A NoSQL storage system

Yinan Li
A53217445
yil691@ucsd.edu

Yuchen Wnag
A53213477
yuw520@eng.ucsd.edu

Abstract
We design and implement a NoSQL database storage. A database table's storage is a set of independent column storage. For each column storage, there are in-memory storage, memtable, and on-disk storage, sstable, both of which store a set of key-value pairs. Memtable is roughly cache for sstable. When memtable's size exceed a predetermined threshold, it's dumped to disk as a sstable. To enhance read latency, we implement in-memory bloomfilter and LRU cache. To reduce the number of disk access to once at most, an in-memory indextable for looking up sstable offset is provided. The column storage provide service mainly by two API, get(key) and put(key, value). We implement several queries on top of them, e.g. select, join, sum and count.

1. Implementation
The following implementation is all concerned with storage for a single column, and we will describe at last for the organization of the whole table.

1.1 memtable and sstable
The whole storage's function can be provided by just memtable and sstable, while others are for improving performance. So we first describe implementation of memtable and sstable.
memtable and sstable is the storage for a single column of a database. They together provide two operations, get(key) and put(key, value). Data modification, insert, and delete are provided by put(key, value). Data lookup is provided by get(key). memtable is a conventional in-memory hashtable with a size upper bound. When memtable size is larger than the threshold, its entries are sorted by key and dumped to disk as sstable. For get(key), we first lookup the key in memtable. If not found, we search sstable from the newest record, until the first key is found. For put(key, value), we update the record if the key exist in memtable or insert a record if not. Therefore, put (key, value) never access disk, unless memtable is full.
For data insert and update, it's straightforward to use put(key, value). For data deletion, we use put(key, NULL), i.e. we insert or update the value for the key as a tombstone, but not erase it from memtable.
The on-disk sstable should be merged at some time. For simplicity, we merge the sstable when we dump memtable.

1.2 Optimization
1.2.1 Bloomfilter
To avoid disk access as much as possible, we first implement bloomfilter. Bloomfilter is kept in memory. For put(key, value), we update memtable, and then insert key to bloomfilter. For get(key), if key is not found in memtable, then it's looked up in bloomfilter. If it's not in bloomfilter, it's ensured that the key is not in sstable either, so we avoid disk access. Otherwise we search sstable as usual.

1.2.2 Indextable
Indextable is an in-memory hashtable. To further eliminate disk access, sstable is organized to blocks, where each block is the size of disk block and can be read in memory with a single seek. When memtable is dumped to sstable, the offset and key range of each block is recorded in indextable, where the key range is indextable's key and offset is value. Therefore, indextable record the block offset of all keys' newest record. When we search a key in sstable, instead of sequentially read sstable, we first find out the offset of the key in indextable, and the seek disk with the offset to read in a block and search the key in the block. As a result, we limit disk access or get(key) to at most once.
1.2.3 LRU

LRU is an in-memory cache for recent used key-value pair. It's conveniently implemented with python *OrderedDict* data structure. For put(key, value) or get(key), LRU is both updated and push the key to newest position as most recently used. Least recently used key is popped out when LRU size is beyond threshold. LRU further eliminates disk access. For get(key), before searching sstable, LRU is checked for the key.

2. Table organization and query

All columns of a table are grouped together, which means they all have the same key as the table's key, and they can only be accessed through the table class. The table class contains the mapping between column name and column class.
A table can be initialized in two ways: create a new table or load a table from disk. If it's loaded from disk, indextable, bloomfilter are initialized by scanning all sstables. A more efficient solution would be to persist indextable and bloomfilter on disk when table is properly closed. However, it would involve dealing with consistency after system failure. For simplicity, we assume all in-memory cache will be lost after the table is closed and need to be reloaded from disk next time.
When closing a table, we dump remaining memtables.

We implement the following queries:
select():
select required columns of rows where the rows' record satisfies the conditions. It's similar to SQL's SELECT...WHERE...
set():
update or insert rows.
hasKey():
return whether the table contains the key.
join():
join two tables by primary key and return the result rows.
count():
count how many times a value appear in a certain column
sum():
return the sum of a numeric column
max():
find the maximum of a numeric column

3. Performance test

We generate a series of workload and test the elapsed time for each operation. We first insert N records to the table, then performance select(), hasKey(), join(), and sum() on the table. Specially, because whether the key actually exist in the table greatly affects time cost, we separately test hasKey() in two cases, where the key actually exist or doesn't exist in the table.
As memtable size is a key parameter which influence performance, we choose N respect to the size of memtable size M. Specifically, we choose N = M / 10, M / 7, M / 5, M / 3, M / 2, M * 2, M * 3, M * 5, M * 7, M * 10.
We set the system's M = 20000 and block size = 200. The result is as follows.

