|  |
| --- |
| **Assignment Cover Letter**    **(Group Work)** |

**Student Information**: **Surname**  **Given Names Student ID Number**

1. **Tandra** **Vincentius** **2301894804**

2. **Tanjaya** **Ravel** **2301894804**

**Course Code**  **: COMP6571**  **Course Name**  **: Data Structure and Algorithms**

**Class** **: L2AC**  **Name of Lecturer(s)** **:** **Andreas Kurniawan**

|  |
| --- |
|  |

**Major**  **: CS**

**Title of Assignment** :

(if any)

**Type of Assignment**  **: Final Project**

**Submission Pattern**

**Due Date**  **:** 18/06/2020   **Submission Date**  **:** 18/06/2020

**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. Several structures were tested but there were some that we failed to implement.

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:

* Hash Table
* Hash Tree

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).

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.

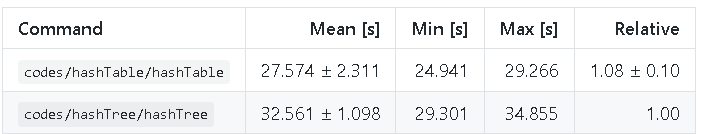
To generate/regenerate these random data please run:

npm run generate

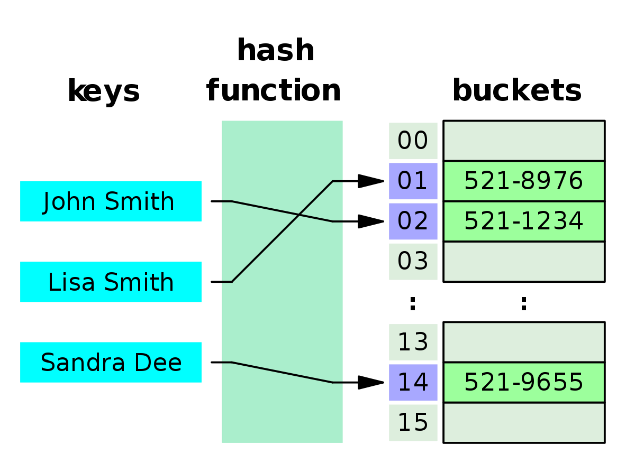
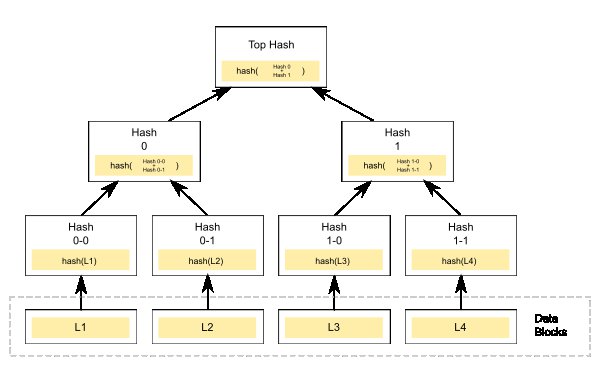
**Benchmarking**

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. So through these results we can see the hash table being faster but besides the differences in time complexity, we can attempt to figure out why this is exactly the case.



Here we see a respresentation of both structures, the hash table assigns a key value pair to a bucket where they are stored while a hash tree is structured similar to a binary search tree in which they have a branching factor of 2 (the number of children at each node). However, unlike a binary search tree where generating the root node is simple, with a merkle tree you would have to compute and store a hash at every node which adds up as the tree gets larger and larger. This combined with its branching factor makes the hash tree substantially slower than the hash table when handling large amounts of data.

One of the advantages a tree would hold over the hash table is its ability to store data in an organized and ordered structure. However, our use-case describes the best way to store non-uniform data in a nosql database and so, the non-ordered data makes this advantage less favorable for the hash tree and better shows why the hash table is faster. The hash table is a structure that functions relative to its function and only slows down when running into collisions which are always expected to happen, especially with larger data sets even with the ideal hash function. However, with proper collision resolution using open-addressing methods we are able to resolve these collisions so the hash table works at its best.

**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**