A HIGH PERFORMANCE CPU-GPU DATABASE FOR STREAMING DATA ANALYSIS

by
ANES ABDENNEBI

Submitted to the Graduate School of Engineering and Natural Sciences
in partial fulfilment of
the requirements for the degree of Master of Science

Sabancı University
August 2020

A HIGH PERFORMANCE CPU-GPU DATABASE FOR STREAMING DATA ANALYSIS

Approved by:

Asst. Prof. Kamer Kaya .....
(Thesis Supervisor)

Asst. Prof. Erdinç Öztürk .....

Assoc. Prof. Tolga Ayav .....

Date of Approval: August 31, 2020

ABSTRACT

A HIGH PERFORMANCE CPU-GPU DATABASE FOR STREAMING DATA ANALYSIS

ANES ABDENNEBI

Computer Science and Engineering, Master's Thesis, 2020

Thesis Supervisor: Asst Prof. Kamer Kaya

Keywords: OLAP Databases, CPU, GPU, Big Data, Bloom Filter

The outstanding spread of database management system architectures in the last decade, plus the increasing growth, volume, and velocity of the data, which is known nowadays as "*Big Data*", are continuously urging researchers, businessmen and companies to build robust and scalable database management systems (DBMS) and improve them in a way they adjust smoothly with the evolution of data.

On the other hand, there is a tendency to support the conventional processing units (PUs), which are the *Central Processing Units* (CPUs), with additional computing power like the emerging *Graphical Processing Units* (GPUs). The research community has accepted the potential of vigorous computing power for data-intensive applications. Several research studies were conducted in the last years that ended up in building remarkable DBMSs by integrating GPUs and using them according to different workload distribution algorithms and query optimization protocols. Thus, we try to address a new approach by building a hybrid columnar-based high-performance database management system calling it DOLAP which adopts the *Online Analytical Processing* (OLAP) infrastructure. Distinctively from previous hybrid DBMSs, our database, DOLAP, depends on Bloom filters while performing different operations on data (ingesting, checking, modifying, and deleting). We implement this probabilistic data structure in DOLAP to prevent unnecessary memory accesses while checking the database's data records. This method is proved to be useful by reducing the total running times by 35%. Moreover, since there exist two main PUs with different characteristics, the CPU and GPU, a workload distribution model that effectively decides the query's executing unit at a time T should

be defined to improve the efficiency of our system. Therefore, we suggested 3 load balancing models, the *Random-based*, *Algorithm-based* and the *Improved Algorithm-based* models. We run our tests on the *Chicago Taxi Driver dataset* taken from *Kaggle* and among the 3 load balancing models, the improved algorithm-based model demonstrates its effectiveness in well distributing the query load between the CPUs and GPUs where it outperforms the other models in nearly all the test runs.

ÖZET

AKIŞ VERİ ANALİZİ İÇİN YÜKSEK BAŞARIMLI CPU-GPU VERİTABANI YÖNETİM SİSTEMLERİ

ANES ABDENNEBI

Bilgisayar Mühendisliği, Yüksek Lisans Tezi, 2020

Tez Danışmanı: Dr. Öğr. Üyesi Kamer Kaya

Anahtar Kelimeler: OLAP veritabanları, CPU, GPU, Büyük Veri, Bloom Filre

Günümüzde Büyük Veri olarak bilinen verilerin artan hacmi ve hızı, araştırmacıları, analistleri ve şirketleri veritabanı yönetim sistemlerini sağlam, ölçeklenebilir, ve veri ile sorunsuz bir şekilde uyum sağlayabilecek şekilde oluşturmaya teşvik etmektedir.

Öte yandan, Merkezi İşlem Birimleri olan geleneksel işlem birimlerini (PU), Grafik İşlem Birimleri gibi ek bilgi işlem gücüyle destekleme eğilimi vardır. Araştırmacılar, veri yoğunluklu uygulamalar için güçlü bilgi işlem gücünün potansiyelini kabul etmektedirler. Son yıllarda, GPU'ları eldeki sisteme entegre ederek ve bunları farklı iş yükü dağıtım algoritmalarına ve sorgu optimizasyon protokollerine göre kullanarak dikkat çekici DBMS'lerin oluşturulmasına neden olan çeşitli araştırma çalışmaları yapılmaktadır. Bu nedenle, Çevrimiçi Analitik İşleme altyapısını benimseyen DOLAP adını verdiğimiz, hibrit, sütun tabanlı yüksek performanslı bir veritabanı yönetim sistemi oluşturarak yeni bir yaklaşımı ele almaya çalışıyoruz. Önceki hibrit DBMS'lerden farklı olarak, veritabanımız DOLAP, veriler üzerinde farklı işlemler gerçekleştirirken (alma, kontrol etme, değiştirme ve silme) Bloom filtreleri kullanmaktadır.

Veritabanının veri kayıtlarını kontrol ederken gereksiz bellek erişimlerini önlemek için bu olasılıklı veri yapısını DOLAP'ta uygulamaktayız. Yaptığımız deneylerde, toplam çalışma süresini %35 azaltarak kullanışlı olduğunu kanıtladık. CPU ve GPU olmak üzere farklı özelliklere sahip iki ana PU üzerinde sistemimizin verimliliğini artırmak amacıyla, sorgunun yürütme birimine etkin bir şekilde karar veren bir iş yükü dağıtım modeli tanımladık. *Rastgele tabanlı*, *Algoritma tabanlı* ve *Geliştirilmiş*

Algoritma tabanlı modeller olmak üzere 3 yük iş dağıtım modeli önerdik.

Testlerimizi *Kaggle*'dan alınan *Chicago Taxi Driver* veri kümesi üzerinde gerçekleştirdik, Bu deneylerde 3 yük dengeleme modeli arasında, iyileştirilmiş algoritma tabanlı model, sorgu yükünü CPU'lar ve GPU'lar arasında iyi bir şekilde dağıtmadaki etkinliğini kanıtlamakta ve neredeyse tümünde diğer modellerden daha iyi performans göstermektedir.

ACKNOWLEDGEMENTS

This work is supported by TÜBİTAK project number 118E044.
I offer my sincere thanks and appreciations to my family, to Manel who supported me during my studies for the past two years, to my professor Kamer hoca for supporting me all the time and guiding me through, to Barış bey from IT for being so helpful with the technical problems that we have faced.
I also offer my special thanks to my teammate Anıl who worked hard and offer his time to reach several achievements in this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

The world is witnessing data explosion in several technological fields, especially on the streaming data for which the flow speed is so fast, and the data volume is enormous. For example, in Internet-of-Things (IoT), the data is exchanged and widely expanded between smart devices, vehicles, sensors, beacons, and other millions or even billions of devices across networks. Other fields where it shows the massive volume and great velocity of data are recommendation systems, social networks, stock markets, business transactions, and data analytics in Artificial Intelligence (AI). All these fields have a centric and challenging problem which is analyzing the streaming data in a fast and most efficient way. Successful businesses or accurate scientific research have to possess the requirements to cope with the speed and the continuously changing data to be able to extract data reports, customer behaviors, market tendencies, and even feed well-pre-processed data to Machine Learning models for various data and business-related purposes. Consequently, processing this fast and voluminous data faces two main challenges, finding the proper scalable data management framework and choosing a suitable, powerful and efficient system that handles the considerable workload of data analysis.

Concerning the data management infrastructure, the burden has become heavier on the relational database management systems (DBMS) in handling fast and voluminous streaming data, a non-relational database management system has become a strong candidate for dealing with terabytes or more of data records in a fast pace. Here, non-relational database management systems are advantageous over the relational one because of the scalability property. The relational DBMS failed to keep up with the horizontal growth rate of data size where the non-relational DBMS does, additionally, the latter can cope with the unstructured data like social media originated data, satellite imagery, data generated from collaboration software and phone recordings, sensor data gathered from traffic, temperature or weather, and the list is still wide open since it is expanding massively.

Distributed solutions may not be a perfect choice here because of the communication overhead that increases latencies while responding to the queries of the users.

Therefore, we focus on building a centralized solution and build a hybrid database management system called *DOLAP*. By hybrid, we mean introducing a columnar structure for the database management system for analytic queries inspired from the Online Analytical Processing (OLTP) and using the combination of CPU (central processing unit) and GPU (graphical processing unit). With sufficient storage capacity, we create a powerful centralized data analysis system which is cheaper compared to previously developed tools and efficient in delivering accurate readings and statistics about the data under the scope. Moreover, since this system is a hybrid one (using both CPU & GPU in parallel), there should be a well-established workload balancing between different processing units to guarantee that the query load is equally distributed between the CPU and GPU. Thus, we propose a random model, an algorithm-based model, and an improved version of it to achieve the best performance. Supporting the CPUs with the intensive computation capacity of the GPUs does not necessarily mean that GPUs will run queries faster than the CPUs. There is a substantial drawback represented in the communication overhead between the host and device. Whenever a user sends a query to get executed on the GPUs, the system moves the selected data columns to the GPU's memory and that costs a critical amount of time, therefore, the proposed load balancing model should be adaptive to this important spot.

Another substantial part of this system is the space-efficient data structure called *Bloom filter* which is a compact structure that tells whether an item e belongs to a set S or not. The role of the Bloom filter in *DOLAP* is to minimize the number of memory accesses whenever a query is executed, so one may check the existence of a record in the database by checking its existence first if it does exist then the data will be retrieved from the database.
Our database management system is built based on the OLAP (*Online Analytical Processing*) architecture using a columnar structure -introduced in the 3^{rd} chapter- which makes this system a strong element in the big data world. Other hybrid systems that made it out to the market like *Kinetica* (Kinetica, 2018) based on the OLAP architecture which uses both CPU and GPU power for analyzing datasets and running SQL queries. Another tool that runs SQL queries is *SQream* (SQream, 2014) designed to analyze streaming data in seconds using the GPU computation power to give faster data analyzing times. Another GPU-optimized database is *BlazingSQL* (BlazingDB, 2015) which is an open-source SQL engine that provides a python library to create database tables and run SQL queries on raw data. Nevertheless, our designed DOLAP is intended to use smart load balancing methods to evenly distribute the workload on the CPUs and GPUs. Additionally, it has a query optimization part which improves the system's performance. Moreover, DOLAP

uses data sketches like the Bloom filter membership queries. DOLAP has the option of integrating data sketches into any part of its system, in query answering and optimization, in Load Balancing and data manager modules.

In the next section, we will give a brief background about the processing units (GPU & CPU), the Bloom filter data structure, and its integration in our system, plus the used data visualization tool Grafana. In the third chapter, we will introduce the DOLAP structure, next in Chapter 4, we introduce the query structure that we will be testing our system with, in addition to the query types that we chose to use during the querying processing of DOLAP. Chapter 5 is about the load balancer and the proposed models, schemes, and algorithms while in the 6^{th} chapter, all the tests, and results are introduced with a thorough study and comparisons between different models and setups.

2. BACKGROUND AND NOTATION

2.1 The Processing Units

As stated previously, DOLAP is built to benefit from both CPU and GPU architectures simultaneously, since the CPU cannot stand up to the high requirements posed by the expected huge number of queries to be answered. The graphical processing units are used to increase the performance of DOLAP and handle multiple concurrent users.

2.2 Graphical Processing Units (GPU)

A GPU is an accelerator that can perform rapid and intensive operations and reduces the burden on the central processing unit by performing heavy tasks. Moreover, GPUs are heavily used for training deep learning tools for AI applications and applying operations on high volume streaming data.
Even though GPUs are designed to perform basic computing tasks, they are more suitable for data-parallelism which best suits our requirements in building our database management system DOLAP. Thus, a platform is needed to facilitate the usage of GPU, and the recent most robust one is the *Compute Unified Device Architecture (CUDA)*, CUDA allows developers to program the GPUs using *C* or *C++* programming languages with additional keywords to be added to distinguish between the functions destined to be running on CPU and the ones to run on GPU. Here, we introduce the main terminology spots of CUDA which are mainly used during the development of DOLAP:

- *Kernel*: is a function written in host (CPU side) and ran on the device (GPU). It runs on multiple device threads.
- *Thread*: is the essence entity that runs a kernel, with its own memory space and registers.
- *Block*: it groups the threads into one memory space to run parallel tasks for better data mapping.
- *Streaming Multiprocessor (SM)*: is the unit forming the whole GPU structure, each SM contains blocks of threads and warps.
- *Warp*: it is a group of threads in a block, generally 32 threads in recent implementations. A warp is run by a streaming multiprocessor, and the latter can run several warps at a time.
- *Grid*: the grid is highest container that groups many thread blocks in dimensions up to 3.

