The Design of CuttDB - Brief Introduction

CuttDB is a key-value database engine. Each record in CuttDB contains a key and a value, key and value are schemaless strings. It is a UNIX DBM-liked database and only the three basic operations Get(key)/Set(key, value)/Delete(key) are supported. Other concepts like data types, transaction, relation query are omitted.

Key-value databases got remarkable development with the NoSQL movement. There are many products which have similar function, like TokyoCabinet, LevelDB, BerkeleyDB, Memcached etc. They are targeted at different usage levels and environments. Even the DBM-liked databases have very different designs, which made them show different performance behaviour. CuttDB emphasizes on the performance when storing massive small records in limited memory. It was initially designed for URL storage and de-duplication in web crawler and to be a replacement of TokyoCabinet. Additionally, it is a hash based engine. It means a record can only be retrieved by certain key. Range query, prefix query and sequential read are not supported.

To achieve the goal of fast set/get speed, the design is tried to accommodate to the physical characteristic of hardware. Today most storage equipments are still rotation disks. They have good sequential I/O performance while random I/O are terribly slow. SSD performs much faster than an rotation disk, but its random I/O are still less efficient than sequential operations. My approach to the problem is a simple way of data arrangement on disk, and a compact index to store more records in limited memory.

The files of database have two types, data file storing records and index file storing index pages. Both of them are log structured and they always write ahead to get max throughput. Once the size of writing file increased to a certain size, it will be closed and a new file will generated. Even if to update or delete an item, which refers to a record or an index page, the old one on disk would neither be erased nor marked invalid at the time. However, the size of garbage space in each file is stored in memory's meta-info structure. It will be updated immediately. When the garbage space ratio in one file reached a threshold, the file will be mapped into memory and got a whole scan. The item does not match the current state should be discarded, valid items will be appended to the current writing file. After the scanning, the mapped file can be erased. This is the space recycle procedure. Every database instance has a separate thread to do space monitoring and recycle tasks on both data files and index files.

The index of engine is a huge hashtable. It is different with conventional hash index which maps hashes of keys into records information. The index in CuttDB contains two parts: main table and pages. Main table is a nonexpendable array which only stores on-disk offsets of every pages. It helps to locate the corresponding page for a key. A page is a small sub-hashtable. It compactly stores some records' sub-hash values and their offsets. All offsets in the program are 6 bytes long. It means that the size limit of a instance is 2^(48+4) bytes with 16 bytes aligned. The sub-hash value of a record is limited to 3 bytes. The keys of records and their meta-info include their sizes, are not stored in pages, only after one record was read out, it could be

possible to check if the record matches the query key. The range of sub-hash (2^24) ensures very low collision rate in a page. If a page contains 100 records as recommended, the probability of no hash collision is 99.94%. So in nearly all cases, if the page is already in memory, only one read from disk is enough to locate a record.

Since the total size of index pages could be greater than physical memory size, or user may wants to sacrifice the speed to reduce memory usage, all pages are swappable. This is a little similar to the OS memory management. Pages are maintained in LRU hash tables as caches. There are two page caches, a dirty page cache and a clean page cache. A page's being modified or written to index file may cause its move between the two caches. The task thread mentioned before is also responsible to flush the oldest dirty pages out to index file. Main table is only several megabytes in size, it must be always in memory.

Every write operation in the program is buffered. Though OS itself has disk buffer, context switches caused by write(2) syscall would consume CPU heavily when writing short data frequently. Get() operation may cause reading from any data files, there is a file descriptor cache helps to not open too much file simultaneously or do open() and close() too frequently.

Because of the simple arrangement of data file, it is very easy to recover damaged data or to restart from crash. The data file itself is operation log, so just to iterate the records in data file and rebuild the index is enough to restore. Actually there is 'check point' in CuttDB, like some other databases, makes the recovery very fast. Data in write buffer may be lost when abnormal close occurred . However, data in buffer will not stay for more than several seconds before flushed out. The short time's data loss was acceptable in my system environment.

The design aims at reducing disk seek and index as much records as possible in memory. With the help of buffer, a Set() operation typically cause no actual disk write if the page which the key belongs to is in cache, or one disk read to load page if it is not in cache. There is also a record-level LRU cache, so a Get() operation cause no read if matched record itself is cached, one read if page is cached, or two reads at most for both page and record if nothing is cached. Since only about 9 bytes extra space is necessary for index one more record(3 bytes hash, 6 bytes offset), for a instance stored 500,000,000 records, 5GB memory is enough to full cache the index pages and ensure the best performance, which achieved the performance of about less than 5 microseconds for Set() and 10 milliseconds for Get(). The design of CuttDB is enlightened by Bitcask in Riak. There are two significant differences between them. CuttDB needs not to keep the index in memory completely, all index pages are swappable. Secondly, only the hash of key is stored in index instead of the whole key and its meta-info. These features made CuttDB more flexible to store massive records in a single machine. However, a defect is introduced, update operation on existing record will cost an extra disk seek to check if the record with the same hash matches.

This is the general idea of CuttDB. Some parts like bloom filter and lock mechanism which has less relation to the core was not introduced. To see the complete implement, please refer to: https://github.com/fusiyuan2010/cuttdb.

Appendix: Diagram of data arrangement in CuttDB