UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RAFAEL FERREIRA LONGO VARGAS

A Performance Comparison of Data Lake Table Formats in Cloud Object Storages

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Claudio Fernando Resin Geyer
Coadvisor: Dr. Julio Cesar Santos dos Anjos

Porto Alegre
September 2022

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões Mendes
Vice-Reitora: Prof[a]. Patricia Helena Lucas Pranke
Pró-Reitor de Graduação: Prof[a]. Cíntia Inês Boll
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Marcelo Walter
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

ABSTRACT

The increasing informatization of processes involved in our daily lives has generated a significant increase on the number of software developed to meet these needs. Considering this, the volume of data generated by applications is increasing, which is generating a bigger interest in the usage of it for analytical purposes, with the objective of getting insights and extracting valuable information from it. This increase, however, has generated new challenges related to the storage, organization and processing of the data. In general, the interest is in obtaining relevant information quickly, consistently and at the lowest possible cost. In this context, new approaches have emerged to facilitate the organization and access to data at a massive scale. An already widespread concept is to have a central repository, known as *Data Lake*, in which data from different sources, having variable characteristics, are massively stored, so that they can be explored and processed in order to obtain new relevant information. These environments have been implemented in object storages lately, especially in the Cloud, given the rise of this model in recent years. Frequently, the data stored in these environments is often structured as tables, and stored in files, which can be text files, such as CSVs, or binary files, such as Parquet, Avro or ORC, that implement specific properties, like data compression, for example. The modeling of these tables resembles, in some aspects, the structures of Data Warehouses, which are frequently implemented using Database Management Systems (DBMSs), and are used to make data available in a structured way, for analytical purposes.

Given these characteristics, new specifications of table formats have emerged, applied as layers above these files, which aim to implement the support to usual operations and properties of DBMSs, using object storages as a storage layer. In practice, it is intended to guarantee the ACID properties during these operations, as well as the ability to perform operations that involve mutability, like updates, upserts or merges, in a simpler way. Thus, this work aims to evaluate and compare the performance of some of these formats, defined as Data Lake Table Formats: Delta Lake, Apache Hudi and Apache Iceberg, in order to identify how each format behaves when performing usual operations in these environments, like: inserting data, updating data and querying data.

Keywords: Data Lake. Big Data File Formats. Data Lake Table Formats.

RESUMO

A crescente informatização dos processos envolvidos em nosso cotidiano gerou um aumento significativo no número de softwares desenvolvidos para atender a essas necessidades. Diante disso, o volume de dados gerados tem crescido, o que está gerando um maior interesse na utilização dos mesmos para fins analíticos, com o objetivo de obter insights e extrair informações valiosas deles. Esse aumento, no entanto, gerou novos desafios relacionados ao armazenamento, organização e processamento dos dados. Em geral, o interesse está em obter informações relevantes de forma rápida, consistente e com o menor custo possível. Nesse contexto, surgiram novas abordagens para facilitar a organização e o acesso a esses dados em grande escala. Um conceito já difundido envolve ter um repositório central, conhecido como *Data Lake*, no qual dados de diferentes fontes, com características variáveis, são armazenados massivamente, para que possam ser explorados e processados de forma a obter novas informações relevantes a partir deles. Esses ambientes vêm sendo implementados em *Object Storages* ultimamente, em especial, na Nuvem, dada a ascensão desse modelo nos últimos anos. Frequentemente, os dados armazenados nesses ambientes costumam ser estruturados como tabelas e armazenados em arquivos, que podem ser arquivos de texto, como CSVs, ou arquivos binários, como Apache Parquet, Apache Avro ou Apache ORC, que implementam alguma propriedade específica, como compressão de dados, por exemplo. A modelagem dessas tabelas se assemelha, em alguns aspectos, às estruturas de *Data Warehouses*, que são frequentemente implementados usando Sistemas de Gerenciamento de Banco de Dados (SGBDs), sendo utiizados para disponibilizar dados de forma estruturada, para fins analíticos.

Diante dessas características, novas especificações de formatos de tabelas, têm surgido, visando aplicar camadas acima desses arquivos, a fim de implementar o suporte às operações e propriedades comuns em SGBDs, utilizando *Object Storages* como camada de armazenamento. Na prática, pretende-se garantir as propriedades ACID durante essas operações, bem como a capacidade de realizar operações que envolvam mutabilidade, como atualizações, *upserts* ou *merges*, de forma mais simples. Assim, este trabalho tem como objetivo avaliar e comparar o desempenho de alguns desses formatos, definidos como "formatos de tabela de Data Lakes": Delta Lake, Apache Hudi e Apache Iceberg, a fim de identificar como cada formato se comporta ao realizar operações usuais nesses ambientes, como: inserção de dados, atualização de dados e consulta aos dados.

