高性能Key/Value存储引擎SessionDB

## 简介

随着公司业务量的逐年成长，粘性会话(Sticky Session)越来越成为应用横向扩展(Scale Out)的瓶颈，为消除粘性会话，支持应用无状态(Stateless)，我们SOA团队在今年发起了集中式会话服务器(Centralized SessionServer)项目，该项目的核心是一个我们独立设计和开发的高性能持久化的Key/Value存储引擎，我们称为SessionDB，本文介绍SessionDB存储引擎的特性，架构和设计，我们的性能优化，并做出性能评测和分析。

我们的Key-Value存储引擎基于LSM(Log Structured Merge Tree)[1]算法思想，借鉴了Google LevelDB[2]的一些设计思想，同时在读写方面做了很多性能优化，具备如下特点：

1. 高读写性能，写入性能接近O(1)内存访问，读取性能最差平均O(1)次磁盘操作，适合高性能会话数据的存取，同样也适合其它缓存类数据的存取；
2. 数据持久化，所有数据都存储在磁盘文件中，没有Memcached等缓存数据库的踢出丢弃(Eviction)问题，适合会话数据场景。
3. 容量大，可存储超过内存容量的数据。
4. 有效利用内存，Heap内存占用量小，采用三级存储机制，只有近期插入的新鲜数据驻留在Heap内存中，大量次新鲜数据驻留在内存映射文件(Memory Mapped File)中，巨量老数据驻留在磁盘文件中，三级存储机制确保高性能读写，且Heap GC对整体读写性能影响不大。
5. 线程安全，支持多线程并发和非阻塞(non-blocking)式读写。
6. 抗宕机(Crash-Resistance)，所有数据是持久化durable的，宕机或进程死，只需重启机器或进程，即可快速自动恢复数据。
7. 支持自动的到期数据和删除数据清理(compaction)，避免磁盘和内存空间浪费。
8. 设计和实现简单轻量，简单的类Map接口，仅支持Get/Put/Delete操作，基于Java实现可跨平台，代码量少，目前core jar只有48K，可作为嵌入(Embeddable)使用。

## LSM原理

当代数据存储引擎主要基于两类数据结构，B+树和LSM树。传统的SQL数据库(例如BerkeleyDB)主要基于B+树结构，B+树的读性能好，一次读取通常只需一次I/O磁盘操作，但B+树的写入性能相对差，一次写入常常需要多次随机磁盘I/O操作。和B+树不同，LSM树是一种写优化的数据结构，LSM利用磁盘顺序写性能远好于随机写这一事实，将随机写转变为顺序批量写。简化的LSM树有两个部件组成(Figure 1)，C0和C1部件，C0部件驻留在内存，C1部件驻留在磁盘上，C0和C1都可以是B+树，写操作都发生在C0部件，基本是纯内存操作，性能高；当C0树的大小超过一定的阀值，它就会和磁盘上C1树进行合并(compaction)，合并成更大的一颗C1树，读操作从C0树开始查找，如未找到则继续查找C1树。扩展的LSM树一般有多(K)个部件(Figure 2)，除C0驻留内存，其它则以新鲜度分层(Level)方式驻留磁盘，每一层都有大小限制，归并时从Ci到Ci+1向下归并。随着层次的增加，LSM树在查找时所需检查的层次就会变多，所以总体LSM树的读性能要低于写性能，但有一些优化的手段，比如增加布隆过滤器(Bloom Filter)，来有效减少读取时所需查找的部件数量。当前流行的HBase，Cassandra，LevelDB等NoSQL数据库的核心存储引擎都是基于LSM树的思想发展而来的。

我们的SessionDB也基于LSM树的算法思想，和LevelDB比较相似，但做了简化和优化，可以认为SessionDB是一个简化版的LevelDB。和LevelDB的主要差异是，SessionDB并不按Key进行排序(仅按Key的哈希值进行排序)，所以SessionDB仅支持随机Get/Put操作，不支持顺序遍历等操作。在我们的会话数据场景和其它多数缓存场景中，顺序遍历是不需要的。我们的简化一方面简化了设计和实现，同时还大大提升了数据检索(Get操作)的性能。

