阿里巴巴中间件性能挑战赛第二赛季订单查询系统

谭钧升

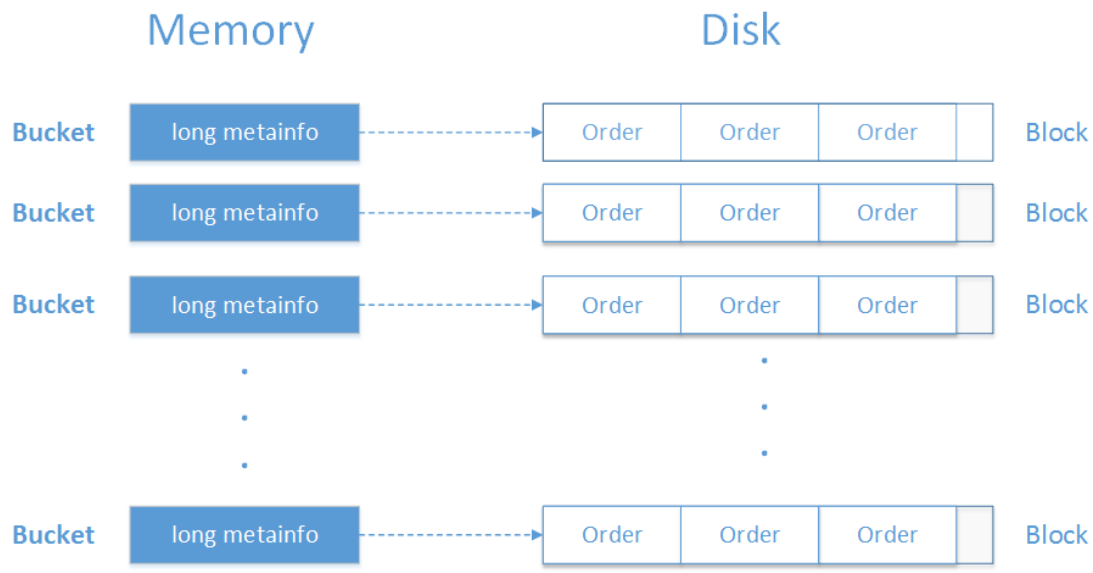
- **题目背景**

  略

- **实现思路**

  比赛给的环境配置是：3 块 1T 的机械硬盘，5G jvm 内存，5G 堆外内存，所以很明显，我们需要实现的系统是类似于传统的基于硬盘的数据库查询系统。当然，只是一个针对特定业务的非常轻量级的“数据库”。而我只知道传统基于硬盘的数据库是用 B 树实现数据索引的，但并不懂得 B 树的具体实现，因此，在赛季刚开始时的一周时间，我都在学习 B 树，看一些关于数据库实现原理的书籍。其中还借了一本关于 MySql InnoDB 引擎的书来看，不仅各种翻书看，还在网上找了一个开源的数据库引擎 MapDB 的代码来看。然而，对于第一次接触数据库底层实现的我来说，这些都显得过于复杂了。在看了一周各种资料以后，我开始想：对于这次比赛，这个题目，真的需要用 B/B+树来实现吗？B 树的优势是能动态保持树的平衡，而且减少磁盘 IO，也能支持范围查询。但是，B 树在构建时节点的分裂操作会比较耗时。而且，在这个题目里，一旦索引构建完毕，就只需要查询，也就是不再有增、删、改的操作。类似于 B 树将部分索引节点放在内存，其他节点放在硬盘里这种特点，我联想到了哈希表。同样的，如果使用基于链接法构造的哈希表，我只需将某些 Bucket 索引放在内存里，而哈希到同一个 Bucket 的数据都放在文件里，就能达到 B 树同样的结果了。而且，Bucket 的数目是我可以控制的。于是，我决定使用哈希表来构建原 100G 交易数据的索引。

  在确定使用的数据结构后，下一步就是要确定如何索引的表示。刚开始，我所有的设计都是为了减少查询时的硬盘 IO 次数。因此，刚开始时，我设计的索引是这样的：



  也就是，我用数组实现了一个哈希表，哈希表中的 bucket 存的是 64bit 的一个 long，我命名为 metainfo，也就是数据的元信息。通过该 64bit 的 metainfo，我可以获得哈希到该 bucket 的所有订单数据。而存在硬盘中的数据的单位是一个 block 一个

block 的，每个 block 的大小是固定的，比如是 64KB, 256KB。这样，假设每个 block 64KB,每条订单数据 64B，那么该 block 最多能存 1000 条订单。然而，通常一个 block 不会存满订单数据，因此 block 经常会产生内部碎片。那么 metainfo 如何表示一个 block 的信息？充分利用每个 bit 即可。比如，在这里，我把 64bit 的 metainfo 划分为:

也就是: 1，第 63 位~第 59 位： 5 位文件位，用来表示该 block 在哪个文件；

2，第 58 位~第 24 位：35 位用来表示该 block 在文件中的位置。2^35=32GB，因此每个文件最大 32G 多；

3，第 23 位~第 0 位：24 位用来表示每个 block 的大小，2^24=16MB，因此每个 block 最大 16MB。

这样，在存储和读取数据时，都是用位运算。下面介绍一下在构建索引时的过程：

1， 读到一条订单数据，根据订单 ID 哈希选择放在哪个 bucket（在文件里表示为放到哪个 block）；

2， 根据 bucket 保存的 metainfo，将该订单数据直接 append 到该 block 的当前末尾位置即可。

这样，在构造完之后，要查询某条订单就很简单了：根据订单 ID 哈希到的 bucket，根据 metainfo 到对应的文件、对应的位置将一整块 block 读取出来放到内存中，然后再在内存中找到对应 ID 的订单数据。

这样，每次查询，都只需一次 IO 即可。但是，这样相当于将原本的 100G 交易数据全部读出来写到另外的文件，3 块磁盘 1 个小时的时间很难构建完，所以后来就改成了“每次查询需要 2 次 IO，但索引文件数据只需 10 几 G”方案。

关于优化构建时间：读写分离，批量顺序写。3 块硬盘，2 块读，一块写。要写的数据先攒在内存中，到了一定量（比如几 M），再批量顺序写到文件中。