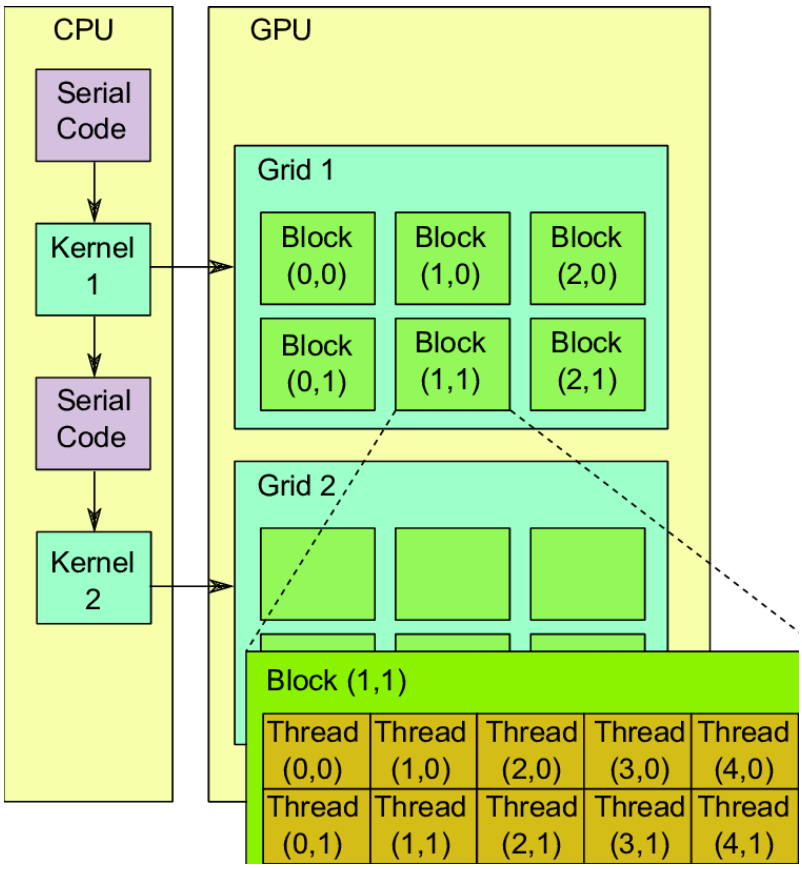


Figure 2.1 The GPU architecture illustration

DOLAP is a hybrid database management system, it runs the incoming queries on both host and device. Therefore, the GPU should also have the database either fully

or partially transferred to its memory if it fits, and the main memory is managed by the host. Hence, while DOLAP is running and receiving queries, some will be executed on host and others on the device. Sending the queries to the device requires also sending the related data blocks to the GPU's memory so the kernels would operate on those copies. However, the communication bus between the host and device form a shortcoming because of the low bandwidth rate. We tried to minimize the data transfers between CPU and GPU through some data management procedures which involve copying the data once and keeping the pointer preserved within the GPU's memory as long as it is still being used frequently, and delete it once it considered a cold data.

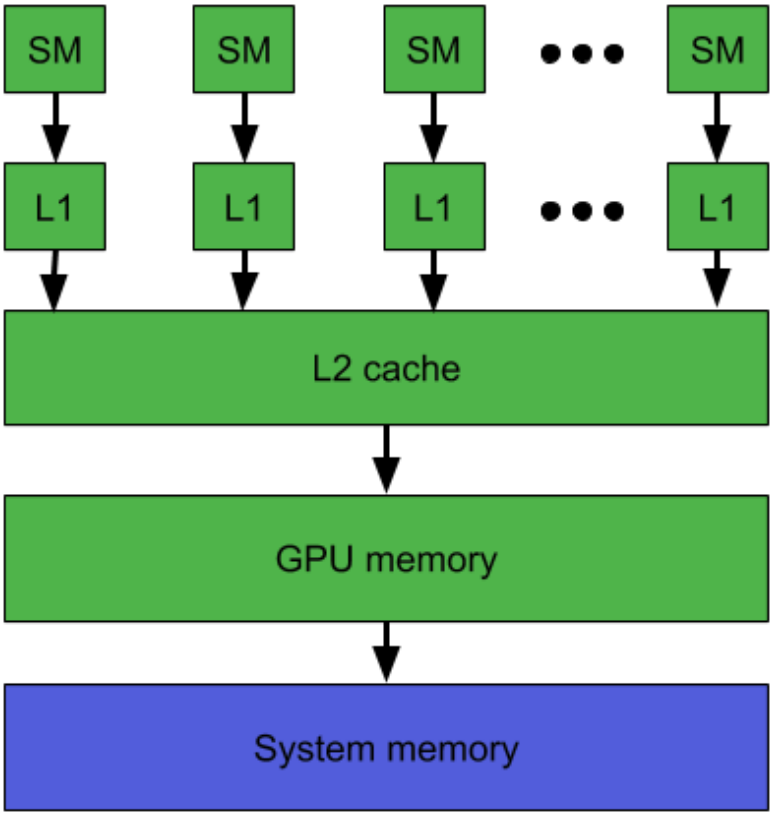


Figure 2.2 The GPU different memory levels

Figure 2.2 shows the hierarchy of the GPU's memory starting from the streaming multiprocessors' L1 cache memory and ending with the GPU memory. Here the GPU cache memory, L1 & L2, and the GPU memory are relatively smaller than the ones in CPU, however, their bandwidths are higher. It is important to notice that the L1 caches associated with each SM are not coherent which means that two different SMs may write on the same memory address without noticing the changes made by each other immediately.

2.3 Bloom Filter

The Bloom filter is a space-efficient data structure that answers elements membership queries. It was first introduced by (Bloom, 1970). A Bloom filter uses a bit vector `bf`[.] of size m and k hash functions. Each hash function takes an element e from the universal set $\mathcal{U}$ as an input and outputs a location in the bit vector. That is $h_i : \mathcal{U} \rightarrow \{0, \ldots, m-1\}$ for all $1 \leq i \leq k$. A BF has two main operations; *insertion* & *query*. Initially, all bits are set to 0. To *insert* an item x, the bit positions $h_1(x), h_2(x), \ldots, h_k(x)$ are computed and the corresponding bits, $\texttt{bf}[h_1(x)], \texttt{bf}[h_2(x)], \ldots, \texttt{bf}[h_k(x)]$ are set to 1. To *query* an item's existence, the hash functions $h_1(x), h_2(x), \ldots, h_k(x)$ are computed again. If all the corresponding bits for these positions are 1 BF returns *true*. Otherwise, it returns *false*.

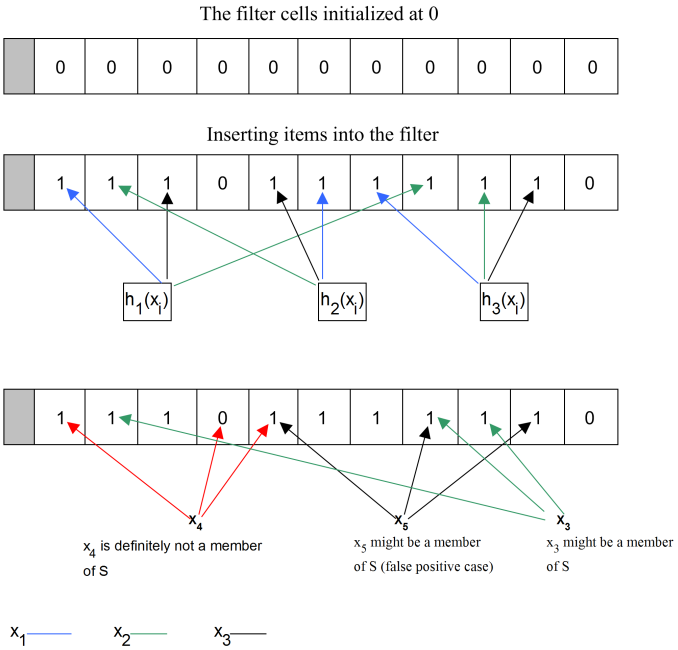


Figure 2.3 A toy BF with $m = 11$ and $k = 3$. Three items are inserted to the BF; the hash function outputs for x_1 are 0, 5 and 6. For x_2, they are 2, 4 and 9 and for x_3, they are 1, 7 and 8. Two extra items are queried; x_4 with hash positions 0, 3 and 4 and x_5 with hash positions 4, 7 and 9.

A toy BF with $m = 11$ bit vectors and $k = 3$ hash functions is shown in Figure 2.3. To query an item, the corresponding bit positions are checked; for x_4, these bits are $\texttt{bf}[0], \texttt{bf}[3]$ and $\texttt{bf}[4]$. Since $\texttt{bf}[3]$ is 0, $x_4 \notin S$. For x_5, although all the corresponding bits are 1, this is a false positive response (due to a hash collusion at $\texttt{bf}[8]$). Querying the item x_3 would return "true" after checking that the bits in positions 1, 7 and 8 are all set to 1 and therefore, x_3 is not in the filter.

One of the main metrics to evaluate the effectiveness of a BF is the *false positive probability* which can give information about how accurate the BF is. Assuming the hash functions uniformly distributes the items to their range, the probability having a given bit equal to 0 is

$$p = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} \tag{2.1}$$

where m is the vector size, n is the number unique items in the stream, and k is the number of hash functions used. Starting with p, the false positive probability is

$$\epsilon = (1-p)^k \approx \left(1 - e^{-kn/m}\right)^k. \tag{2.2}$$

2.3.1 Parameters

To ensure that the filter performs in the best manner in DOLAP, we tuned the filter's parameters (k, m, and n) so we obtained the best number of hash functions k and filter's size m. The method followed to achieve the definitive best parameters of the Bloom filters is based on selecting the optimal number of hash functions with the most convenient bit-vector size such that we should avoid high number of hash functions which can make our Bloom filter slow, besides, the bit-vector size should not be too small to avoid increasing the false positive probability. Therefore, we used the mathematical Equation 2.3 that links k, m, and n:

$$k = (m/n) \times ln(2) \tag{2.3}$$

We wrote a C++ code to make several trials on the Bloom filter's size and its optimal hash functions number, it was *trial and error* approach. We try several values of k and m, we compare the error estimates of each run then we picked the best parameters that return the least false positive probability. Similarly, we can also retrieve the best m and k values according to a predefined false positive probability.

2.4 Grafana

Grafana is a visualization and data analysis tool. It provides the possibility of connecting time-series databases to a graphical user interface to display statistical graphs and tables. Moreover, it provides the service of querying the database through an Adhoc filter variable (Grafana, 2014). Figure 2.4 is a screenshot of the used Grafana dashboard while querying DOLAP. Different dashboards might be used to query the database after connecting to a datasource which is provided by DOLAP.

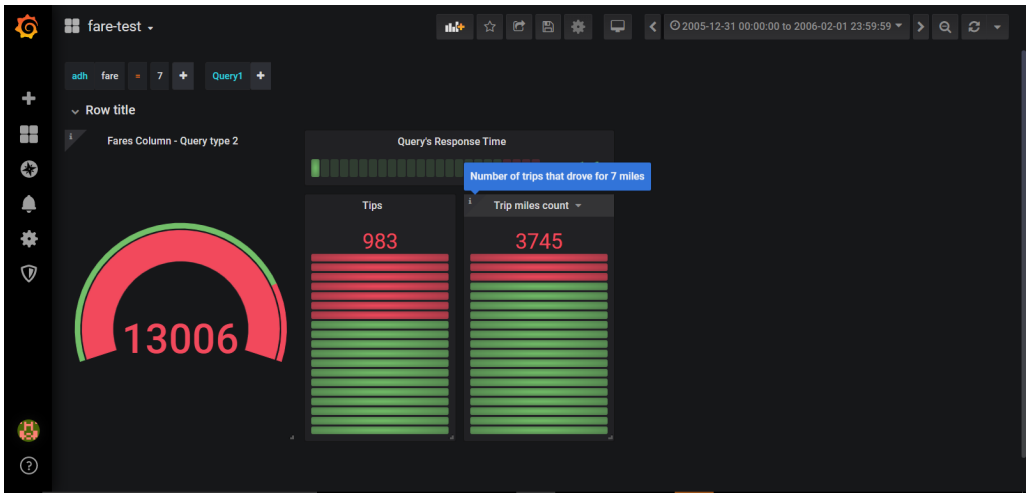


Figure 2.4 A Grafana dashboard

2.5 OLAP

OLAP is an acronym for *Online Analytical Processing* (OLAP, 1993) , and the benefit of this technology is what it provides for business-related data analysis, discovering trends and insights from data resulted from users transactions, and designing sophisticated data modelings. OLAP is a part of a wider category of Business Intelligence where it allows the analysis of multidimensional data that is gathered from several resources like databases. It groups them in a way that it facilitates extracting readings, behaviors, building business models, adapting to new customers' changes in actions, business performance evaluating and financial reporting.

