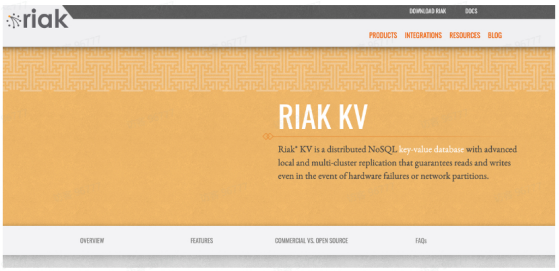
02 从零实现 KV 存储—bitcask 论文详解

论文地址：https://riak.com/assets/bitcask-intro.pdf

bitcask 存储模型最初是由一个做分布式存储系统的商业化公司 Riak 提出来的。

Riak 有很多产品，其中就包括一个分布式 KV 存储系统 Riak KV，他们的分布式 KV 产品具有可插拔的存储引擎，可以独立于整个系统，单独开发和测试新的存储引擎。



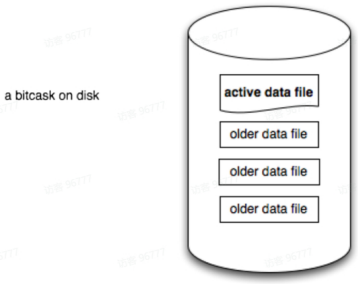
基于此，他们想打造一个全新的存储引擎，在最理想的情况下，满足下面的这些条件：

- 读写低延迟
- 高吞吐，特别是对大量的随机写入
- 能够处理超过内存容量的数据
- 崩溃恢复友好，能够保证快速恢复，尽量不丢数据
- 简单的备份和恢复策略
- 相对简单、易懂的代码结构和数据存储格式
- 在大数据量下，性能有保障
- 能够有自由的授权使用在 Riak 的系统中

现有的存储引擎，没有一个能够很好的满足这些条件，于是 Riak 团队重新设计了一个简洁高效的存储引擎 bitcask。

一个 bitcask 实例就是系统上的一个目录，并且限制同一时刻只能有一个进程打开这个目录。目录中有多个文件，同一时刻只有一个活跃的文件用于写入新的数据。

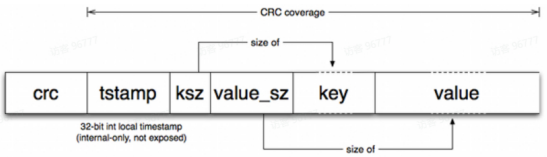
当活跃文件写到满足一个阈值之后，就会被关闭，成为旧的数据文件，并且打开一个新的文件用于写入。
所以这个目录中就是一个活跃文件和多个旧的数据文件的集合。



当前活跃文件的写入是追加的（append only），这意味着可以利用顺序 IO，不会有多余的磁盘寻址，减少了磁盘寻道时间，最大限度保证了吞吐。

写入到文件的数据，具有固定的格式，大致有这些字段：

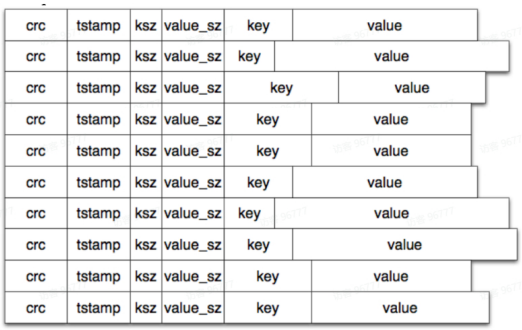
- crc：数据校验，防止数据被破坏、篡改等
- timestamp：写入数据的时间戳
- ksz：key size，key 的大小
- value_sz：value size，value 的大小
- key：用户实际存储的 key
- value：用户实际存储的 value



