Ticket Distribution, Benchmarking and Revenue Computation for a Public Transport Company

Gustavo Martins
University of Coimbra
Department of Informatics Engineering
Coimbra, Portugal
gamart@student.dei.uc.pt

João Valença
University of Coimbra
Department of Informatics Engineering
Coimbra, Portugal
valenca@student.dei.uc.pt

ABSTRACT

Many cities in the modern and developed world have seen a substantial urban growth in the last two decades. With this growth came different solutions to public transportation. In order to create an inexpensive, transparent and fair system, the department of transport has issued a special ticket that allows users to take advantage of different transportation without having to purchase different tickets for the different services.

This created a problem for the various services, which are funded independently from one another, since each ticket price should be distributed across all of them in a fair and precise manner. The system should also withstand the growing use of the services and needs to be scalable, so as to guarantee a fail-safe service regardless of the stress it is put through.

Since each ticket and line is stored in a shared database, the best solution is to improve the current system and make it more resistant to stress. The current setup is a single node setup, and benchmarks suggest that it is prone to overload.

The aim of this paper is to propose a new architecture to better withstand system overload (with alternative engines if need-be), and to present its results of such an architecture when stressed through the use of relevant benchmarks. This will be done to prove that such scalability is possible and within reach.

Keywords

Bigdata Systems, Database Scalability, Database Clustering, Data Warehouse, Cloud Services, Benchmarking

1. INTRODUCTION

Everyday in large cities, hundreds of thousands of people use public transportation. Between buses, trains and subway systems, the cities' public transportation networks manage to cover, between all of them, the whole urban area, and most important points can be reached easily, with just a couple transitions.

Wherein these transitions, however, lies a problem for the common user of the metropolitan area. Each passenger would, in the worst case scenario, have to buy multiple tickets for all three of the services to get around across the city. Added to the hassle of buying three different tickets, each person would also have to keep track of how many uses each ticket had and how much they would each cost individually.

In order to ease the passengers' lives and streamline the whole process, a city could choose to implement a shared system between all three ways of transport. We use this paper to describe a new system to help with the management. In this *Transport Management System* (or T.M.S., for short), a user only has to buy a single ticket, and can use it regardless of the means chosen or required to get to their destination, so long as they are used within a pre-established time period. The ticket price is then distributed evenly across to whichever means the passenger used. If a given citizen had to take a train and two buses to get to their destination, two thirds of the ticket price would go to the bus company and the remainder third to the train company, and nothing would go to the subway company, since it took no part in the passenger's trip.

Such a system, however, poses a technical problem. The database system behind this service has to be able to scan across the over hundreds of thousands of entries and compute the value to distribute to each company in a fair and precise way, so that none of the companies is at risk of losing money.

Handling such enormous amounts of data is no easy task and a specialized system must be employed. The system must be able to withstand the increased overhead that comes with scale, ideally without much additional cost for the company.

To solve this problem, we approached it with a Relational Cluster Database approach. This system should be the most appropriate for the problem at hand. Also, since no similar problems were found, we decided to design our own benchmarks in order to prove the efficiency and stability of both systems, as well as present a fair comparison between them.

2. RELATED WORK

A similar study[1] with the same topic and purpose, which is to prove that a ticketing system can be scalable, was already made in the past. In that study the technologies Apache Hadoop and Apache Hive, explained later, were the ones used, and a Data Warehouse system was the only one they simulated in their work. First, they built an architecture with a single node, and afterwards one with five nodes, in order to compare a standalone architecture with a scalable one and see which one is the best for this kind of system.

Unfortunately, they did not get and present any results for the scalable version of the architecture, and the only ones are for the single node version. From the results presented, it can be concluded that the single node cluster did not performed so well, taking a great amount of time to process a very little number of lines and size in the database.

3. BACKGROUND

To perform this specific work and taking into account all the requirements that something like this demands, some services and structures had to be chosen. In this section, those services and structures, and not only the ones that were chosen, are explained in detail, how they work, their strenghts and weaknesses, etc..

3.1 MySQL

MySQL[2] is one of the most popular SQL servers in the world. Its popularity derives from being simple to use, and light enough to be used in web services as well as it being open source and free software.

As the name implies, it is a SQL server, meaning it is a relational database system (as opposed to a noSQL server). It makes use of a sequencial programming, meaning that only one action at a time can be performed by the server, in order to reduce complexity and increase ease of use. This makes it suitable for small scale Databases, since it provides a headache-free experience for the engineer, and works very stable with very little planning besides the schema designing process.

Since is the most simple and common system, it will not be more efficient than the others, but will provide a good control test to measure against for the other approaches.

3.2 MySQL Cluster

MySQL Cluster[3] is a MySQL implementation that makes use of parallel programming. With the parallelization, however, come the concurrency problems. All processing nodes will have access to a shared memory, so every read and write must be done carefully and orderly, so as to preserve the database integrity.

This system provides a much faster and efficient Database than standard MySQL, but it poses new challenges, such as designing a system that doesn't duplicate or delete important data. Since MySQL cluster offers automatic sharding, if configured correctly should be ACID-compliant, and should prove to be stable enough to run a large system, but attention must be payed during configuration and debugging the system.

MySQL Cluster uses several machines, or nodes, to divide the tasks, and there are two types of them. The Management Nodes are the machines that control the whole cluster and are the ones where the user can make the SQL queries. The Storage Nodes are the machines that contain the data itself.

The purpose of MySQL Cluster is to divide the data search between the several Storage Nodes, in order to the reduce the time it takes to get the output. After that search, the Management Node is responsible to join the information given by the Storage Nodes and present it to the user. Sometimes it is not possible to divide the work through the several Storage Nodes, and in that case the query becames slower.

3.3 Data Warehouse

A Data Warehouse is a normal database, just like any other, but instead of being used to serve as Operational Database, where day-to-day operations are made, is used to serve as a Central Repository. Thus, they are used to accumulate data from several disperse sources and contain not only the current data but the historical one too, that can already not exist in the sources.

Since they store all the information that was ever created they are often used to make reports and analysis of a company, which can prove very useful. Just so, Data Warehouses are often integrated with high-processing machines.

3.4 Cloud Database

A Cloud Database is the same as a normal Database, but instead of being stored and executed on a local machine it is on a Cloud service. Being internet based, this system might prove useful for easing the development of the Database, since there are no physical machines to maintain.

3.4.1 Cloud Virtualization

There is no need, however, of a Cloud to simulate a Cloud Database. Taking advantage of the Virtualization process, several Databases can be used inside of Virtual Machines, each one representing a Cloud Database. But this virtualization brings another problem, which is the fact that for every single Database there is only one disk, the bottleneck, and the system is going to be slower.

3.5 Apache Hadoop

Apache Hadoop[4] is a free framework, built in Java, that is used to serve as a Cluster manager. Just like MySQL Cluster it was designed to work with a large amount of data and to make use of a Database Cluster to improve performance. Hadoop has several modules, two of which are the Hadoop Distributed File System (HDFS) and Hadoop MapReduce, derived from Google File System (GFS) and Google's MapReduce, respectively. These modules are the essencial ones to work with clusters and a large amount of data.

