# CSE291: TinyDB: NoSQL database

Junyao Wang | A53217637 | juw086@eng.ucsd.edu
Rongqi Yan | A53203966 | roy007@eng.ucsd.edu

June 14, 2017

## 1 Introduction

### 1.1 Overview

We implement our NoSQL storage called TinyDB. TinyDB is key value store following the typical design of most k-v stores like Hbase. Data is organized in column oriented way with simple API of get and put for single column table. Column families and databse is structured upon single column table. Optimization methods like BloomFilter, Block level cache, compaction and data compression are implemented to improve performance. JUnit test files are included to test the basical functions and a simple bentchmark is conducted to measure the performance.

### 1.2 Component

TinyDB is mainly composed of four three parts: Coltable, DB and test.

Coltable is the basical data store part for a single column. Memtable, SStable and table design details are implemented in this part.

DB is the wrapper based on Coltable. By grouping Coltabls into column families and managing column families, basic DB functions are implemented. A simple SQL-like parser is also implemented.

Test part is mainly JUnit tests for Coltables including low level tests about data store and higher level tests of handling large amounts of requests.

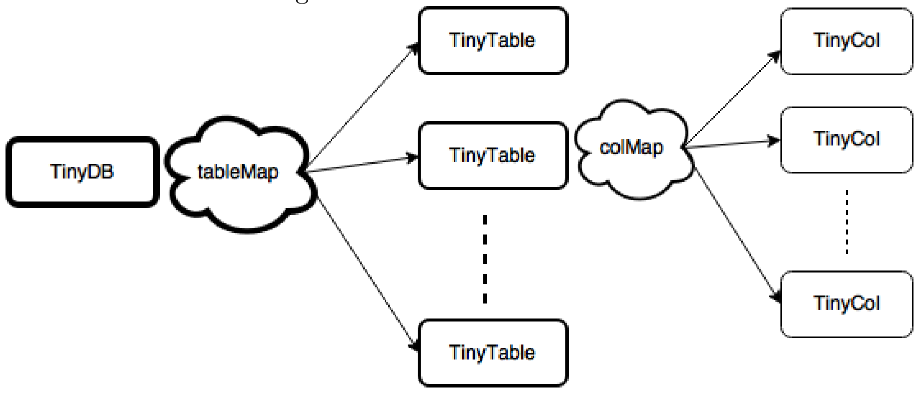
## 2 Database and API Implementation

### 2.1 Architecture

#### 2.1.1 DB Architecture

Below is a figure showing how our database is organized:

Figure 1: Database Architecture



In high-level, our storage system consists of several databases, each one of them is called a TinyDB, distinguished by its name and path. If our system is to support multiple users, we can assign each user a TinyDB. A TinyDB has functions for initialization, loading data and manage many TinyTables within it. A TinyTable is a table for column families in the TinyDB, which can communicate with its upper TinyDB and create new columns, each TinyTable maintain a map to store and manage columns within a TinyTable. TinyCol is the column in a TinyTable. We can take TinyCol the base unit of our storage. And TinyCol is actually an instance of ColTable with put() and get() operations to add and access data. Thus, we have connect the table with high-level database. The basic data unit for data manipulation is data entry, which is a key-value pair.

#### 2.1.2 ColTable Architecture

Our ColTable takes idea from LevelDB. Data store is organized heirarchically with Memtable and different levels of SStables. SStable size in different levels are identical and capcity of each level is ten times larger than capacity of the previous. There are two major condiserations in this design. One is that different levels of storage scales better and is easy for future optimizations like sharding. The other one is to improve read performance. Recent data will be put into memtable then to upper level sstables and gradually be pushed to lower levels so that only upper level SSTables will be looked up during a read query considering that recent pushed data are typically accessed more frequently. Below is some implementation details about some important parts:

SSTable. SSTables are divided into three groups of blocks. The first group is header block with data blocks numbers and Bloom Filter of 2048 bytes stored in it. The second group is meta data block with the first and last key of each data block stored in it. The third group is data blocks storing the all compressed key-values. Size of each block is 4K and most store operations are finished in block and write level.

Compaction and Compression. Our compaction strategy is based on level design. Minor compaction will be conducted automatically when level-1 is full and compaction results will pushed to level-1 with level-1 cleared after minor compaction. As for major compaction, the same compaction interface will be for lower level SSTables except that number of SSTables will be involved and it will be scheduled. Efficient block wise compression is conducted before data is put into each data block by serializing our BlockEntry.

BlockCache and BloomFilter A BlockCache is managed by the each ColTable for caching blocks from different SStables. Later test results will show that block-wise cache will dramatically improve read performance. A FIFO cache algorithm is maintained for simplicity. A bloomfilter is also created for each SSTable to improve read performance. The bloomfilter stays in memory and will also be dumped to the header block of each SSTable.

### 2.2 API

We also develop some API for data operations. To make them more convenient for user that are familiar with traditional query language like SQL, we define the grammar similar to SQL's. Up to now we have implement API for create table, insert data entry and select specific data from target table.

For an input command to a TinyDB for data query, we first check if the command is valid. If the command is unknown to our grammar, we can't parse it. The checking method is to compare the first word of the command with our defined grammar. For an INSERT command, we add data to specific columns according to parameters in command, for a SELECT command, we will define scale according to FROM parameters, make constraints according to WHERE parameters and search for data in TinyCol.

