Tampere University

Sergei Panarin

COMPARING THE IMPACT OF VIRTUAL DISTRIBUTED FILE SYSTEM ALLUXIO ON THE PERFORMANCE OF SQL QUERIES DONE IN HIVE, SPARK AND PRESTO.

Bachelor's thesis
Faculty of Engineering and Natural Sciences
Examiners: Prof. Davide Taibi
April 2020

ABSTRACT

Sergei Panarin: Comparing the impact of virtual distributed file system Alluxio on the performance of SQL queries done in Hive, Spark and Presto.
Bachelor's thesis
Tampere University
International Bachelor's Degree in Science and Engineering
April 2020

Data architecture in the cloud is an important topic nowadays and finding solutions to improve data accessibility and performance of various applications working with databases is a crucial task for many businesses and researchers around the globe. In this work the basic performance of 3 SQL architectures integrated with Alluxio VDFS was tested and compared. Tests were done in the cloud storage provided by S3. Alluxio showed increased performance compared to querying data directly from S3.

Keywords: Data orchestration, SQL, cloud, Alluxio

PREFACE

The idea to test Alluxio on top of SQL engines came to me while I was doing my internship for Nokia in a DevOps team. Our supervisor was highly interested in the topic and after having worked in this group for half a year, I decided to focus my thesis on this topic. I would like to thank my team, my University supervisor as well as my family and friends for supporting me through the process of conducting this work.

Tampere, 6th April 2020

Sergei Panarin

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

API	Application Program Interface
AWS	Amazon Web Services
BB	BigBench Benchmark
CLI	Command Line Program
HDFS	Hadoop Distributed File System
I/O	Input/Output designation
QL	Query Language
SQL	Standard Query Language
SSB	Star Schema Benchmark
VDFS	Virtual Distributed File System

1. INTRODUCTION

Big data processing is one of the most important challenges that tech companies face nowadays. The ability to quickly manipulate large amounts of data is crucial when it comes to delivering reliable and innovative solutions in the tech industry. Most of the data is stored in various databases that provide data structuring solutions which make it easier to store and preserve the data. Some of the examples of such databases are Hadoop's DFS [1] and Amazon S3. In order to be able to manipulate or access the data in those databases, the Structured Query Language (SQL) was introduced. SQL uses queries to retrieve the data in a certain database and manipulate it. Many frameworks use SQL for data processing and analysis such as Apache Spark or Apache Hive.

In this work Apache Spark, Apache Hive and Presto were used. Apache Spark is a computing framework that is based on Hadoop MapReduce structure [2], it works on a cluster and processes data sets located there. Hive, on the other hand, is a database, that comes with its own set of tools, including its own SQL: HiveQL. Presto is yet another SQL engine, it can be used with a variety of databases, like AWS S3, Hadoop, MongoDB, etc.

Alluxio, formerly "Tachyon", is a virtual file system, that sits between the storage level (Hadoop, S3, etc) and the compute level (Spark, Hive, etc). Alluxio is scalable, very customizable and relatively cheap.

Theoretically, Alluxio should accelerate the querying process significantly by storing the data in a cache, thus improving performance and reducing querying times.

The goal of the research is to try and set-up 3 SQL engines mentioned above with and without Alluxio, put tabulated data in a cloud storage, query the data by using these set-ups and compare the performances. These 3 SQL architectures have been chosen for their differences and their popularity. We want to see how Alluxio affects different types of architectures and we would like to test some of the most widely used big data services available.

In order to achieve that goal, we are going to:

- Generate the data to be tested by using the Star Schema Benchmark's [3] data generator.
- Upload the data to the selected S3 storage.
- Query the data by using our SQL engines and observe the results.
- Query the data by using our SQL engines coupled with the Alluxio file system and observe the results.
- Compare the results from the previous 2 steps.
- Compare the results for Spark with the work done by Rajaram and Haridass [4].

2. THEORETICAL BACKGROUND

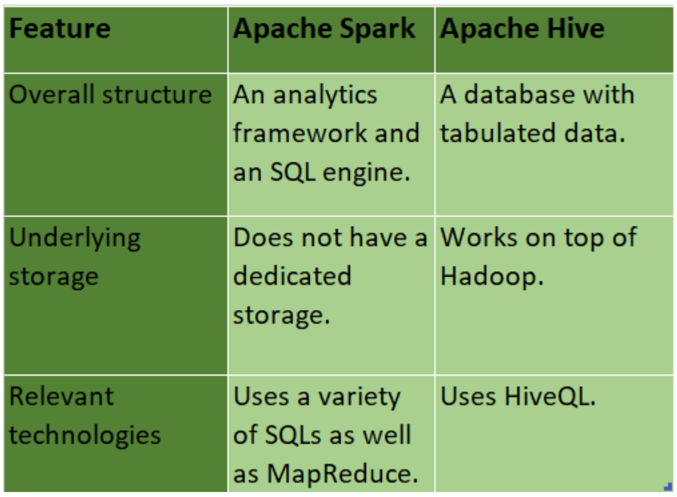
2.0.1 SQL architectures

Much of the data in modern databases is usually stored in some tabulated format. Accessing and manipulating (querying) that data in an efficient manner is a paramount problem, which has had many solutions. A vast amount of query languages have been introduced over the years, all of which solve the problem of how to read and write that data. The complexity of data manipulation, however, has grown as well and tasks have become much more diverse and nuanced. New platforms have been introduced over the years to meet the demand in data management.

One such platform is Apache Hive. Hive is built on top of Hadoop [1], a library of software that revolves around using clusters for processing large sets of data. Hive provides its own HiveQL language as well as tools for accessing and manipulating data stored in the underlying storage. One of the signature features of Hadoop is MapReduce technology, which divides and reassembles the data for better performance and memory usage. Everything else is managed by the YARN manager. Hive can also be configured to work with other storage solutions such as Amazon S3.

Another SQL architecture is Apache Spark [2]. Unlike Hive, Spark is a framework built for analytics and doesn't have a dedicated underlying storage. It provides many application program interfaces (APIs) for the following languages: Python, Java R, Scala. It also supports a variety of QLs and, like Hive, uses MapReduce.

Differences between Spark and Hive can be seen in figure 2.1.



Feature	Apache Spark	Apache Hive
Overall structure	An analytics framework and an SQL engine.	A database with tabulated data.
Underlying storage	Does not have a dedicated storage.	Works on top of Hadoop.
Relevant technologies	Uses a variety of SQLs as well as MapReduce.	Uses HiveQL.

Figure 2.1. *Differences between Spark and Hive*

The final SQL engine from our work is Presto [5]. Presto is openly distributed modern analytical engine, that can be used in combination with a multitude of data sources and can even query data from multiple databases simultaneously.