Palavras-chave: Data Lake. Big Data File Formats. Data Lake Table Formats.

LIST OF FIGURES

Figure 2.1 Spark SQL Query Planning 15
Figure 2.2 Trino Architecture Overview 16
Figure 2.3 Storage Formats Illustration 17
Figure 2.4 Partition Pruning and Predicate Pushdown Illustration 19
Figure 2.5 Avro Structure Illustration 19
Figure 2.6 An overview of the disposition of the Delta table files 22
Figure 2.7 Hudi's Copy On Write Approach Illustration 23
Figure 2.8 Hudi's Merge On Read Approach Illustration 24
Figure 2.9 Iceberg Format Overview 26

Figure 4.1 Experiments' Architecture Overview 33

Figure 5.1 Total Load Phase Duration 42
Figure 5.2 Speedup - Load Phase 42
Figure 5.3 Average Load Duration for each table (using 4 worker nodes) 43
Figure 5.4 Total Query Phase Duration 44
Figure 5.5 Speedup - Query Phase 45
Figure 5.6 Average Query Duration (using 4 worker nodes) 46
Figure 5.7 Total Load Phase Duration by Scale 47
Figure 5.8 Total Update Phase Duration by Scale 48
Figure 5.9 Average Speedup - Update Phase 49
Figure 5.10 Total Query Phase Duration by Scale (using 4 worker nodes) 50
Figure 5.11 Average Execution Duration by Query (using 4 worker nodes) 50

Figure A.1 Average Load Duration for each table (using 1 worker node) 56
Figure A.2 Average Load Duration for each table (using 2 worker nodes) 57
Figure A.3 Average Load Duration for each table (using 3 worker nodes) 58
Figure A.4 Average Query Duration (using 1 worker node) 59
Figure A.5 Average Query Duration (using 2 worker nodes) 60
Figure A.6 Average Query Duration (using 3 worker nodes) 61

Figure B.1 Average Execution Duration by Query (using 1 worker node) 62
Figure B.2 Average Execution Duration by Query (using 2 worker nodes) 62
Figure B.3 Average Execution Duration by Query (using 3 worker nodes) 63

LIST OF TABLES

Table 2.1 Supported Query Types by Table Type 25

Table 4.1 Experiments' Virtual Machines Specification 32
Table 4.2 TPC-DS Tables 35
Table 4.3 Load, Update and Read Performance Experiment Tables 38
Table 4.4 Experiments' Metrics and Variations Overview 40

LIST OF ABBREVIATIONS AND ACRONYMS

SQL	Structured Query Language
HDFS	Hadoop Distributed File System
TPC	Transactional Performance Council
OLAP	Online Analytical Processing
GUID	Global Unique Identifier
DBMS	Database Management System
RDBMS	Relational Database Management System
API	Application Programming Interface
ACID	Atomicity, Consistency, Isolation and Durability

CONTENTS

1 INTRODUCTION **10**
1.1 Context **10**
1.2 Objectives **10**
1.3 Motivation **11**
1.4 Organization **11**
2 BACKGROUND **12**
2.1 Cloud Computing **12**
2.2 Object Storage **12**
2.3 Data Warehouse **13**
2.4 Data Lake **13**
2.5 Distributed Query Engines **14**
2.5.1 Spark SQL 14
2.5.2 Presto/Trino 16
2.6 Big Data File Formats **17**
2.6.1 Apache Parquet 18
2.6.2 Apache ORC 18
2.6.3 Apache Avro 19
2.7 Data Lake Table Formats **20**
2.7.1 Delta Lake 20
2.7.1.1 Data Objects 21
2.7.1.2 Logs 21
2.7.1.3 Log Checkpoints 22
2.7.2 Apache Hudi 22
2.7.2.1 Copy On Write Tables 23
2.7.2.2 Merge On Read Tables 24
2.7.2.3 Query Types 25
2.7.3 Apache Iceberg 25
2.7.3.1 Metadata Layer 26
2.7.3.2 Data Layer 27
3 RELATED WORK **28**
3.1 On Data Lake Architectures and Metadata Management **28**
3.2 The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet **28**
3.3 A study of SQL-on-Hadoop systems **29**
3.4 Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics **29**
4 METHODOLOGY **31**
4.1 Overview **31**
4.2 Architecture **31**
4.3 Experiments **33**
4.3.1 TPC-DS Inspired Experiment 33
4.3.1.1 Dataset Overview 34
4.3.1.2 Experiment Phases 35
4.3.1.3 Implementation 36
4.3.2 Load, Update and Read Performance Experiment 37
4.3.2.1 Dataset Overview 37
4.3.2.2 Experiment Phases 38
4.3.2.3 Implementation 39