每次写入都是追加写到活跃文件当中，删除操作实际上也是一次追加写入，只不过写入的是一个特殊的墓碑值，用于标记一条记录的删除，也就是说不会实际去原地删除某条数据。

当下次 merge 的时候，才会将这种无效的数据清理掉。

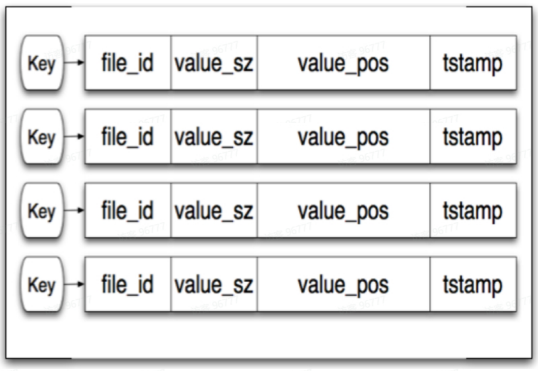
所以一个文件中的数据，实际上就是多个相同格式的数据集合的排列：



在追加写入磁盘文件完成后，然后更新内存中的数据结构，叫做 keydir，实际上就是全部 key 的一个集合，存储的是 key 到一条磁盘文件数据的位置。

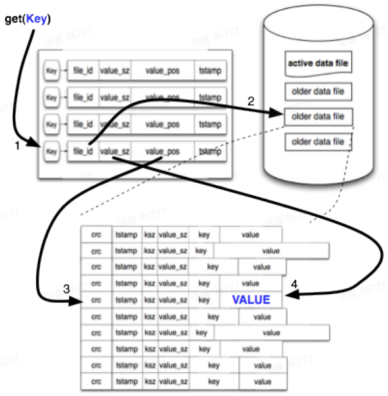
这里论文中说的是使用一个哈希表来存储，实际上这里的选择比较灵活，选用任意内存中的数据结构都是可以的，可以根据自己的需求来设计。

例如哈希表，可以更高效的获取数据，但是无法遍历数据，如果想要数据有序遍历，可以选择 B 树、跳表等天然支持排序的数据结构。



keydir 一定会存储一条数据在磁盘中最新的位置，旧的数据仍然存在，等待 merge 的时候被清理。

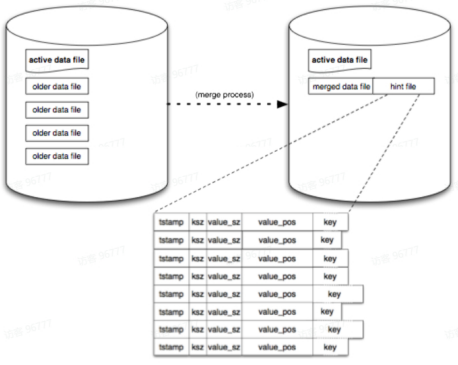
所以读取数据就会变得很简单，首先根据 key 从内存中找到对应的记录，这个记录存储的是数据在磁盘中的位置，然后根据这个位置，找到磁盘上对应的数据文件，以及文件中的具体偏移，这样就能够获取到完整的数据了。



由于旧的数据实际上一直存在于磁盘文件中，因为我们并没有将旧的数据删掉，而是新追加了一条标识其被删除的记录。

所以随着 bitcask 存储的数据越来越多，旧的数据也可能会越来越多。论文中提出了一个 merge 的过程来清理所有无效的数据。

merge 会遍历所有不可变的旧数据文件，将所有有效的数据重新写到新的数据文件中，并且将旧的数据文件删除掉。



merge 完成后，还会为每个数据文件生成一个 hint 文件，hint 文件可以看做是全部数据的索引，它和数据文件唯一的区别是，它不会存储实际的 value。

它的作用是在 bitcask 启动的时候，直接加载 hint 文件中的数据，快速构建索引，而不用去全部重新加载数据文

件，换句话说，就是在启动的时候加载更少的数据，因为 hint 文件不存储 value，它的容量会比数据文件小。

好了，bitcask 总体的设计完成了，我们再回过头来看看，bitcask 是否满足了设计之初的那些要点：

- 首先，bitcask 很快，查询和写入都很快，因为读写都只有一次磁盘 IO
- 写入数据还是顺序 IO，保证了高吞吐
- 内存中不会存储实际的 value，因此在 value 较大的情况下，能够处理超过内存容量的数据
- 提交日志和数据文件实际上就是同一个文件，数据的崩溃恢复能够得到保证
- 备份和恢复非常简单，只需要拷贝整个数据目录即可
- 设计简洁，数据文件格式易懂、易管理

总体来说，bitcask 基本满足了设计的要求，是一个简洁优雅、高效的存储引擎。

最后再来看看 bitcask 的一些面向用户的 API 操作接口，这可以帮助我们在实现的时候提供一些参考。

```
// 打开一个 bitcask 数据库实例，使用传入的目录路径
// 需要保证进程对该目录具有可读可写权限
bitcask::Open(Directory Name);

// 通过 Key 获取存储的 value
bitcask::Get(Key);

// 存储 key 和 value
bitcask::Put(Key, Value);

// 删除一个 key
bitcask::Delete(Key);

// 获取全部的 key
bitcask::list_keys();

// 遍历所有的数据，执行函数 Fun
bitcask::Fold(Fun);

// 执行 merge，清理无效数据
bitcask::Merge(Directory Name);

// 刷盘，将所有缓冲区的写入持久化到磁盘中
bitcask::Sync();

// 关闭数据库
bitcask::Close();
```

我们课程当中的 KV 存储引擎，将基本按照 bitcask 论文的描述来进行设计，并且在很多细节上有更加精细的处理，因为论文当中提及的一些概念并没有涉及到细节。我们还会在基础的功能之上，对 bitcask 存储引擎进行一些优化。