3.5.1 Apache Hive

The Apache Hive[5] is an infrastructure built on top of the Apache Hadoop that provides several services to the user. One of this services gives the user the ability to do SQL-like queries to the database stored in the Hadoop.

4. SYSTEM ARCHITECTURE

In this section, the whole system architecture will be explained and discussed. It will show the different kinds of architectures that can be used in such a system, one that involves public service, i.e. , contact with a great amount of people, central management, and scalability.

Finally, it will be also presented the Database Structure. The components of the Database and how its going to be generated are topics that will be explained in detail. It will also be discussed the Benchmark that was specifically created to test a Public Transport Control Systems in general, and this Database System in particular.

4.1 General Architecture

Although several systems and architectures can be used, there is always a common base that is transversal to all of them. In that base, and taking into account what the objective of this study is, three systems and components were identified. Those are Data Warehouses, Cloud Databases and Submiters. This does not mean that all three will be studied, but are nontheless important systems to consider, and will therefore be explained in more detail.

4.1.1 Data Warehouse

Since a lot of transactions are made every second to the Database System (whether by Deposits and Sales of Tickets in shops or machines, or by Validations in public transports), there is a need to use an architecture that supports a Data Warehouse System, so that any other queries to the Database can be made in separate, without overloading the Database System.

With a system structure like this one, some other things are implied. One of them is the need to transfer all the Data from the Database System to the Data Warehouse in regular intervals. This transfer process is too much Disk and Time consuming, and so it should be made by night, when the transaction activity related to Deposits, Sales and Validation Systems are fewer and further apart. There is also the question of how regular this Data transfer from de Database System to the Data Warehouse should be. Since there is a lot of information in just one day, with numbers rounding 300.000 entries for the Validations alone, it is then recommended that this process is done on a daily basis.

Based on what was described above, a System Architecture was designed and it will resemble with something like the following picture depicts:

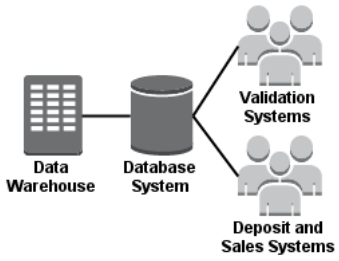

Figure 1: General Architecture

As it can be seen, there are two sets of systems related with the user: the Validation Systems and the Deposit and Sales System. The Deposit and Sales System is related to the purchase of the tickets, called the *Andante* tickets, by the users, and the deposits those users made on those same tickets, so it can be used on public transport. The Validation System refers to the machines that exist in every public transport, be it buses, subways or trains, in which the user has to show the ticket so they can use the public transportation.

All of this information has to go somewhere, and that somewhere is the Database System. This system stores all the information generated by the user related Systems and must be fast and reliable, since the importance and number of transactions submited per second is considerably high. However, as said in the beginning of this report, the main purpose of this system is to distribute the gains by all the tree companies (buses, subways and trains) equally and there are , therefore, some calculations that must be done. These calculations, though, are very time consuming and they cannot be done directly on the Database System, which is already overloaded with transactions. Because of this, a Data Warehouse is created, and on a daily basis all the data in the Databased System is transfered to the Data Warehouse, where those calculations are subsequently done.

But, there is even another advantage for the use of a Data Warehouse in this case. Since all the transactions made to the Database System are insertions about Validations, Deposits and Sales, the information related to the first two can be removed from the Database System once the transfer to the Data Warehouse is completed. As such, only the information about the Tickets needs to be permanently on the Database System, which makes the Database System lighter and faster to its daily use.

4.1.2 Cloud

Another system that can be used in this situation, and that is indeed very often used today, is a Cloud Database. Instead of having a physical Server with a Database on it, the Database is stored in the Cloud, and it is up to the Cloud supplier to have a physical structure to store de Data. This system, however, can turn to be even more expensive than a normal one, simple because the amount of data that is stored in a system like this is very large. Besides, it can make the system slower too. The use of Cloud Services is recommended in some cases, since it relieves some of the work of managing the system, but only if the amount of Data is not that big.

Beyond the benefit of less management work, however, Cloud Databases can bring another ones, for example, the fact that both Ticketing and Cloud Systems may make use of the Internet. This can turn the Deposit process by the users through a possible Internet application easier, since the Database is on the Internet itself. Also, the Validation Systems can make use of the Internet to directly make their transactions to the Database, instead of bringing all the Data to a system that does that.

A system using a Cloud Database System instead of the regular one was desgined and should be described as follows:

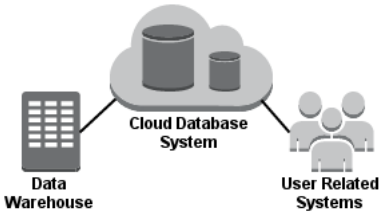

Figure 2: Cloud Architecture

As shown in the picture, the architecture is similar to the default one, the only difference being that now, the Database System is in a Cloud Service. To the user of public transportation the process is the same, which is good, because if a change in the architecture ever occurs, they will be none the wiser. There is also the possibility of having the Data Warehouse on a Cloud Service too, but since the Data Warehouse is where all the information ever created is stored, the size of it would be too big and expensive to have it on the Cloud, not to mention the fact that the revenue computation would be much slower.

4.1.3 Tester

The last of the three components explored here, and the one that can never be forgotten, since it is the most essencial one, is the Tester. For the purposes of any study, not

just this one, due to the impossibility of having tests with real life users, a Tester must be adopted. On one hand there is the disadvantage of not having real life testing. On the other hand there is the advantage of test customization. Using simulation, with the capability of customization, several scenarios can be used, and the System can be tested against an even higher stress than in real life. Besides that, some situations can be prevented and possible failures can be corrected before the System goes "online".

This allows for a wide range of simulations and tests, which can be specifically created for a particular type of System, and based on the requirements of this study and on a System like this one, an architecture of testing was designed, which is presented as follows:

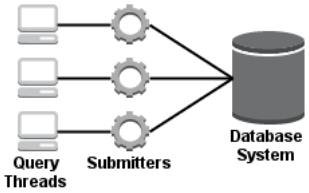

Figure 3: Tester Configuration

The Tester that was created consists of two parts: the Submitter and the Query Threads. The purpose of this study is to verify how good and consistent this System is when put through a series of stress situations, much like those that happen in real life on the public tranportation services. So, a set of Query Threads, whose objective is to simulate several users making deposits and validating their tickets on public transports' machines, was created.

The other part of the Tester is the Submitter, and this a tricky one. On a standalone architecture (which will be explained later), a Submitter is not only not necessary, but also bad for business, because the Query Threads can handle the whole process by theirselves and the existence of a Submitter would only make the system slower. However, on a scalable architecture (which will be also explained later) a Submitter is needed, since the information is divided by several Databases, and something to join that up is required.

4.2 Standalone Architecture

The most basic of the architecture formats that exists for Database displaying is the Standalone one. In this architecture a Central Database is created and it is the only one in the whole system, storing all the Data. Notice that we are only talking about the Data distribution amongst Databases, and not about RAID systems, which are used to have a backup of the Data in case of failure.

