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磁盘 I/O 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)组成，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。

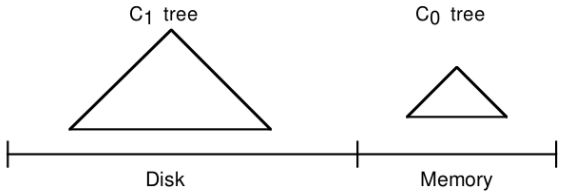

Figure 1, 简化的 LSM 树

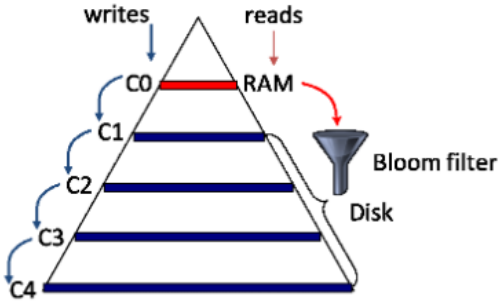

Figure 2, 多层 LSM 树

总体架构和设计

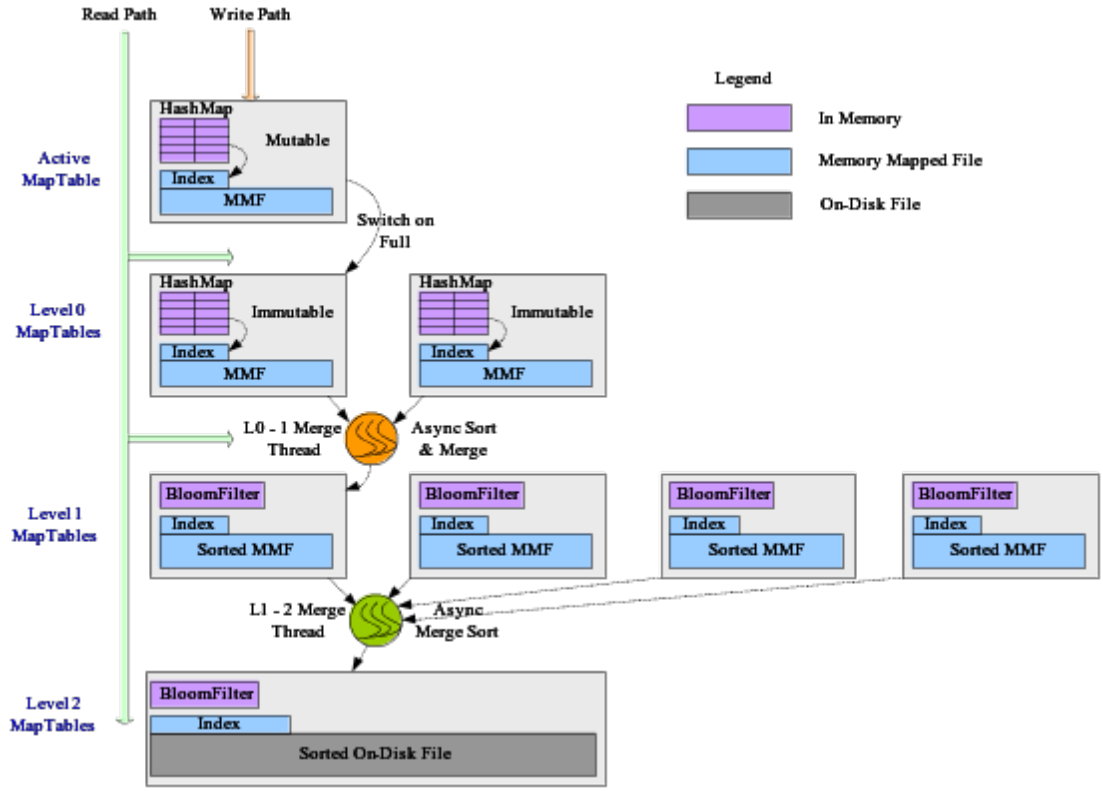

Figure 3 SessionDB 总体架构和设计

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

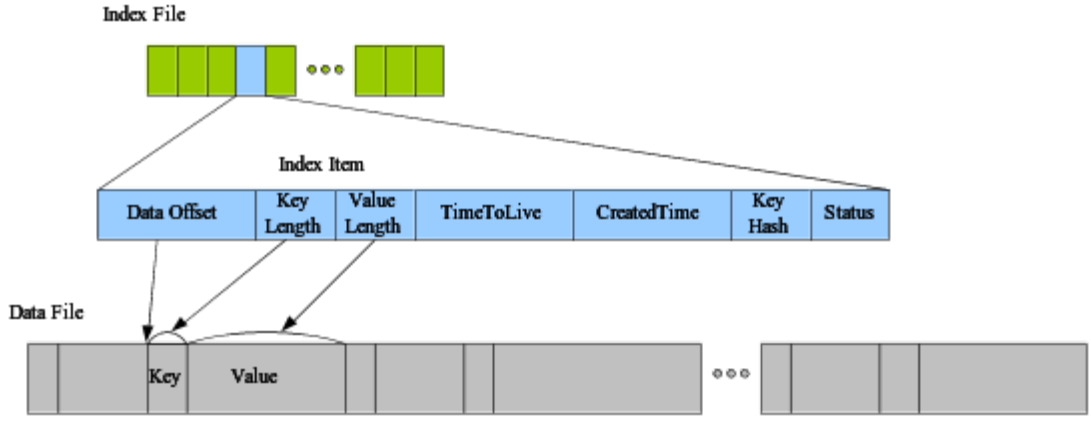

Figure 4 Indexed Datafile

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 SessionDB 的分类存储

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 的一个结构单元(Unit)，考虑到多线程并发读写对 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

我们改写了 Google LevelDB 的 benchmark 程序，对 SessionDB(Java), BerkeleyDB(Java), LevelDB(C), RocksDB(C++)[6]进行了 benchmark 测试，下面是测试结果和分析：

测试环境：

CPU: 4 * Intel Xeon E312xx (Sandy Bridge)

OS: CentOS release 6.5 (Final)

Kernel: 2.6.32-358.el6.x86_64

FileSystem: Ext4

测试的 Key 是 16 个字节，Value 是 100 个字节，Key/Value 项的总数为 1000000，测试结果单位是 micros/op:

	Random write	Random read	Random write(100k)
SessionDB	4.38723	2.83049	526.71828
BerkeleyDB	10.05216	4.68461	453.08231
LevelDB	7.217	6.679	1305.964
RocksDB	6.934	7.692	N/A

标注: N/A 表示测试错误导致没有结果

SessionDB 的总体读写性能要优于基于 B+树的 BerkeleyDB，也优于 Google 的 LevelDB，甚至优于 Facebook 对 LevelDB 的改进版 RocksDB。考虑到 LevelDB 和 RocksDB 是采用 C/C++语言开发，而我们的 SessionDB 是采用 Java 开发的，所以 SessionDB 在比对中的性能优势是比较明显的。另外测试中我们发现，SessionDB 的读取性能要好于写入性能，这一点和其它写优化的 K/V 存储引擎的测试结果不同，我们认为这主要是因为我们对索引优化的结果。

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+树的数据库如 BerkeleyDB[5]，同时也优于 Google 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[8]。

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