2.0.2 Alluxio

Alluxio is a Virtual Distributed File System (VDFS) that is used in data stacks. It can help achieve several goals, including: global data accessibility for compute frameworks, in-memory data sharing, high I/O performance, efficient use of network bandwidth, and optimization of the compute and storage architecture [6]. Alluxio provides an extra data orchestration layer between the compute and storage layers. The schema can be seen in figure 2.2 . Alluxio also gives universal access to the databases in the storage layer, an API and a namespace to the compute layer applications. Those applications can cache the data they access frequently in the Alluxio filesystem. Alluxio also uses tiered storage, which has access to both the hard drive and SSD. This should allow queries to be processed significantly faster on the data used frequently by the applications.

The compute layer usually consists on an application that is used to access the data in the storage layer. Such applications are often SQL engines, such as Spark or Presto, or databases with their own QLs, such as Hive.

These applications are capable of accessing and manipulating data in the storage layer. This is mainly done in the form of SQL queries, sets of instructions written in the Standard Query Language, which are aimed at the files stored in the appropriate tabulated formats, such as JSON, CVS, Parquet, etc. Depending on the nature of those queries, as well as the applications themselves, performance can significantly vary from case to case. Since data processing is of utmost importance in the changing world of technology, solutions that increase the performance and throughput times of data manipulations are

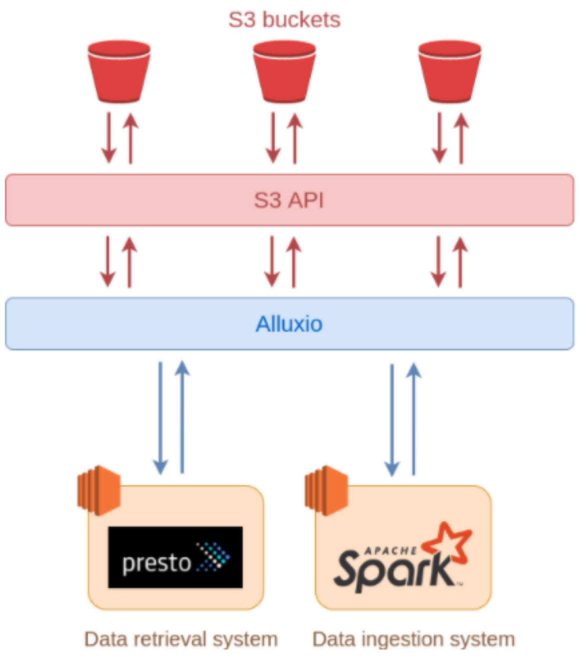


Figure 2.2. *Alluxio VDFS's role*

in high demand. Thus, Alluxio will likely play a vital role in the future development of cloud technology.

2.0.3 EMR

In our work we are going to use a pair of Amazon EMR clusters. Generally speaking, a cluster of computers is a set of machines working together as a unified system. They usually perform the same job and are used as servers for better performance. This also allows for distributions of certain responsibilities between them. For instance, clustering is actively used in cloud computing and data storage. Amazon Web Services (AWS) provide access to EMR cluster technology, which allows users to create their own clusters and automatically integrate a multitude of different services in them. For instance, Amazon EMR is available with Apache Hive, Apache Spark and Presto. Users can also install additional software onto their clusters. EMR automatically configures and tunes its applications, while also providing flexible options with Amazon Machine Images (AMIs), scaling databases and installing third-party software.

3. RELATED WORK

According to the research done by Rajaram and Haridass [4], Alluxio does decrease the execution time of queries on Apache Spark but the total time that is needed for mounting Alluxio with Spark negates some of the time saved by faster query execution. The research yielded interesting results, revealing that overall Spark alone operates more efficiently with smaller databases while Spark with a mounted Alluxio file system is much more efficient for larger databases of JSON and CSV formats since it's strength lies in reducing the computation times when dealing with large quantities of data. This work also presents a benchmark we are going to use for testing Alluxio over Spark.

C. Lawrie also conducted a research [7] that compares the performances of HDFS caching and Spark caching with Alluxio. This work presents several ways in which Alluxio is a better solution than HDFS caching: It has access to tier storage, which provides a tool for Alluxio to store data in other storage levels instead of memory; it is able to write files directly into memory; and it is generally faster than HDFS caching.

Alluixo is still being developed and optimized. There are certain areas in which Alluxio might not be the optimal solution for the data orchestration in the cloud. Moreover, there are multiple challenges when integrating new data orchestration solutions in hybrid or multi-cloud environments.

One of the concerns is data security and its impact on the performance of the applications. Alluxio is no exception and can suffer from memory cache attacks which may result in the failure of integrity and data security. This can lead to a possibility of severe performance decrease of normal users with a degradation of 20%-59% according to [8].

One of the works discusses the implications of various caching solutions, mainly Alluxio, on the throughput and cost performance of various cache architectures by comparing multiple possible cache to SSD ratios [9]. This shows that cache to SSD ratios have pretty significant impact on the performance and costs, for instance, the research indicates that 1:7 ratio provides the best highest performance of the system, while the 5:3 ratio allows for the optimal highest cost performance.

Another research was conducted on Online Analytical Processing (OLAP) with Alluxio by Chang X. [10]. The experiments were conducted on a Cloud platform due to the virtualization factor. Queries were tested on OLAP with Alluxio with a benchmark designed by the author. The SQL engines used in this experiment were Hive, Spark and Presto. The results showed that the read caching did not improve performance significantly for Hive due to the fact that query execution times there are largely dependent on data access times. With Spark it depends on the file type, text files are usually unaffected while querying with Parquet files does seem to be faster due to the nature of the reading process. Presto has shown similar results to Spark. The authors of the paper then proceed to optimize Alluxio by fixing a bug that was found during the experiment. That has increased the performance of querying times by about 50%, thus showing that Alluxio is a good solution for achieving better performance with OLAP and that Alluxio's block size should be increased for Text files and reduced for Parquet files.

A paper by Yang CT. [11] describes a research conducted with Ceph Storage and Hadoop with Alluxio that measures read/write execution times by using Hi-Benchmark testing tool. The results showed that such a system is a reliable solution and provides high read/write speed.

Ivanov and Pergolesi [12] measured and compared query execution times on Hive and Spark by using Hadoop and testing it with the BigBench benchmark while using a 1000GB set of columnar files. Spark showed the best performance with Parquet files while Hive has peaked with Optimized Row Columna (ORC) files.

