abbshr commented on Oct 1, 2016 Owner

今年年初做毕设那段时间, 曾涉及一些关于数据存储方面的技术, 着手实现时也发现了不少乐趣, 这里就梳理一下思路, 做个总结, 也尽量让人读完之后有个全面的了解, 感谢工业界和学术界产出的详实资料文献和笔录经验, 给我学习过程提供了莫大便利! 其中过于底层的技术的原理我可能没法做到了然于胸, 当然我会持续深入学习那些不甚了解/拿不准的东西, 不断完善知识技能树.

几个术语

来区分这两个专有名词:

- 数据库
- 存储引擎

数据库往往是一个比较丰富完整的系统, 目的是大而全, 提供了丰富的查询和一系列复杂的数据操作逻辑, 或者是在网络层面,水平扩展等支持.

然而一个存储引擎目标则是小而精, 因为它是纯粹专注于**读/写/存储**, 一般来说, 直接使用存储引擎的是像数据这样的上层存储系统.

那么来思考几个问题

如果是你, 你要如何:

- 设计一个 k-v 存储引擎?
- 支持随机读写?
- 高效的数据操作?
- 持久化数据?
- 错误容忍?

ok, 我知道你也不是十分了解~ 我也非专业, 那么我们就看看人家是怎么考虑这些问题的~~.

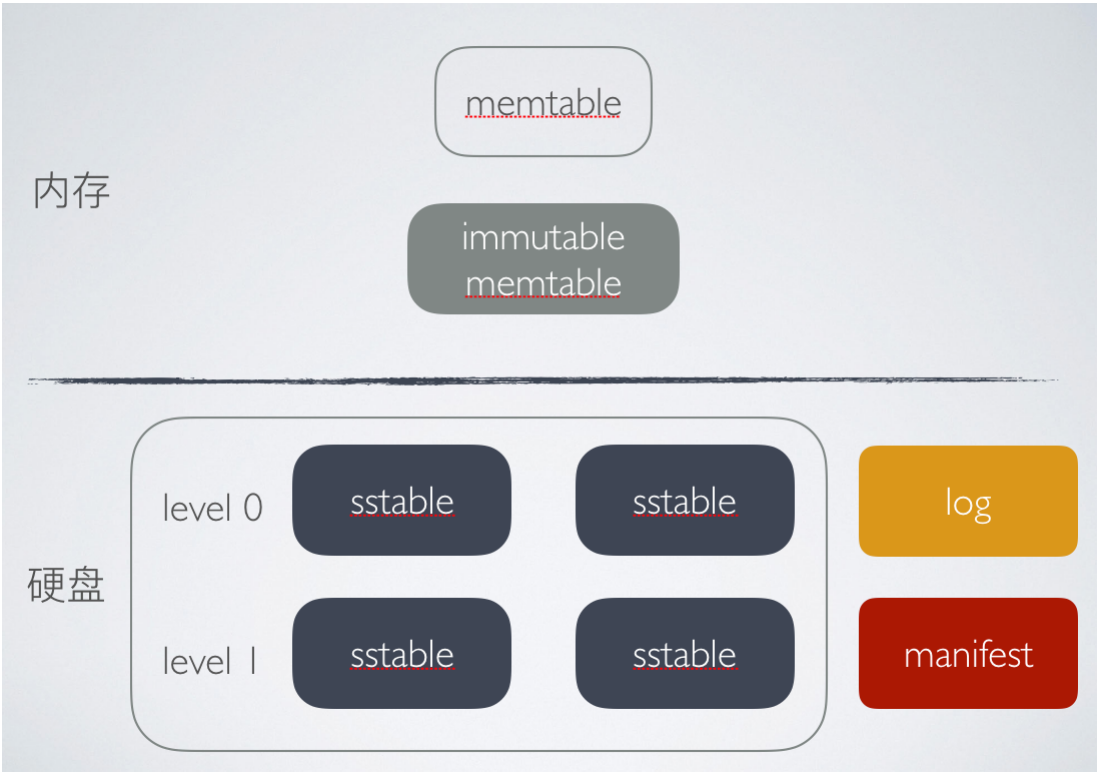
从 leveldb 讲起

为啥说 leveldb? 因为我觉得它是个不错的学习例子, 代码和官方文档都详细说明了一个简单的 k-v 引擎在设计上应该考虑哪些, 注意什么.