4.4 Metrics and Variations **40**
5 RESULTS **41**
5.1 TPC-DS Inspired Experiment **41**
5.1.1 Load Phase 41
5.1.2 Query Phase 44
5.2 Load, Update and Read Performance Experiment **47**
5.2.1 Load Phase 47
5.2.2 Update Phase 48
5.2.2.1 Query Phase 49
5.3 Overall Analysis **50**
6 CONCLUSION **52**
REFERENCES **54**
APPENDIX A — RESULT PLOTS - TPC-DS INSPIRED **56**
APPENDIX B — RESULT PLOTS - LOAD, UPDATE AND READ **62**

1 INTRODUCTION

1.1 Context

The challenges arising from the generation of huge amounts of data by computer applications, and the need for processing it fast for generating insights, are getting bigger and more frequent. Nowadays, the information obtained by processing these datasets are important for a lot of fields, with commercial or academic purposes. Even the computer applications benefit from this trend, which can be seen as a cycle, as the data, that was generated by them, is processed, and then, reused by the same applications as a more valuable information, enabling the implementation of innovative features.

For achieving these objectives, though, the systems need to be scalable enough to be able to store and process the data, as well as maintain it in a consistent and organized way. Since the rise of Cloud Computing, many approaches have been implemented to fit these requirements, many of them involving distributed systems. It is the case of the Cloud Object Storages, that seek to maintain data in a distributed manner, for a low cost. These solutions are being commonly used for the implementation of centralized data repositories, known as Data Lakes. These repositories maintain data of many natures, having varied characteristics. Usually, the datasets that are stored in these environments can be classified as: structured (e.g.: CSV, Parquet), semi-structured (e.g.: JSON, XML) or unstructured (e.g.: image, video, audio).

Given these tendencies, and the ability to store data in a structured way in environments such as Cloud Object Storages, that tend to store large amounts of data for a low cost, new approaches have emerged, with the objective of facilitating the organization of datasets in these environments, keeping the data in a tabular format, and supporting some properties that are usually found in the implementation of DBMSs (such as the support to ACID properties and operations that involve mutability). The formats, organized as tables, have been called Data Lake Table Formats.

1.2 Objectives

This project has the objective of measuring and comparing the performance of the following Data Lake Table Formats: Delta Lake, Apache Hudi and Apache Iceberg, commonly used for storing and managing structured data in Data Lakes. The idea is to

measure the performance of these formats when submitted to usual operations, like inserting, updating or reading data. The main objective is to try to replicate some analytical workload scenarios, as experiments, to measure the performance of each format, as well as comparing them, identifying which format outperforms the others when executing each set of experiments.

1.3 Motivation

The motivations of this work are centered around trying to obtain better insights on the performance of different tables formats that support ACID properties and mutability operations, using Cloud Object Storages as a storage layer. This could be useful, given the lack of information about the performance comparison of these table formats when submitted to multiple sets of operations. The work is also relevant to identify the pros and cons of each format, which can be useful for implementing new formats or identifying points that can be improved on the existing ones.

1.4 Organization

This project is organized as follows: Chapter 2 gives a background, introducing concepts that are relevant to the understanding of the work. Chapter 3 summarizes some works related to this thesis. Chapter 4 describes the methodology used for measuring the performance and comparing the table formats. Chapter 5 shows the results obtained from the experiments, and the comparisons among the formats. At the end, Chapter 6 presents some conclusions, as well as possible improvements and future work.

2 BACKGROUND

This chapter introduces important concepts to the understanding of the work, as well as more information about each table format used in the experiments. The concepts are presented in the sub chapters of this section, organized in a way that more general concepts are introduced first, before the more specific ones.

2.1 Cloud Computing

The definition of Cloud Computing may vary, depending on each author is being is being used as reference. The definition adopted to the understanding of this work is the one from NIST (MELL; GRANCE et al., 2011), that is: “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction”.

This concept is particularly important for the next concepts introduced in this chapter, since many solutions associated with them are offered as Cloud Computing vendors, given the necessity to adapt and scale rapidly, since the workloads usually involve large amounts of data.

2.2 Object Storage

An object storage, as defined in Object Storage: The Future Building Block for Storage Systems (FACTOR et al., 2005), is basically an abstraction that defines data as objects, that can be manipulated by a common interface, to perform operations such as adding, removing or reading an object (or, simply put, a file). This abstraction is different from other storage architectures, such as block storages, in which the operations are performed against arrays of logical blocks. The object storage can be seen as an architecture that implements a higher level of abstraction for manipulating data, resembling key-value stores, in which the key is an unique identifier of an object, and the object is a file, that stores the data itself. Typically, this architecture is implemented in a distributed manner, ensuring replication and easy scalability.