4. RESEARCH METHOD AND INSTALLATION

In this work tests and measurements are divided into two main categories: local tests to ensure the correctness of the queries and generated database, tests on the cloud with measured performance. This section covers the research questions, how the research is going to be conducted alongside with descriptions of the tools used to measure the results.

4.0.1 Goal and research questions

The goal of this work is to analyze and compare the execution times of SQL queries using Alluxio on top of 3 different SQL Engines. For that purpose and based on the goal described above, we have derived the research questions to be the following:

- **Main research question**: What is the impact of Alluxio on the execution time SQL queries among the three selected SQL architectures?
    - **MRQ.1**: What is the difference of the execution time of SQL queries on Apache Hive on top of Alluxio?
    - **MRQ.2**: What is the difference of the execution time of SQL queries on Apache Spark on top of Alluxio?
    - **MRQ.3**: What is the difference of the execution time of SQL queries on Presto on top of Alluxio?

In our main research question, we aim at analyzing and comparing query execution times when querying from/to an S3 database using 3 SQL engines (Apache Spark, Apache HiveQL and Presto) which run with an extra data orchestration level (Alluxio), and the same set-up but without any caching solutions. As an example, we compare the query execution time on a Parquet file in S3 by using Presto running on Alluxio and Presto without any data orchestration. We chose these 3 SQL architectures for their differences and the different approach that is used in each one of them when handling SQL queries.

Additionally, we discuss briefly the suitability of our benchmark and why it was used in this particular study. We explore what other benchmarks are available and discuss their relevance in the context of this project.

The difference in querying times, if observed, is also discussed as well as impact of Alluxio on the querying times of each individual SQL architecture.

4.0.2 Structure

This project has been designed following the approximate structure of a single node as was defined and suggested by Rajaram and Haridass [4]. This includes the structure of the node as well as some ideas for the benchmark used in our project.

The structure of one node is rather simple. It includes a computing SQL engine or an application that is connected to a Data cache. The node accesses the cloud data storage and executes queries on it's data. This can be simulated by using a Kubernetes cluster with Docker containers inside each of the worker nodes. In our case, however, we are primarily interested in how the execution times of the queries are affected by our choice of the engine and the data orchestration tool. The more software we use and more complexity we add the project, the lower the credibility of this research will be in the context of the research questions. Too many factors might be affecting the execution times and unforeseen data bottlenecks are possible. Thus, the project will conducted in the following manner:

- Most of the initial set-up is implemented on a local host machine.
- Each SQL engine is tested separately with and without Alluxio.
- One AWS S3 data storage with several Parquet files of different sizes and 2 AWS EMR clusters with and without Alluxio for easier applications integration.
- Star Schema benchmark consisting of several simple queries is used for every engine.

Star Schema Benchmark [3] simulates classical warehousing applications. It consists of 5 tables: LINEORDER, SUPPLIER, PART, CUSTOMER, DWDATE, all of which are modified versions from the TPC-H benchmark. The Star Schema Bencmark's structure can be seen in figure 4.1.

SSB is a rather straightforward benchmarking tool, which offers flexibility and complexity necessary to perform interesting queries for this experiment. It provides several tables interconnected with foreign keys, sufficient data variance and a well implemented generating tool. In addition to that, it was used as the benchmark in the work that we aim to compare our Spark results to [4]. Our queries are simpler than the ones used in that work and we used each query once and thus increased performance should be expected.

There are other options for the benchmark available, such as the BigBench benchmark [13]. BB is a well-known benchmarking tool that is widely used in big data analysis. It would suit our purposes well but we decided to implement our data by using SSB

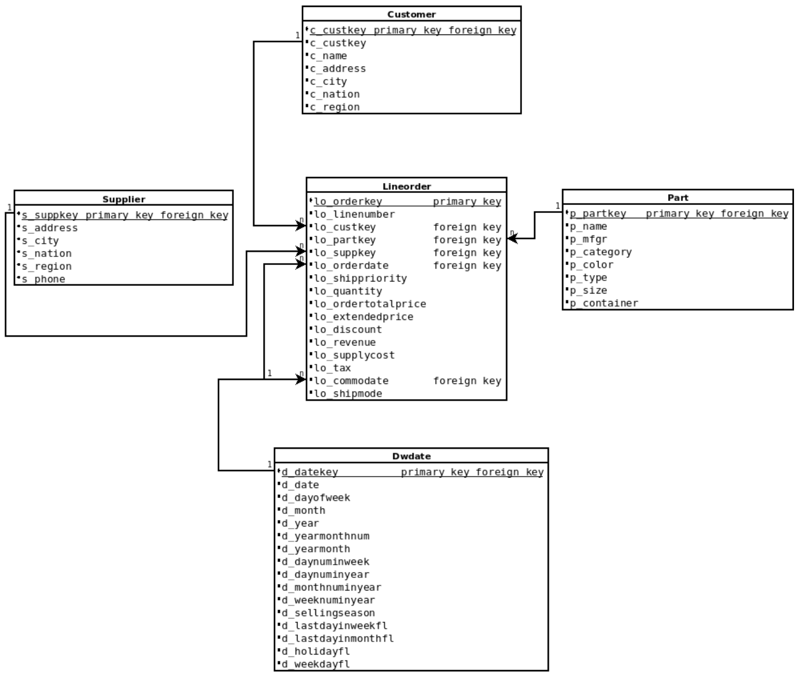


Figure 4.1. *The Star Schema Benchmark.*

because it gives our research more credibility and allows to compare the outcome of the experiment with some of the previous works done with SSB.

SSB itself is a derivative from the TPC-H benchmark which in turn was developed from the TPC-D benchmark [14]. TPC-H is a solid choice when it comes to benchmarking solutions but SSB provides simplicity and accessibility that are more important for the work of our scale. In addition to that, SSB shares all the main aspects of TPC-H, while also providing a reliable benchmarking tool for us to use in this work. The newer version of TPC-H is TPC-DS. It improves upon its predecessor but has the same drawbacks when it comes to analyzing our data and comparing it to the results of other authors.

After configuring the benchmark and generating the test files, the experiment takes place in two phases: local and cloud testing.

In local testing we aim to ensure that the database has been generated and is able to be queried from the SQL engines we installed locally. Work is done on a Linux machine with Ubuntu 20.04. It is enough to test the queries on one working engine, we chose to install Spark first and test the queries on it, since we already have approximate results for Spark and we know what to expect. We need to generate the test files with the benchmark and load them into the HDFS. After that we need to test our selected queries against a sample database in HDFS.

Cloud testing is done by using Amazon S3 storage. Alluxio service from AWS is acquired. It is automatically distributed to our AWS account. 2 EMR clusters are set-up in order to provide easier access to Apache Spark, Hive and Presto. One of the clusters is installed with the Alluxio service. The queries are then tested by SSH connection to the clusters. This is the main research of this project.

