 [Translated from Chinese (Simplified) to English - www.onlinedoctranslator.com](https://www.onlinedoctranslator.com/en/?utm_source=onlinedoctranslator&utm_medium=docx&utm_campaign=attribution)

High-performance Key/Value storage engine SessionDB

## Introduction

As the company's business volume grows year by year, sticky sessions (Sticky Session) are increasingly becoming the bottleneck of application scale out (Scale Out). In order to eliminate sticky sessions and support application stateless (Stateless), our SOA team launched a centralized Centralized Session Server (Centralized SessionServer) project, the core of this project is a high-performance persistent Key/Value storage engine independently designed and developed by us, which we call SessionDB. This article introduces the characteristics, architecture and design of the SessionDB storage engine. We performance optimization, and make performance evaluation and analysis.

Our Key-Value storage engine is based on the algorithm idea of ​​LSM (Log Structured Merge Tree) [1], borrowed some design ideas of Google LevelDB [2], and has done a lot of performance optimization in reading and writing, and has the following characteristics:

1. High read and write performance, the write performance is close to O(1) memory access, the worst read performance is O(1) disk operations on average, suitable for access to high-performance session data, and also suitable for access to other cached data;
2. Data persistence, all data is stored in disk files, there is no kicking and discarding (Eviction) problem of cache databases such as Memcached, suitable for session data scenarios.
3. Large capacity, can store data exceeding the memory capacity.
4. Effective use of memory, Heap memory footprint is small, using a three-level storage mechanism, only recently inserted fresh data resides in the Heap memory, a large number of fresh data resides in the memory mapped file (Memory Mapped File), and a huge amount of old data Residing in disk files, the three-level storage mechanism ensures high-performance reading and writing, and Heap GC has little impact on the overall reading and writing performance.
5. Thread safety, supports multi-threaded concurrent and non-blocking (non-blocking) read and write.
6. Crash-Resistance, all data is durable and durable, just restart the machine or process, and the data can be quickly and automatically restored.
7. Supports automatic expired data and deleted data cleanup (compaction) to avoid wasting disk and memory space.
8. Design and implement a simple and lightweight, simple Map-like interface that only supports Get/Put/Delete operations. Based on Java, it can be implemented across platforms with a small amount of code. Currently, the core jar is only 48K and can be used as an Embeddable.

## Principle of LSM

Contemporary data storage engines are mainly based on two types of data structures, B+ trees and LSM trees. Traditional SQL databases (such as BerkeleyDB) are mainly based on the B+ tree structure. The read performance of the B+ tree is good. A read usually only requires one disk I/O operation, but the write performance of the B+ tree is relatively poor. A write often requires multiple Random disk I/O operations. Unlike the B+ tree, the LSM tree is a write-optimized data structure. LSM takes advantage of the fact that disk sequential write performance is much better than random write, and transforms random write into sequential batch write. The simplified LSM tree consists of two parts (Figure 1), C0 and C1 parts, the C0 part resides in memory, the C1 part resides on the disk, both C0 and C1 can be B+ trees, and all write operations occur in the C0 part , which is basically a pure memory operation with high performance; when the size of the C0 tree exceeds a certain threshold, it will be merged with the C1 tree on the disk (compaction), merged into a larger C1 tree, and the read operation starts from the C0 tree Start the search, if not found, continue to search the C1 tree. An extended LSM tree generally consists of multiple (K) components (Figure 2). Except for C0, which resides in memory, the others reside on disk in a freshness level (Level). Each level has a size limit. When merging, it starts from Ci to Ci+1 are merged downwards. As the level increases, the LSM tree needs to check more levels when searching, so the overall read performance of the LSM tree is lower than the write performance, but there are some optimization methods, such as adding Bloom Filter (Bloom Filter) ), to effectively reduce the number of parts to be searched for when reading. The core storage engines of the currently popular NoSQL databases such as HBase, Cassandra, and LevelDB are all developed based on the idea of ​​the LSM tree.