没听过 leveldb 也不要紧, 几句话介绍一下: google 亲儿子, BigTable 核心技术, Chrome IndexedDB 的内核. 具备极高的写入效率, 同时有不俗的读性能.

结构规划

了解一个系统的普遍方法就是先了解其大体轮廓和工作方法. 为了了解 leveldb 是如何工作的, 我们首先来看它的结构规划是怎样的.



根据此图先初步说一下它的读写流程:

- write: 写入 log 文件一条操作(数据更新)记录, 再往 memtable 里写入 k-v 对.
- read: 读 memtable (没找到)→ 读 immutable memtable (没找到)→ 读 sstable (没找到)→ 空

没错, 每次操作就这么简单的流程. 接下来就来谈谈这样设计的优劣原由以及各个组件所起的作用, 从而讲一下设计一个存储引擎的思路.

内幕

虽说一次读写就那么几步, 但是他们的背后究竟发生了什么? 换句话说, 既然 leveldb 作为一个具备高性能写操作的存储引擎, 是什么提供的性能优势? 来看下操作内幕.

memtable

memtable 是个内存数据结构, 也是读写操作的入口.
作为一个存储引擎, 读无非就是读若干个 key, 然后返回对应的 value; 而写就是存下这些 key-value 对.
所以 memtable 中存的每条数据也都是一个键值对.
关键点是 memtable 中的数据是按 key 的字母表顺序排序的.

然而对一个具备随机读能力的排序结构执行插入操作往往是有开销的, 通常写瓶颈也都是集中在这里. 但是也有很多数据结构提供了加速随机写, 比如链表, AVL 树, B 树, skiplist. memtable 的核心结构就是用了跳表.

结构简单, 概率上近似 O(log N) 读写复杂度的跳表在很多存储系统里得到应用, 比如 elasticsearch 的高速搜索就是基于跳表,位图以及倒排索引实现的. memtable 采用这种结构初步实现快速读写.

接下来, 既然是存储引擎, 那么肯定能持久化数据, 而 leveldb 核心技术就是围绕持久化过程而构建的

接下来, 既然是存储引擎, 那么肯定能持久化数据, 而 leveldb 核心技术就是围绕持久化过程而构建的.

Log Structed Merge Tree

我们在谈论 leveldb 的存储方案之前, 有必要先介绍一下另外一种数据结构: lsm tree.

lsm tree 是针对写入速度瓶颈问题而提出的. mysql 这种数据库的存储引擎使用了 B+ 树来持久化数据, B+ 树是一个索引树, 可以说是同时考虑了读写均衡, 其结构上对树高进行了优化, 搜索耗时相比 AVL 树降下来. 然而问题依然是前面我们谈到的 "对一个随机读优化的排序结构执行随机写是有很大开销的", 所以对那些需要高频写操作的系统来讲, B+ 树作为存储结构可能并不合适.

这时 lsm tree 可能是个更好的选择, 它是一种类似日志的数据结构, 将随机写变为顺序写, 核心思想是:

- 对变更进行批量 & 延时处理
- 通过归并排序将更新迁移到硬盘上

自从上世纪某年代一篇关于日志结构文件系统的论文发表之后, 有人基于这个想法提出了解决写瓶颈的问题: 使用顺序写(追加)替代随机写. 为什么要变为顺序写呢? 因为顺序写不需要多次寻址, 速度能达到硬盘理论传输速度, 而随机写则受限于硬盘寻址速度.