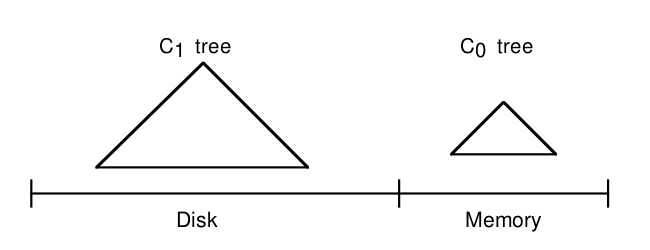


Figure 1, 简化的LSM树

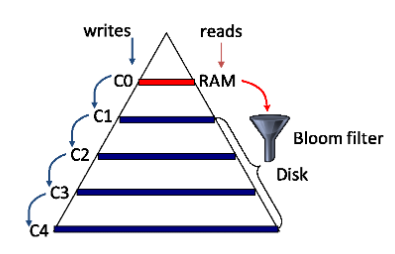


Figure 2, 多层LSM树

## 总体架构和设计



Figure 3 SessionDB总体架构和设计

整个架构(见Figure 3)由四个层次组成，最顶上的一个是当前活跃的ActiveMapTable，相当于LSM树的C0部件，Put/Delete操作发生且仅发生在ActiveMapTable上，当ActiveMapTable的大小超过一定阀值，则它会被插入到Level0队列头，变成一个只读的ImmutableMapTable，同时系统会创建一个新的MapTable作为当前活跃的ActiveMapTable。ActiveMapTable(ImmutableMapTable相同)由两个子部件组成，InMem-Hashmap + IndexedDatafile，Put操作时数据项Key/Value先追加(append)到IndexedDatafile，这点类似于持久化的WAL(Write Ahead Log)，而后Key和数据项在数据文件中的索引index被put到InMem-Hashmap中；Get操作时先检索InMem-Hashmap，找到index后再从IndexedDatafile中读取数据项的Value，为加快数据在磁盘文件中的读写速度，IndexedDatafile以内存映射(Memory Mapped)方式加载并访问。

当Level0的ImmutableMapTable达到一定的数量(比如2个)，一个称为Level0Merger的背景线程会将多个ImmutableMapTable排序和归并(Sort & Merge)为一个SortedMapTable，然后将其插入Level1队列头。 Level0Merger归并时会消除对重复key的Put/Delete数据，仅保留最新的一份数据。Level1的SortedMapTable有两个子部件组成，BloomFilter和SortedDatafile，归并排序时数据同时写入BloomFilter和SortedDatafile; Get操作时先检索BloomFilter，如报告可能存在，再通过两分查找 (binary search) 算法查询SortedDatafile，SortedDatafile也以内存映射(Memory Mapped)方式加载并访问，以加快读写速度。

当Level1的SortedMapTable达到一定的数量(比如4个)，一个称为Level1Merger的背景线程会将多个SortedMapTable归并排序(Merge & Sort)为一个OnDisk-SortedMapTable，归并后将其插入Level2队列头。如果OnDisk-SortedMapTable已经存在，则一起参与归并排序。OnDisk-SortedMapTable是最后一路归并排序的结果，所以Level1Merger归并时不仅会消除对重复key的Put/Delete数据，而且还会彻底消除删除(Deleted)和到期(Expired)的数据。OnDisk-SortedMapTable的组成结构和Level1的SortedMapTable基本相同，唯一区别是OnDisk-SortedMapTable中的SortedDatafile直接驻留在磁盘上，没有采用内存映射方式，这样设计的主要考虑是最后一层的数据量可能会比较大，驻留磁盘可以不受内存容量限制。