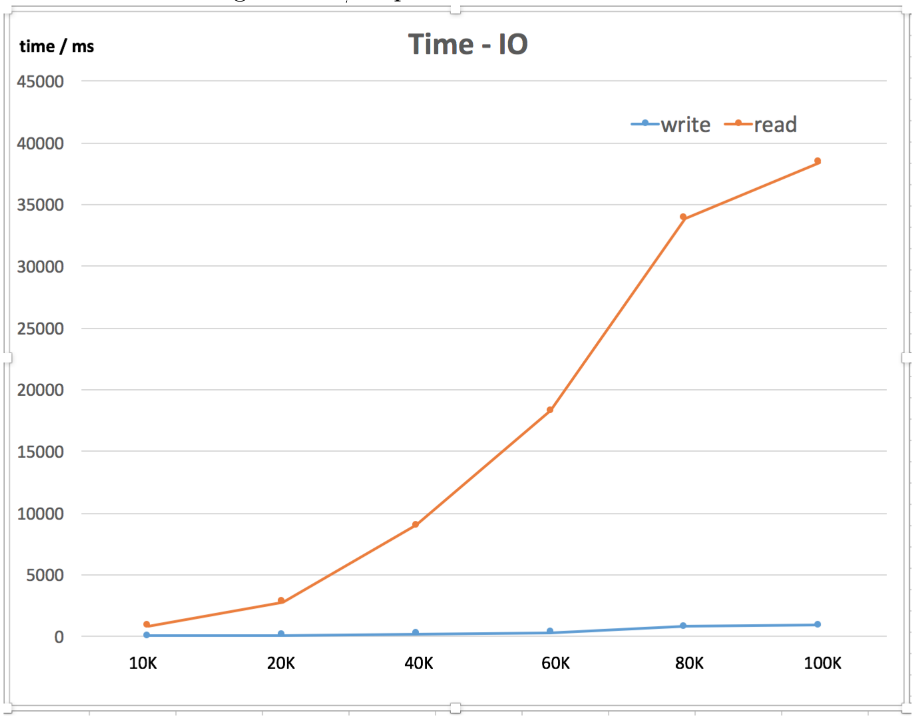
For lower level ColTables, our API is as simple as possible because we want to make it transparent and easy to use. Simple put() and get() operations are provided so that upper level will not need to care about details about store.

## 3 Test Result Rand Performance Analysis

For ColTable, base stress test about IOPS is conducted. Random strings of lengh 1 to 8 is generated and R/W performance with no block cache is measured as below:

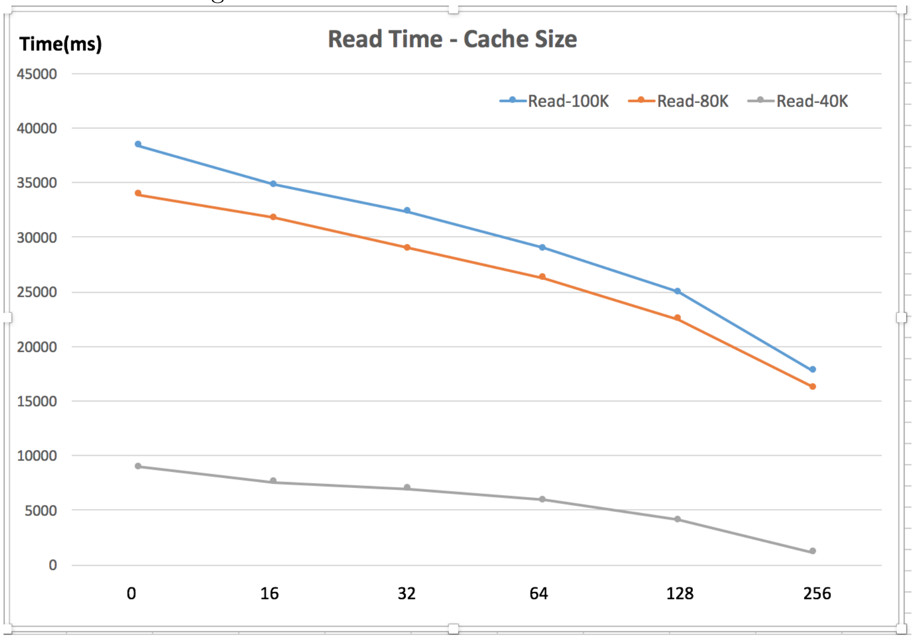
As can be seen from the figure, write performance is much better than read performance. 100,000 writes can be finished within one second. However, reads can be 40 times slower than

Figure 2: R/W performance with no cache



writes. This suggests that we should use block cache to improve write performance. Below is the read performance of using different numbers of block cache:

Figure 3: Write Performance of different cache



As can be seen from the figure, with increasing cache size read performance improves greatly(IOPS almost doubles). Here we are using a simple FIFO cache strategy without hot data. It can be expected that with a better cache strategy like LRU caching on hot data better read performance can be achieved.

For API test we do a query test and find that our INSERT and SELECT operations work well as we expected.

## 4 Future Work

Up to now we have complete most basic structures for a NoSQL database and make the data storage and access operations works as we expected. However, out read performance is still not satisfying. Future work will focus on improving read performance.