Having only one Database to store all the Data makes the system easier to manage, since only one Server needs management, and, in case of a query, only one Database is requested. However, the existence of only one Database makes the system much slower, and inefficient, because there is only one machine to handle all the requests that can be made. Moreover, there is also the fact that all the Data is stored in one pyshical disk, making the Database too big, which makes every single one of the requests even slower.

An example of a Standalone Architecture, the one that will be used in this study, was designed and it should look like the following:

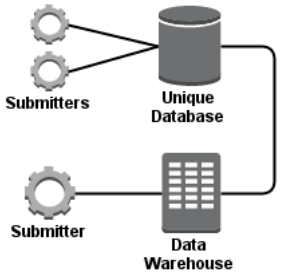

Figure 4: Standalone Architecture

As it can be seen, there is a Unique Database to handle the User related transactions (Validations, Deposits and Sales). These transactions will be made by several submitters that are represented by the Query Threads explained earlier, to simulate a great amount of simultaneous users using the Database.

There is also the Data Warehouse, to where the Data present in the Unique Database is copied to on a daily basis. This Data Warehouse is where all the information ever created is stored, since it can be regularly removed from the Database to make it lighter, and , as such, is where the Revenue Computation is going to be made. This process, that involves a very large amount of calculations and requests to the Data Warehouse, is going to be managed by a submitter, also represented in the picture.

This structure, having only one Database, is a very simple one and works well on a small scale. However, it proves to be inefficient when it begins to be used on a larger scale, where the frequency of transactions is very high and the amount of information stored is enormous. To solve this problem, another architecture was created, in which the Data can be distributed. This architecture is the Scalable one.

4.3 Scalable Architecture

The Scalable Architecture is very similar to the Standalone one, with just one difference: the Database System. In a Scalable Architecture, instead of having a unique Database to handle all the transactions that are made by the User related systems, there is a Cluster of Databases. This Database Cluster can have any number of Databases necessary, and generally, the more Databases there are, the faster the System.

The process behind the Database Clustering and what makes the System faster is the parallelisation of disks and distribution of requests. Unlike a single Database, where all the Data is stored in one physical disk, a Database Cluster has the Data distributed by several physical disks, and with them working in parallel, the same amount of information can be researched in a less amount of time. Notice that each physical disk has his own piece of Data, different from the other ones, and backups are implemented using RAID systems, which will not be explored in this study.

Making the necessary changes to the previous architecture presented, a new design was made, which is presented as follows:

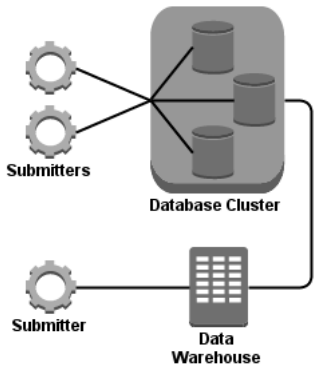

Figure 5: Scalable Architecture with Clustering

As presented in the picture, the only difference to the Standalone architecture is in the Database System, where we now have a Database Cluster instead of a Unique Database. We continue to have a set of submitters, that simulate the several users making requests to the Database, but now these submitters do not represent Query Threads anymore, because, as previously stated, since the information is dispersed through several physical disks, there is a need to have an intermediate agent, a submitter, to join these pieces and return them as one to the Query Thread.

The Data Warehouse, also present on a Standalone architecture, remains unchanged, as the submitter that is responsible for doing the Revenue Computation.

This architecture, where tha Data can be distributed to several Databases, as many as needed, proves to be very efficient on a large scale System, which makes the Database Cluster on of the most used structures today. Of course that using a Scalable architecture has his own disadvantages, such as the fact the management becomes more complex, since the Data has to be well distributed by all the Databases. Another disadvantage, and perhaps the most important one, is how expensive a structure like this can be, since there are more physical systems to mantain. However, and despite all this disadvantages, a scalable structure is strongly recommended to any system that might have a medium sized Database.

4.4 Database Structure

For the purposes of this study, which has as topic the management of a Public Transportation Company, a specific Database Structure had to be created. As discussed earlier in this section, there are three main systems for which we need to save Data. Those systems are the Sales, Deposits and Validations.

The Sales system relates to the purchase of Tickets by the users. These tickets, whose purchase can only be made in stores belonging to the Tranport Management System (TMS) company, are the most important part of the whole Transportation system, since it is through them that the users can use Public Transportation. To represent this Sales system a table with the name Tickets was created.

Another main system is the Deposits one. This system relates to the depasits that are made by the user on their tickets in order to use public transportation. This process can be made in stores belonging to the TMS company, in any machines located near Public Transports, or in applications created specifically to that purpose for smartphones. Making these deposits, the users can use their tickets to travel in public transports. To represent this Deposits system a table with the name Deposits was created.

The last system is the Validations, and it relates to the usage of the tickets by the users when entering a public transport. This process is the most important one to the Revenue Computation, since it is based on it that the calculations on the Data Warehouse can be made, and the gains can be equaly distributed by the three existing companies on the TMS company. To represent this Validations system a table with the name Validations was created.

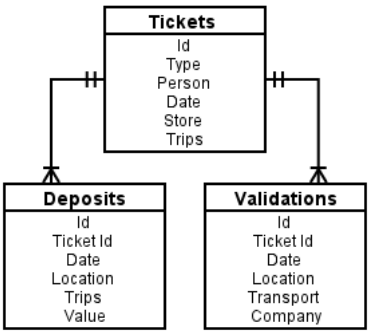

Figure 6: Schema Diagram

As described above, three tables - as many as the number of main systems that were identified to be part of a Public Transportation company - were designed and created to be the container of all the information that this study requires. Each one of these tables and their attributes will now be explained in more detail.

The first table is the Tickets one. In this table, and for the purposes of this study, six attributes were created. The “Id” serves as the identifier of the ticket itself, and, as such, it is the primary key of the table. The type “Integer” was used for this attribute. Then there are another two attributes, “Type” and “Person”, both with the type “Text” and represent the type of the ticket (normal or premium) and the person who bought it, respectively. The “Date” is a “DateTime” attribute and represents the date and time at which the ticket was sold to the user in a store. This store in which the user bought his ticket is represented by the attribute “Store” and has the type “Text”. The last attribute is the “Trips”, with the “Integer” type, and indicates how many trips a user can make in public transports with their ticket before making another Deposit.

The second table is called Deposits, and six attributes were created to represent this process. The first one, “Id”, has the type “Integer” and it is the identifier of the deposit, and so, the primary key. Following that there is the “Ticket Id”,an “Integer”, which links this table to the Tickets table, and acts as a foreign key. Finally there are four values that represent the remaining information of each Deposit, “Date”, “Location”, “Trips” and “Value”, the first two being a system date (“DateTime”) and a string (“Text”) respectively, describing the time and place in which the deposit took place, and the latter ones, two “Integers” that count the number of trips bought and the value payed by the costumer.

Finally, the third table, Validations, represent each validation the user have made, and it contains six attributes. First, the "Id", which is an "Integer" type, and serves as the identifier of the validation, i.e. , the primary key. The second attribute, "Ticket Id" is a Foreign key from the table Tickets, and it has the same proprierties and functions. The attribute "Date" is a "DateTime" attribute and represents the date at which the validation was made. The three last attributes, " Location", "Transport" and "Company", have the type "Text", and represent the transport stop at which the validation was made, the transport identifier, and to which company the transport belongs, respectively.