2.5.1 OLAP in DOLAP

DOLAP is a hybrid database management system which allows users to ingest datasets and query them in various ways. Moreover, it also gives the option of running ad-hoc queries on the database in a multi-dimensional fashion which is based on the OLAP structure. This method allows the system to build insights out of the users' queries, extracting the hot and cold data, allows for Machine Learning models to be built and trained upon the data collected from users' interaction with DOLAP.

2.6 Database Management Systems

A database management system (DBMS) is a software having functions that form an interface that eases the interactions between the users and the database including ingesting, modifying, deleting, and defining data. In other words, the DBMS should contain all means of data manipulation and organization. The DBMS should function in a way that it receives the command directly from the administrator and direct them to the system to perform the desired functions. There are various database management systems, in-memory database management systems, cloud-based, NoSQL, and columnar DBMS.

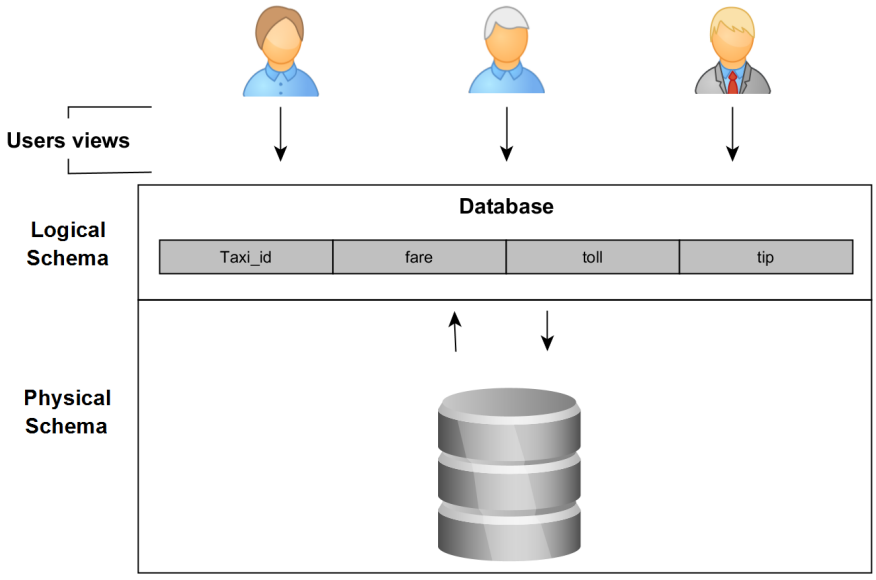
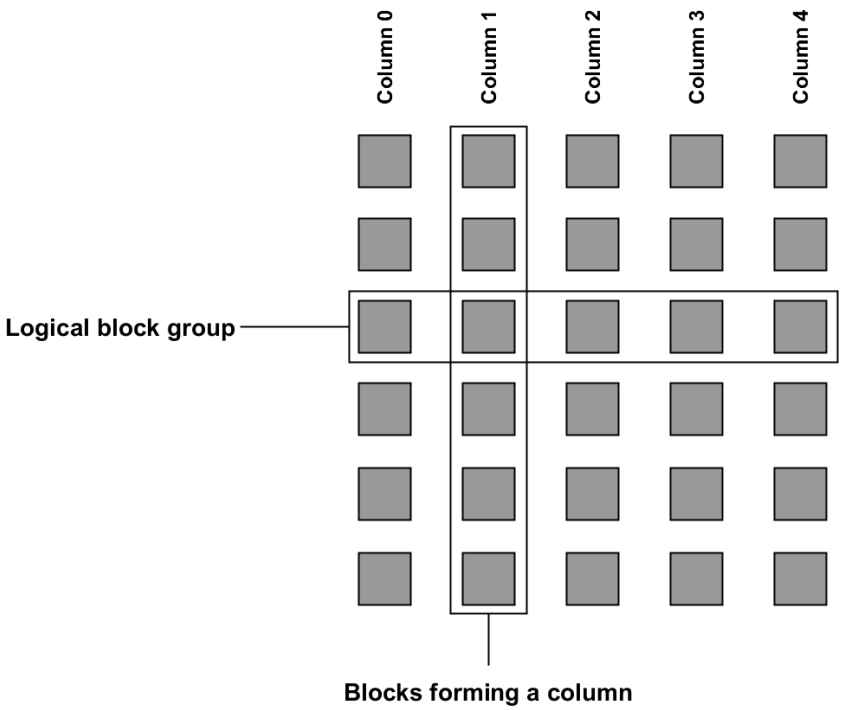


Figure 2.5 A DBMS abstract schema

3. DOLAP STRUCTURE

In our high-performance database management system, getting the searched records from DOLAP in the fastest execution time is one of the aims of this project. Therefore, one of the aspects that was focused on is the data placement in the memory. Figure 3.1 illustrates the organization of the database records into blocks where vertically these blocks form a column of the same type, and horizontally it forms a *logical block group* which refers to one record block in the database. This scheme provides a spatial locality between the blocks of different columns residing in a close storage location in the memory, thus, using this scheme would guarantee returning data of one record faster than other data placement schemes.

Figure 3.1 The data holding scheme for DOLAP



3.1 Ingesting Data to DOLAP

Loading the data into DOLAP occurs in parallel by benefiting from the multi-core architecture of the CPU where each core is responsible for one column at a time. First of all, the number of records that will be inserted into one block should be defined. After that, each CPU core will start ingesting the data into the blocks in parallel, in other words, all the cores (the number of cores depends on the number of the columns) will be inserting records to data blocks in parallel. For example, assume that there are 20 columns then in one iteration, there are 20 blocks getting filled in parallel. Algorithm 1 shows the steps of the ingesting process. This method of ingesting allows later on a smooth Bloom filter checking. Whenever a query comes to the database, and before accessing it, the corresponding Bloom filter of each block will be queried first to check the membership of the searched value(s). If the filter's answer was negative then there is no need for the query to be executed. However, if the value(s) does exist in the Bloom filter belonging to its block then the query will proceed on to retrieve the matching results. It is substantial to note that the positive answers of the Bloom filters might be false positives which means that the value(s) do not exist in the block(s) but still returning a positive answer about their existence.

The next section shows the scheme of a data block and its Bloom filter such that accessing a specific Bloom filter should pass through the corresponding block.

Algorithm 1: Ingesting data to DOLAP

```
Num_Block_Groups = number_of_total_groups;
Num_records = BLOCK_SIZE;
database = database_address;
record = array[Num_records];
for (d = 0; d <Num_Block_Groups) in parallel do
    for (l = 0; l <Num_records) do
        record[l] = VALUE;
    for (i = 0; i <Num_records) do
        database.block[d][i].insert(record[i]);
        database.block[d][i].bloom_filter.insert(record[i]);
```

3.2 Bloom filter in DOLAP

As stated previously, the Bloom filter is used to reduce the burden on the memory and exclude unnecessary accesses. Therefore, we implemented it in our database management system in a way that each block has its Bloom filter so while running a query, the blocks records won't be accessed unless the corresponding Bloom filters return positive responses. Figure 3.2 demonstrates the designed structure of the data block and its filter. Still, there is a possibility that a block is accessed but returns nothing as a result of the query, and it may happen due to the probabilistic property of the Bloom filter structure. One possible solution for reducing this possibility is by increasing the filter's size. However, it presents another bottleneck that manifests in the additional use of memory which is an undesirable trade-off in this case.

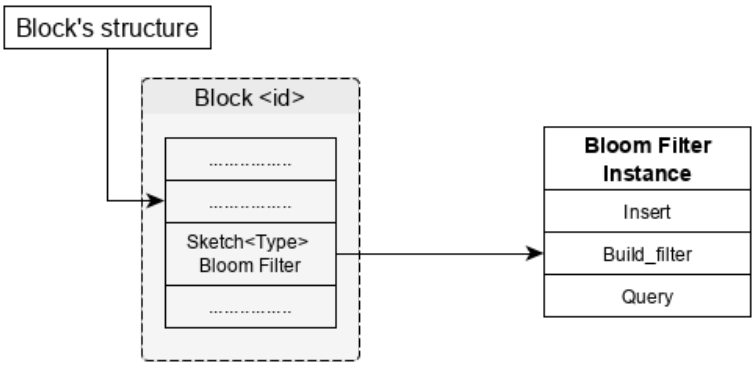


Figure 3.2 A Bloom filter for each block

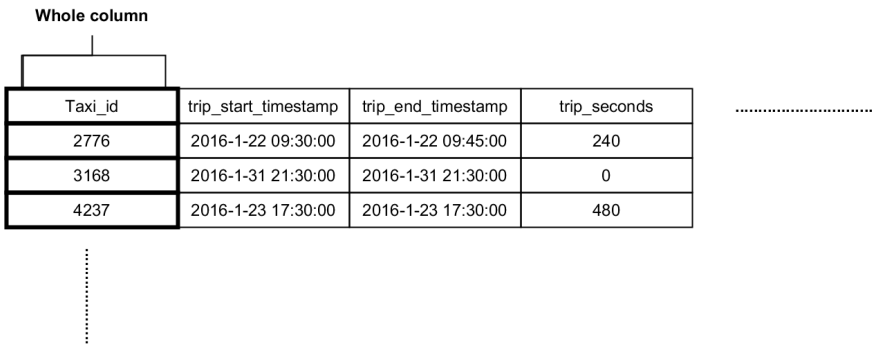
4. QUERY PROCESSING

The users of DOLAP would be interacting with our DBMS through a visualization tool which is Grafana in this project. Therefore, and for performance measuring purposes, we have set two main query types, *Simple query* & *Lucene query*. The users, for different purposes, may run either of these query types, they may want to display a whole database column or extract specific records that match the entered query. Examples of these queries will be given in the next sub-sections.

4.1 The simple DOLAP query

This type was the first one to be tested by us while implementing the DOLAP querying system, and we used it frequently for testing purposes, like stress testing. The simple query returns all the records of a specified column. For example, if the *taxi_id* column is requested then all taxi ids will be recovered and returned to Grafana to display it.

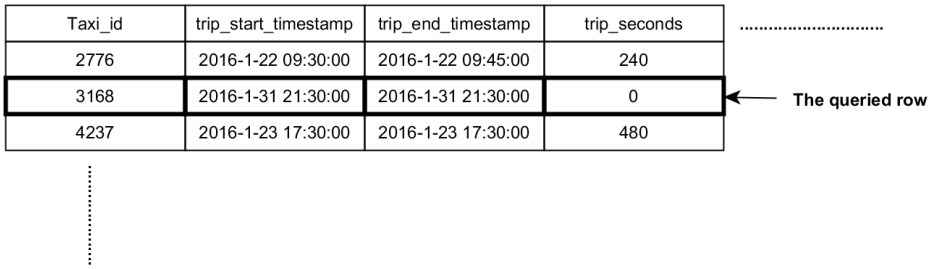
Figure 4.1 Illustration of a database column getting queried by a simple query



4.2 Lucene Query

Lucene query is more dedicated to user's needs, each user would request different data records from DOLAP at any time t for various purposes. Here, the query is collected from the visualization tool as a *JSON* file where the target columns and the entered values are forwarded to the C wrappers to run the intended functions either on the host or the device.

Figure 4.2 Illustration of a database row getting queried by a lucene query



Figures 4.1 and 4.2 show the way each query type perform when called by a user, however, it is substantial to note that in a Lucene query, there might be more than 1 result, it varies between 0 to the number of the database's rows. For the *Lucene queries*, we specified two sub-types for testing purposes, namely *query type 2* & *query type 3* where the *simple query* is labeled as *query type 1*. This division of query types is chosen for the sake of showing performance differences on both host and device. Equation 4.1 is an example of a *Lucene query* including only one column, *fare*, which represents a query of type 2.

$$fare = 7 \tag{4.1}$$

Where Equation 4.2 is an example of a more complicated query that asks for a list of drivers who got paid 7$ and tipped with 2$ where the payment was in cash.

$$\begin{aligned}&(fare = 7) \wedge (tips = 2) \wedge (extras = 0)\\ &\wedge (payment\_method = Cash)\end{aligned} \tag{4.2}$$

4.3 Querying Algorithms

Here, we present the simplest query response algorithms used to retrieve results from DOLAP for each query type varying from 1 to 3.