Put操作发生且仅发生在当前活跃的ActiveMapTable，操作涉及一次内存映射文件写入和一次内存Hashmap的写入，可以认为写入性能接近O(1)内存访问；Delete操作是一种特殊的Put操作，相当于Put一个特殊的墓碑(Tombstone)数据，所以Delete可以统一成Put操作。Get操作从当前活跃的ActiveMapTable开始，按新鲜度从上往下依次搜索，同一层内按新鲜度从左向右搜索。在ActiveMapTable和Level0的ImmutableMapTable中查找时，开销就是一次内存Hashmap的Get操作，和一次内存映射文件的读取操作(如果存在)，在Level0和Level1的SortedMapTable中查找时，开销是对内存映射文件的一次两分查找，和一次数据读取操作(如果存在)，最坏情况下，数据要在最后一层中才能找到，这个时候除了之前在内存和内存映射文件中的查询开销，还涉及一次磁盘的读取操作。可以认为总体读取性能最差平均O(1)次磁盘操作。

## 索引和数据文件结构



Figure 4 Indexed Datafile

SessionDB每个层级的数据文件都是带索引文件的，称为IndexedDatafile，数据项的Key和Value直接记录在数据文件中。索引项(Index Item)都是定长记录，目前索引项的大小是40个字节，包括：

|  |  |  |
| --- | --- | --- |
| 项目 | 大小(字节) | 用途 |
| Data offset | 8 | 数据项(Kev/Value)在数据文件中的偏移 |
| Key Length | 4 | 数据项Key的长度 |
| Value Length | 4 | 数据项Value的长度 |
| TimeToLive | 8 | 数据项的存活时间 |
| CreatedTime | 8 | 数据项的创建时间 |
| Key Hash | 4 | 数据项Key的hash值 |
| Status | 4 | 标记删除，压缩等信息 |

Table 1 索引项结构

## 优化

#### BloomFilter

BloomFilter是一种时间和空间效率很高的随机数据结构，它利用位数组很简洁地表示一个集合，并能判断一个元素是否属于这个集合。SessionDB为Level1和Level2的MapTable都增加了BloomFilter，这样在检索时可以快速判断一个Key是否存在于该MapTable中，如存在，则对该MapTable中的SortedDatafile进行相对耗时的两分查找，如不存在，则直接略过该MapTable，继续检查后续的MapTables。BloomFilter的一个问题是它可能会误报(False Positive)，换言之，如果BloomFilter报告不存在，则元素一定不存在；但如果BloomFilter报告存在，则元素可能真的存在，也可能不存在(误报)。我们采用Google Guava中提供的BloomFilter，它有一个误报率参数，通过将误报率设置为一个比较小的值（比如0.001，代价是需要占用更多的内存），我们可以有效地控制误报率，提高总体查询效率。

#### 存储优化

我们知道JVM Heap内存的存取性能很高，但JVM Heap内存操作有一个Heap GC的问题，所以存储量不能太大，而且还有宕机数据丢失的问题；纯磁盘文件的存取基本没有大小限制，但是它的性能要比内存低几个数量级。内存映射文件[4]是一种介于纯内存和纯磁盘之间的存储机制，它的性能介于内存和磁盘之间，它的数据也是持久化的，宕机数据基本不丢失，同时它不受Heap GC影响。

SessionDB的所有Index文件采用内存映射机制，一方面确保较高的数据检索性能，另一方面保证数据持久化。BloomFilter都驻留内存，因为它的大小比较小。新鲜的数据文件(Datafile)都存放在内存映射文件中，不受Heap GC影响，且访问速度较高。大量的老数据文件都存放在最后一层的磁盘文件中，不受内存大小限制。所有SessionDB中的数据都是直接或者间接持久化的，宕机或者进程死，只需重启即可快速恢复。总体上，SessionDB的存储机制充分考虑了数据的局部性(Locality)，大小，新鲜度(Freshness)和持久化，在效率和存储之间做了较好的平衡。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MapTable** | **级别(Level)** | **子部件** | **存储区域** | **大小** |
| ActiveMapTable/  ImmutableMapTable | 当前活跃和Level0 | HashMap | 内存Heap | 最多128 \* 1024项 |
| Index | 内存映射 | 固定4M |
| Datafile | 内存映射 | 固定128M |
| SortedMapTable | Level1 | BloomFilter | 内存Heap | <1M |
| Index | 内存映射 | 小于2 \* 4M |
| Datafile | 内存映射 | 小于2 \* 128M |
| OnDisk-SortedMapTable | Level2 | BloomFilter | 内存 | 约1-10M量级 |
| Index | 内存映射 | 40字节 x 数据量 |
| Datafile | 磁盘文件 | 取决与实际数据量 |