Nowadays, many Cloud vendors offer this kind of solution, as a managed service, that implements a common interface for performing operations to manipulate data, as objects, in their infrastructure. Some of the examples, from different cloud vendors, are: Amazon S3, Google Cloud Storage and Azure Blob Storage.

2.3 Data Warehouse

The term "Data Warehouse" was coined in the late eighties, by Devlin and Murphy (1988). The concept was introduced, at the time, as a single logical repository to store data, coming from applications, that could be used for analytical purposes, based on business needs. The concept was then popularized when the book "Building the Data Warehouse", authored by Inmon (1990), was published, giving a more practical guide on the implementation of these systems, defining the concept as follows: “A data warehouse is a subject-oriented, integrated, nonvolatile, and time-variant collection of data in support of management’s decisions”. Another possible definition was given later by Kimball (1996), as follows: “A data warehouse is a copy of transaction data specifically structured for query and analysis.”. Both defining Data Warehouses as collections (or repositories) that store data in a suitable way for improving the usage of data for decision making processes.

More practically, nowadays, the Data Warehouses can be seen as repositories that store data in a structured way (typically, using tabular formats), facilitating the usage of data, coming from various systems, for analytical purposes. The implementations of Data Warehouses vary, storing data in relational databases, for example, or even as structured files in object storages, using abstraction layers and mechanisms to query the data using SQL. This is an important information, given that the term is often been used to refer to structured zones of Data Lakes (concept that is defined in the section 2.4), that can use Data Lake Table Formats for defining structures that store processed data that can be used in decision support processes.

2.4 Data Lake

The concept of Data Lake can be a little fuzzy for researchers or practitioners, given the evolution the term gained since it was introduced, by Dixon (2010). Nowadays,

Data Lakes are often defined as central repositories that store massive amounts of data, that has varied characteristics and come from multiples sources, to be used future analysis or operational purposes (SAWADOGO; DARMONT, 2021). Usually, a Data Lake has a logical and physical organization, where data is usually separated by zones, associated with the refinement level of data: the data directly coming from a source is stored in a "raw" zone, and, after processing steps, that are executed to ensure data quality and cleansing, storing the data in intermediate zones, the processed data is, eventually, stored in "consumption" zones. There are, though, other logical organizations that can be used, depending on the author or practitioner (SAWADOGO; DARMONT, 2021).

Nowadays, Data Lakes are usually implemented using Cloud Object Storages, although implementations using the Hadoop Distributed File System (HDFS) are still being used, in conjunction with other tools from the Apache Hadoop ecosystem.

2.5 Distributed Query Engines

When analyzing data stored in Data Lakes, it is important to have a layer that makes querying data easily possible, specially, when the data can be queried in a declarative way, using SQL (Structured Query Language), for example. For this purpose, distributed query engines are built. Distributed query engines can be seen, nowadays, basically, as data virtualization layers, that provide access to data that comes from multiple sources, in a distributed fashion. They can be seen as what was referred to as SQL-on-Hadoop systems, that aimed to process SQL queries using Hadoop data processing frameworks (e.g.: Tez and MapReduce) (FLORATOU; MINHAS; ÖZCAN, 2014).

Since the emergence of this concept, many solutions have been implemented. Apache Hive and Apache Impala, for example, were the first to gain popularity. In this chapter, however, only the distributed query engines that support the Data Lake Table Formats that were evaluated in this work are presented.

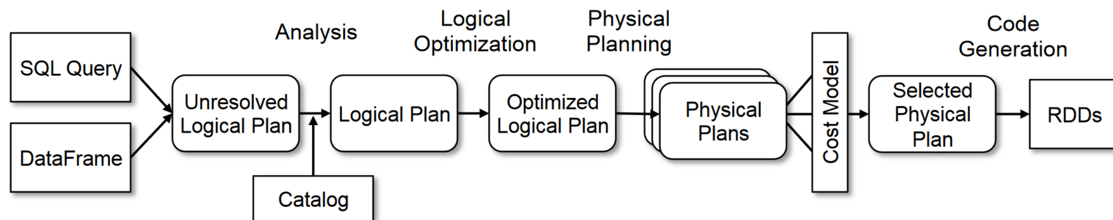
2.5.1 Spark SQL

Spark SQL is a Spark (a distributed analytics engine from the Hadoop ecosystem) module for structured data processing that can act as a distributed query engine (DATABRICKS, 2022). This is the query engine that was chosen for running the ex-

periments of this work, that evaluated the performance of the data lake table formats. It uses a query optimizer called Catalyst, that supports multiple data sources and analytics workloads, using some established DBMS (Database Management System) techniques (ARMBRUST et al., 2015). The optimizer is extensible, thus, supporting the addition of custom optimization techniques and features. This is an important information, given the fact that each of the Data Lake Table Formats analysed in this work provide a specific extension of the optimizer, what is directly related to the possible impact on the performance obtained when when executing the experiments for each format.