Algorithm 2: Algorithm of the simple query

```
Result: Results array
Num_Block_Groups = number_of_total_groups
Num_records = BLOCK_SIZE (ex: 1024)
col_num = total number of columns
target_col_num = the column to be queried
database = database_address
result_array = Initialize a new array
for (d = 0; d <Num_Block_Groups) in parallel do
    target_block_index = d × col_num + target_col_num
    for (l = 0; l <Num_records) do
        result_array[i] = database[target_block_index].getLine(i)
    end
end
```

Algorithm 3: Algorithm of the query of type 2

```
Result: Results array
Num_Block_Groups = number_of_total_groups
Num_records = BLOCK_SIZE (ex: 1024)
col_num = total number of columns
target_col_num = the column to be queried
database = database_address
result_array = Initialize a new array
Value = the value to be searched
In parallel, each thread will be responsible for a group of blocks, so each thread
 has a starting and ending group
start_group
end_group
for (b = start_group; b <end_group) in parallel do
    for (l = 0; l <Num_records) do
        returned_value = database[filter_block_index].getLine(i)
        if Value == returned_value then
            result_array[l] = returned_value
    end
    Update start_group for each thread
    Update end_group for each thread
end
```

To retrieve a whole column from DOLAP, Algorithm 2 uses k threads where each

thread is responsible for getting the data records from several blocks in parallel. The number of total logical groups and the block size are defined, these values depend on the size of the database and the number of the used threads in processing the query.

Algorithm 3 shows how a *Lucene query* of type 2 is executed on DOLAP. Since our database management system's structure divides the columns into blocks, and each row is a logical group of blocks, and no matter what is the maximum number of threads used in answering the query, we assigned a thread to a group of blocks belonging to one column (since it is a type 2 query) to assure the parallelism and fast response times.

Algorithm 4: Algorithm of the query of type 2 with Bloom filter

```
Result: Results array
Num_Block_Groups = number_of_total_groups
Num_records = BLOCK_SIZE (ex: 1024)
col_num = total number of columns
target_col_num = the column to be queried
database = database_address
result_array = Initialize a new array
Value = the value to be searched
In parallel, each thread will be responsible for a group of blocks, so each thread
 has a starting and ending group
start_group
end_group
for (b = start_group; b <end_group) in parallel do
    if database[filter_block_index].BF.query(Value) == true then
        for (l = 0; l <Num_records) do
            returned_value = database[filter_block_index].getLine(i)
            if Value == returned_value then
                result_array[l] = returned_value
            end
        end
        Update start_group for each thread
        Update end_group for each thread
    end
end
```

Algorithm 5: Algorithm of the query of type 3 with Bloom filter

```
Result: Results array
Nd = how many columns are queried
Num_Block_Groups = number_of_total_groups
Num_records = BLOCK_SIZE (ex: 1024)
col_num = total number of columns
target_col_num = the column to be queried
database = database_address
result_array = Initialize a new array
Value = the value to be searched
In parallel, each thread will be responsible for a group of blocks, so each thread
 has a starting and ending group (start_group & end_group)
for (b = start_group; b <end_group) in parallel do
    if the Nd Bloom filters return positive answers then
        for (l = 0; l <Num_records) do
            returned_record = get the record having all the queried columns
            if dependency values match the returned record then
                result_array[l] = returned_record
            end
        end
        Update start_group for each thread
        Update end_group for each thread
    end
end
```

Furthermore, since we are integrating Bloom filters in DOLAP, we built a Bloom filter version for queries of the second type. The main difference between Algorithms 4 and 3 is performing the Bloom filter check before accessing the database records. Based on the filter's answer, it will be decided to either enter the corresponding data block or not. We apply the checking process on all data blocks forming the column under querying.

Algorithm 5 represents the function responsible for executing *Lucene queries* of type 3. The query may contain 2, 3 columns or more. The algorithm should detect the number of queried columns, the block number, and the block size. Then each thread will process n blocks but before that, the Bloom filters linked with the data blocks are checked for the existence of the searched values. If at least one of these Bloom filters return a negative answer then the threads move to the next bunch of blocks (move to next iteration of the outer loop). Otherwise, the returned records are checked against the entered values and if a match is found then it will be put in the result array.

5. THE LOAD BALANCER

5.1 Background

Load balancing is a process of distributing the workload among the executing units in case of a central system which may include several CPUs and GPU boards or in case of a distributed system, it redistributes the load among the processes on different machines connected through a network. The aim of implementing a load balancing module is to improve the performance of a parallel system by distributing the workload intelligently.

In centric or distributed systems, there are two main types of load balancers; *static* & *dynamic*. The static type defines the method of sending the work to executing units before the run of the system, it may collect information about the system state during the run to update the load balancing method, however, the update occurs manually before running the system again.

On the other hand, the dynamic load balancer distributes the load at runtime, moreover, it can update its model dynamically while the system is running based on the information collected during the system's work.

Several works have been conducted in the load balancing field, in heterogeneous databases like the work introduced in (Ilic, Pratas, Trancoso & Sousa, 2011), also, *Omnidb* is such an example of query processing on parallel CPU/GPU databases (Zhang, He, He & Lu, 2013) as well as (Breß & Saake, 2013), (Fang, He, Lu, Yang, Govindaraju, Luo & Sander, 2007) and (Breí, Beier, Rauhe, Sattler, Schallehn & Saake, 2013). There are database management systems that process queries depending only on the GPU processing power like (He, Lu, Yang, Fang, Govindaraju, Luo & Sander, 2009) and (Root & Mostak, 2016)

5.2 Load Balancing in DOLAP

One of the serious aspects of this work is the workload distribution between the CPUs and GPUs. We do not want the work to be a heavy burden on one of the main processing units (CPU & GPU). The aim is to distribute the incoming queries according to the current situation of the host and device plus other criteria. Since the load balancer is a substantial requirement in the structure of DOLAP several models are proposed to investigate the best load balancing that provides better performance in a matter of fast query responding time and equally distributed workload on the host and the device.

Algorithms and methods to be designed and implemented in this part will be implemented in a way that will only work on a CPU-centric system in the first stage. In the beginning, a comprehensive performance analysis will be made by running all processes on the DBMS on the CPU (software). As a result of performance analysis, process-intensive and/or parallelizable basic function blocks will be defined and their performance-oriented optimized implementations will be made on GPU and CPU. The result of the performance analysis to be done for CPU and GPU in DOLAP of each functional building block will be kept in a table. The analysis of workloads that will occur while the DBMS is working will be dynamically performed by the CPU, and process-intensive tasks will be distributed to both CPU and GPU. During the task distribution, besides the values in this formed table, other processing loads running on all hardware will be taken into consideration.

The load balancing technique is to distribute the workload among the computing hardware on the server to provide parallelization, reduce power consumption, reduce response time, and increase performance and efficiency. With a successful load distribution, the response time of the users can be reduced by keeping the waiting time to a minimum. Two of the work distribution models used today are static and dynamic load balancing models. In this project, a dynamic load distribution model that takes into account the current loads of the devices is used. This model is summarized in Figure 5.1 A queue structure has been created for both CPU and GPU. When the query comes, many parameters summarizing the current state of the system are collected and processed instantly by the load balancing model to distribute the workload.

We use NVIDIA GV100 as the main GPU to be used on the system. These GPUs support the unified memory feature that will facilitate data exchange between GPUs during data management in the project. However, it is not sure that the DOLAP

will depend on unified memory in the GPU section due to the high costs paid while running the kernels, therefore, the normal approach of exchanging data between the host and the device will be used.

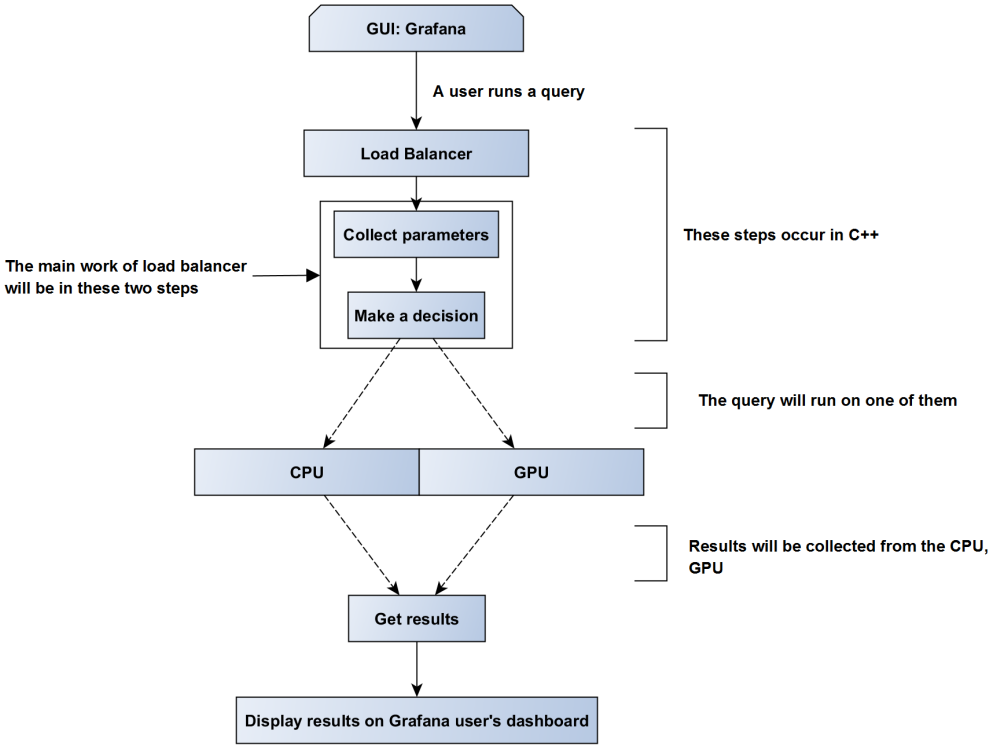


Figure 5.1 The Load Balancer integration in DOLAP

We are proposing 3 load balancing models that are heavily tested with more than a single user to check the total running time of the queries coming from either a single user or more.

5.3 Random-based Load Balancing

First, we introduced the *Random-based* model which does not take into consideration any information about the current state of the CPUs and the GPUs which includes the expected running time of the query according to its type, the current load on the CPUs and the load on the GPUs. Despite the possible overload on either of the main processing units, this model may decide to run the query on the busy one. On the other hand, and due to the randomness property of this model, for different test runs, this model may perform better than other models. A thorough study is introduced in Chapter 8.

5.4 Algorithm-based Load Balancing

This model is more sophisticated than the *Random-Based* model because it takes into account the query type, the loads on the host, and the device and according to the current state. A decision is made to distribute the load between the executing units.

Algorithm 6: Algorithm-Based Load Balancer

Result: GPU or CPU
dev = device_With_Least_Avg_ExecTime()
cpu_utilization = get_CPU_Usage()
gpu_utilization = get_GPU_Usage()
τ (Device Utilization difference threshold)

if *CPU_utilization - GPU_utilization* $> \tau$ **then**
  decision = GPU
else if *GPU_utilization - CPU_utilization* $> \tau$ **then**
  decision = CPU
else if *(CPU_utilization == GPU_utilization)* **then**
  decision = random(CPU, GPU)
else
  decision = dev
end

5.5 Optimized Algorithm-Based Load Balancing

Starting from the partially randomly taken decision in Algorithm 6 when both CPU and GPU utilization is equal, an optimized version of this algorithm is proposed to cover this shortcoming based on the average running times of query types. Whenever both utilization percentages are equal, the load balancer would check the query type first, if it is of type 1 then it will assign it to the faster-executing unit which the GPUs. If the query type is either 2 or 3 then the query will be assigned to the host which is faster than the GPU in this case. The results of this load balancing model are introduced in the results chapter.

Algorithm 7: Optimized Algorithm-Based Load Balancer

Result: GPU or CPU
dev = device_With_Least_Avg_ExecTime()
cpu_utilization = get_CPU_Usage()
gpu_utilization = get_GPU_Usage()
τ (Device Utilization difference threshold)

```
if CPU_utilization - GPU_utilization > τ then
    decision = GPU
else if GPU_utilization - CPU_utilization > τ then
    decision = CPU
else if (CPU_utilization == GPU_utilization) then
    if query_type == 1 then
        decision = GPU

    else if query_type == 2 or query_type == 3 then
        decision = CPU
    end

else
    decision = dev
end
```

In both Algorithms 6 & 7, there is a threshold τ and it represents the maximum difference in percentage between the CPUs and GPUs. At each time when a decision needs to be made in the load balancer model, the CPUs' and GPUs' average usages are retrieved using a script through predefined *C++* libraries. Then we check whether the difference between percentages exceeds the threshold τ. If it does, then a *switch* in executing units will occur which means if the host was the last executing unit then switch to device and vice versa. Yet, the comparison between the CPUs' & GPUs' utilization percentages is not fair since both, the host and device's computing capabilities are different. For example, a 75% CPU usage is not on a par with 75% of GPU usage, still, in this situation, we can follow the optimized algorithm to achieve a decision. However, in an edge example, assuming both percentages are 90% in this case, it would be an ignorant decision from the load balancer model if it assigns the next query to the CPUs for two reasons:

1.1 The device may have more idle threads and thread blocks to execute the query.

1.2 The host is also busy with other side tasks like receiving the queries and making the decision through the load balancer, therefore, it is better to lower the load on the CPUs for a while.