Our SessionDB is also based on the algorithm idea of ​​the LSM tree. It is similar to LevelDB, but it has been simplified and optimized. It can be considered that SessionDB is a simplified version of LevelDB. The main difference with LevelDB is that SessionDB does not sort by Key (only sorts by the hash value of Key), so SessionDB only supports random Get/Put operations, and does not support operations such as sequential traversal. In our session data scenario, and in most other caching scenarios, sequential traversal is unnecessary. On the one hand, our simplification simplifies the design and implementation, and at the same time greatly improves the performance of data retrieval (Get operation).

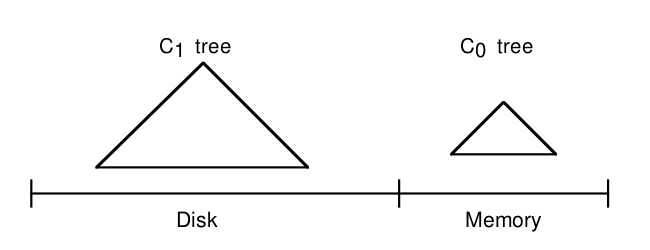


Figure 1, Simplified LSM tree

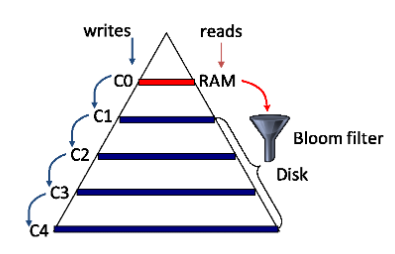


Figure 2, multi-level LSM tree

## Overall Architecture and Design



Figure 3 SessionDB overall architecture and design

The entire architecture (see Figure 3) consists of four levels. The top one is the currently active ActiveMapTable, which is equivalent to the C0 component of the LSM tree. The Put/Delete operation occurs only on the ActiveMapTable. When the size of the ActiveMapTable exceeds a certain threshold, it will be inserted into the head of the Level0 queue and become a read-only ImmutableMapTable, and the system will create a new MapTable as the currently active ActiveMapTable. ActiveMapTable (the same as ImmutableMapTable) consists of two sub-components, InMem-Hashmap + IndexedDatafile. During the Put operation, the data item Key/Value is first appended (append) to the IndexedDatafile, which is similar to the persistent WAL (Write Ahead Log), and then Key and The index Index of the data item in the data file is put into the InMem-Hashmap; when the Get operation is performed, the InMem-Hashmap is searched first, and then the Value of the data item is read from the IndexedDatafile after finding the Index, in order to speed up the reading and writing of data in the disk file Speed, IndexedDatafile is loaded and accessed in a memory mapped (Memory Mapped) manner.

When the ImmutableMapTable of Level0 reaches a certain number (such as 2), a background thread called Level0Merger will sort and merge multiple ImmutableMapTables (Sort & Merge) into a SortedMapTable, and then insert it into the head of the Level1 queue. Level0Merger will eliminate Put/Delete data for duplicate keys when merging, and only keep the latest data. The SortedMapTable of Level1 is composed of two sub-components, BloomFilter and SortedDatafile. When merging and sorting, the data is written into BloomFilter and SortedDatafile at the same time; when the Get operation is performed, the BloomFilter is first retrieved. If the report may exist, then the binary search (binary search) algorithm is used to query the SortedDatafile, SortedDatafile It is also loaded and accessed in the form of memory mapping (Memory Mapped) to speed up reading and writing.

When the SortedMapTable of Level1 reaches a certain number (for example, 4), a background thread called Level1Merger will merge and sort (Merge & Sort) multiple SortedMapTables into an OnDisk-SortedMapTable, and insert it into the Level2 queue head after merging. If OnDisk-SortedMapTable already exists, it will participate in merge sort together. OnDisk-SortedMapTable is the result of the last merge sort, so Level1Merger will not only eliminate Put/Delete data for duplicate Keys, but also completely eliminate Deleted and Expired data. The composition structure of OnDisk-SortedMapTable is basically the same as that of Level 1 SortedMapTable. The only difference is that the SortedDatafile in OnDisk-SortedMapTable resides directly on the disk and does not use memory mapping. The main consideration of this design is that the data volume of the last layer may be relatively large. Large, resident disks can be unlimited by memory capacity.