5. CASE STUDY

In this section the actual study will be described. Since this study has no reference, the database's data needs to be generated, and here that will be shown. The benchmark that was created specifically will also be explained. It's queries and their objective, as weel as the Revenue Computation and how it's done.

5.1 Database Generation

Since this study, or a study concerning Public Transportation was not done or publicly available yet, the Database's Data generation had to be done. For this generation, a real case was simulated, and the generation became divided in two parts. The first one is an Initial Generation of Tickets and the second one a Stress Test that involves the generation of both Deposits and Validations. This division was seen as an opportunity to test not only the main purpose of this study, the Revenue Computation, but also to test how the Database System (not the Data Warehose) works on a real life situation, where people make deposits and validations with their tickets.

5.1.1 Initial Generation

The data to be generated in the first stage of the Database Generation are the tickets. Since a fully developed Transport Management System is already in place in most cases, almost all the Tickets are already sold and in use. Also, since the amount of sales per amount of time is very low, this table was chosen to be generated beforehand and loaded direcly onto the database, so as to simulate a real case.

To generate the Tickets, a script in Python language had to be made. This script should receive the number of rows to be generated and write the randomly generated data to a file called "tickets.db". In this file the columns are separated by Vertical Bars (|) and the rows are separated by New Lines (\n).

This script should look like the following:

Algorithm 1 Ticket Generation Script Pseudocode

```
1: for number of rows do
2:     id ← getNextId()
3:     type ← getRandomType()
4:     name ← getRandomName()
5:     date ← getNextRandomDate()
6:     store ← getRandomStore()
7:     trips ← getRandomTrips()
8:     writeToFile(id|type|name|date|store|trips)
```

Finally, to load the data to the Database, the specific Database's data load command has to be used.

5.1.2 Stress Test

The second stage of the Database Generation is the Stress Test, where the Deposits and Validations are inserted to the Database. Since these happen very often and everyday in a Public Transport Company, they will be used to test the Database System in a stress situation, in order to simulate what happens in a real life scenario.

To do so, two Python language scripts were made. The first one is similar to the one used on the ticket generation, explained above, but instead of creating a file from which the data could be directly loaded to the database, the script creates files containing Database's insert queries. However, and to simulate a real case Database System, these insert queries should not contain just one row of information, since the public transports are not always connected to the network.

Another issue to consider when generating these insert commands is the simultaneous users that make concurrent transactions to the Database System. To represent this, instead of creating one file with all the data to be inserted, several files were created, and so each one of the users can have their own file to make transactions to the Database System at the same time as the others, thus recreating a more accurate depiction of what would actually happen in a concurrent system.

The next algorithm shows how this first script should be:

Algorithm 2 Insert Generation Script Pseudocode

```
1: for number of inserts do
2:     query ← 'INSERT INTO TABLE VALUES'
3:     for size of insert do
4:         columns ← getRandomValues()
5:         query ← query + columns
6:     writeToFile(query)
```

After running this script, several compressed files containing the Database insert queries are created, ready to be used by the second script.

This second script created should then read those files created by the first one and execute the insert queries to the database. To simulate real life cases, where multiple users make concurrent transactions to the Database System, this script makes use of the CPU multi-processing capabilities, in order to achieve the highest overload possible in the Database, so to see how it works under stress. After each one of the queries submitted to the Database the script saves its times to several files, so an analysis comparing different system architectures can be made afterwards.

In the next algorithm, a representation of the concurrent process is shown:

Algorithm 3 Stress Test Script Pseudocode

```
1: createDatabaseConnection()
2: for rows in input file do
3:     time ← getCurrentTime()
4:     submitInsertQuery()
5:     writeToFile(getCurrentTime()−time)
```

When all the concurrent processes are finished doing their respective queries to the Database, several files containing

the times are created for posterior analysis. The number of files will vary with the number of processes.

5.2 Benchmark

In order to perform this study, whose objective is to compare different system architecures and make some conclusions about their performance in a Transport Management System environment, a Benchmark was made. This Benchmark is divided in two parts. In the first part a set of twelve queries was created. This set of queries aims to simulate day-to-day operations that can possibily be made by the managers of the Transport Management System, in order to get some usefull information about the Company and all its components. As such, the presence of the Business related information was taken into account and it is very present in the set of queries. The second part of the Benchmark is the most relevant one, as it is the main object of this study, the Revenue Computation, the details of which will all be explained further.

5.2.1 Query Definitions

In this section, the first part of the created Benchmark will be explained in more detail. In order to get the most useful information out of the Database, the set of queries was divided in several parts. The first one contains only queries that use the Tickets table and relate to queries 1 and 2. The next three queries, 3 through 5, only use the Deposits table, and the following three, 6 through 8, the Validations table. Then, combinations between tables were made. The 9^{th} query aggregates the Tickets and Deposits tables, the 10^{th}, the Tickets and Validations, and the 11^{th}query, the Deposits and Validations tables. Finally, the last query uses all the tables. For each query in the created set, a brief summary and the Business perspective behind it will then be shown.

Q1 - Number of sales by Year and Ticket Type.

Summary: This query only uses the Tickets table and counts how many sales were made, dividing them by Year and Ticket Type.

Business Context: For the manager of the Transport Management System, this query can be very useful, since it provides information about the evolution of ticket sales along the years.

```
SELECT
  YEAR(t_date) AS 'Year',
  t_type AS 'Ticket Type',
  COUNT(*) AS 'Number of Sales'
FROM
  tickets
GROUP BY
  1,
  2
ORDER BY
  1,
  2
;
```

Q2 - Most used stores on Ticket Selling.

Summary: This query outputs the number of sales per store.

Business Context: This query can be useful for a manager or analyst of the Transport System to see the distribution of the sales throughout the city, as well as the most and least used stores. This may help decide placement for future selling spots.

```
SELECT
  t_store AS 'Store',
  COUNT(*) AS 'Number of Sales'
FROM
  tickets
GROUP BY
  1
ORDER BY
  2 DESC,
  1
;
```

Q3 - Pricing in Deposits by Month.

Summary: This query shows the total earning of the whole ticket selling system by month.

Business Context: This query can be potentially used to keep track of the time periods with most affluence, so as to keep track of the busiest times of the year when the system might take extra workloads.

```
SELECT
  CONCAT(YEAR(d_date), ' - ', MONTHNAME(d_date)) AS
       'Date',
  ROUND(SUM(d_value),2) AS 'Gain in Deposits'
FROM
  deposits
GROUP BY
  1
ORDER BY
  d_date
;
```

Q4 - Number of Deposits by Month.

Summary: This query outputs the total amount of deposits by month.

Business Context: Much like the previous query, this one helps to give an overview of the sales over a long timespan, so as to draw conclusions about the season variations of sales.

```
SELECT
  CONCAT(YEAR(d_date), ' - ', MONTHNAME(d_date)) AS
       'Date',
  COUNT(*) AS 'Number of Deposits'
FROM
  deposits
GROUP BY
  1
ORDER BY
  d_date
;
```

Q5 - Most used stores and machines in Deposits.