5.6 The Data Manager

Since DOLAP handles a significant amount of data transfers, updates, and retrieving between the cores of the CPU or withing the GPUs and between the host and the device, it forms a challenge because there is a huge overhead that DOLAP is paying repetitively on each query run and for each user. Therefore, by building this model, we try to minimize the data transmission overhead to achieve better performance.

5.6.1 Bottleneck spots

Since we are working on a time-series database, there is a `timeseries` column in the data source that we are receiving data from, and for displaying purposes on Grafana, this column is queried and attached to the results set on each query run which causes around 0.3 seconds additional time. Additionally, while answering the

queries, different data blocks are checked for different columns at each time, and some blocks might be queried frequently. Nevertheless, they keep getting queried and the results are retrieved at each run, this is an undesirable behavior that we want to avoid in our management system. Therefore, we built the data manager model in a way that it preserves the most queried columns aside. Thus, multiple queries that require one or all of these columns would find the data of these columns ready by only passing their preserved pointers. Meanwhile, an update process is kept running to check the demand frequency of the kept columns. If other columns have an increasing access frequency compared to the current ones then the least frequent column among the k preserved ones is deleted from memory and the new one is stored for further use. Following this method, we believe that we reduce the burden on the memory. Moreover, we avoid the bottleneck of adding unnecessary data copying overhead to the query response time.

6. EXPERIMENTS & RESULTS

All the experiments in this work are performed on a single machine running on 64 bit Ubuntu 18.04.03 LTS equipped with 1 TB RAM and a dual-socket Intel Xeon Gold 6152 clocked at 3.70 GHz where each socket has 44 cores (88 in total). For the multicore implementations, we used OpenMP and all the codes are compiled with g++ enabling the -O3 optimization flag. We also used `Python 3` as an intermediary between DOLAP and the visualization tool `Grafana v6.0`. Concerning the GPU implementations, we used `CUDA` 7.5 on an NVIDIA Quadro GV 100 with 80 Streaming Multiprocessors and 163840 maximum threads in total and 32GB memory. We organized the experiments based on Bloom filters and the Load Balancer previously proposed models plus the number of users, in our case, we made experiments up to 2 concurrent users. The experiments are divided into 4 main sections: **a)** 1 user without Bloom filter, **b)** 1 user with Bloom filter, **c)** 2 users without Bloom filter, **d)** 2 users with Bloom filter. Each section of these has sub-sections concerning the load balancer models and the different given values of τ.

6.1 The Dataset

We chose a dataset from *Kaggle* collected from the taxicabs of Chicago, Illinois, called *Chicago Taxi Rides 2016* Kaggle (2016) It includes taxi trips from 2015 to 2016 and it has 20 columns of different data types. Some trips durations are rounded up. Some of the rows for some columns are empty due to either non-recorded data about that trip or it was hidden for privacy purposes. The taxi IDs are long and unique keys of 128 character string however it was replaced by at most 4 digits also for privacy matters.

6.2 Implementation

We implemented our system, DOLAP, using `C++` in the functions and kernels side, while we used `python` to call the wrapped `C++` functions and kernel and run the users' queries, for communication between *Grafana* and `python` codes, we used `JSON` scripts.
We run our tests on a query set of 300 different queries of all 3 types on both host and device. For algorithm-based load balancing models and each threshold value, we run the test 5 times and took the average, whereas for the random-based model, we run the test 25 times and we took the average since this model is independent of the threshold value's effect.

6.3 Performance Metrics

To assess our new database management system, we settled several metrics to evaluate it and check its capability of providing fast responses and its ability to host parallel users. Since we are testing DOLAP with a query set of 300 queries of different types with 1 and 2 users, we want to measure the total running time of the whole set in seconds. Moreover, we want to prove that DOLAP can handle many concurrent users at a time, therefore, we measure the average CPU and GPU utilization in percentages.

6.4 Results related to the usage of Bloom filter

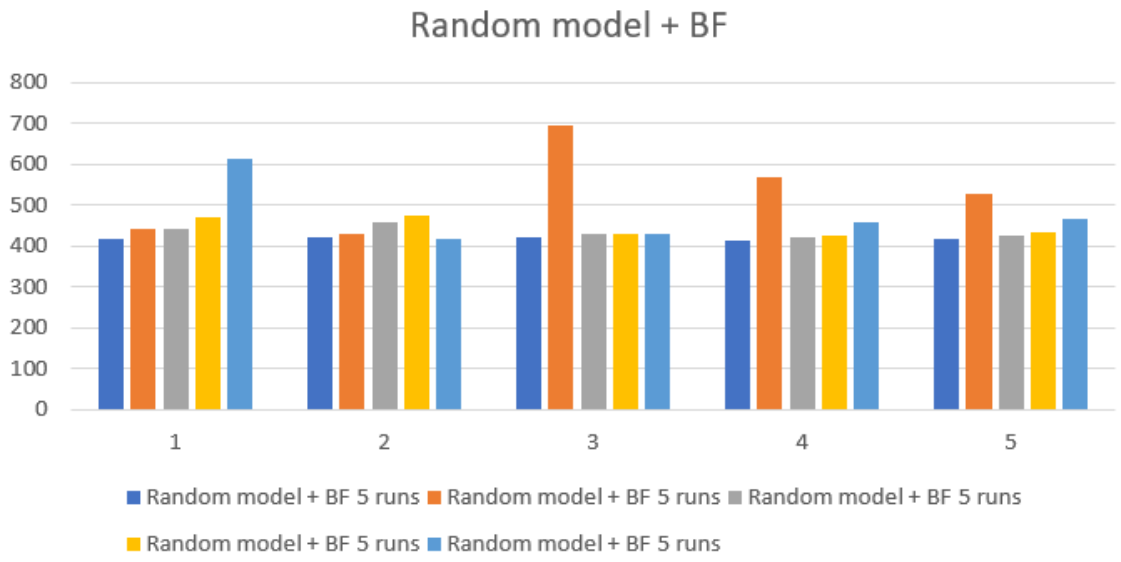


Figure 6.1 Comparing five runs of the total 25 runs of the random-based model (times in seconds)

As Figure 6.1 shows, *Random-based* model performs better than the other threshold values in any run when $\tau = 5\%$ while the 3^{rd}, 4^{th} and 5^{th} runs show spikes in the running times when $\tau = 10\%$, the threshold values here do not have any impact on the resulting total running time of the query set because the random-based model does not take into account the threshold value while making the execution device decision. The performance of the *Random-based* model might be bad as same as it may show better performance than the first and second *Algorithm-based* models due to its randomness property. It does not take into consideration the current state of the host and the device, therefore, it may tend to send the type 3 queries to the host which is the fastest executing unit, and forward the type 2 queries to the device which is faster in responding than the host. However, this is still not a trustful method of load balancing since it is not based on information from the current system's state.

In the 1^{st} *Algorithm-based* model, despite that it checks only the excess of the threshold between the CPU and GPU utilization, it still has a random part where it assigns the query randomly to either the host or the device which explains the unstable performance as shown in Figure 6.2. As a result, we cannot decide whether it is the most suitable model for DOLAP's load balancer or not.

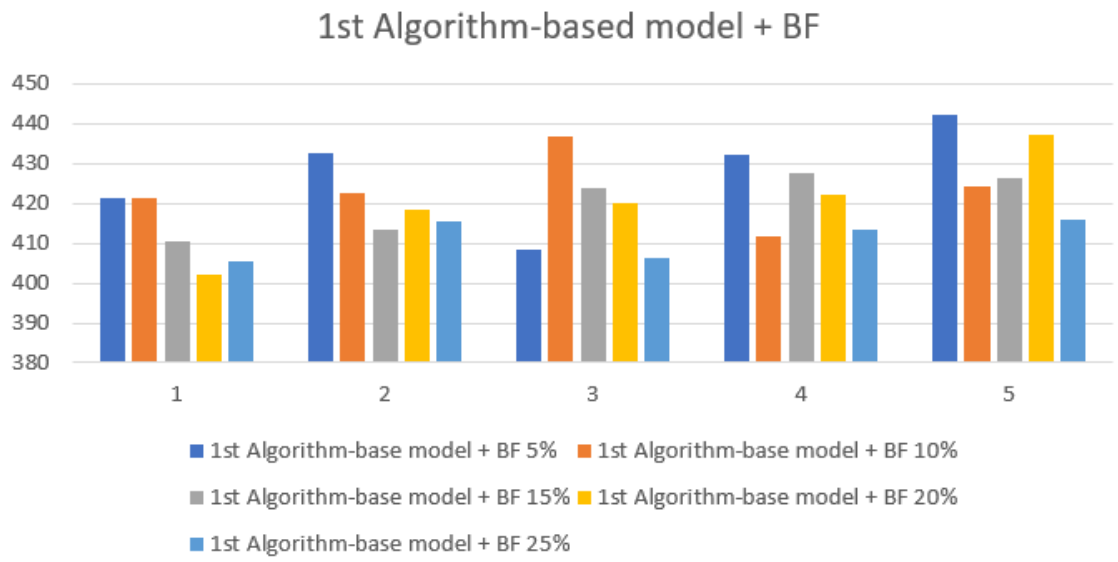


Figure 6.2 Comparing five runs of the first Algorithm-based model with different threshold values (times in seconds)

We can consider the 2^{nd} *Algorithm-based* load balancing model as the most stable and well-performing model because it takes into account each query type's running time, the fastest executing unit (CPU or GPU), and both CPU & GPU utilization percentages. Therefore, and as the Figure 6.3 demonstrates a steady performance and in total, it gives running times less than the other two models which prove the usefulness of our proposed algorithm.
The thresholds 5% and 10% show the best performances among other τ values and the reason behind that is the likelihood of exceeding the utilization threshold is higher when it is between 5% and 10% than the other τ values. This claim is supported also by the results in Table 6.3

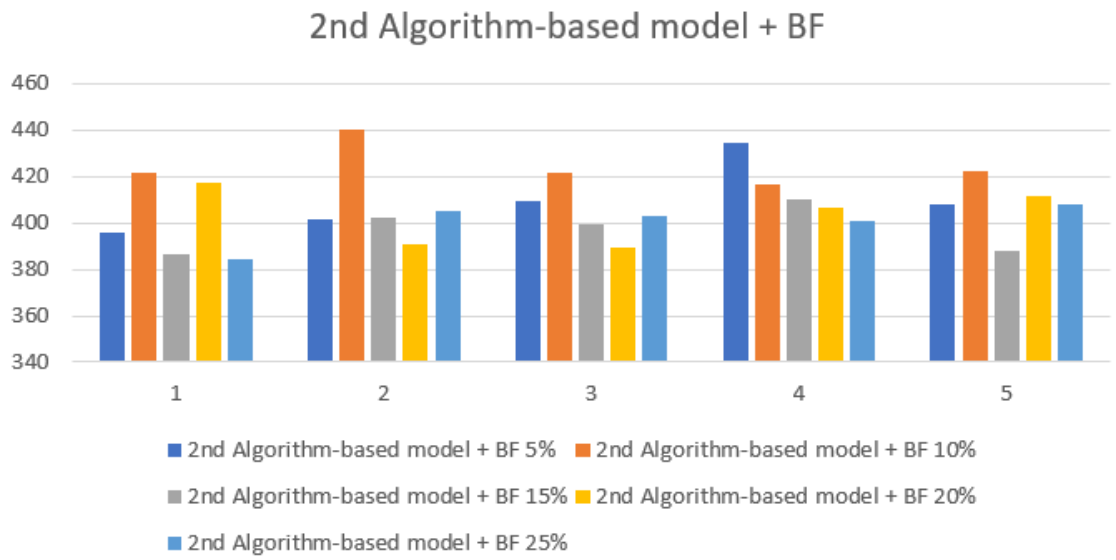


Figure 6.3 Comparing five runs of the second Algorithm-based model with different threshold values (times in seconds)

