来保证内存中key的有序。

- **Immutable MemTable**

  当 MemTable达到一定大小后，会转化成Immutable MemTable。Immutable MemTable是将转MemTable变为SSTable的一种中间状态。写操作由新的MemTable处理，在转存过程中不阻塞数据更新操作。

- **SSTable(Sorted String Table)**

  有序键值对集合，是LSM树组在磁盘中的数据结构。为了加快SSTable的读取，可以通过建立**key**的索引以及布隆过滤器来加快key的查找。



LSM树会将所有的数据插入、修改、删除等操作记录**(**注意是操作记录**)**保存在内存之中，当此类操作达到一定的数据量后，再批量地顺序写入到磁盘当中。这与B+树不同，B+树数据的更新会直接在原数据所在处修改对应的值，但是LSM数的数据更新是日志式的，当一条数据更新是直接**append**一条更新记录完成的。这样设计的目的就是为了顺序写，不断地将Immutable MemTable flush到持久化存储即可，而不用去修改之前的SSTable中的key，保证了顺序写。

（1）（10分）当数据从内存**flush**到外存时，为什么采用**append**方式？

[ans]

LSM的核心思想就是放弃部分磁盘读性能来换取写的顺序性，因此LSM将写入推迟(Defer)并转换为批量(Batch)写，并日志式地将每一次写入都直接顺序写入到磁盘中。

这样设计的目的就是为了顺序写，不断地将Immutable MemTable flush到持久化存储即可，而不用去修改之前的SSTable中的key，保证了顺序写。

总而言之，LSM为了提高写性能，必然采取append方式。

（2）（10分）**LSM**树结构中**compaction**操作的目的是什么？

[ans]

log append的方式带来了高吞吐的写，内存中的数据到达上限后不断刷盘，数据范围互相交叠的层越来越多，相同**key**的数据不断积累，引起读性能下降和空间膨胀。因此，**compaction**机制被引入，通过周期性的后台任务不断的回收旧版本数据和将多层合并为一层的方式来优化读性能和空间问题，其主要作用是数据的gc和归并排序，是ISM-tree系统正常运转必须要做的操作，尽管其任务运行期间会带来很大的开销。

由于LSM不断地将内存中数据flush到持久化存储，而不去修改磁盘之前的数据中的key，这样虽然保证了顺序写，但同时也会带来一些问题：

- 冗余存储，对于某个key，实际上除了最新的那条记录外，其他的记录都是冗余无用的，但是仍然占用了存储空间。因此需要进行**Compact**操作**(**合并多个**Table)**来清除冗余的记录。
- 读取时需要从最新的倒着查询，直到找到某个key的记录。最坏情况需要查询完所有的Table，可以通过前面提到的索引/布隆过滤器来优化查找速度。

因此为了防止磁盘中数据的不断膨胀，compaction操作是必要的。

Compact的两种基本操作：

- size-tiered策略
- leveled策略

（3）（10分）应用于**column-family**数据库时，**Buffer**中的数据如何组织？

[ans]

以HBase为例，使用row key做分片，并保证其全局有序。每个row key下可以有多个column family。每个column family下可以有多个column。

ROW	COLUMNS
Row1	{ID,Name,Phone}
Row2	{ID,Name,Address,Title}
Row3	{ID,Address,Email}