The structure of our project can be seen in figure 4.2.

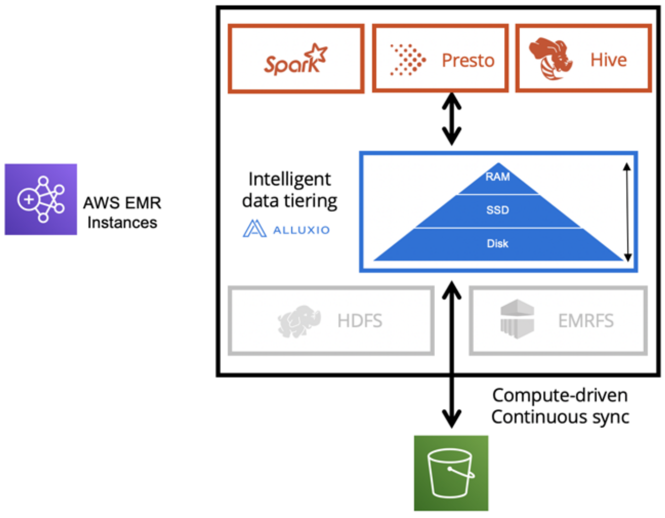


Figure 4.2. *The architecture of the experimental set-up.*

4.0.3 Installation

Our data is generated in the form of *Parquet* tables using the Star Schema Benchmark. The benchmark can be retrieved from the open source Github repositories. There are several modified versions of it but they all follow the basic structure used by [3], we simply need the makefile in order to create the *DBgen* tool, that creates the tables. The repository must be cloned first:

```
git clone https://github.com/electrum/ssb-dbgen.git
```

Next we go to the cloned repository and use the *make* command to create the tool.

After the tool has been successfully created, we can generate the data in the form of *tbl* tables. Every table must be created separately using the following commands:

```
./dbgen -s scale_factor -T c
./dbgen -s scale_factor -T l
./dbgen -s scale_factor -T p
./dbgen -s scale_factor -T s
./dbgen -s scale_factor -T d
```

The *scale factor* is a numerical value in the range [0.01, 1000]. It determines the size of the generated database. The last argument determines the table to be generated. All five generated tables will be in the *tbl* format and can be transformed to any format we choose.

4.0.4 Local installations

For local tests we installed Ubuntu 20.04 on our working machine. In order for our SQL engines to work, we need to install JDK, Scala and Git dependencies. It is also important to install the latest Java version. After that is done, the latest version of Spark is downloaded, in our case it was 3.1.1. The next step is to extract the archive into a selected Spark locatin. Alongside that install the compatible version of Hadoop, for instance Hadoop 2.7. Download the archive and extract it to the Hadoop folder. The final stage is to configure the Spark and Hadoop environments and edit the configuration files so we could access the Spark and HDFS APIs from our localhost addresses from several different ports. After that is done, HDFS files can be accessed from pyspark or scala terminals by specifying their local addresses.

After that Spark and HDFS should be all set to work.

4.0.5 Cloud installations and EMR

EMR without Alluxio

Our first goal is to set up an Amazon EMR cluster for easier use of Spark, Hive and Presto. In order to do that, we went to the Amazon EMR service and created a cluster from there. While setting up the cluster, EMR service automatically offers to install all 3 applications. 1 Master node and 1 Worker node m5.Xlarge are created. Each node has 4 cores, and each core has 16GB of memory available. This sums up to a total of 128GB. After the installation process, the cluster is almost ready to use. The last step is to enable SSH connection in order for the cluster to be accessible from local terminals, which is done from the applications user interfaces settings. In order to open an SSH tunnel to the Amazon EMR cluster, a program PuTTY is used. PuTTY is an open source terminal emulator that supports several network protocols, including SSH. Host name should be retrieved from cluster information and pasted in a local running instance of PuTTY along with the private key of EC2 file that was used to create the EMR cluster. After the configuration is complete, launch local terminal through PuTTY.

EMR with Alluxio

The next step is to set-up and EMR cluster with the Alluxuo service. The major difference with the previous section is to activate the Alluxio AWS service provided by Amazon before creating the cluster, which is done by navigating AWS website. The cluster set-up procedure is similar to the previous section. AWS IAM roles need to be created if they have not been already. Creating default roles should be enough. After that a bootstrap action must be performed during the installation procedure. Bootstrap actions in AWS allow to install additional software to the cluster by running additional scripts. When the script is specified, cluster is then installed. The necessary script can be found from the AWS website [15]. The running cluster now has Alluxio service in it. It is useful to verify that Alluxio is running on the cluster. For this Alluxio has a script called runTests:

```
$ sudo runuser -l alluxio -c "/opt/alluxio/bin/alluxio runTests"
```

Alluxio, if set up correctly, then will show you the result of the tests and tell if the tests passed. Now all SQL architectures are set up to use Alluxio when querying data.

4.0.6 Data generation and queries

After everything is set up, the data needs to be generated and queries should be tested locally. Local tests are only needed to verify the appropriateness of queries selected from the test queries provided by the Star Schema Benchmark[3]. First, the data is generated by using Star Schema Benchmark's generator tool. We used the scale factor of 10 to create a database of an appropriate size for our experiment. The result is a database of roughly 6.8 GB in size. Hadoop home directory is configured, queries are then written directly into Scala terminal and processed. Queries for working with SSB can be found from open-source repositores such as the one from open source database management system ClickHouse [16]. We selected 12 successful queries. In cloud tests performance was measured across all 3 engines by running 12 selected queries. The queries are selected so they could simulate real warehouse working environment while retaining relative simplicity and diversity. All 12 queries are run on each engine with Alluxio VDFS and without it and the time to execute each query is measured.

Selected queries can be found from Appendix A.

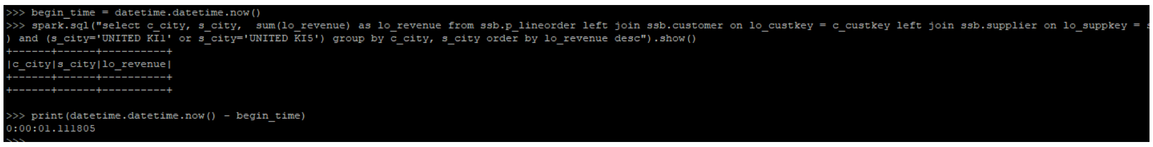
5. DATA ANALYSIS

The tests were conducted on the 12 selected queries in both EMR clusters. Each query was run 20 times and the average execution time for each individual query was measured. Since we are interested in the overall performance of each SQL engine and not the individual queries, the averaging method was judged sufficient. It is also expected that the execution times will not vary as much due to the nature of our installation and the secure connection to the AWS clusters. There will be no additional load and no simultaneous computation present in the cluster. Thus, possible deviations are not expected to impact the results and the necessary statistical tools, like standard deviation, will not be necessary. The queries are rerun to simply confirm their average execution time.