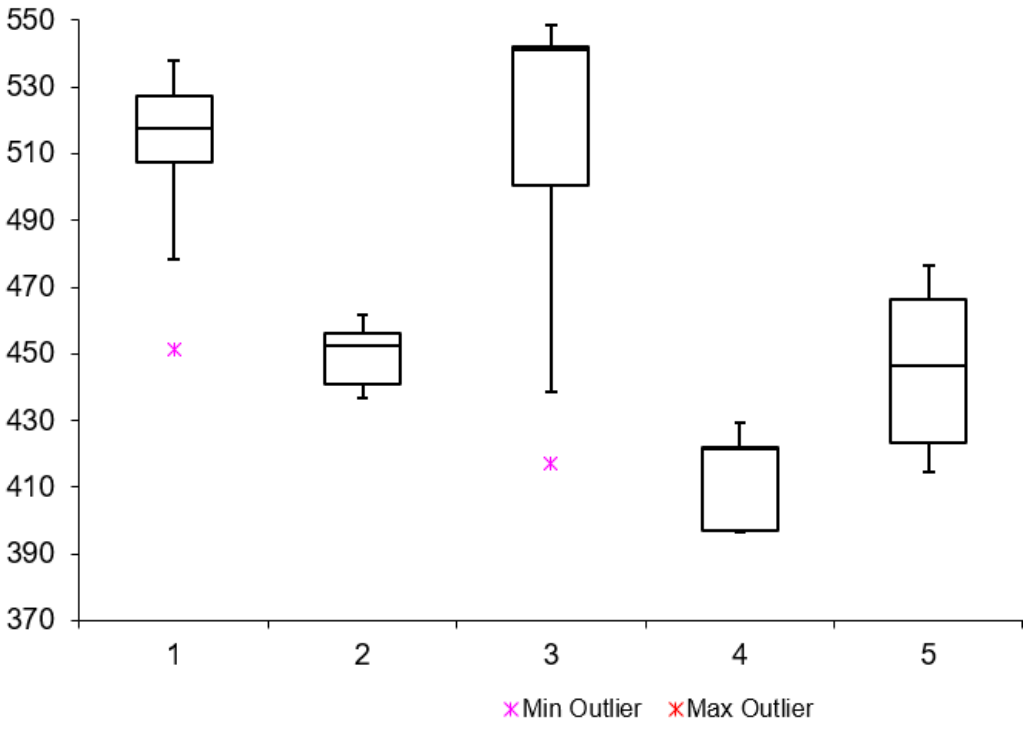


Figure 6.4 Dispersion boxes that shows ranges, quartiles, and interquartile of the query set's running times under each 5 runs for the random-based model

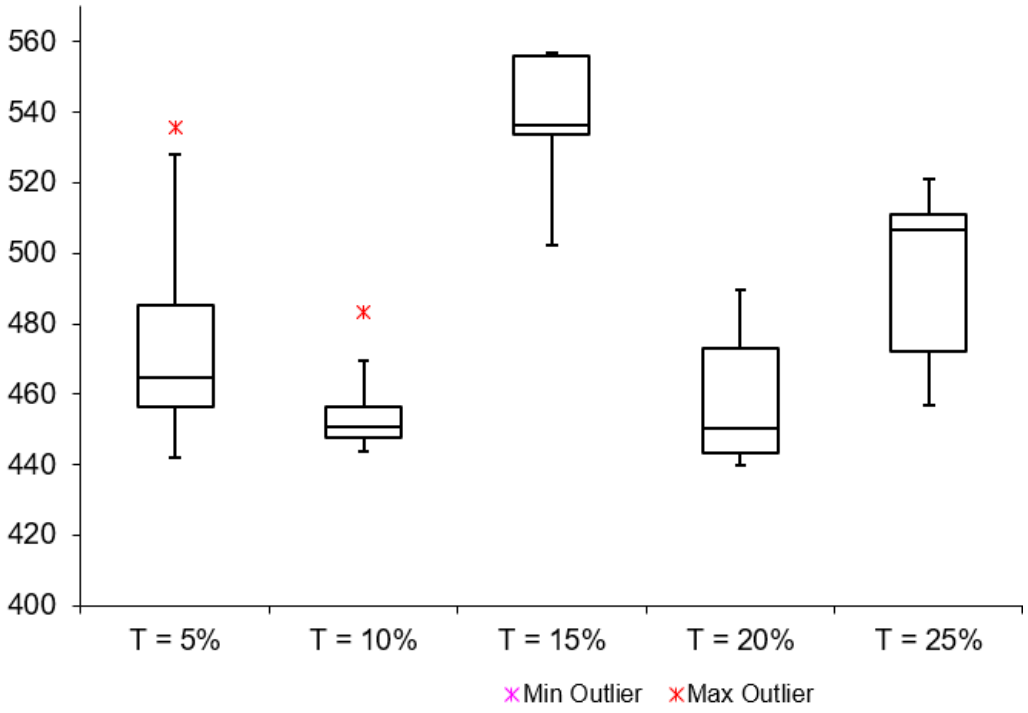


Figure 6.5 Dispersion boxes that shows ranges, quartiles, and interquartile of the query set's running times under each threshold value for the first algorithm-based model

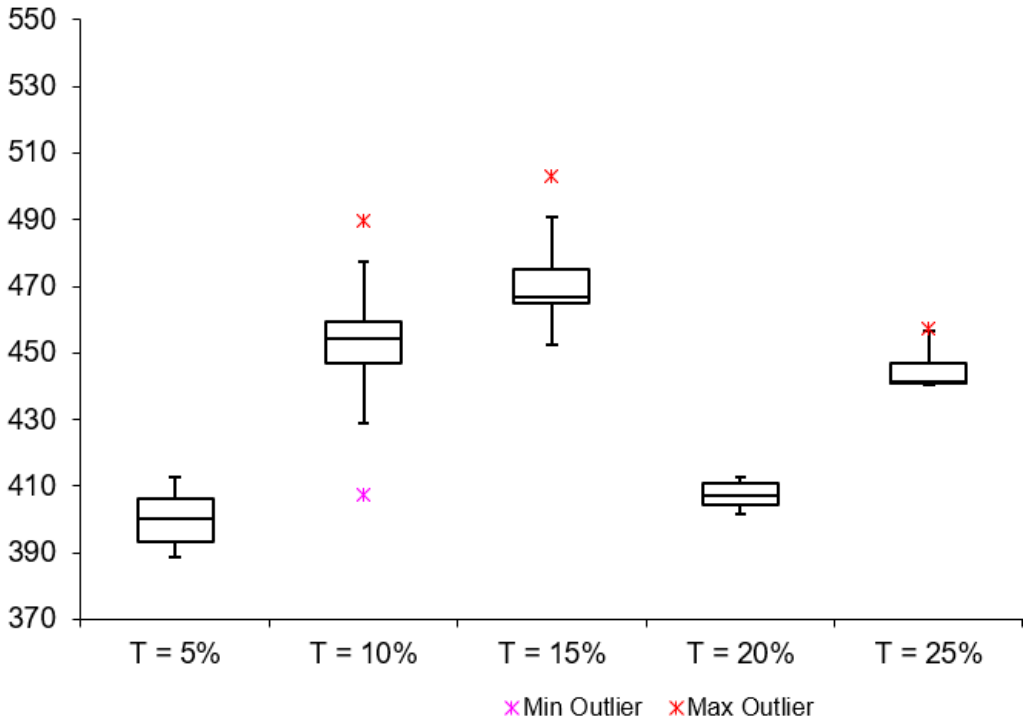


Figure 6.6 Dispersion boxes that shows ranges, quartiles, and interquartile of the query set's running times under each threshold value for the improved algorithm-based model

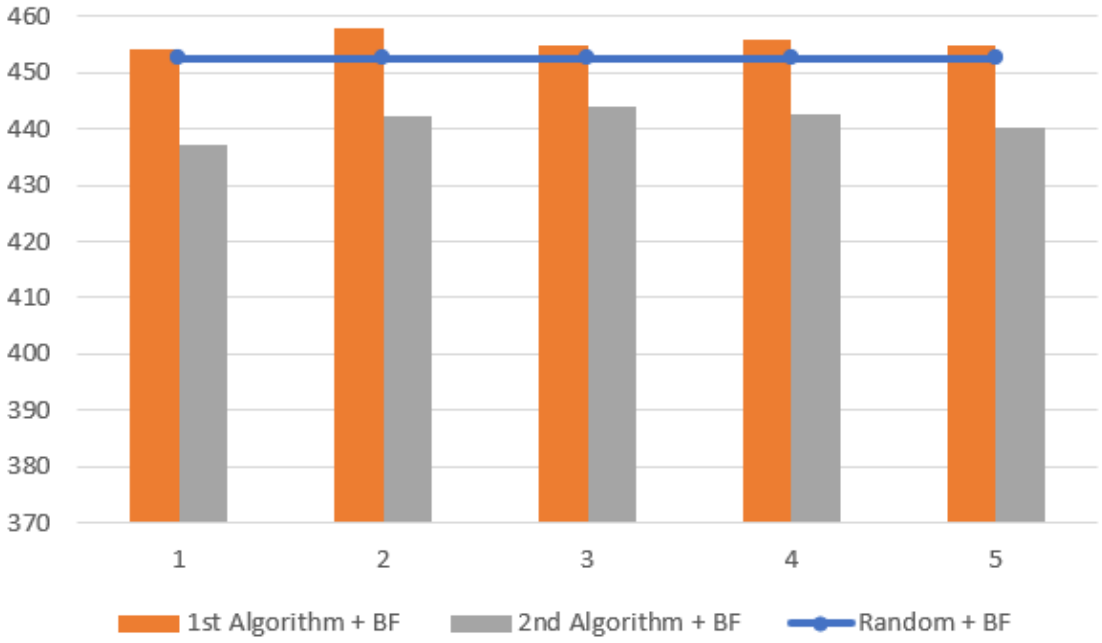


Figure 6.7 The average running times, in seconds, of each load balancing model launched by 1 user

The second *Algorithm-based* load balancer model gives better performance than the other models in each threshold value which is due to the thorough checking of the system's state at each decision making phase, while the first *Algorithm-based* model shows a bad performance compared to the *Random-based* model. However, both models give unstable execution times where sometimes are high and sometimes low and near the second *Algorithm-based* model running times. Figure 6.7 demonstrates these interpretations.

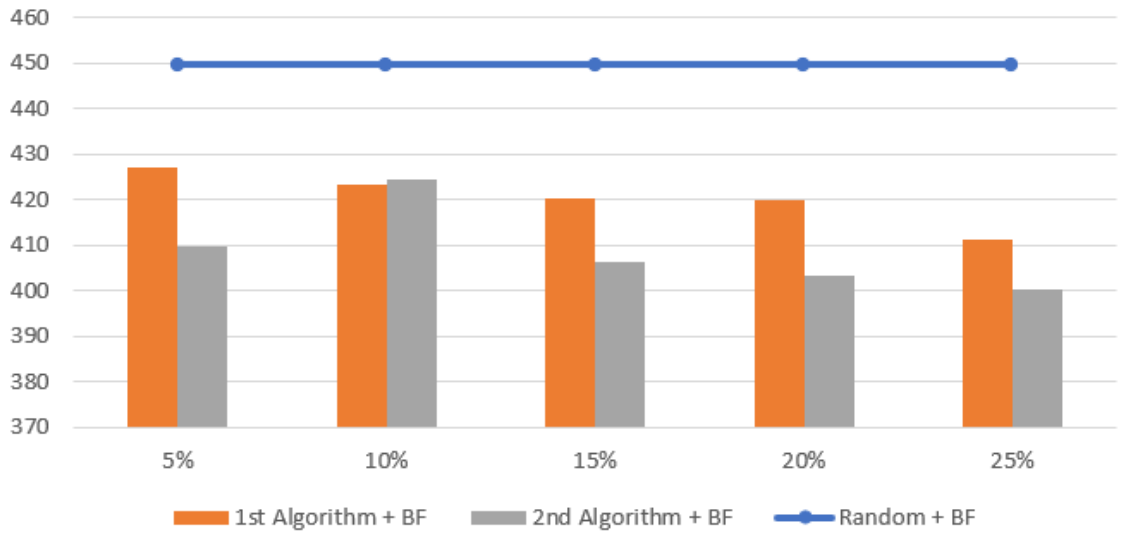


Figure 6.8 The average running times, in seconds, of each load balancing model launched by 2 users. As noticed, by comparing Fig. 6.7 and Fig. 6.8, the *random-based* model's performance is still bad, the gap between the latter and the other 2 models widens as the number of concurrent users rise. On the other hand, the 2^{nd} *Algorithm-based* model outperforms the first *Algorithm-based* model in almost all threshold values which demonstrates the benefit of our proposed load balancing model, the second algorithm-based model.