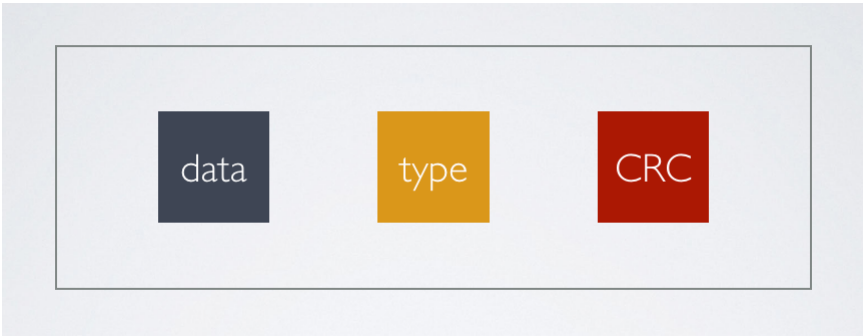
这种思想跟当今的 immutable 结构, 函数式中的不变量, 比特币区块链如出一辙, 因为跟日志很像, 所以称之为 "日志结构合并树", 原论文花了 30 来页从数据结构, 操作策略, 低层次磁盘优化等等方面阐述了 lsm tree. (我读 5 页就放弃了...)

leveldb 在持久化上借鉴了 lsm tree 的设计. 具体实现就是 memtable + sstable.

sstable

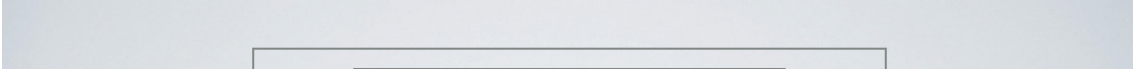
sstable 全名 sort-string table, bigtable 使用的存储技术. 顾名思义, sstable 中的数据都是有序的.
除了日志之外, leveldb 的数据统统存储在 sstable 中. 对于 sstable, 我们先来看下其布局.

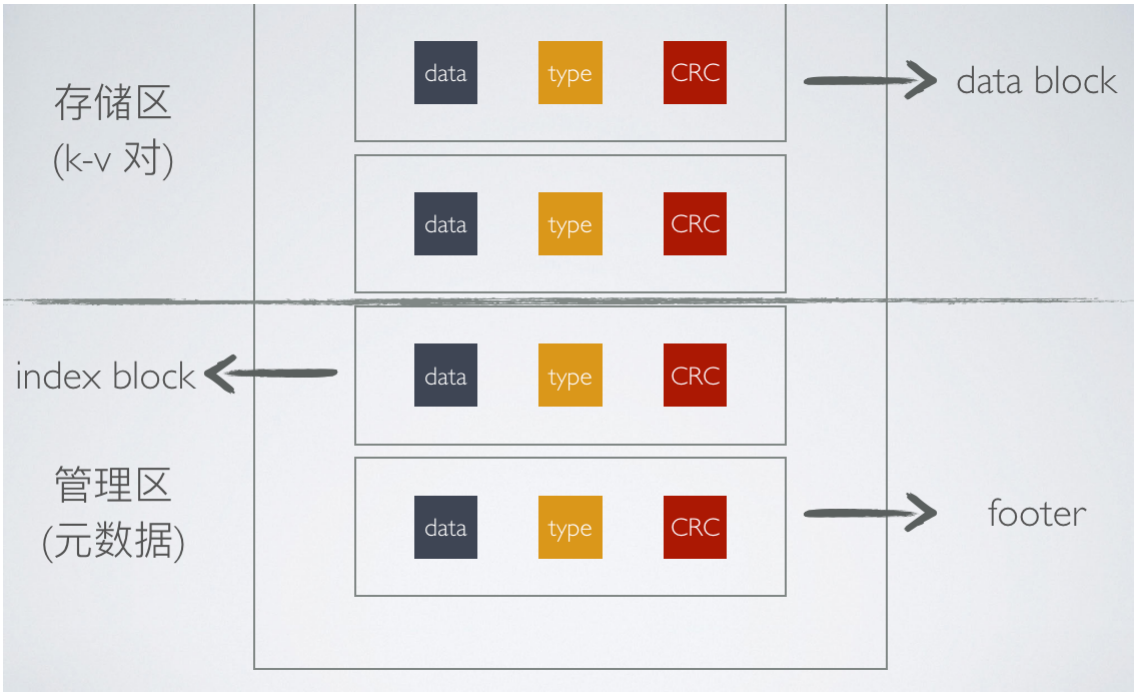
每个 sstable 内部按 block 划分. 物理布局如图.



type 用于表示 data 域采用何种压缩算法. (因为 leveldb 使用了 snappy 压缩算法)
crc 循环冗余编码用于数据的检错. 这个 IP 数据报头也有用到.