The resulting 12 numbers will also then be averaged into 1 number in order to get the general idea of the SQL engines' performances. They will be compared against each other with individual queries as well as the generalized results.

The queries are run on all 3 engines and the time is measured by using Python's DateTime module. The time is normalized there, hence we only have to compare the raw results obtained from our console.

An example of an SQL query and time measuring can be seen in figure 5.1.



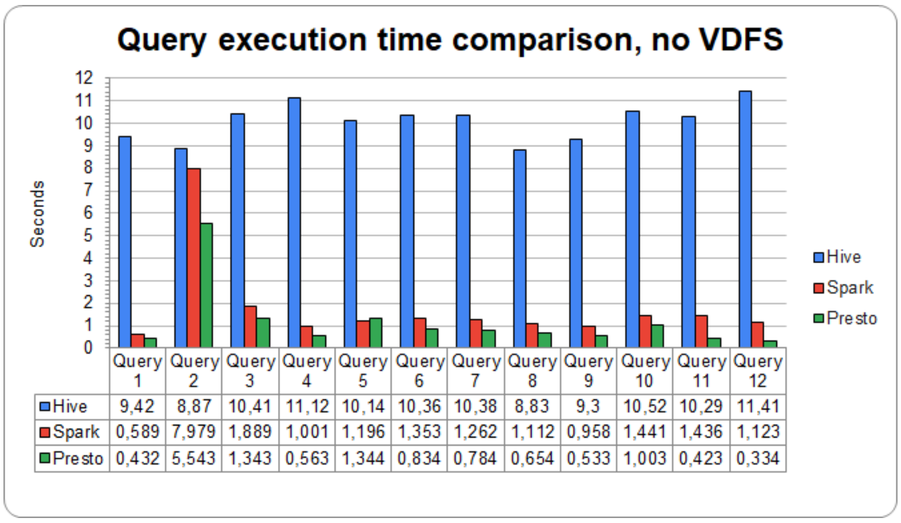
```
>>> begin_time = datetime.datetime.now()
>>> spark.sql("select c_city, s_city,  sum(lo_revenue) as lo_revenue from ssb.p_lineorder left join ssb.customer on lo_custkey = c_custkey left join ssb.supplier on lo_suppkey = s
) and (s_city='UNITED KI1' or s_city='UNITED KI5') group by c_city, s_city order by lo_revenue desc").show()
+------+------+----------+
|c_city|s_city|lo_revenue|
+------+------+----------+
+------+------+----------+

>>> print(datetime.datetime.now() - begin_time)
0:00:01.111805
>>>
```

Figure 5.1. *Query execution example.*

6. RESULTS

6.0.1 Tests with no VDFS

The results of testing all 3 SQL platforms without Alluxio VDFS are shown in figure 6.1

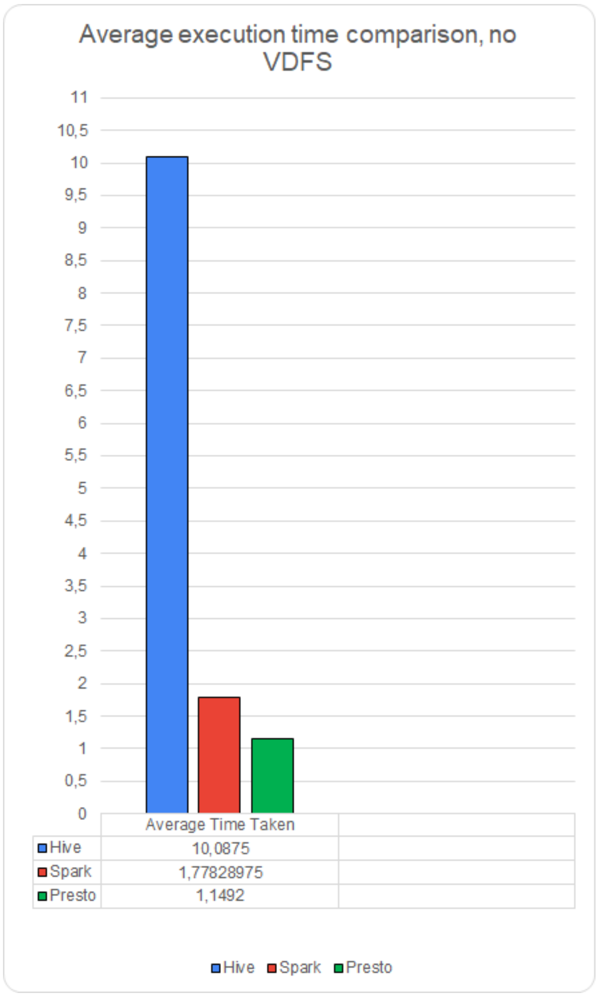


	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9	Query 10	Query 11	Query 12
🟦Hive	9,42	8,87	10,41	11,12	10,14	10,36	10,38	8,83	9,3	10,52	10,29	11,41
🟥Spark	0,589	7,979	1,889	1,001	1,196	1,353	1,262	1,112	0,958	1,441	1,436	1,123
🟩Presto	0,432	5,543	1,343	0,563	1,344	0,834	0,784	0,654	0,533	1,003	0,423	0,334

Figure 6.1. *Comparison with no Alluxio.*

As it can be observed from the results, Hive performed the worst out of all 3 engines, while Presto outperforms the other two in nearly all cases. This is expected, since Hive spends most of the time accessing the data. Presto has a very linear straight-forward architecture, while Spark is composed of a variety of levels, where Spark must do a ton of management before executing the job. Spark also spends some time managing resources and negotiating with the cluster manager. This allows Spark to work with more types of queries while Presto is a more strictly SQL engine.

Table in figure 6.2 shows the average execution time for each SQL platform with no Alluxio preinstalled.