Summary: This query is very similar to Q2, in that it too outputs the number of uses of each store and machine sorted from the most used to the least used.

Business Context: Just like Q2, it can be useful to draw conclusions about the most frequently used machines or stores. It may also help any manager to decide placement for future selling spots.

```
SELECT
  d_location AS 'Location',
  COUNT(*) AS 'Number of Deposits'
FROM
  deposits
GROUP BY
  1
ORDER BY
  2 DESC,
  1
;
```

Q6 - Validation percentage by Transport Type.

Summary: This query outputs the usage percentage of each individual transport type in the system.

Business Context: Although this query does not directly show the revenue percentage to distribute, it can be used to get the percentage of the actual use of each transport since the system has been started.

```
SELECT
  totals.type AS 'Transport Type',
  CONCAT(ROUND(totals.total/total.total*100,2),' %') AS
      'Validations Percentage'
FROM
  (SELECT
    COUNT(*) AS total
  FROM
    validations
  ) AS total,
  (SELECT
    v_company AS type,
    COUNT(*) AS total
  FROM
    validations
  GROUP BY
    1
  ORDER BY
    2 DESC
  ) AS totals
;
```

Q7 - Number of Validations by Day.

Summary: This query outputs the number of validations per day on the system.

Business Context: This query might prove useful for statistics sake, since it can show the overtime behaviour of the usage of the whole transport system.

```
SELECT
  CONCAT(DATE(v_date), ' - ', DAYNAME(DATE(v_date))) as
      'Day',
  COUNT(*) AS 'Number of Validations'
FROM
  validations
GROUP BY
  1
ORDER BY
  v_date,
  2
;
```

Q8 - Most used Transports.

Summary: Much like Q6, this query outputs the most used transports to give some information on the most commonly used means of transportation.

Business Context: Again, like Q6, Q8 shows the popularity of each transport type, albeit in an absolute, rather than relative fashion.

```
SELECT
  v_transport AS 'Transport',
  COUNT(*) AS 'Number of Validations'
FROM
  validations
GROUP BY
  1
ORDER BY
  2 DESC,
  1
;
```

Q9 - Deposits information by Person.

Summary: This query lists information about the deposits made by each user.

Business Context: This query might be useful to check how many deposits a user has made within the system, the most used locations by the users to make deposits and the preferences for a certain ticket type.

```
SELECT
  t_person AS 'Person',
  t_type AS 'Ticket Type',
  SUM(d_trips) AS 'Total Trips',
  frequencies.location AS 'Most Used Location'
FROM
  tickets,
  deposits,
  (SELECT
    locations.id AS id,
    locations.location AS location
  FROM
    (SELECT
      d_t_id AS id,
      d_location AS location,
      count(d_location)
    FROM
      deposits
    GROUP BY
      1,2
    ORDER BY
      3 DESC
    ) AS locations
  GROUP BY
    1
  ORDER BY
    1
  ) AS frequencies
WHERE
  t_id = d_t_id AND
  t_id = frequencies.id
GROUP BY
  t_id
ORDER BY
  t_id
;
```

Q10 - Validation percentage by Ticket Type.

Summary: This query measures the most used between the types of tyckets offered by the company.

Business Context: This one has use to see which of the ticket types is more popular, which might be useful to decide

over price changes or special promotions, since it gives an overview on what most people use.

```
SELECT
  totals.type AS 'Ticket Type',
  totals.company AS 'Transport Type',
  CONCAT(ROUND(totals.total/total.total*100,2),' %') AS
       'Validations Percentage per Ticket Type'
FROM
  (SELECT
    types.type,
    COUNT(*) AS total
  FROM
    (SELECT
      t_type AS type,
      v_company AS company
    FROM
      tickets JOIN validations
    WHERE
      t_id = v_t_id
    ) AS types
  GROUP BY
    1
  ) AS total,
  (SELECT
    types.type,
    types.company,
    COUNT(*) AS total
  FROM
    (SELECT
      t_type AS type,
      v_company AS company
    FROM
      tickets JOIN validations
    WHERE
      t_id = v_t_id
    ) AS types
  GROUP BY
    1,
    2
  ) AS totals
WHERE
  total.type = totals.type
GROUP BY
  1,
  2
ORDER BY
  1,
  2,
  3 DESC
;
```

Q11 - Number of Validations by Deposit.

Summary: This query displays the ratio between validations and deposits made by each person.

Business Context: This query should show an estimate on how many trips each user buys each time, to maybe gauge the potential of possible promotions. Arguably, this is not significant busines- context-wise, although it does prove useful in benchmarking the the use of both the validations and deposit tables, to better understand potential slowdowns in future functions.

```
SELECT
  d_t_id AS 'Person',
  ROUND(validation.total/deposit.total,2) AS
       'Validations per Deposit'
FROM
  deposits,
  (SELECT
    d_t_id AS id,
    count(*) AS total
  FROM
    deposits
  GROUP BY
    1
  ) AS deposit,
  (SELECT
    v_t_id AS id,
    count(*) AS total
  FROM
    validations
  GROUP BY
    1
  ) AS validation
WHERE
  d_t_id = deposit.id AND
  d_t_id = validation.id
GROUP BY
  1
ORDER BY
  1
;
```

Q12 - Person's full information.

Summary: Finally, Q12 retriefves all information of a given user within the system. Every ticket purchase, every deposit and every validation should appear for each user.

Business Context: This query can be used to trace the movements of each user to detect any incongruent uses and detect possible system errors or fraud.

```
SELECT
  tickets.person AS 'Person',
  tickets.type AS 'Ticket Type',
  tickets.trips AS 'Trips Left',
  deposits.total AS 'Total Deposits',
  deposits.sum_trips AS 'Total Trips Deposited',
  deposits.sum_value AS 'Total Money Deposited',
  IFNULL(validations.total,0) AS 'Total Validations',
  CONCAT(IFNULL(validations.bus,0.00), ' %') AS 'Bus
       Usage',
  CONCAT(IFNULL(validations.subway,0.00), ' %') AS
       'Subway Usage',
  CONCAT(IFNULL(validations.train,0.00), ' %') AS
       'Train Usage'
FROM
  (SELECT
    t_id AS id,
    t_person AS person,
    t_type AS type,
    t_trips AS trips
  FROM
    tickets
  ) AS tickets LEFT JOIN
  (SELECT
    d_t_id AS id,
    COUNT(*) AS total,
    SUM(d_trips) AS sum_trips,
    ROUND(SUM(d_value),2) AS sum_value
  FROM
    deposits
  GROUP BY
    1
  ) AS deposits ON tickets.id = deposits.id LEFT JOIN
  (SELECT
    v_t_id AS id,
    COUNT(*) AS total,
    ROUND(AVG(v_company = 'bus')*100,2) AS bus,
    ROUND(AVG(v_company = 'subway')*100,2) AS subway,
    ROUND(AVG(v_company = 'train')*100,2) AS train
```

```
  FROM
    validations
  GROUP BY
    1
  ) AS validations ON tickets.id = validations.id
ORDER BY
  tickets.id
;
```

5.2.2 Revenue Computation

The second part of the created Benchmark is the main purpose of this study, the Revenue Computation. As said earlier, the user can use the same ticket on both three tranports available (bus, subway and train), which alleviates the work of the users, and condense all the information about a certain user in one place. But, there is also another specificity in this ticket. If the user validates his ticket within a margin of two hours after the first validation, both these validations count as one trip, and not as two trips. This is a way to benefit the users that frequently use more than one transport (not necessarily diferent transport types) to reach their destination. This system implies that some calculations to evenly distribut the profit across the three companies has to be made. For example, if within two hours a user validates his ticket two times in subways and one time in a bus, then, two thirds of the trip cost goes to the subway company and the remaining one third goes to the bus company.