The figure 2.1 defines the phases of a query planning in Spark SQL. Then, brief descriptions of each phase of the execution of a SQL query in Spark SQL are presented.

Figure 2.1: Spark SQL Query Planning



Source: Armbrust et al. (2015)

1. **Analysis Phase**: all the query planning starts at the analysis phase. Initially, an "unresolved logical plan" tree is built, then, after the construction of the tree, the unresolved attributes (such as columns with unknown types or unknown table references) are resolved. After the execution of the phase, a logical plan of the query execution is obtained.
2. **Logical Optimization Phase**: after obtaining the logical plan, some performance optimizations are done. It involves the application of techniques like partition pruning and predicate pushdown (illustrated in the figure 2.4), generating new plans, and computing their cost, trying to optimize the logical plan.
3. **Physical Planning Phase**: the phase when multiple physical plans are generated, by assigning physical operations, based on information about the engines' operators/nodes. The physical operation defines, for example, how a join operation between two tables is going to be executed. Multiple plans are generated, and only one is selected by using a cost model.
4. **Code Generation Phase**: the final phase, when Java byte-code is generated to run

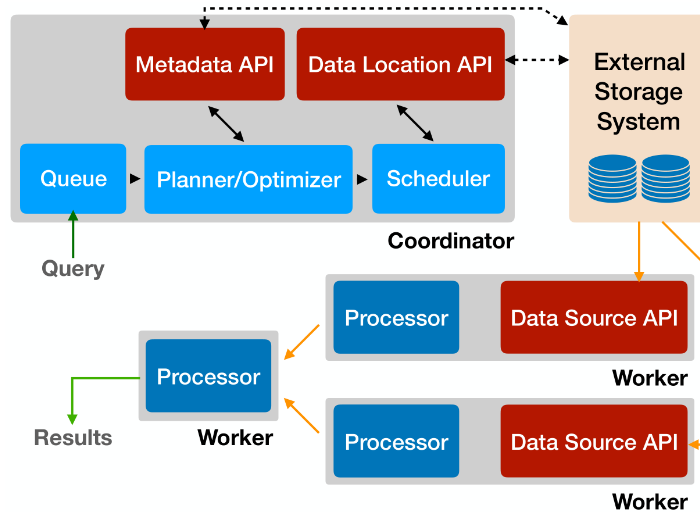
on each node of the cluster.

2.5.2 Presto/Trino

Presto is an open source distributed query engine, created at Facebook, that supports ANSI SQL for querying and processing data from multiple sources. Over the years, the participants of the project created a fork, known as Trino (formerly PrestoSQL). Both follow the same core design principles. Although these engines were not used for executing the experiments of this work, they are briefly presented, since they are supported by Data Lake Table Formats compared in this project.

The Presto/Trino engine's cluster consists of one single coordinator node and worker nodes. The coordinator node is responsible for admitting, parsing, planning and optimizing queries, as well as orchestrating them, while the worker nodes are responsible for their processing (SETHI et al., 2019). An overview of the architecture is shown in the figure 2.2.

Figure 2.2: Trino Architecture Overview



Source: Sethi et al. (2019)

Briefly summarizing the execution of a query, the coordinator analyzes the data's metadata and creates an execution plan, which is distributed to the cluster's workers. When dealing with the connection to external storage systems and formats (like the ones compared in this work), Presto/Trino has plugins that provide connectors, implementing an API that is composed of four parts: Metadata API, Data Source API, Data Location

API and Data Sink API, used to communicate with these sources.

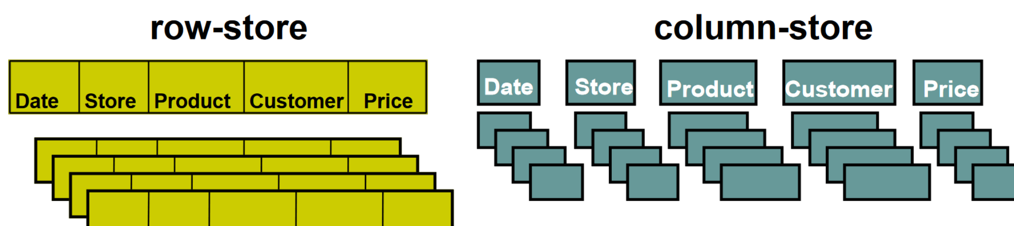
2.6 Big Data File Formats

In Data Lakes, the format used to store the data in it has, usually, a great impact on the performance of the analytical workloads that process and extract new information from the data. Given that, formats optimized for this type of workloads, such as binary formats that implement compression algorithms, are still arising, even with the facilities that Cloud Computing solutions have provided, such as easy scalability and cheap storage.