The Put operation occurs and only occurs in the currently active ActiveMapTable. The operation involves a memory-mapped file write and a memory Hashmap write. It can be considered that the write performance is close to O(1) memory access; the Delete operation is a special Put operation , which is equivalent to Put a special tombstone (Tombstone) data, so Delete can be unified into Put operation. The Get operation starts from the currently active ActiveMapTable, searches from top to bottom according to freshness, and searches from left to right according to freshness within the same layer. When searching in ActiveMapTable and ImmutableMapTable of Level0, the overhead is a Get operation of the memory Hashmap, and a read operation of the memory-mapped file (if it exists). When searching in the SortedMapTable of Level0 and Level1, the overhead is for the memory-mapped file. A two-point search, and a data read operation (if it exists), in the worst case, the data can only be found in the last layer. At this time, in addition to the previous query overhead in memory and memory-mapped files, it also involves a disk read operation. It can be considered that the overall read performance is the worst with an average of O(1) disk operations.

## Index and Data File Structure



Figure 4 Indexed Datafile

The data files at each level of SessionDB are indexed files, called IndexedDatafile, and the Key and Value of data items are directly recorded in the data file. Index items (Index Item) are all fixed-length records, and the current index item size is 40 bytes, including:

|  |  |  |
| --- | --- | --- |
| project | size (bytes) | use |
| Data offset | 8 | The offset of the data item (Kev/Value) in the data file |
| Key Length | 4 | The length of the data item Key |
| Value Length | 4 | The length of the data item Value |
| Time To Live | 8 | data item lifetime |
| CreatedTime | 8 | creation time of the data item |
| Key Hash | 4 | The hash value of the data item Key |
| Status | 4 | Mark deletion, compression, etc. |

Table 1 Index Entry Structure

## optimization

#### BloomFilter

BloomFilter is a random data structure with high time and space efficiency. It uses bit array to represent a set concisely and can determine whether an element belongs to this set. SessionDB adds BloomFilter to both Level 1 and Level 2 MapTables, so that it can quickly determine whether a Key exists in the MapTable during retrieval. If it exists, it will perform a relatively time-consuming binary search on the SortedDatafile in the MapTable. If it does not exist , then directly skip the MapTable and continue to check the subsequent MapTables. One problem with BloomFilter is that it may be false positive (False Positive). In other words, if BloomFilter reports that it does not exist, then the element must not exist; but if BloomFilter reports that it exists, the element may or may not exist (false positive) . We use the BloomFilter provided in Google Guava, which has a false positive rate parameter. By setting the false positive rate to a relatively small value (such as 0.001, the cost is that it takes up more memory), we can effectively control false positives. rate, improving the overall query efficiency.

#### storage optimization

We know that the access performance of JVM Heap memory is very high, but there is a problem of Heap GC in JVM Heap memory operation, so the storage capacity should not be too large, and there is also the problem of data loss during downtime; the access of pure disk files basically has no size limit, but its performance is orders of magnitude lower than memory. Memory-mapped file [4] is a storage mechanism between pure memory and pure disk. Its performance is between memory and disk. Not affected by Heap GC.

All Index files of SessionDB adopt a memory mapping mechanism, which ensures high data retrieval performance on the one hand and data persistence on the other. BloomFilters are memory resident because of their small size. Fresh data files (Datafile) are stored in memory-mapped files, not affected by Heap GC, and have a high access speed. A large number of old data files are stored in the last layer of disk files, which are not limited by the memory size. All the data in SessionDB are persisted directly or indirectly. If the process crashes or the process dies, it can be recovered quickly just by restarting. Generally speaking, the storage mechanism of SessionDB fully considers the locality (Locality), size, freshness (Freshness) and persistence of data, and makes a good balance between efficiency and storage.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **MapTable** | **Level** | **subcomponent** | **storage area** | **size** |
| ActiveMapTable/  ImmutableMapTable | Currently Active and Level0 | HashMap | Memory Heap | Up to 128\*1024 items |
| Index | memory map | Fixed 4M |
| Datafile | memory map | Fixed 128M |
| SortedMapTable | Level1 | BloomFilter | Memory Heap | <1M |
| Index | memory map | Less than 2\*4M |
| Datafile | memory map | Less than 2\*128M |
| OnDisk-SortedMapTable | Level2 | BloomFilter | Memory | About 1-10M order of magnitude |
| Index | memory map | 40 bytes x amount of data |
| Datafile | disk file | depends on the actual data volume |