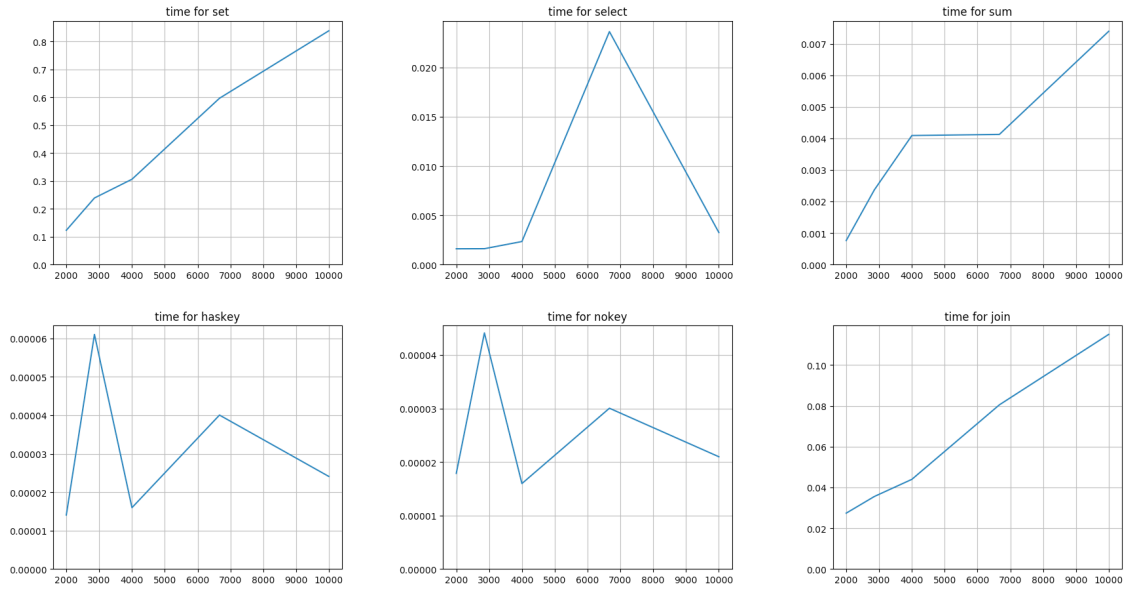

Figure 1. Performance for N < M. X axis is N, y axis is time cost, in seconds.

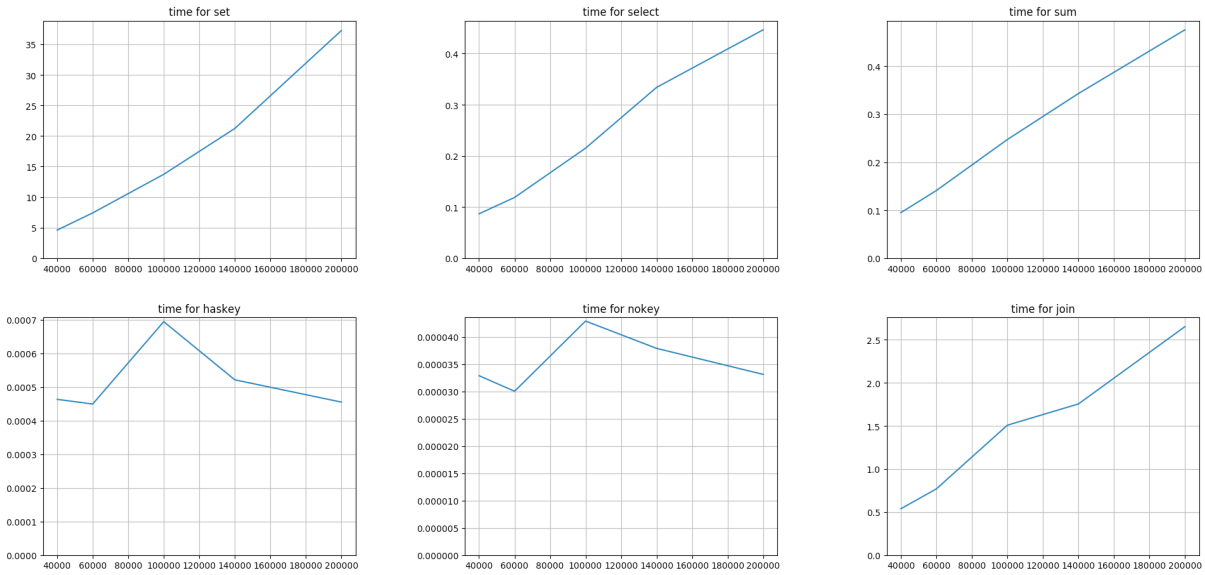

Figure 2. Performance for N > M. X axis is N, y axis is time cost, in seconds.

For Figure 1 and Figure 2, the most significant result is whether the line is linear. We denote hasKey(k) where k exists as hasKey(), and hasKey(k) where k doesn't exist as noKey(). For N < M, select(), hasKey() and noKey() are not linear. For N > M, hasKey() and noKey() are not linear. It's trivial that hasKey() and noKey() should not be linear for N < M, since it's only a random hashtable lookup in memory. But for N > M, since we might access disk, the result shows that for single key query the number of disk access don't grow with number of records. The reason why select(), join() and sum() grows linearly for N > M is that the query result size grows linearly, and search for a single key costs constant time, so the number of disk access also grows linearly. However, for query like hasKey(), we are querying single key.

