**Cover Page**

Data Structures and Algorithms Final Project Report

Date of Submission : 18/06/2020

Class : L2AC  
Course : COMP6571 - Data Structures and Algorithms

Group Members:

Vincentius Gabriel Tandra 2301894804

Ravel Tanjaya 2301890320

**Introduction:**

Our project mainly focuses on finding best data structure for storing specific non uniform data inside a nosql database. We chose this topic because we felt like it was a problem that we could apply to the real world where the market for the nosql database continues to expand.

**Problem Description:**

One of the ways to store non uniform data is by using a nosql database because this kind of database does not use a predefined structure which means all of the data that is stored inside the database may be unique.

In recent years, the nosql paradigm of databases’ popularity is increasing greatly, this is caused by the fact that data nowadays is not structured and uniform enough which means that storing this data on a traditional RDBMS (relational database management system) may not be efficient and that’s why nosql database is a very hot topic these days with a lot of nosql paradigm emerging on the market, like redis, mongodb firestore etc.

With the large variety of nosql paradigms currently on the market, users may not be able to deduce the most efficient one out of the bunch. This is why we will choose a certain condition and then evaluate the best paradigm for that condition.

**Condition:**

    In this project, we will be evaluating which data structure that is best suited for a food delivery application using a benchmarking applicatoin (written in JavaScript object format):

|  |
| --- |
| data = {     Customer: [         Budi: 128000,         joko: 50000,     ],     Orders: [         Fsdfkkjgkfdgjfkd: {             Food: [                 "pizza", "donut"             ],             Price: 12500,             status: "waiting"         },         fsdagfdgfdgfdhh: {             Food: [                 "pizza", "donut"             ],             Price: 12500,             status: "waiting"         },     ] } |

**Data Structures Chosen:**

For the problem we have chosen, we had to narrow down to a selection of various data structures. In this case, we were to use structures that could be used to implement a nosql database. These included trees or hash based structures. We were able to test many structures in the limited time we had and were able to evaluate the best structure from the group.

During the time we took to develop an app for the project we used the previous condition to benchmark a number of data structures to find the most effective one for our implementation. For our tests we were testing the insertion and deletion functions of each structure

Data Structures Tested:

* B-Tree
* Hash Table
* Cuckoo Filter
* Hash Tree

Btree: A B-Tree is a variation of the binary search tree that allows nodes to have more than two children, it is a self-balancing data structure where all the children are on the same level. It maintains sorted data sand allows insertions, deletions and searches. Some advantages of using B-Trees include sorted keys for traversing, balances the index through a recursive algorithm, it does this through the movement and splitting of elements. However, B-Trees are also quite inefficient since deleting and inserting elements involves lots of movement between the key/values.

It is a structure that is commonly used for data base indexing and that is why we chose to test it. Its average time complexity for its insertion and deletion algorithms in Big O notation are O (log n).

Hash Table: A hash table is a data structure which stores data in an associative manner. In a hash table, the data is stored in an array format, where each data value has its own unique index value. The data inside a hash table is stored in key value pairs. It uses a technique called hashing to generate the indexes where these key value pairs are then stored. An advantage of using a Hash Table is that they are efficient for inserting and searching data. However, when running into collisions they become quite inefficient.

Many databases now implement the hash table for database indexing, we decided to test it to see why it is such a popular choice for handling data. Its average time complexity for its insertion and deletion algorithms in Big O notation are O (1).

Cuckoo Filter: A cuckoo filter is a data structure that is typically used to check whether an item of data is a part of a set. A cuckoo filter is an optimization of the bloom filter, a similar structure but one that by standard, does not implement deletion. A cuckoo filter uses a 4-way set-associative hash table based on cuckoo hashing to store the fingerprints of all items. An advantage of using a cuckoo filter data structure over other structures such as a hash table is that it has a space advantage.

Structures such as a cuckoo filter are useful in processing very large sets of data, similar structures such as a hash map or hash table could be used but it wins out here due to the space advantage. We tested the filter to see whether its advantages mattered for our benchmark. Its average time complexity for its insertion and deletion algorithms in Big O notation are O (1).

Hash Tree: A hash tree is a hash-based data structure composed of a tree of hashes where the leaves of the tree are hashes of data blocks. It is a generalization of a data structure known as a hash list. One of its advantages is that it is very useful in verifying data.

A good example of where a hash tree is used is in the digital cryptocurrency, Bitcoin where transactions using bit coin are stored in blocks known as a blockchain, leaves of the hash tree are typically hashes of single blocks, and is used to verify changes within the blockchain. We tested this structure in order to see whether or not it could compare to the other structures we were testing. Its average time complexity for its insertion and deletion algorithms in Big O notation are O (log2n).

**Benchmarking Application:**

Each of these data structures were tested in a benchmarking application we developed in order to compare the relative speeds of each of their functional algorithms. In our case, these are their search and delete algorithms. The application functions as follows:

It uses a Nodejs based build system to automate the build process, a hyperfine based benchmark system and Random Data generation based on faker.js and was developed under a unix based system.

**NodeJS build system**

Currently to test the speed and efficiency of the data structure nodejs is used to automate the whole process including the building, generation of random data and benchmarking.

In order to use these scripts please make sure you have installed the following:

* NodeJs
* npm
* Hyperfine

To use this script please first, change directory to the benchmark folder with:

cd benchmark

the use npm to install all of the required depedency.

npm install

to install all required npm depedencies.

**Automatically build all of the codes**

To automatically build all of the available codes, we can run this command:

npm run build

The script will change directory to each of the codes and will use gcc to build all files with c++17 standard.

**Generate the random data**

The random data that is generated is of JSON(Javascript object notation) these data will later be loaded by the programs and used as a sample data to measure the speed of each data structure.

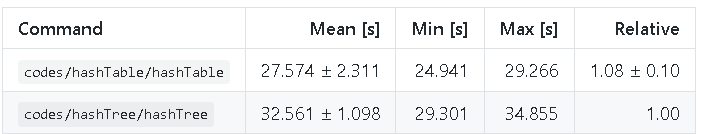
To generate/regenerate these random data please run:

npm run generate

**Benchmarking**

In order to benchmark, please make sure you have already build for the latest changes and generate the json data,then run:

npm run benchmark

**Benchmarking Results**

After benchmarking each structure, we found the **hash table** to be the most efficient data structure for the choice. Here we see the hash table head to head with the hash tree and as expected from the differencess in time complexity, it beats it out in speed even while handling the large amounts of data. This means that the other structures such as the b-tree and the cuckoo filter would also be adequate but not as good as the hash table for the implementation we will be using in the final applicatoin.

**Final Application Demo**

**Demo Video**

**Github Link**

https://github.com/ravelgwong/nosql-cpp

Within this github link, all of our code is stored, this includes our main application, the benchmarking application, previous reports and other files which were used to create this project.

**Conclusion**