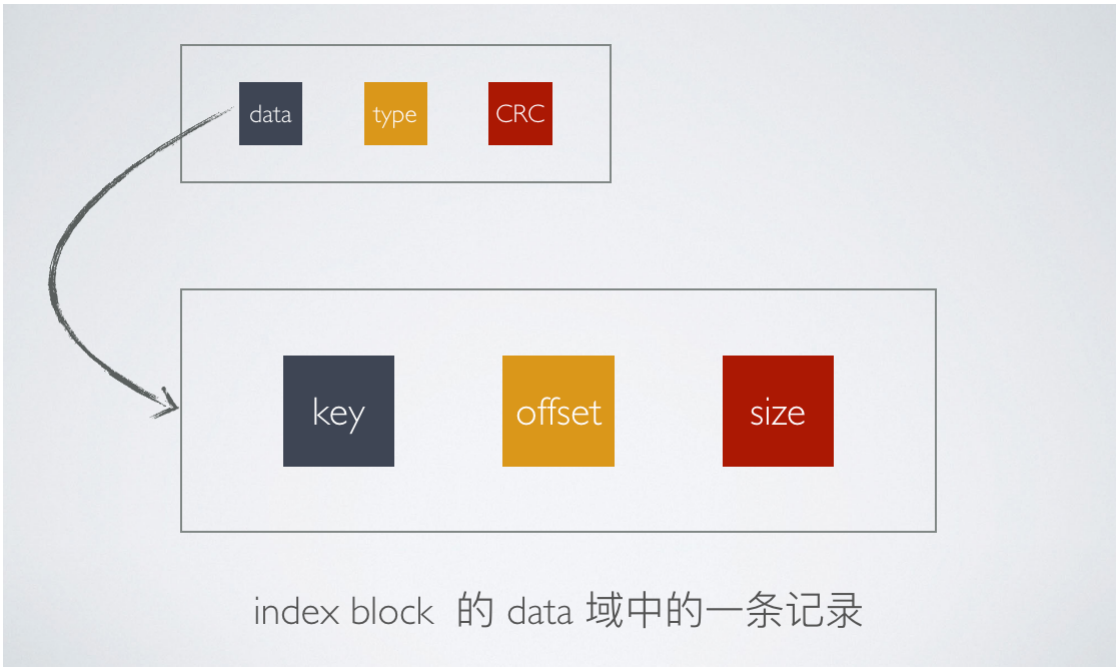
而在逻辑视图上, 将整个 sstable 看成是由 存储区 和 管理区 组成:





下面分别介绍不同种类的 block 作用.

index block



index block 中的每条记录会对某个 data block 建立索引.
key 保存大于等于前一个 data block 中最大 key 并且小于下一个 data block 中最小 key 的一个值.
offset 指明被索引的 data block 的起始位置在 sstable 中的偏移量.
size 记录 data block 大小.

footer

footer 是整个 sstable 的末尾 block, 记录了 indexblock 的起始偏移量和大小, 很容易看出其存在意义就是为

了方便读取 index block.

data block

对于重中之重 —— 数据块, 其 data 域存储的就是所有的 k-v.
更具体来说, 是 data 域的 record 段存储的. record 里的 k-v 对全部按 key 有序排列.

record 字段也并不是简单的存储了 k-v, 而是将 key 分解为前缀和后缀存储.
为何如此复杂设计? 原因很简单, 压缩存储, 尽量减少重复数据占用的空间, 思路类似 trie 树 (公共前缀树).

而上面的 restart 字段表示从某条记录开始重新存储新的前缀, 并且该条记录完整存储一个 key. (切换公共前缀)

sstable 跟 lsm tree 啥关系?