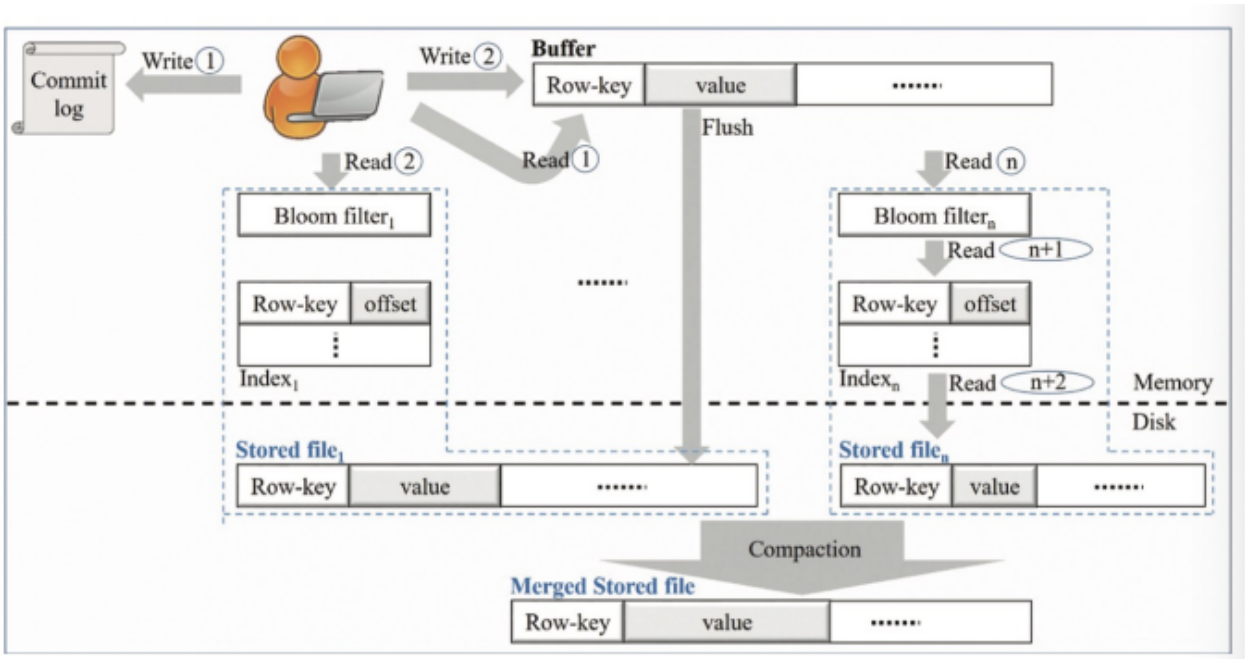


Fig. 3. The use of a simplified LSM-tree architecture per column-family in wide-column stores.

2、解释**Bloom Filter**的原理。

布隆过滤器（Bloom Filter）可以用来优化查找速度，是一种多哈希函数映射的快速查找算法，通常应用在一些需要快速检测一个元素是否在一个集合中，但是并不严格要求100%正确的场合。

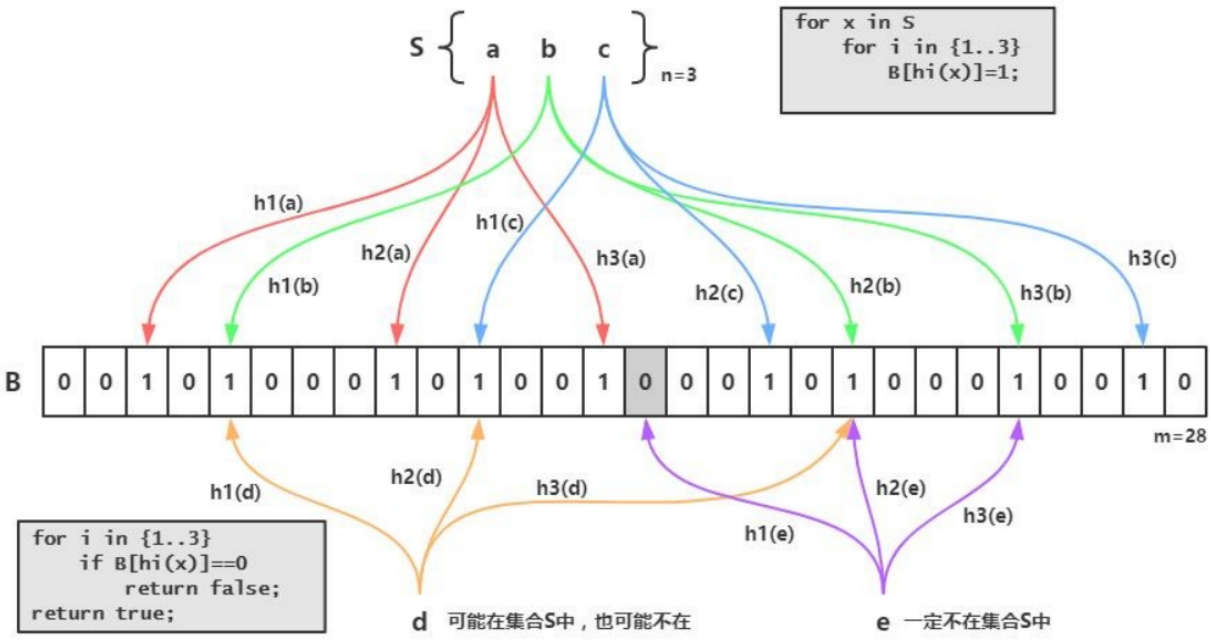
布隆过滤器（Bloom Filter）是一个高空间利用率的概率性数据结构，由二进制向量（即位数组）和一系列随机映射函数（即哈希函数）两部分组成。