When dealing with structured data, there are, mainly, two ways to store it: in text files, like CSV, JSON and XML, or in binary files. The text files have benefits for the end user, given that they are easy to read and display, but they impose some constraints related to performance when processing it, because, generally, more data needs to be read from the storage medium, thus more I/O time is consumed. The binary formats, on the other hand, can be implemented using techniques like compression that reduce the physical amount of data that is stored in the storage mediums. The I/O time, thus, tends to be lower, and even, the processing time, given that, sometimes, data does not even need to be uncompressed to compute new information.

Considering the model of these file formats, optimal for analytical purposes, there are usually two approaches used for storing data: row-oriented and column-oriented. The row-oriented formats store each row contiguously, whereas column oriented formats store each column values contiguously. Each one can have its own advantages, that'll be briefly presented in the next subsections. The figure 2.3 tries to illustrate these approaches:

Figure 2.3: Storage Formats Illustration



Source: Abadi, Boncz and Harizopoulos (2009)

On each of the next subsections, file formats that are used and supported by the implementations of the Data Lake Table Formats compared in this work are presented.

2.6.1 Apache Parquet

Apache Parquet is a binary and column-oriented file format, used to store structured data. It supports some compression codecs, like Snappy, GZIP, BZIP2 and deflate, which is essential to achieve its purpose: store more information using less storage and, consequently, reducing the amount of I/O and CPU resources used to deserialize data (VOHRA, 2016). The file has also a portion reserved for storing some metadata about its content, like the name of the columns, the column types, the compression used, and, even some statistics about the data.

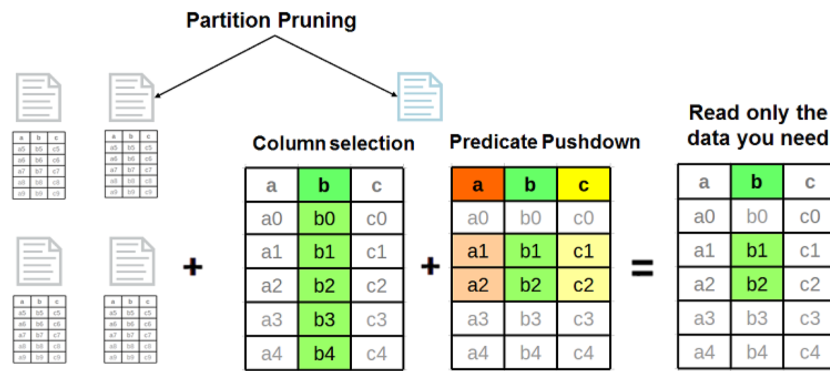
One important information about the Parquet files is that they are immutable. If records need to be updated or written, it implies the cost of a complete rewrite of the file. A common approach, thus, is to create new files when new records need to be stored (which can be viewed as creating a new partition). When records need to be updated, new files can be created to write the delta against the base file. These types of approaches are going to be discussed more explicitly in the sections that define the Data Lake Table Formats that support mutability operations.

2.6.2 Apache ORC

Apache ORC is a binary and column-oriented file format. It supports the usage of a technique called predicate push-down (supported by Apache Parquet as well), that consists in using a predicate to read only the portion data that satisfies it (APACHE, 2022c). In the case of ORC, this behavior can be resumed as the possibility of reading only a portion of a specific column, given that, for being column-oriented, reading only the data of a column is supported by default. The idea, thus, is that a full scan in the column's data is not always necessary. The behavior can be visualized in the image below:

This behavior is possible because the file is structured to have segments called stripes. The idea is that the stripes store a subset of rows, where the data of a same column of these rows are stored adjacently. The file also supports compression codecs like *zlib* and *Snappy* (APACHE, 2022c).

Figure 2.4: Partition Pruning and Predicate Pushdown Illustration



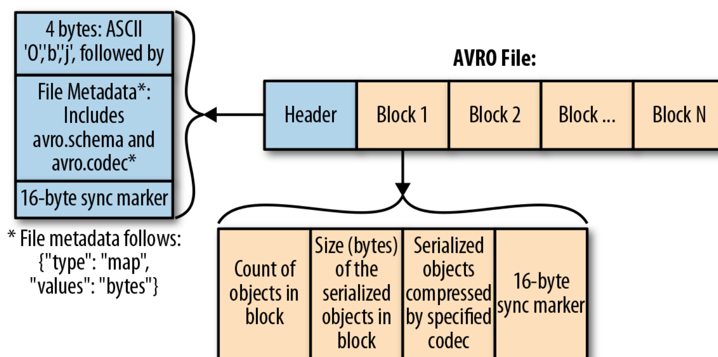
Source: Vertica (2022)

2.6.3 Apache Avro

Unlike formats such as Apache Parquet and Apache ORC, Apache Avro is a row-oriented file format. It can also be seen as a serialization framework, given that its serialization techniques can be applied in cases that do not necessarily involve writing data to a file.