To do this calculation, a script in Python language was made. This script, that is supposed to run in the Data Warehouse, does not make use of the multi-processing capability of the CPU, in order to get consistent results, so to a easier comparison between different System Architectures. To get the profit distribution between the companies, the script starts by selecting all validations for each user, ordering them on the way. Then, it sweeps the validations and, for each two hour interval, counts how many validations were made and wherein company they were made. During this process, all the trip fractions corresponding to each company are sumed up. This will give us at the end the distribution percentage for each company.

The next algorithm should give a general idea of how the script should be:

Algorithm 4 Revenue Computation Script Pseudocode

```
 1: finalTrips ← [0,0,0]
 2: createDatabaseConnection()
 3: for each user in the database do
 4:     getUserValidations()
 5:     beginDate ← firstValidationDate
 6:     trips ← [0,0,0]
 7:     for each validation do
 8:         if validationDate − beginDate < 2 hours then
 9:             trips[transportType]++
10:         else
11:             finalTrips ← finalTrips + trips
12:             beginDate ← validationDate
13:             trips ← [0,0,0]
14:             trips[transportType]++
15: display(finalTrips)
```

After the execution of this script the profit distribution percentage is then returned, so to be used by the managers of the Transport Management System to guarantee that each company receives what it deserves.

6. EXPERIMENTAL SETUP

This section will show the setup that was used to realize the study, both in standalone and cluster. The several machines that were used to do that, as well as the database and its properties will be presented. The structure of machines in the cluster and the database engine that were used will also be refered.

6.1 Structure

For this study, even being supposed to do a simulation a Data Warehouse architecture, where there is a Database System and a Data Warehouse, this will not be used. As explained in that section, the Database System is where the INSERT, UPDATE and DELETE commands (Initial Generation and Stress Test) are made, and after transfer the data to the Data Warehouse, it is in the latter that the SELECT commands (Benchmark) are made.

To avoid the use of different machines, the Data Warehouse will be simulated on the same machine as the Database System, because, like the Data Warehouse is a stable system (without modifications to the data), so it is the Database System after the Initial Generation and Stress Test. To summarize, only one machine (or set of machines for the scalable version) will be used. Initially it will be used as a Database System, to do the Generation and Stress Test, and after that as a Data Warehouse, to do the Benchmark.

6.2 Database

As said earlier, the data for the database is generated specifically for this study, and both algorithms were already shown and explained. There are three database tables, Tickets, Deposits and Validations, whose specifications are as follows:

Table	Length (lines)	Size (MB)
Tickets	100.000	10
Deposits	1.000.000	110
Validations	10.000.000	1200

Table 1: Database specifications

As it can be seen, the whole database takes about 1300 MB, which is little, but still suficient for the study. There are , on average, ten deposits for each ticket (user) and one hundred validations for each ticket, which is a very reasonable value for the purposes of the Revenue Computation. The word average was used, because in order the get the random values a normal distribution was used, making even better data for the Benchmark.

For the tables Deposits and Validations the data will be inserted through SQL INSERT commands, instead of a LOAD command,as used fot the table Tickets. But for each one of these INSERT commands, more than one line is introduced, as shown in the algorithm 2, by the variable “size of insert”. For the Deposits table, the value that was used was 1.000 lines per insert, and 50.000 lines for the Validations table.

6.3 Standalone

For the standalone version of the study two computers were used. The main difference between them, that will allow to have more comparison factors, is that one uses a HDD and the other a SSD as storage. For nomenclature purposes the names **gmlt** and **jvub** were adopted. The next table shows the specifications of both machines:

	gmlt	jvub
OS	Arch Linux 3.14.4	Arch Linux 3.14.4
	64 bits	64 bits
Storage	Hard-Disk Drive	Solid-State Disk
	5400 rpm	-
	75 MB/s (read)	300 MB/s (read)
Memory	4 GB	8 GB
	1333 MHz	1600 MHz
CPU	Intel i7 Quad-Core	Intel i7 Dual-Core
	1.6 GHz	2 GHz

Table 2: Standalone machines specifications

The database used on both machines was MySQL, with the engine itself being MariaDB[6]. The configuration files were kept the default ones on both computers, with one exception for the "max_allowed_packet" size, that was increased in order to do the INSERT commands, which can take 50.000 lines each.

6.4 Scalable

For the scalable version of the study four computeres were used. To apply the scalable architecture, as explained earlier, the MySQL Cluster was the choice, since it is basically the same Database Management System that was used in the Standalone version. Three of the four machines were used as Storage nodes (or Data nodes) and the fourth machine was used as the Management node (or SQL node). In the following table the four machines specifications are presented:

	Management Node	Storage Nodes
OS	CentOS 2.6.32	CentOS 2.6.32
	32 bits	32 bits
Storage	Hard-Disk Drive	Hard-Disk Drive
	7200 rpm	7200 rpm
	75 MB/s (read)	75 MB/s (read)
Memory	1.5 GB	1.8 GB
	667 MHz	667 MHz
CPU	Intel Celeron	Intel Celeron
	2 GHz	2 GHz

Table 3: Scalable machines specifications

As said before, the database that was used to support and connect all four machines was MySQL CLuster. These four machines are on a local network, connected with themselves through a switch, with Category 5 network cables. That means that the top speed between them is about 1.000 Mb/s, or 12 MB/s, which is a speed below the minimum requirements for this kind of architecture.

7. EXPERIMENTAL RESULTS

After a detailed explanation of the setup that was used to perform this study, this section presents the results of the case study, shown earlier, which involves the testing of the standalone and scalable versions. Initially, some expectations for the tests, based on the specifications of both architectures and how they work, are described, and after that, the results are shown. Finally some conclusions and observations about the results are made.

7.1 Standalone

7.1.1 Expectations

Given the nature of the queries, the database engine and the machines used, we expect jvub to have better times overall. Difference between processors is small, and the main discriminant is the disk speed instead. Since jvub has a SSD, it is expected to out-perform gmlt by a considerable margin in most of the queries.

On the other hand, however, if there were any queries that relied more on arithmetic and logic operations (such as averages or joins), they would prove faster on gmlt given its superior processing power and cache size per core. Since no such queries were made, we cannot see this advantage in the tests. The number of processor cores should not affect the results, since the processing is sequencial in these tests.

7.1.2 Results

Insertion.

Two of the tables were loaded into the database via Insert queries. To simulate real-world stress, we used a distributed approach, with 10 different processes for each of the tables at the same time. The two following graphs describe the insert times. Each point represents the average time it took the threads to handle each of its 20 Insert queries.