布隆过滤器使用`exists()`来判断某个元素是否存在于自身结构中。当布隆过滤器判定某个值存在时，其实这个值只是有可能存在；当它说某个值不存在时，那这个值肯定不存在，这个误判概率大约在 1% 左右。

①原理：

- 创建一个m位的BitSet位数组，并将所有位初始化为0；
- 选择k个不同的哈希函数/散列函数，当一个元素被加入集合时，通过K个散列函数将这个元素映射成位数组中的K个点，把它们置为1。
- 在判断y是否属于这个集合时，我们对y应用k次哈希函数，如果所有hi(y)的位置都是1（1≤i≤k），那么我们就认为y是集合中的元素，否则就认为y不是集合中的元素。

Bloom Filter跟单哈希函数Bit-Map不同之处在于：Bloom Filter使用了k个哈希函数，每个字符串跟k个bit对应。从而降低了冲突的概率。



②Redis中的布隆过滤器：

（1）应用场景：

布隆过滤器是 Redis 的高级功能，虽然这种结构的去重率并不完全精确，但和其他结构一样都有特定的应用场景，比如当处理海量数据时，就可以使用布隆过滤器实现去重。

具体例子：

1) 百度爬虫系统每天会面临海量的 URL 数据，我们希望它每次只爬取最新的页面，而对于没有更新过的页面则不爬取，因策爬虫系统必须对已经抓取过的 URL 去重，否则会严重影响执行效率。但是如果使用一个 set（集合）去装载这些 URL 地址，那么将造成资源空间的严重浪费。