之前提到了 leveldb 借鉴 lsm-tree 实现了顺序写结构, 然后紧接着介绍 sstable, 那么 sstable 如何实现的 lsm-tree 这种思想? 接下来就不得不提 leveldb 中的操作策略了.

sstable(s) 是复数

看过第一幅图你应该发现了, sstable 不仅仅有一个, 而是一批.

leveldb 将 sstable 划分为了不同层次(level). level i+1 层的 sstables 由 level i 层的 sstables merge 得来. 而最上层称为 level 0.

那么第 0 层的 sstables 是怎么来的呢?

memtable + sstable = lsm tree

- memtable → immutable memtable (内存中)
- immutable memtable → (level 0) sstable (内存 → 外存)

还记得最开始提到的 memtable? 那个读写入口数据结构. leveldb 给了他一个阈值, 但凡 memtable 里的数据大小达到了阈值, 后台任务就将 memtable 标记为只读, 也就是变成了 immutable memtable.
然后创建一个新的 memtable 用于接下来的读写.

immutable memtable 经过一段时间会被迁移到硬盘上, 成为 sstable, 这一过程称之为 compaction.
因为 memtable 的结构是有序的, 因此 sstable 也是有序存储的.

compaction

compaction 是执行 lsm-tree 中 merge 的过程.
对于不同应用情况, 分为 minor compaction 和 major compaction.

minor compaction

minor compaction 用于内存到外存的迁移过程. 就是简单的遍历跳表, 依次写入新的 sstable record, 最后建立 index block 并完善一些其他的重要元信息. 这也就是从 immutable memtable 到 level 0 sstable 的迁移.

major compaction

major compaction 用于 level 之间的迁移.
当某个 level 的 sstables 数量超过一个给定的阈值, 就会触发 major compaction.
这里有两点差异:

- 对 level > 0 的 sstables, 选择其中一个 sstable 与 下一层 sstables 做合并.
- 对 level = 0 的 sstables, 在选择一个 sstable 后, 还需要找出所有与这个 sstable 有 key 范围重叠的 sstables, 最后统统与level 1 的 sstables 做合并.

不知之前是否注意到, level 0 的 sstables 可能有键范围的重合. (因为 level 0 的 sstable 是直接由 memtable 变过来的, 而不同的 memtable 之间并没有约束 key 必须独立.)

merge 策略

compaction 过程中有提到选择 sstable, 如何选择? 这就看 leveldb 的 merge 策略了:

- level i 按顺序选择.
- level i+1 选择所有与 level i 所选 sstable(s) 有 key 范围重叠的 sstables.
- 将这些 sstables 做 K 路归并排序. 对于相同的 key, 只保留最新的(上层 sstable, 同层中新的 sstable).
- 清除参与此次 merge 的所有 sstables, 保留新的 sstable.

有了前面提到的 compaction 第二个约束和 merge 策略, 间接解释了 level 0 和其他 level 不同的原因: 在 level 0 的 compaction后, level 1 产生的 sstable是没有 key范围重叠的, 因此向高层 level 的 compaction 也不会有同一个 level 下 key 范围重叠的 sstable 产生.

到这里, leveldb 如何实现的 lsm-tree 就比较明朗了: 内存结构中写数据, 每次写只考虑当前 memtable, 不涉及其他结构的更新, 内存数据顺序迁移到外存中, 形成一个日志结构, 这就是 leveldb 顺序写的思路.

关于 k 路归并

关于 k 路归并

就是我们上数据结构课学的那个外部排序. 这里有个简单的思路: 用最小堆实现.

- k 个有序的 sstable 取最小 key, 组成最小堆, 取定点key, 用其所在 sstable 的下一个 key 填充(当一个 sstable 所有的 key 都被取完, 则从堆的末尾取 key), 然后重新调整使其符合堆性质, 如此循环往复.
- 而所有取出的最小堆顶点依次组成一个列表, 就形成新的 sstable 了.

如何做到高效的随机写?