Table 2 分类存储

#### 索引优化

LevelDB有一个特性是支持对Key的顺序遍历查询，SessionDB不支持这一特性，因为我们的会话场景(也包括很多缓存场景)只需简单类Map的Get/Put/Delete操作，不需要顺序遍历。为此，我们对索引结构进行了一个优化，我们将Key的Hash值存在索引文件中，排序时我们按Hash值进行排序，Hash值相同（Hash碰撞）再按Key排序，也就是说索引文件中的索引项是按Key的Hash值顺序存放的，在数据文件中，相同KeyHash值的Key/Value对则按key顺序存放。查询时，我们只需要在索引文件上对Hash值进行两分查找，定位到索引项后再从数据文件里头读取对应的Key进行比对，由于Hash碰撞可能的存在，我们可能还要在定位索引项的左右两边进行比对查询，但是因为Hash碰撞的概率很低，基本一次就可以定位到数据文件中的Key/Value对，所以总体性能就是一次索引文件的两分查找+一次数据文件的读取。相对于数据文件，索引文件的大小比较小（40个字节每项，100万的数据量也只占用40M），同时索引文件是内存映射的，两分查找基本是内存读取，另外，和不定长度的key比对查询相比，对定长整数Hash值的比对性能要快很多，所以我们的索引优化大大减少了两分查询和比对的数据量，提升了总体查询性能。

#### 数据分片(Sharding)

Figure 3仅是SessionDB的一个架构单元(Unit)，考虑到多线程并发读写时对ActiveMapTable的压力，我们引入了一种数据分片(Sharding)策略，也就是一个SessionDB可以配置成多个单元(缺省4个)，每一个单元都是SessionDB的一个分片(shard)，数据写入时，SessionDB会根据Key的hash值和单元数求模获得对应的单元，然后再写入该单元，数据读取时也以同样方式先定位到对应的单元，再在单元内检索数据。数据分片可以有效缓解对单个单元SessionDB的读写压力，提升总体性能。

## 性能测试和分析

## 结论

为满足实际项目需要，我们设计和开发了一个高性能的基于LSM算法的Key/Value存储引擎SessionDB，我们在LSM算法(特别是参考Google LevelDB设计)的基础上，对SessionDB进行了很多优化，例如引入BloomFilter, 分级存储机制，索引优化和数据分片。经过实际性能测试和分析，SessionDB的总体随机读写性能要优于传统的基于B+数的数据库如BerkeleyDB[5]，同时也优于Goolge LevelDB，甚至要好于Facebook对LevelDB的改进版RocksDB[6]。

后续我们将根据实际生产环境中获得的反馈对SessionDB做进一步的调优，同时会考虑开发服务器版本的SessionDB，支持多语言客户端的接入，长期我们会考虑将SessionDB扩展成分布式的Key/Value数据库，主要参考Amazon的Dynamo[7]思想。

SessionDB是一个开源项目，其源代码可以从github上获得。

参考：

1. The Log-Structured Merge-Tree (LSM-Tree)

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.2782&rep=rep1&type=pdf>

2. Google LevelDB

<http://code.google.com/p/leveldb/>

3. Bloom Filer

<http://baike.baidu.com/view/1912944.htm>

4. 10 Things to Know about Memory Mapped File in Java

<http://www.codeproject.com/Tips/683614/Things-to-Know-about-Memory-Mapped-File-in-Java>

5. BerkeleyDB Java Edition

<http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index-093405.html>

6. Facebook RocksDB

<http://rocksdb.org/>

7. Amazon Dynamo Paper

<http://s3.amazonaws.com/AllThingsDistributed/sosp/amazon-dynamo-sosp2007.pdf>

8. SessionDB Source Code on Github

<https://github.com/ctriposs/sessdb>