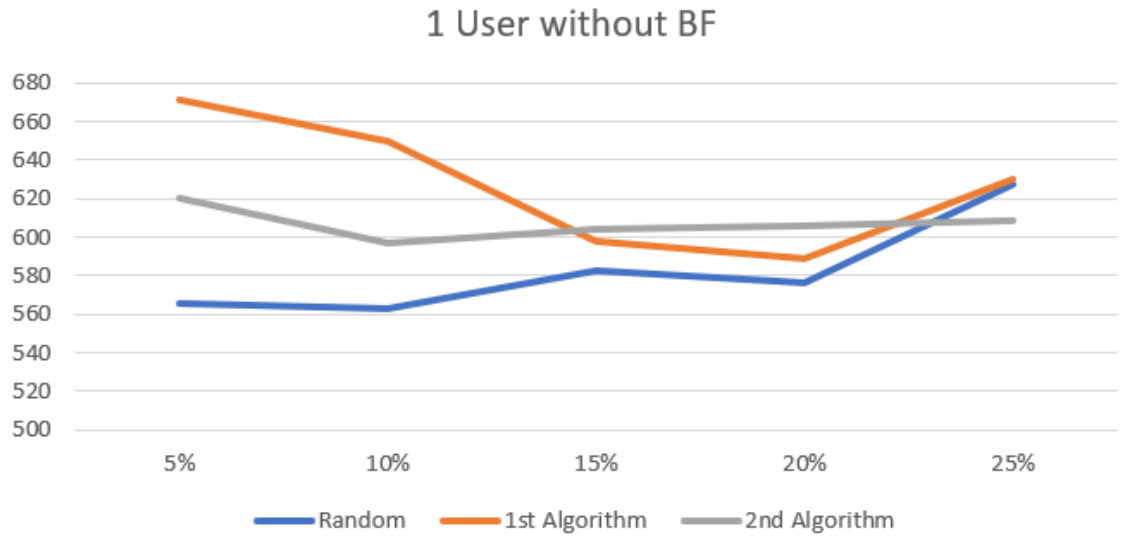


Figure 6.9 Comparing the load balancing models without using Bloom filters in matter of the total running time in seconds of the query set (300 query)

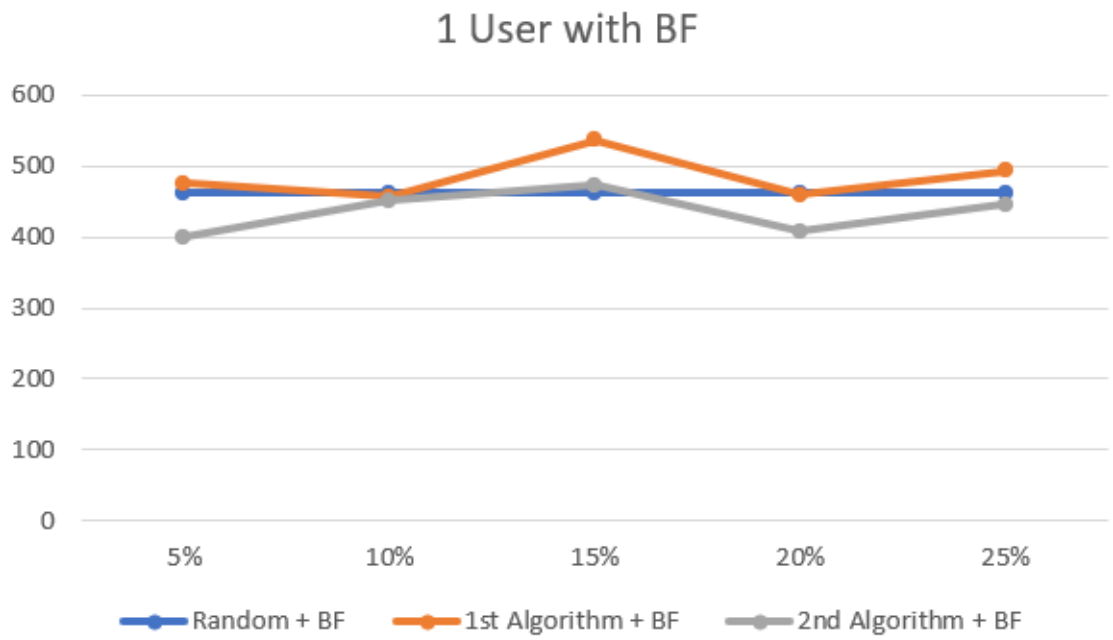


Figure 6.10 Comparing the load balancing models using Bloom filters in matter of the total running time in seconds of the query set (300 query)

The steady performance and full awareness of the system's state at each time a load balancer decision is made is what makes the second *Algorithm-based* model more desirable to be adapted in DOLAP. Unlike the other two models in Figure 6.9, they show ups and downs especially when the utilization threshold exceeds 20%. Moreover, we can notice the difference in these models before and after using the Bloom filters when comparing it with Figure 6.10. In the latter, again the ups and downs in performance of the *random-based* and 1^{st} *algorithm-based* models are detectable, however, here the 2^{nd} *Algorithm-based* is in overall performing better than the other models while using the Bloom filters.

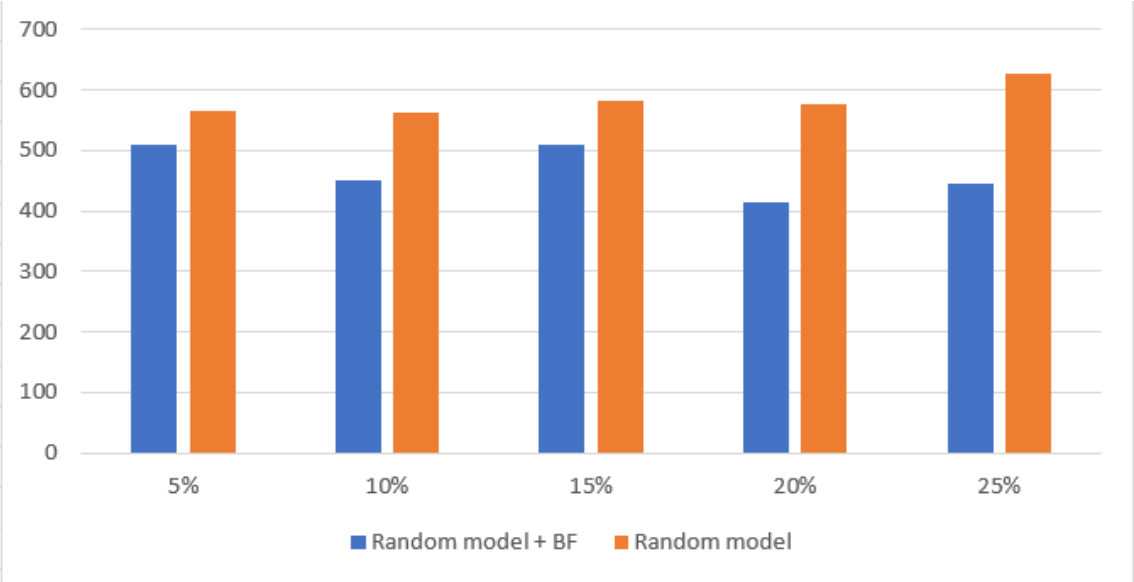


Figure 6.11 Comparing the Random-based load balancer model's total running time of the query set in seconds, with and without using the Bloom filters averaged on each 5 runs

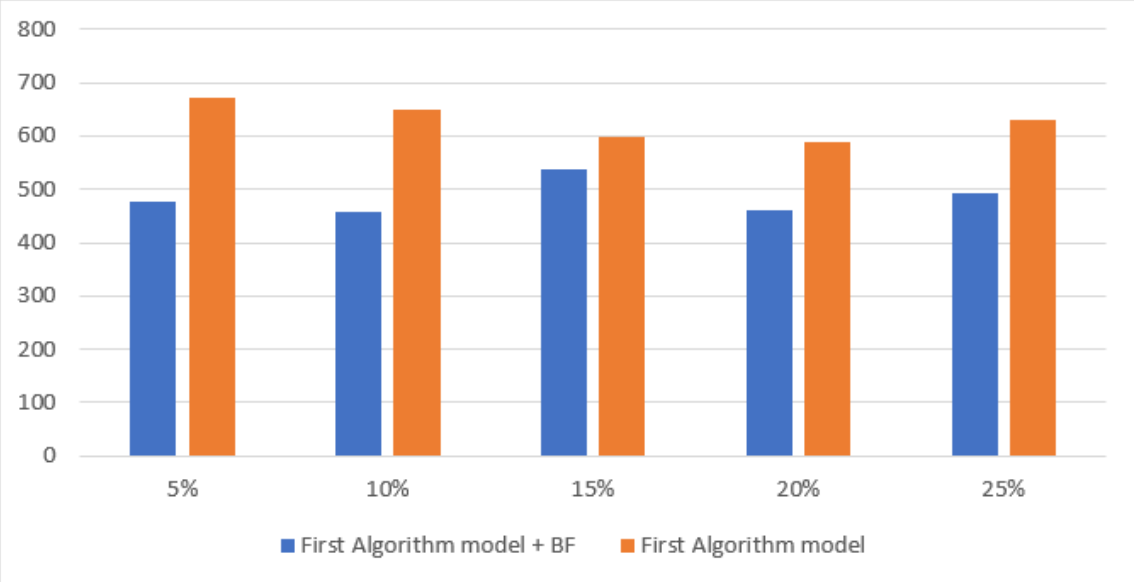


Figure 6.12 Comparing the 1^{st}Algorithm-based load balancer model's total running time of the query set in seconds, with and without using the Bloom filters averaged on each threshold value

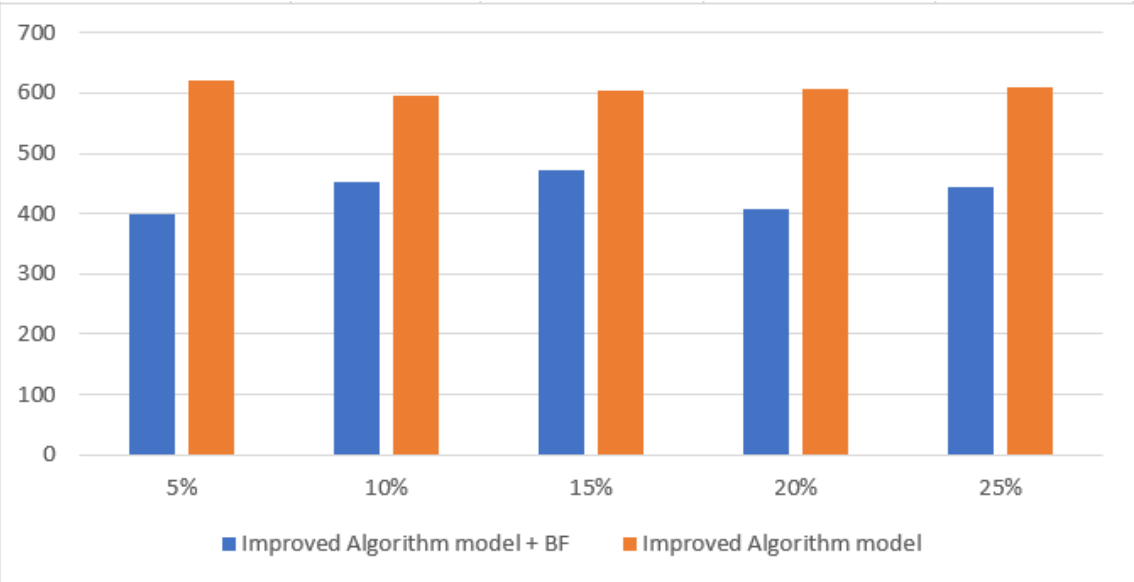


Figure 6.13 Comparing the Improved Algorithm-based load balancer model's total running time of the query set in seconds, with and without using the Bloom filters averaged on each threshold value

Figures 6.11, 6.12, and 6.13 clearly show the difference of using the Bloom filter in the load balancing models. As it is noticeable, Bloom filter improves the total running time of the query set under execution. The ameliorations vary from 10% to 29% for the *Random-based* and 1^{st} *Algorithm-based* models, and between 21% and 35% for the 2^{nd} *Algorithm-based* model.

Table 6.1 Comparing the queries running times, in seconds, of the Load Balancer Models with and without using Bloom filters

Mode	**Query type 2**		**Query type 3**	
	CPU	**GPU**	**CPU**	**GPU**
Without Bloom filter	1.545	1.100	2.732	1.182
With Bloom filter	1.834	1.335	0.960	1.431

Despite that the average execution times of the second type query on CPU and GPU, plus the running time of query type 3 on GPU without using Bloom filters is better than the ones with Bloom filters. Still, the presence of Bloom filters in DOLAP achieves improvements since it may cause additional overhead in some queries but it also may respond faster by preventing the unnecessary access to blocks in which the search items aren't there. Table 6.2 supports this by showing the minimum running times of *Lucene* queries of type 2 and 3 which are less than the average running times without depending on Bloom filters.