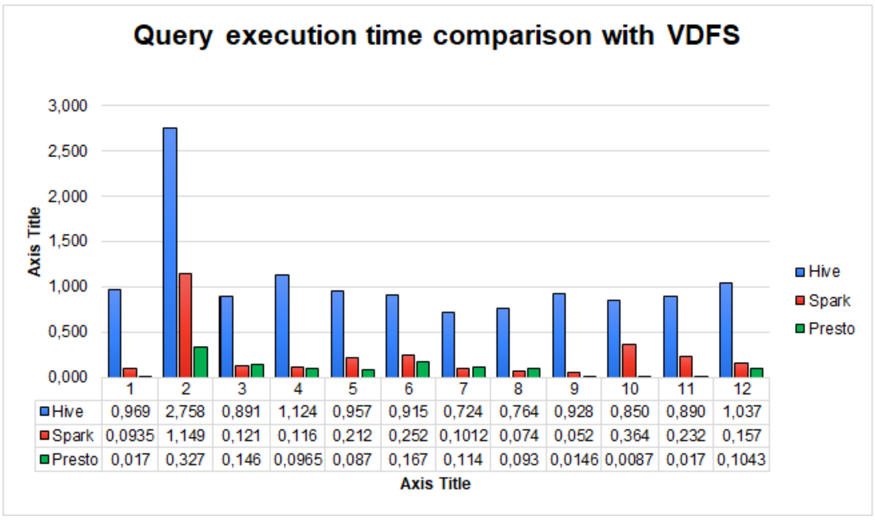
	Average Time Taken	
🟦Hive	10,0875	
🟥Spark	1,77828975	
🟩Presto	1,1492	

Figure 6.2. *Average execution time with no Alluxio.*

6.0.2 Tests with VDFS

It is important to note, that Alluxio needs to read and store the data before it can be accessed through cache. Thus, the very first query executions on the tables saved in our S3 storage will perform on roughly the same level as the queries run on top of just SQL architecture. The queries are thus run one time in order for Alluxio to be able to cache all the data from S3. We ran a couple of very basic queries such as SELECT * in order to load the files into Alluxio file system. After that all 12 previously selected queries are run from each of the API's of SQL engines. We averaged the result from each query in order to get the general picture of each engine's performance. Just a simple averaging operation is enough in our case since we aim at general understanding of each engine's performance when integrated with Alluxio and not the nature of each individual query.

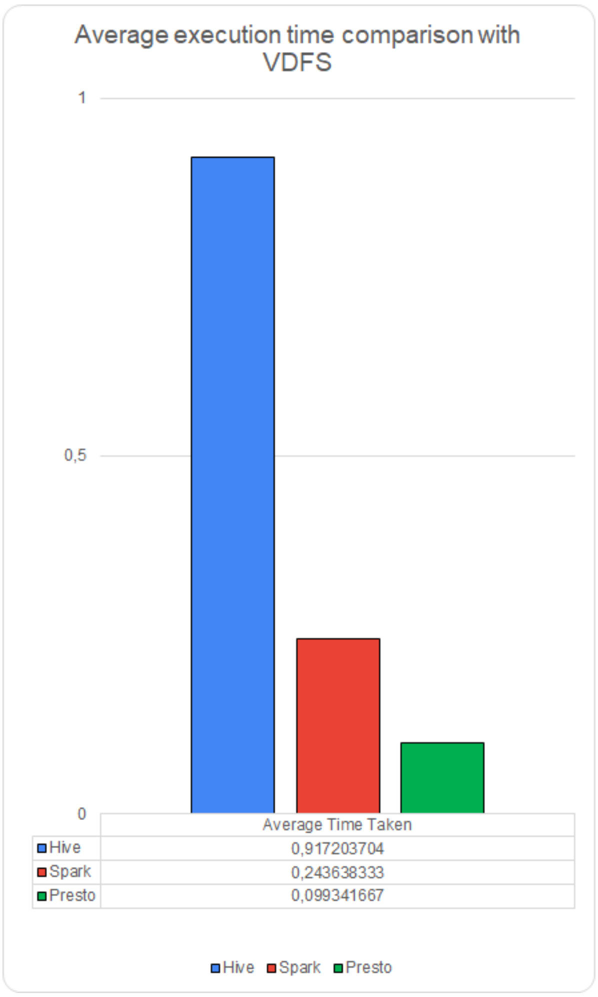
The results of testing all 3 SQL platforms with data being cached in Alluxio VDFS are shown in figure 6.3 and the average result is shown in figure 6.4.



	1	2	3	4	5	6	7	8	9	10	11	12
🟦Hive	0,969	2,758	0,891	1,124	0,957	0,915	0,724	0,764	0,928	0,850	0,890	1,037
🟥Spark	0,0935	1,149	0,121	0,116	0,212	0,252	0,1012	0,074	0,052	0,364	0,232	0,157
🟩Presto	0,017	0,327	0,146	0,0965	0,087	0,167	0,114	0,093	0,0146	0,0087	0,017	0,1043

Figure 6.3. *Comparison with Alluxio.*

The cloud tests with Alluxio VDFS verify the claim that Alluxio does improve the performance of SQL queries done on cached data. As with normal testing, Hive performed the worst while Presto remained in the lead. This can be explained by the fact that Alluxio worked closely with Presto and optimized its architecture in order to be compatible with Presto, and the fact that Presto has a rather straight-forward structure that is easy to work with. The results for Spark do show the same level of performance increase as in the work by Rajaram and Haridass [4] for Parquet files. The increase in the performance is big but gets smaller the smaller the file sizes are and eventually would become irrelevant as was stated in the mentioned work.



	Average Time Taken
🟦 Hive	0,917203704
🟥 Spark	0,243638333
🟩 Presto	0,099341667

Figure 6.4. *Average execution time with Alluxio.*

Here we can see the results for each individual SQL architecture. Thus, we can provide answers to our research questions.

Main research question 1: The difference in query executions times in our Hive installations can be seen in figure 6.5.

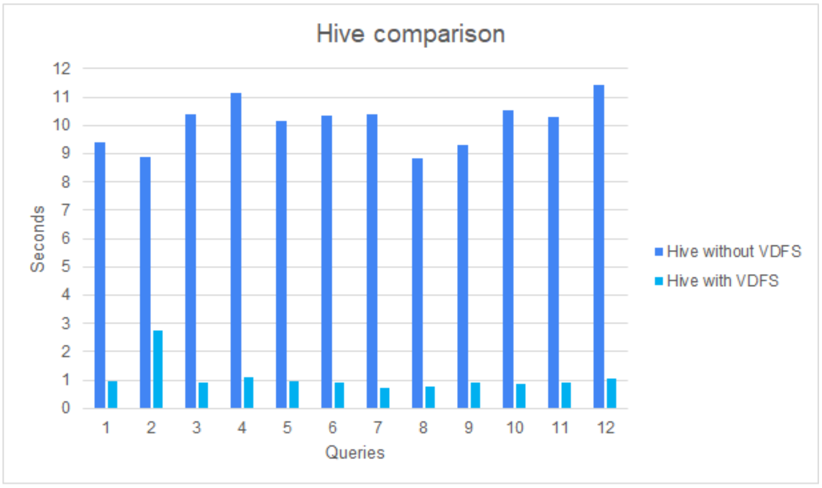


Figure 6.5. *Hive comparison.*

Main research question 2: The difference in query executions times in our Spark installations can be seen in figure 6.6.