2) 垃圾邮件过滤功能也采用了布隆过滤器。虽然在过滤的过程中，布隆过滤器会存在一定的误判，但比较于牺牲宝贵的性能和空间来说，这一点误判是微不足道的。

（2）工作原理

1) 工作流程-添加元素

布隆过滤器主要由位数组和一系列 **hash** 函数构成，其中位数组的初始状态都为 0。如下图：

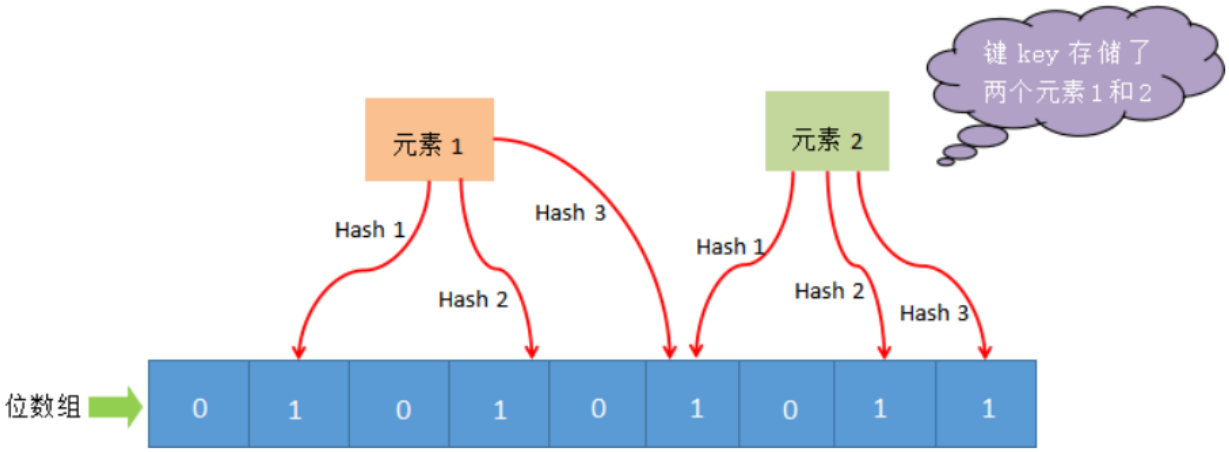


图1：布隆过滤器原理

当使用布隆过滤器添加 key 时，会使用不同的 hash 函数对 key 存储的元素值进行哈希计算，从而会得到多个哈希值。根据哈希值计算出一个整数索引值，将该索引值与位数组长度做取余运算，最终得到一个位数组位置，并将该位置的值变为 1。每个 hash 函数都会计算出一个不同的位置，然后把数组中与之对应的位置变为 1。通过上述过程就完成了元素添加(add)操作。

2) 工作流程-判定元素是否存在

当我们需要判断一个元素是否存时，其流程如下：首先对给定元素再次执行哈希计算，得到与添加元素时相同的位数组位置，判断所得位置是否都为 1，如果其中有一个为 0，那么说明元素不存在，若都为 1，则说明元素有可能存在。

3) 为什么是可能“存在”

为什么是有可能存在？因为那些被置为 1 的位置也可能是由于其他元素的操作而改变的。比如，元素1和元素2，这两个元素同时将一个位置变为了 1（图1所示）。在这种情况下，我们就不能判定“元素 1”一定存在，这是布隆过滤器存在误判的根本原因。

（3）基本命令：

命令	说明
bf.add	只能添加元素到布隆过滤器。
bf.exists	判断某个元素是否在于布隆过滤器中。
bf.madd	同时添加多个元素到布隆过滤器。
bf.mexists	同时判断多个元素是否存在于布隆过滤器中。
bf.reserve	以自定义的方式设置布隆过滤器参数值，共有 3 个参数分别是 key、error_rate(错误率)、initial_size(初始大小)。

其中：

- **bf.add**：添加元素到布隆过滤器中，bf.add命令只能一次添加一个元素，如果想一次添加多个元素，可以使用bf.madd命令。
- **bf.exists**：判断某个元素是否在过滤器中，类似于集合的sismember命令，不过bf.exists命令只能一次查询一个元素，如果想一次查询多个元素，可以使用bf.mexists命令。

③布隆过滤器的缺点

bloom filter之所以能做到在时间和空间上的效率比较高，是因为牺牲了判断的准确率、删除的便利性。

- 存在误判。可能要查到的元素并没有在容器中，但是hash之后得到的k个位置上值都是1。如果bloom filter中存储的是黑名单，那么可以通过建立一个白名单来存储可能会误判的元素。

- 删除困难。一个放入容器的元素映射到bit数组的k个位置上是1，删除的时候不能简单的直接置为0，可能会影响其他元素的判断。

④布隆过滤器解决的问题：

1. 解决缓存穿透的问题

一般查询情况下，都先查询缓存是否有该条数据，若缓存中不存在再查询数据库。当数据库也不存在该条数据时，每次查询都要访问数据库，这就是缓存穿透。缓存穿透带来的问题是，当有大量请求查询数据库不存在的数据时，会给数据库带来大量压力，这种是通过数据库写回缓存机制无法解决的。

可以使用布隆过滤器解决缓存穿透的问题，把已存在数据的key存在布隆过滤器中。当有新的请求时，先到布隆过滤器中查询是否存在，如果不存在该条数据直接返回；如果存在该条数据，再查询缓存查询数据库。

2. 黑名单校验（或其他能容忍误差的过滤情况）

可以将邮箱的所有黑名单都放在布隆过滤器中，再收到邮件时，判断邮件地址是否在布隆过滤器中即可。