Table 6.2 Minimum and maximum response times in seconds of CPU and GPU queries of Lucene queries using the Bloom filter

	MaxCPU	**MinCPU**	**MaxGPU**	**MinGPU**
Query type 2	3.773	1.354	3.172	0.957
Query type 3	2.355	0.648	3.082	1.007

Table 6.2 shows the minimum and maximum CPU and GPU query running time respectively for both query type 2 and 3. The reason of displaying such measurements is to show the effect of using Bloom filters while querying, for example, the minimum CPU query running time for query type 2 using the Bloom filter is less than the average query running time of the same type without using Bloom filter (see Table 6.1). The same goes for the GPU in query type 2, the minimum running time is 0.967s while the average without using Bloom filter is 1.1s. Therefore, the same can be noticed in query type 3 which demonstrates the effect of using Bloom filters on the queries' running times.

Table 6.3 CPU and GPU usage statistics (in %) for each load balancing model

	Random	1^{st} **Algorithm**	2^{st} **Algorithm**
Avg CPU Util	3.052	2.402	2.387
Avg GPU Util	2.087	0.662	0.654
Max CPU Usage	28.2	19.7	19.0
Max GPU Usage	49.0	17.0	16.0
Min CPU Usage	0.7	0.0	0.0
Min GPU Usage	0.0	0.0	0.0

Table 6.3 points to an important side of this work, which is the capability of hosting and answering multiple users' queries in parallel, the 3 load balancing models have CPU and GPU average utilization percentages ranging from 0.6% to 3% and these numbers are taken from tests with 2 parallel users which means that DOLAP can take up to 40 users in parallel with a single GPU board, of course, after taking into account the heat of the CPUs and GPUs and their maximum allowed utilization percentages.

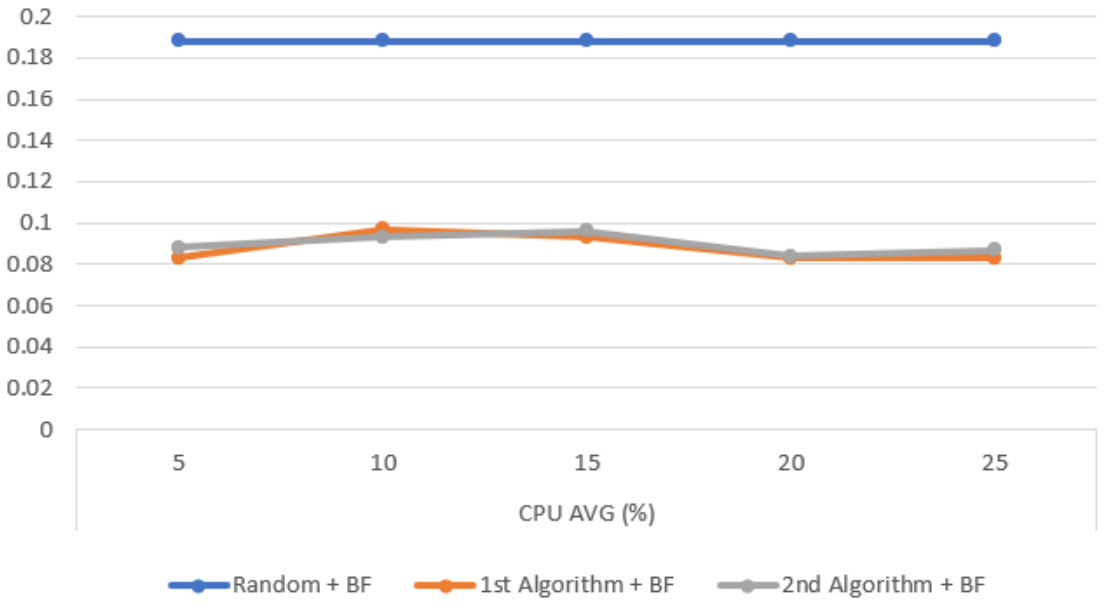


Figure 6.14 Average CPU usage percentages (running 300 queries) when answering the queries scenario for all load balancing models

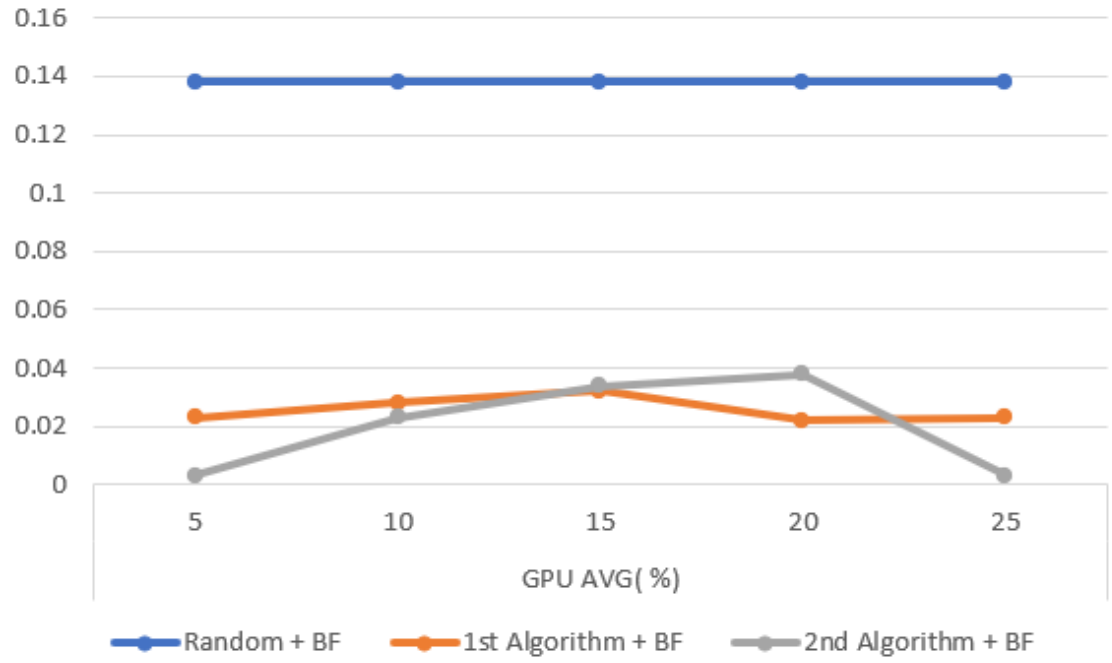


Figure 6.15 Average GPU usage percentages (running 300 queries) when answering the queries scenario for all load balancing models

6.5 Discussion

We have stated that there are several overhead spots where additional time is aggregated in the query execution time, especially in the CPU and GPU C++ wrapping functions, since the query order comes from a user interface connected to a python script which in turn, calls either a CPU or GPU wrapping function in the C++ part. Moreover, there is the intrinsic issue of the communication between the host and the device. At each query running on the GPU side, the data columns under investigation will be copied from the CPU to the GPU's memory. Then the results will be copied from the device to the host so it can be returned to Grafana (the visualization tool) and displayed as readable statistics.
This process has two additional times added to the total running time of a query on GPU. The first one is when data is copied to the device's memory, while the second is when the results are copied back. Moving to the host side, there is no such overhead because the data is already there and the C++ function will operate on it without the need to move them from the RAM to anywhere. The only overhead paid here is when the C++ wrappers are being called to run the executing function. Table 6.4 shows the overheads paid at both CPU and GPU sides and their whereabouts.

Table 6.4 Overhead types and percentages according to query types for the GPU rounded to the whole integer

Overhead Type	**Query type 2**	**Query type 3**
Memory Allocation	29-30%	23-26%
Memory Copying	1%	1%
Results Recovering	17-21%	16-17%
Calling wrapped functions	20-26%	20-26%

Table 6.4 shows the percentages of overheads experienced on the GPU side due to the implicated communication cost that should be paid at each GPU wrapped function call. There are 4 main spots that aggregate to the total query's execution time which are memory allocation, memory copying from the host to the device and inversely, plus the results sending back to Grafana and the waiting time of calling the wrapping functions of the executing functions in the C++ part. The percentages displayed in the table refer to the ratios of how much these overheads is to the total query's execution time.
The first 3 overheads in Table 6.4 do not exist in the CPU side, however, the 4^{th} one does impact the query's running time and it is similar to the cost paid for the GPU side. It varies between 20% and 26%.

7. CONCLUSION

In this thesis, we proposed a hybrid database management system that uses both CPUs and GPUs with a novel mechanism of workload balancing between these processing units. We introduced two main types of queries, simple and Lucene queries, and two sub-types of Lucene queries, type 1 and 2, and for answering them, We designed CPU functions and GPU kernels responding in seconds and milliseconds. Moreover, distributing the load between the CPUs and GPUs formed a challenge and for that we suggested 3 load balancing models written in `C++`, the *Random-based*, $1st$ *Algorithm-based* and 2^{nd} *Algorithm-based* models. The random one showed a bad performance compared to the other models which is logical because the randomness ignores the system's state, While the first proposed algorithm showed a slight improvement compared to the random model, however, there was a randomness spot that gave it a disadvantage, therefore, we could do better in the second algorithm that demonstrates a good stable performance since it looks over the whole current system's state. There was up to 20% improvement introduced by the second algorithm compared to the first algorithm and the random models which confirms our predictions about the advantage of the second algorithm. Another dimension of the presented improvement in this work was due to the use of Bloom filters probabilistic data structures, we tested our functions and kernels with and without Bloom filters and as expected, there was a refinement of the total running times of the query set. The percentage of enhancement reached 33.3% when using Bloom filters while answering the queries. This side of the work was not implemented by the previous GPU-based or hybrid databases like *Kinetica*, *SQream*, or *BlazingSQL* in which we believe that our work has an advantage against the other ones. The tests were conducted by 1 and 2 users separately to prove the capability of DOLAP to answer several concurrent users in milliseconds where the average CPU and GPU utilization didn't exceed 3% and 2% respectively when running 2 users in parallel. This work is open for more improvements and complementary additions, there still a window for a new load balancing algorithm proposal that outperforms ours, moreover, concerning the used Bloom filter, there might be some modifications that would be brought to it in a way that it enhances the filter's performance, by either reducing its false

positive probability and therefore reduces the number of faulty query responses or by implementing a better hash function that gives better response times. Additionally, this project depended on only a single GPU board, by forming a network of GPU boards with acceptable communication bandwidth and network architecture, the DOLAP system would have a tremendous capability of responding to a bigger number of simultaneous users instantly.

Improvement spots also include the mechanisms of how to run multiple queries sent by a user in parallel, therefore, a queue should be constructed to enqueue queries that cannot be handled immediately, this way, the database management system would be able to manage several parallel users with instant query sets.

BIBLIOGRAPHY

BlazingDB (2015). Blazingsql. `https://blazingsql.com/`. Accessed: 2020-07-20.

Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, *13*(7), 422–426.

Breí, S., Beier, F., Rauhe, H., Sattler, K.-U., Schallehn, E., & Saake, G. (2013). Efficient co-processor utilization in database query processing. *Inf. Syst.*, *38*(8), 1084–1096.

Breß, S. & Saake, G. (2013). Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, *6*(12), 1398–1403.

Fang, R., He, B., Lu, M., Yang, K., Govindaraju, N. K., Luo, Q., & Sander, P. V. (2007). Gpuqp: Query co-processing using graphics processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, (pp. 1061–1063)., New York, NY, USA. Association for Computing Machinery.

Grafana (2014). Grafana visualization tool. `https://grafana.com/docs/grafana/latest/getting-started/what-is-grafana/`. Accessed: 2020-08-01.

He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., & Sander, P. V. (2009). Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, *34*(4).

Ilic, A., Pratas, F., Trancoso, P., & Sousa, L. (2011). *High-Performance Computing on Heterogeneous Systems: Database Queries on CPU and GPU*, volume 20, (pp. 222).

Kaggle (2016). Chicago taxi rides. `https://www.kaggle.com/chicago/chicago-taxi-rides-2016`. Accessed: 2019-08-01.

Kinetica (2018). Kinetica gpu-database. `https://www.kinetica.com/`. Accessed: 2020-08-02.

OLAP (1993). Online analytical processing. `https://olap.com/`. Accessed: 2020-07-21.

Root, C. & Mostak, T. (2016). Mapd: A gpu-powered big data analytics and visualization platform. In *ACM SIGGRAPH 2016 Talks*, SIGGRAPH '16, New York, NY, USA. Association for Computing Machinery.

SQream (2014). Sqream data warehouse. `https://sqream.com/`. Accessed: 2020-07-20.

Zhang, S., He, J., He, B., & Lu, M. (2013). Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proc. VLDB Endow.*, *6*(12), 1374–1377.