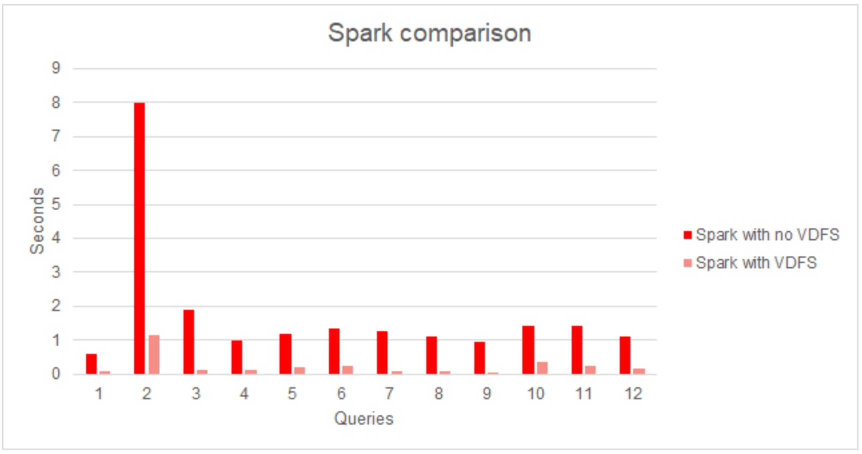


Figure 6.6. *Spark comparison.*

Main research question 3: The difference in query executions times in our Presto installations can be seen in figure 6.7.

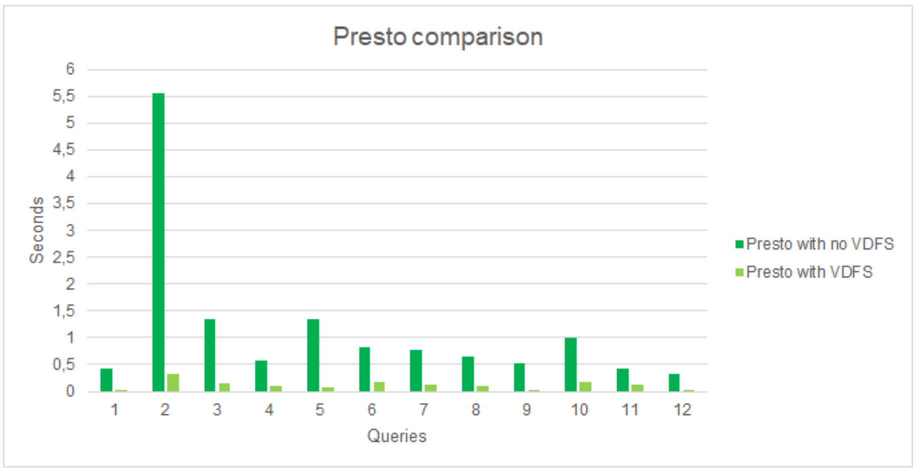


Figure 6.7. *Presto comparison.*

7. CONCLUSION

This paper has analyzed the impact of the Virtual Distributed File System Alluxio on the execution times of SQL queries done by using 3 different SQL architectures: Apache Hive, Apache Spark and Presto. 12 SQL queries were tested locally and then on the cloud with application of Alluxio integrated with the 3 SQL architectures. All necessary applications were installed by using AWS EMR clusters and the data was stored in the AWS S3 storage. Queries were done by using SSH connection to the clusters.

The results obtained in this work coincide with the expected values and measurements obtained by other researchers, while also providing some insight on how different architectures work with extra data orchestration levels applied on top of them. Alluxio architecture does indeed provide a reliable caching solution when it comes to enhancing the performance of basic queries on SQL engines. The tests were kept simple in order to make sure that the impact on the execution times is only dependent on the presence of Alluxio and not on the structure of the queries, memory capacities or other external factors.

There is a lot of room for future research in that area. Moving big data to the cloud is a global trend in the modern technology world and Alluxio can provide a reliable caching solution to the problem of cloud data storage. This has many implications for companies that mainly rely on older storage systems like Hadoop to store their data. With Alluxio, they can achieve much better performance and will be able to stay competitive in the ever evolving technology market.

REFERENCES

[1] Borthakur, D. The Hadoop Distributed File System: Architecture and Design. (2007). URL: http://svn.apache.org/repos/asf/hadoop/common/tags/release-0.16.1/docs/hdfs_design.pdf (visited on 02/27/2021).

[2] Zhijie Han, Y. Z. Spark: A Big Data Processing Platform Based on Memory Computing. *Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)* (2015). DOI: 10.1109/PAAP.2015.41.

[3] Pat O'Neil Betty O'Neil, X. C. Star Schema Benchmark. (2009). URL: https://www.cs.umb.edu/~poneil/StarSchemaB.PDF (visited on 02/15/2021).

[4] Rajaram, K. and Haridass, K. A Benchmark for Suitability of Alluxio over Spark. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)* 10-1 (2020). DOI: 10.35940/ijitee.A8190.1110120.

[5] Paul Dourish W. Keith Edwards, A. M. S. Presto: an experimental architecture for fluid interactive document spaces. *ACM Transactions on Computer-Human Interaction* (1999). DOI: 10.1145/319091.319099.

[6] Li, H. Alluxio: A Virtual Distributed File System. UCB/EECS-2018-29 (May 2018). URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-29.html (visited on 03/03/2021).

[7] Lawrie, C. Evaluation of the Suitability of Alluxio for Hadoop Processing Frameworks. *CERN Summer Student Program 2016* (2016).

[8] Yizhe Yang Qingni Shen, W. X. Memory Cache Attacks on Alluxio Impede High Performance Computing. *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)* (2018). DOI: 10.1109/BDCloud.2018.00069.

[9] Chang, X. and Zha, L. The Performance Analysis of Cache Architecture Based on Alluxio over Virtualized Infrastructure. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (2018). DOI: 10.1109/IPDPSW.2018.00088.

[10] Chang X. Liu Y., M. Z. and L., Z. Performance Analysis and Optimization of Alluxio with OLAP Workloads over Virtual Infrastructure. *Big Scientific Data Management. BigSDM 2018. Lecture Notes in Computer Science* 11473 (2019). DOI: 10.1007/978-3-030-28061-1_31.