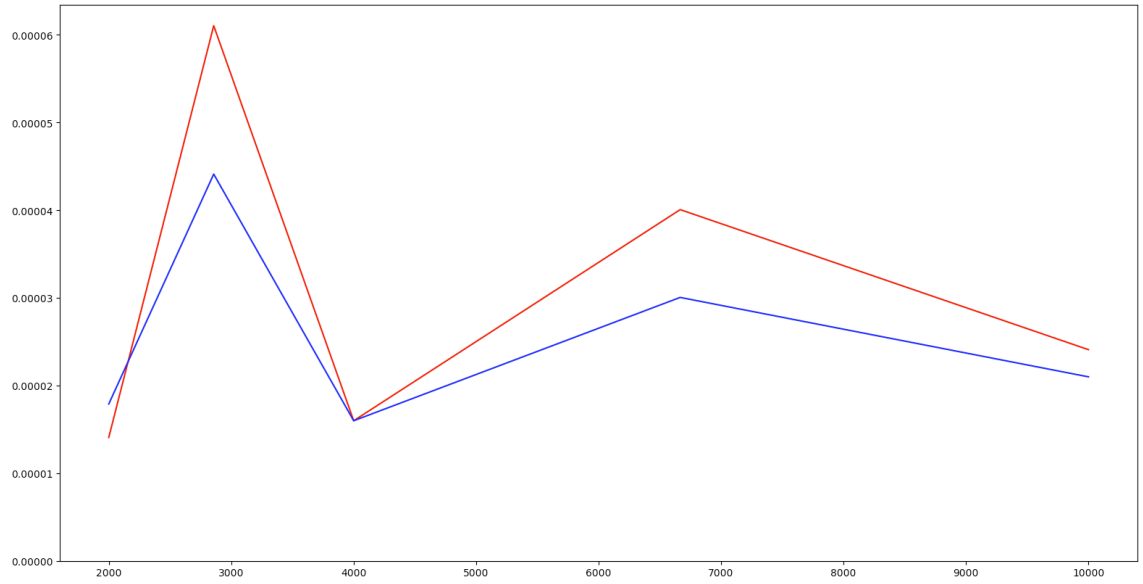

Figure 3. Performance of hasKey() for N > M, where red line is when key exists, and blue line is when key doesn't exist. X axis is N, y axis is time cost, in seconds.

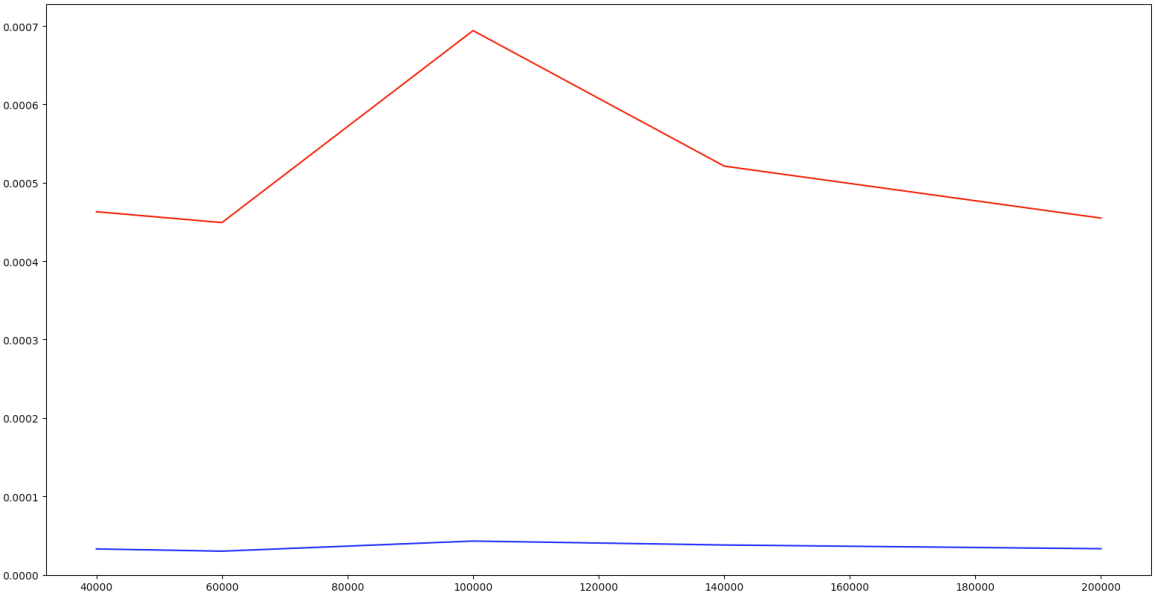

Figure 4. Performance of hasKey() for N < M, where red line is when key exists, and blue line is when key doesn't exist. X axis is N, y axis is time cost, in seconds.

Figure 3 and Figure 4 show that difference between hasKey() and noKey() for N > M and N < M. It shows that for N > M, noKey() is significantly faster than hasKey(), since we implement bloomfilter to completely eliminate disk access for noKey().