Avro stores a schema, using the JSON format, to define the names of columns of a dataset, as well as its types. The schema is defined in the file's header, followed by blocks that can serialize data as binary or JSON values (VOHRA, 2016). If the binary encoding is used, some compression algorithms, like *Snappy*, *deflate*, and *bzip2* can be applied. The structure can be visualized in more detail in the figure 2.5.

Figure 2.5: Avro Structure Illustration



Source: Ackerman and King (2019)

One of the advantages of the Avro format is that schema evolution can be done

easily, given that, when a column is modified, added or removed, the schema, specified in JSON, can be used to deserialize the data properly. It is important to the implementation of some techniques related to mutability that are going to be presented in the Data Lake table formats's section.

2.7 Data Lake Table Formats

Dealing with structured data in Data Lakes is a recurring task. The decision on how the data is organized and made available to customers is important for performance, usability and even consistency of the datasets. Considering these aspects, specifications and implementations of table formats for Data Lakes are becoming popular.

In summary, these formats aim to provide the combination of characteristics found in Data Warehouses and Data Lakes: standard DBMS features implemented over Object Storages. This is challenging, given the fact that most Cloud Object Storages are merely key-value stores, with no cross-key consistency guarantees (ARMBRUST et al., 2020).

The idea of these table formats consist, basically, in enabling mutability and ACID properties when processing data in these environments. In this section, three of these formats are presented: Delta Lake, Apache Hudi and Apache Iceberg. These are the table formats are going to have their performance compared when some usual operations, related to analytical workloads, are performed. It is worth noting that, during the presentation of each format, some important concepts, that are shared among all the formats, are presented.

2.7.1 Delta Lake

The Delta format is an open source ACID table storage layer over cloud object stores (ARMBRUST et al., 2020). Like described previously, it is one of the formats that was evaluated, in terms of performance, by this work. At its core, the format maintains logs about which data objects (maintained as Apache Parquet files) are part of a table, and stores these logs in the Object Storage. The logs, therefore, are used to ensure the ACID properties of the operations made over the tables (ARMBRUST et al., 2020). The design of the format also provides other features, like:

- Time travel capability: the ability to query a specific version of a dataset, before the

execution of operations that updated data, for example.

- UPSERT and MERGE operations: support operations that update or insert rows based on unique identifiers of tables
- Schema evolution: schema change support (such as adding a column to a table) without rewriting historical data

Like cited previously, the Delta format implementation uses a set of files, that have different purposes and are used to construct the table and ensure some properties. These files are: the data objects, the logs, the log checkpoints. They are going to be described with more detail right next.

2.7.1.1 Data Objects

The data objects (or files) are Apache Parquet files. They are used to store the table contents. Each file, usually, is named with a GUID. Thus, when a new version of a file/partition is created (after a data update operation, for example), a new GUID is generated. The file, then, is referenced in a transaction log, identifying that it belongs to the most recent version of the table (after a sequence of operations, defined in the transaction, was executed).

In figure 2.6, an example of a table named `mytable`, that is partitioned by `date` column. Each Parquet file represents a version of a partition.

2.7.1.2 Logs

All the logs of the Delta format are stored in the `_delta_log` folder (as shown in figure 2.6). The folder stores a sequence of JSON files, that are used to register actions that need to be applied to a specific version of a table, in order to generate the new version. The actions that can be registered in a log file vary. The main ones are:

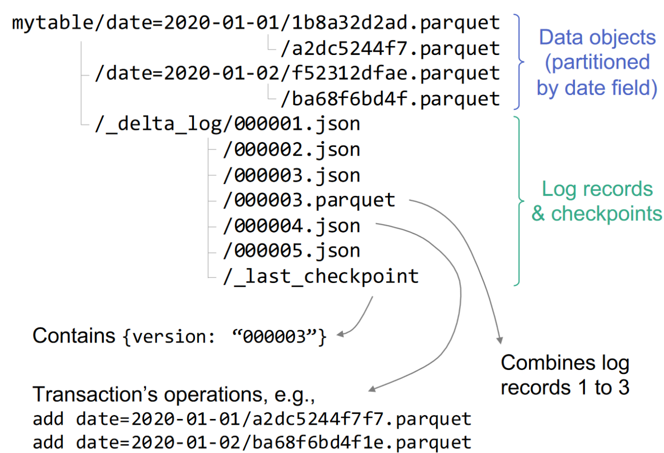
- Metadata Changes: used to register changes related to the table's metadata, like a modification of the schema, for example;
- Addition of files: used to register the addition of data objects (the Parquet files);
- Removal of files: used to register the removal of data objects (the Parquet files).