[11] Yang CT. Chen CJ., C. T. Implementation of Ceph Storage with Big Data for Performance Comparison. *Kim K., Joukov N. (eds) Information Science and Applications 2017. ICISA 2017. Lecture Notes in Electrical Engineering* 424 (2017). DOI: 10.1007/978-981-10-4154-9_72.

[12] Ivanov, T. and Pergolesi, M. The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience* (2019), e5523. URL: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5523 (visited on 03/05/2021).

[13] Ahmad Ghazal Tilmann Rabl, M. H. BigBench: towards an industry standard benchmark for big data analytics. *SIGMOD '13: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), pp. 1197–1208. DOI: 10.1145/2463676.2463712.

[14] Peter Boncz Thomas Neumann, O. E. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. *Nambiar R., Poess M. (eds) Performance Characterization and Benchmarking. TPCTC 2013. Lecture Notes in Computer Science* 8391 (2014). DOI: 10.1007/978-3-319-04936-6_5.

[15] Borkar, D. Accelerate Amazon EMR Spark, Presto, and Hive with the Alluxio AMI. (2020). URL: https://aws.amazon.com/blogs/awsmarketplace/accelerate-amazon-emr-spark-presto-and-hive-with-the-alluxio-ami/ (visited on 03/17/2021).

[16] Star Schema Benchmark. ClickHouse documentation. (). URL: https://clickhouse.tech/docs/en/getting-started/example-datasets/star-schema/ (visited on 10/03/2021).

APPENDIX A: QUERIES

Query 1:

```sql
SELECT sum(LO_EXTENDEDPRICE * LO_DISCOUNT) AS revenue
FROM lineorder_flat
WHERE toYYYYMM(LO_ORDERDATE) = 199401
AND LO_DISCOUNT BETWEEN 4 AND 6
AND LO_QUANTITY BETWEEN 26 AND 35;
```

Query 2:

```sql
SELECT sum(LO_EXTENDEDPRICE * LO_DISCOUNT) AS revenue
FROM lineorder_flat
WHERE toISOWeek(LO_ORDERDATE) = 6
AND toYear(LO_ORDERDATE) = 1994
AND LO_DISCOUNT BETWEEN 5 AND 7
AND LO_QUANTITY BETWEEN 26 AND 35;
```

Query 3:

```
SELECT sum(LO_REVENUE), toYear(LO_ORDERDATE) AS year, P_BRAND
FROM lineorder_flat
WHERE P_CATEGORY = 'MFGR#12'
AND S_REGION = 'AMERICA'
GROUP BY year, P_BRAND
ORDER BY year, P_BRAND;
```

Query 4:

```
SELECT sum(LO_REVENUE), toYear(LO_ORDERDATE) AS year, P_BRAND
FROM lineorder_flat
WHERE P_BRAND >= 'MFGR#2221'
AND P_BRAND <= 'MFGR#2228' AND S_REGION = 'ASIA'
GROUP BY year, P_BRAND
ORDER BY year, P_BRAND;
```

Query 5:

```
SELECT sum(LO_REVENUE), toYear(LO_ORDERDATE) AS year, P_BRAND
FROM lineorder_flat
WHERE P_BRAND = 'MFGR#2239'
AND S_REGION = 'EUROPE'
GROUP BY year, P_BRAND
ORDER BY year, P_BRAND;
```

Query 6:

```
SELECT C_NATION, S_NATION, toYear(LO_ORDERDATE) AS year,
    sum(LO_REVENUE) AS revenue
FROM lineorder_flat
WHERE C_REGION = 'ASIA'
AND S_REGION = 'ASIA'
AND year >= 1992
AND year <= 1997
GROUP BY C_NATION, S_NATION, year
ORDER BY year ASC, revenue DESC;
```

Query 7:

```sql
SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year,
    sum(LO_REVENUE) AS revenue
FROM lineorder_flat
WHERE C_NATION = ’UNITED␣STATES’
AND S_NATION = ’UNITED␣STATES’
AND year >= 1992
AND year <= 1997
GROUP BY C_CITY, S_CITY, year
ORDER BY year ASC, revenue DESC;
```

Query 8:

```sql
SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year,
    sum(LO_REVENUE) AS revenue
FROM lineorder_flat
WHERE (C_CITY = ’UNITED␣KI1’ OR C_CITY = ’UNITED␣KI5’)
AND (S_CITY = ’UNITED␣KI1’ OR S_CITY = ’UNITED␣KI5’)
AND year >= 1992
AND year <= 1997
GROUP BY C_CITY, S_CITY, year
ORDER BY year ASC, revenue DESC;
```

Query 9:

```sql
SELECT C_CITY, S_CITY, toYear(LO_ORDERDATE) AS year,
    sum(LO_REVENUE) AS revenue
FROM lineorder_flat
WHERE (C_CITY = ’UNITED␣KI1’ OR C_CITY = ’UNITED␣KI5’)
AND (S_CITY = ’UNITED␣KI1’ OR S_CITY = ’UNITED␣KI5’)
AND toYYYYMM(LO_ORDERDATE) = 199712
GROUP BY C_CITY, S_CITY, year
ORDER BY year ASC, revenue DESC;
```

Query 10:

```
SELECT toYear(LO_ORDERDATE) AS year, C_NATION,
    sum(LO_REVENUE - LO_SUPPLYCOST) AS profit
FROM lineorder_flat
WHERE C_REGION = 'AMERICA'
AND S_REGION = 'AMERICA'
AND (P_MFGR = 'MFGR#1' OR P_MFGR = 'MFGR#2')
GROUP BY year, C_NATION
ORDER BY year ASC, C_NATION ASC;
```

Query 11:

```
SELECT toYear(LO_ORDERDATE) AS year, S_NATION, P_CATEGORY,
    sum(LO_REVENUE - LO_SUPPLYCOST) AS profit
FROM lineorder_flat
WHERE C_REGION = 'AMERICA'
AND S_REGION = 'AMERICA'
AND (year = 1997 OR year = 1998)
AND (P_MFGR = 'MFGR#1' OR P_MFGR = 'MFGR#2')
GROUP BY year, S_NATION, P_CATEGORY
ORDER BY year ASC, S_NATION ASC, P_CATEGORY ASC;
```

Query 12:

```
SELECT toYear(LO_ORDERDATE) AS year, S_CITY, P_BRAND,
    sum(LO_REVENUE - LO_SUPPLYCOST) AS profit
FROM lineorder_flat
WHERE S_NATION = 'UNITED␣STATES'
AND (year = 1997 OR year = 1998)
AND P_CATEGORY = 'MFGR#14'
GROUP BY year, S_CITY, P_BRAND
ORDER BY year ASC, S_CITY ASC, P_BRAND ASC;
```