到此为止, leveldb 的写入策略就介绍完了, 这里做个总结:

1. 内存 + 跳表
2. 外存中追加, 使用顺序写

读取效率如何?

说完了写, 那么并不具备读优势的 leveldb 读取是怎样设计的呢?

毕竟凡事很难达到两全, 顺序写的这种结构就注定了丧失读效率(需要多次遍历寻址).
然而 leveldb 仍然费尽心思在代码层面而非算法层面做了优化.

查询过程中优化的顺序如下:

- 内存 + skiplist
- sstables B-search (通过 manifest 文件)
- 页缓存
- bloom filter

(周期性 compaction 做辅助)

在查询 level 0 sstables 时, 因为不同 sstables 的 key 范围可能有重叠, 所以只能把但凡包含 key 的 sstables 找出来, 排序取最新的 (manifest 文件记录了当前所有sstable 的信息, 包括文件名, 所处 level, key 的范围等等). 因为 sstable 是有序的, 所以在 level > 0 寻找 sstable 这一过程可以应用二分搜索完成. 找到 sstable 后, 读出其 index block, 加载到内存作为 table cache, 再根据索引去查具体的 data block. 而 data block 也包含了一组 k-v, 为了提高定位 key 的效率, leveldb 又引入了布隆过滤器这一数据结构.

bloom filter

这里引用一下数学之美中的介绍: 它是一个判断元素是否在集合中的高效做法, 常用于垃圾邮件过滤.

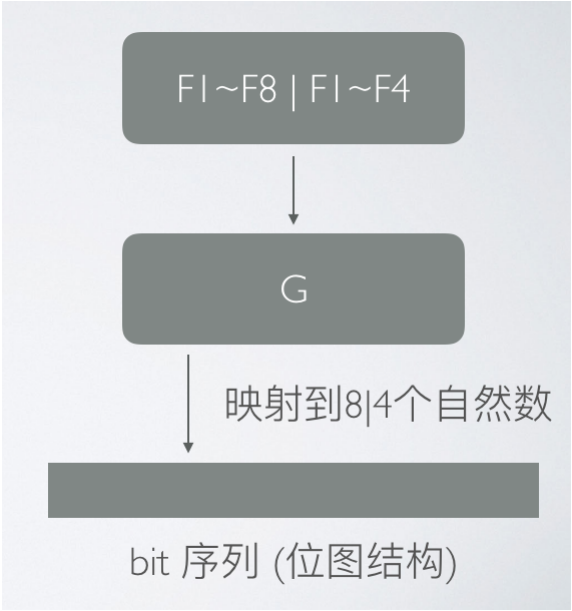
优势:

```
+ 快速
+ 节省空间(为散列表的 1/8 或者 1/4)
+ 不存储数据(安全性)
```

劣势:

+ 存在误判率（但是微乎其微，也可以用白名单方法解决）

原理简单说下:



f1~f8 和 G 为不同的随机数产生器, 把一个元素通过 f 计算分别得到 4 / 8 个值, 然后分别通过 G 映射到 4 / 8 个自然数, 这些数用位图的方法存储在一个 bit 序列中(置 1 的索引), 因此判断时只要检查这几个位置的值是否全为一即可.

有了 bloom filter, 在 block 的判定中就可以快速跳过不含有目标 key 的 block 了, 在一定程度上减少了磁盘的随机访问次数. leveldb 中 bloom filter 是针对每个 sstable block 而建立的.

如何节省存储空间?