Table 2 Classification storage of SessionDB

#### index optimization

A feature of LevelDB is that it supports sequential traversal queries on Keys. SessionDB does not support this feature, because our session scenarios (including many cache scenarios) only need simple Get/Put/Delete operations of the Map class and do not require sequential traversal. To this end, we have optimized the index structure. We store the Hash value of the Key in the index file. When sorting, we sort by the Hash value. The Hash value is the same (Hash collision) and then sort by the Key. That is to say, in the index file The index items are stored in the order of the Hash value of the Key. In the data file, the Key/Value pairs with the same KeyHash value are stored in the order of the Key. When querying, we only need to perform a two-point search on the Hash value on the index file, and then read the corresponding Key from the data file for comparison after locating the index item. Due to the possibility of Hash collisions, we may also need to locate The left and right sides of the index item are compared and queried, but because the probability of Hash collision is very low, the Key/Value pair in the data file can be located basically at one time, so the overall performance is a two-point search of the index file + a data file. read. Compared with the data file, the size of the index file is relatively small (40 bytes per item, and the data volume of 1 million only occupies 40M). At the same time, the index file is memory-mapped, and the two-point search is basically memory reading. In addition, and Compared with variable-length key comparison queries, the comparison performance of fixed-length integer hash values ​​is much faster, so our index optimization greatly reduces the data volume of binary queries and comparisons, and improves the overall query performance.

#### Data Sharding (Sharding)

Figure 3 is only a structural unit (Unit) of SessionDB. Considering the pressure of multi-threaded concurrent read and write on ActiveMapTable, we introduced a data sharding (Sharding) strategy, that is, a SessionDB can be configured into multiple units (absent Save 4), each unit is a shard of SessionDB, when data is written, SessionDB will obtain the corresponding unit according to the Hash value of the Key and the number of units, and then write the unit, the data read When fetching, the corresponding unit is located in the same way, and then the data is retrieved in the unit. Data sharding can effectively relieve the read and write pressure on a single unit SessionDB and improve overall performance.

## Performance Testing and Analysis

We rewrote the benchmark program of Google LevelDB and conducted benchmark tests on SessionDB(Java), BerkeleyDB(Java), LevelDB(C), RocksDB(C++)[6]. The following are the test results and analysis:

test environment:

CPU: 4 \* Intel Xeon E312xx (Sandy Bridge)

OS: CentOS release 6.5 (Final)

Kernel: 2.6.32-358.el6.x86\_64

FileSystem: E.xt4

The tested Key is 16 bytes, the Value is 100 bytes, the total number of Key/Value items is 1,000,000, and the test result unit is 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 |

Note: N/A indicates that the test error resulted in no results

The overall read and write performance of SessionDB is better than BerkeleyDB based on B+ tree, Google's LevelDB, and even Facebook's improved version of LevelDB, RocksDB. Considering that LevelDB and RocksDB are developed in C/C++ language, and our SessionDB is developed in Java, the performance advantage of SessionDB in the comparison is quite obvious. In addition, in the test, we found that the read performance of SessionDB is better than the write performance, which is different from the test results of other write-optimized K/V storage engines. We think this is mainly because of our index optimization results.

## in conclusion

In order to meet the needs of actual projects, we designed and developed a high-performance Key/Value storage engine SessionDB based on the LSM algorithm. We have optimized SessionDB on the basis of the LSM algorithm (especially referring to Google LevelDB design), such as Introduce BloomFilter, hierarchical storage mechanism, index optimization and data sharding. After actual performance testing and analysis, the overall random read and write performance of SessionDB is better than traditional B+ tree-based databases such as BerkeleyDB[5], and also better than Google LevelDB, and even better than Facebook’s improved version of LevelDB, RocksDB[6]. .

In the future, we will further optimize SessionDB according to the feedback obtained in the actual production environment. At the same time, we will consider developing a server version of SessionDB to support the access of multilingual clients. In the long run, we will consider expanding SessionDB into a distributed Key/Value Database, mainly refer to Amazon's Dynamo[7] idea.

SessionDB is an open source project, and its source code is available from github [8].

refer to:

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>