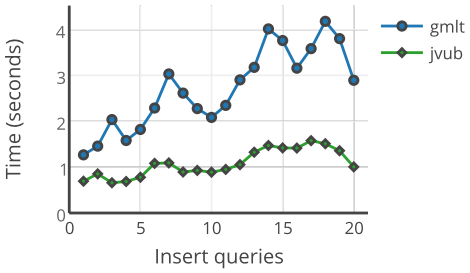

Figure 7: Deposits table insertion

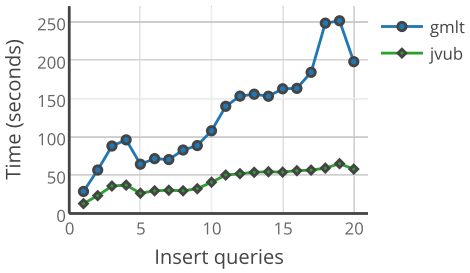

Figure 8: Validations table insertion

As we can see, the reduced size of the deposits' table makes it much faster to process. By looking at the validations' times, we can clearly see where the deposits' processes finished. By the time the 4th validation query ends, we can see a drop in the times, since the processor only needs to manage 10 processes from this point on. We can also see that the bottleneck is the disk writing speed, since the gmlt machine has more processor cores and faster clock speed. Finally, we can clearly see an increase in the queries' time as the database grows, since it needs its indexes to update.

Queries.

The following graphs describe the time needed to process each set of queries per system.

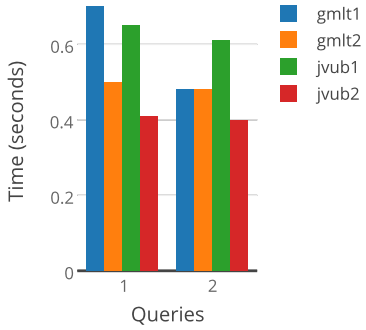

Figure 9: Using only Tickets table

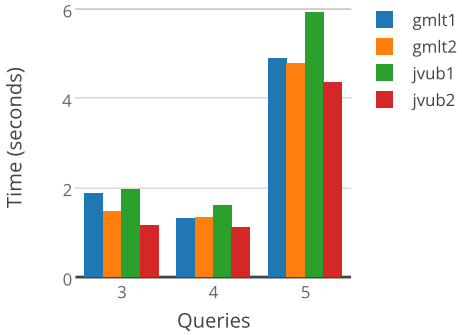

Figure 10: Using only Deposits table

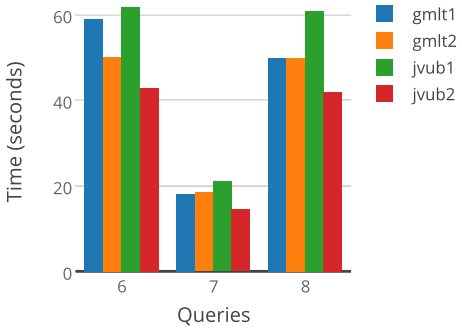

Figure 11: Using only Validations table

Looking at these the graphs we can draw some conclusions about the performance.

The tickets' table is very small, so the difference between mahines turns out to be quite small. This is what was expected initially.

In the other two tables on the other hand, the variation is more significant, more so in the validations table. We can see that, for a database this size, the difference between the two machines grows proportionally to the size of the tables. This can be observed with gmlt being about 5% faster in the first run and 5% slower in the second run.

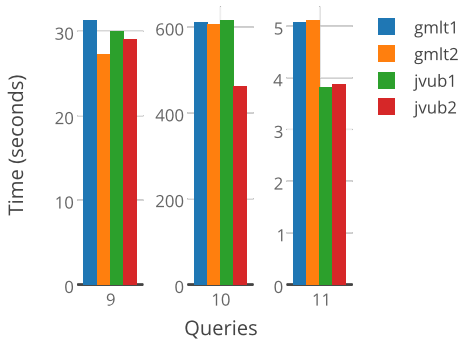

Figure 12: Using each pair of tables

Query 9 uses the tickets and deposits tables. Query 10 uses the tickets and validations, and query 11 uses deposits and validations. This way we tested every pair of tables that could be queried.

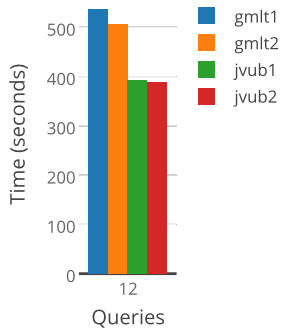

Figure 13: Using all tables

In these graphs, the queries' times become more consistent, implying the cached memory was not used. These queries have the added overhead of having to join two tables, so the gmlt machine does have some advantage, but not enough to have better results.

Revenue Computation.

The next chart describes the average time it took each machine to compute 1000 queries at a time.

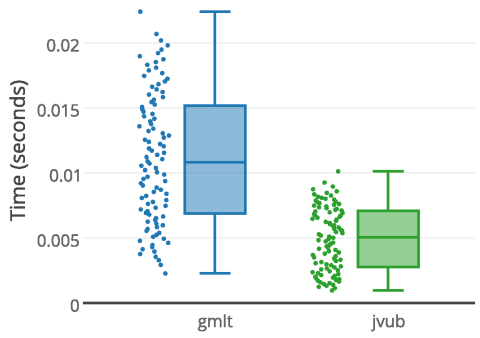

Figure 14: Revenue Computation

Here we can see again that a faster I/O speed outperforms a faster CPU clock speed, with the SSD machine taking half the time to run, and a lot less deviation from the average time.

7.1.3 Conclusions

As we can see, looking at all the results, the best strategy when implementing a standalone approach to this problem is clearly using a disk with faster reading and writing speeds. Even though in some cases it seems to be outperformed by the gmlt machine, the much faster speed in calculating the revenue makes the SSD machine a better choice overall. Another factor to consider, is the size difference between the available RAM memory and the database. Since both machines have a large ram, not a lot of reads need to be made after loading the database into memory. This aproximates the times between the two types of disks and negates most of the overhead. In conclusion, it seems clear that having a machine with good disk I/O seems to be a relatively good and inexpensive solution for databases of this size.

7.2 Scalable

7.2.1 Expectations

Despite having three machines dedicated to process the queries, we should not see a huge improvement in processing the queries.

The network we had access to perform the tests had two downsides to it: the machines used had weaker processors and slower disks than the standalone tests, and the network had a relatively slow connection between the machines. Also, the setup used had one management node, through which all the data must go through to be joined together.

Since the queries return large tables overall, and not a lot of heavy calculations need to be done, we expect the single management node, as well as the slow connection, to act as a bottleneck, and show poorer results compared to the standalone tests. We might see an exception in the last query, which is the heaviest, computational-wise.

7.2.2 Results

Insertion.

The following graphs show the time it took to run all the insert queries in all three systems. Much like 7.1.1, each point represents the average time it took each of the 10 processes to process each of the 20 queries.