做到了读写优化后, 节省空间也是要考虑的, 所幸基于这种设计架构, 已经做到了把空间压缩, 具体:

- 公共前缀 key
- sstables compaction
- snappy 压缩

综上

我们可以看到 leveldb 在存储引擎设计/实现过程中对各个层面做出的优化, 包括:

- 架构层面
- 算法层面
- 代码层面

- 代码层面

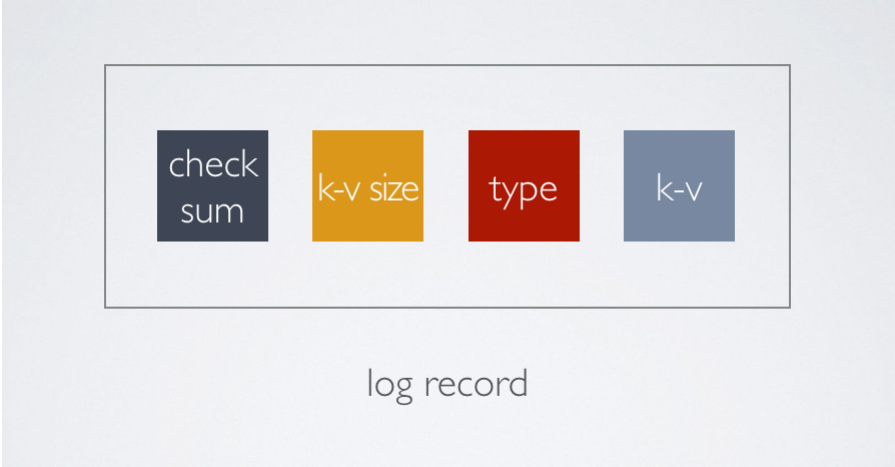
容错 & 数据恢复

对于一个存储引擎来说, 健壮性也是必不可少的, 进程意外退出或者机器故障很容易导致数据断层(丢失), 对此, leveldb 在每次写操作前都会在 log 文件里追加一条日志.

log 跟 sstable 很像, 因为 log 就是日志.
唯一的区别是 log 忠实按照操作顺序记录, 因此不考虑 key 是否有序.

日志文件

log 文件的单位是固定大小的 block. 每个 block 中包含了一系列记录.



其中 type 有以下四种值:

- full: 该记录完整存储在一个 block 里
- first: 该记录的开头部分
- middle: 该记录的中间部分
- last: 该记录的最后部分

leveldb 会根据类型拼接出逻辑记录, 供后续处理.

what's more

除了前面讲的技术, leveldb 还有更多可以深入研究的点, 比如快照(snapshot)和版本控制(数据恢复)技术, 这里暂时没有研究, 就不扩展了.

引用

- log 文件格式: https://raw.githubusercontent.com/google/leveldb/master/doc/log_format.txt
- sstable 文件格式: https://raw.githubusercontent.com/google/leveldb/master/doc/table_format.txt
- 设计要点: https://rawgit.com/google/leveldb/master/doc/impl.html

- 设计要点: https://raw.github.com/google/leveldb/master/doc/impl.html
- 多路归并: https://zh.wikipedia.org/wiki/%E5%A4%96%E6%8E%92%E5%BA%8F
- 跳表: https://zh.wikipedia.org/wiki/%E8%B7%B3%E8%B7%83%E5%88%97%E8%A1%A8
- leveldb 实现原理: http://www.cnblogs.com/haippy/archive/2011/12/04/2276064.html

然后这里有个自己设计的存储(读优化), 当然其架构和性能跟 leveldb 是不能比的, 不过是在研究 leveldb 之前写的, 之后发现好多地方的设计竟然有异曲同工之妙, 我也偷偷的高兴一阵子~~

- Hive-fs (https://github.com/abbshr/hive-fs)
- Leviathan (https://github.com/abbshr/Leviathan/)

To Be Continued ...

我感觉上文中还有一些概念和技术的说明比较模糊, 并且也只介绍了 leveldb. 不过存储技术也是在下今后的研究方向之一, 我要让这篇日志会变得更加严谨, 内容更充实.

13 1

abbshr added logger 存储引擎 数据 labels on Oct 1, 2016

zouzls commented on Nov 21, 2016

博主写的挺好，希望继续更新，赞!

abbshr commented on Nov 21, 2016 Owner Author

@zouzls 好! 我会继续学习

cosmtrek commented on Dec 1, 2016

想问下博主插图是用什么软件画的，挺好看的，谢了。

abbshr commented on Dec 1, 2016 Owner Author

@cosmtrek 行不改名坐不改姓 ---> Keynote 是也

1