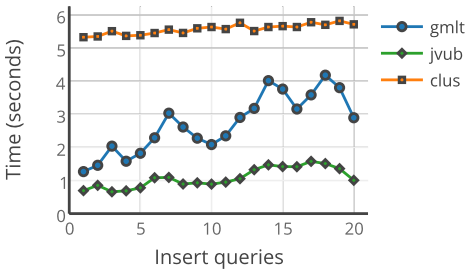

Figure 15: Deposits table insertion

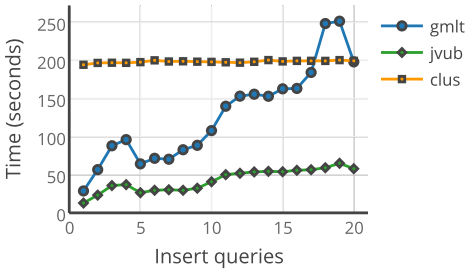

Figure 16: Validations table insertion

This time around we compare the cluster results to the standalone implementations. It is clearly slower to process the data, since only one machine does the processing and distributes the commands over the other ones. We can see however, that it seems to be more constant. Since the processes of actually inserting and sorting the database is done independently, it takes a smaller toll on the SQL node and allows it to run more stably. This can be seen happening when it becomes faster than gmlt in the latter queries.

Queries.

The following histograms contain the results of the select queries for each of the systems.

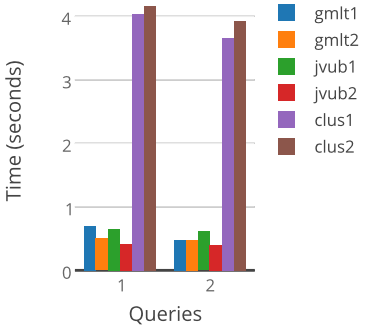

Figure 17: Using only Tickets table

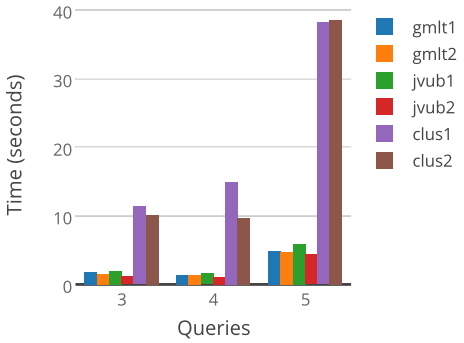

Figure 18: Using only Deposits table

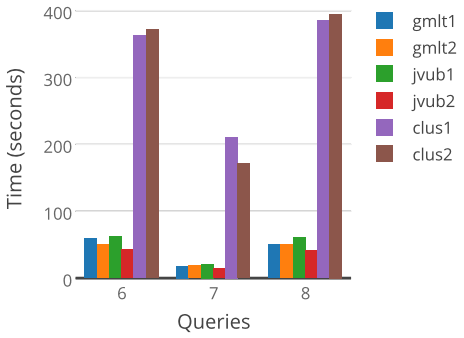

Figure 19: Using only Validations table

Just as expected, the slow network and clock speeds bottleneck the system, leaving it to show worse results.

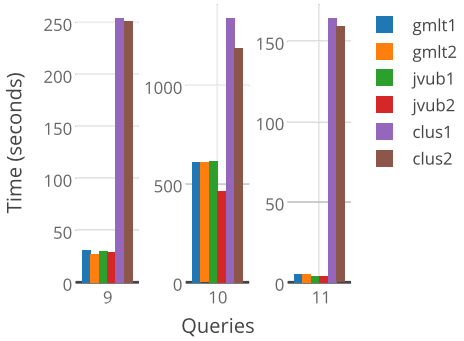

Figure 20: Using each pair of tables

Query 9 uses the tickets and deposits tables. Query 10 uses the tickets and validations, and query 11 uses deposits and validations. This way we tested every pair of tables that could be queried.

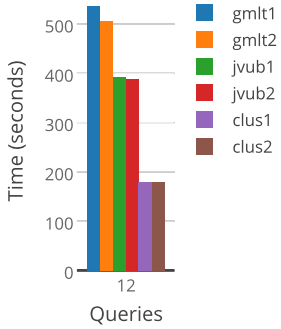

Figure 21: Using all tables

Even in the composite queries, there needs to be one computer that joins the output tables, so we can see the slow down once more. The only exception to this trend is the very last query. Since this query makes use of a lot of arithmetic operations before the join instruction, it benefits greatly from being processed in a parallel fashion. Leaving the final table to be briefly joined by the SQL node, which has to do no more additional calculations of its own.

Revenue Computation.

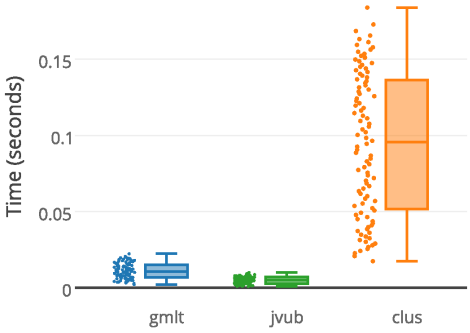

Figure 22: Revenue Computation

Finally, the revenue computation is a much slower process, since it takes a lot of computing power as well as disk I/O access. Because of this we can see that the slower machine simply cannot compete with the ones used in the standalone versions. Since this operation would have to be calculated only once a day, it could be done at a time of lesser traffic, such as at night. Although it took a few hours, it still runs within a feasible time constraint, and could still be done without much real delay. More so if the machines and network were just slightly faster.

7.2.3 Conclusions

In sum, although the distributed aproach is slower than the standalone, it should be kept in mind that it is still a viable option, and it was made using a very non-favorable setup, with slower machines and a benchmark design to work against its advantages. With this in mind, it still preformed reasonably well, and with some setup it should still be feasible to implement it.

8. CONCLUSIONS AND FUTURE WORK

In summary, we can attest that both a standalone system or a distributed approach are both indeed feasible to perform this task, at least in the scale of a city with a few hundreds of thousands habitants. Despite the slower times in the distributed system, they are still manageable, although they do require a closer management and setup by the system administrator.

However, in order to make this solution scalable a few changes will need to be made: there should be an investment on a fast network, so as to really take the advantage of the distributed database. Not only that, but most of the queries should be written in such a way that the arithmetic operations are done before merging the result tables. This way, the storage computers will handle a fraction of the workload simultaneously, thus speeding up the processing and without overloading the SQL node, which is, in the current system, a SPOF (assuming no raid or data replication).

A noteworthy detail is the fact that the technology was used out-of-the-box, and no specific optimizing was made to help the cluster. This was done as a proof-of-concept, to test if it was within reason to approach the problem in this manner.

For the future, we suggest that an effort to modify the queries should be made a priority, since these were made with stress-testing specifically in mind. Another solution may also lie in distributing the revenue computation across more than one machine, either through a distributed system in a low-level language, or maybe even by integrating it within the select queries themselves, if at all possible. This will certainly make more use of the available resources and reduce idle time.

Any single one of these strategies, or a combination of them should improve the overall performance of the system, and good results are expected in that path.

9. REFERENCES

[1] Milton Sêco and Rodrigo Nogueira. Ticket direct. May 2013.

[2] MySQL, mysql.com/products/standard.

[3] MySQL Cluster, mysql.com/products/cluster.

[4] Apache Hadoop, hadoop.apache.org.

[5] Apache Hive, hive.apache.org.

[6] MariaDB, mariadb.com.