A *transaction* can be viewed as successfully applied when all the actions described in the logs are applied atomically.

2.7.1.3 Log Checkpoints

The log checkpoints, as can be seen in the figure 2.6, are, basically, log files that are compressed periodically. The file stores all information about the actions until a log ID, removing redundant actions. The periodic compression has the objective of improving the performance of the client when reading the table's metadata. The client can find the ID of the last checkpoint in the `_last_checkpoint` file (as shown in the figure 2.6), thus, avoiding LIST operations in the Object Storage, which are usually expensive.

Figure 2.6: An overview of the disposition of the Delta table files



Source: Armbrust et al. (2020)

2.7.2 Apache Hudi

The Apache Hudi table format is another one that brings Data Warehouse capabilities to Data Lakes. At its core, the format aims to support features like transactions, table upserts/deletions and schema evolution (APACHE, 2022d). The format uses two types of files for storing data: columnar based files (e.g.: Apache Parquet) and row based files (e.g.: Apache Avro), having its own purpose depending on the type of the table format that is being defined.

In the next subsections, some of the properties that are important for this format, like the types of tables that it has, and the types of queries that can be performed over these tables are presented. The structure of the files that compose the format are illustrated as well.

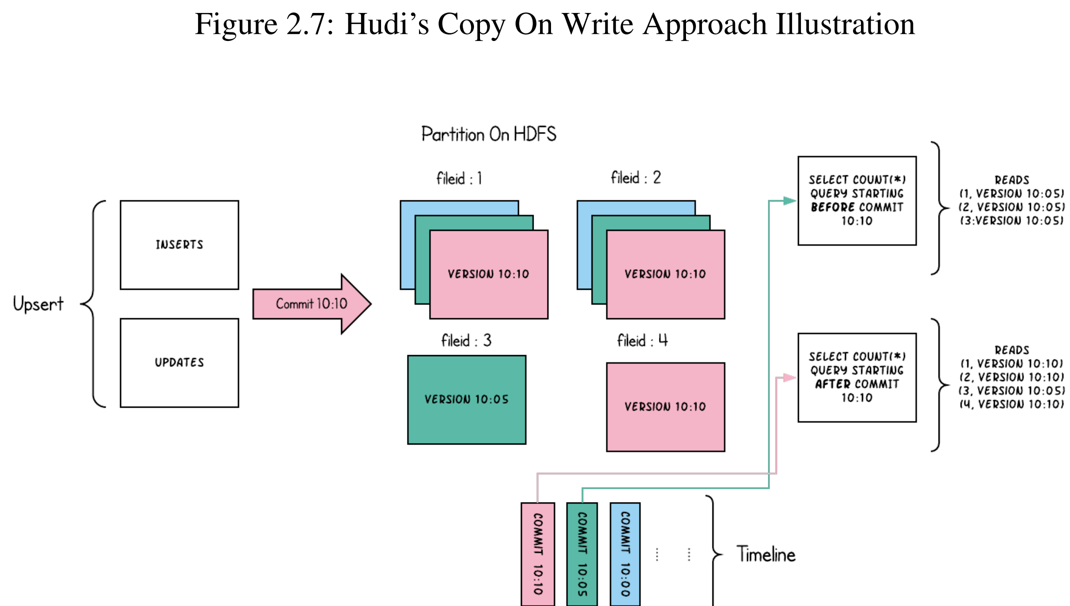
2.7.2.1 Copy On Write Tables

There are two types of tables in the Apache Hudi format: Copy on Write and Merge on Read. In this section, the Copy On Write type is presented. This type consists, basically, in generating a new version of a data file when a operation that modifies the data (such as an update) is performed. The data stored using this table format uses, exclusively, the columnar based formats file formats. The default one is the Apache Parquet format.

This approach increases the cost of writes, since it consists in rewriting all the base file (or partition of a dataset), considering the data modifications made by the user. The read cost, though, continues almost the same, since only one file (the file with the latest version of the dataset) is read. This is ideal for analytical workloads, that are usually read-heavy (APACHE, 2022e). This approach was not mentioned explicitly, but is the one that is used in the Delta Lake format.

The figure 2.7 illustrates an operation that is performing data insertions and updates in a table, generating new versions of the base data files on each commit (shown using different colors). In this case, each group of files (fileid 1, 2, 3 and 4) represent a partition, that store data using columnar based files.

Figure 2.7: Hudi's Copy On Write Approach Illustration



Source: Apache (2022e)

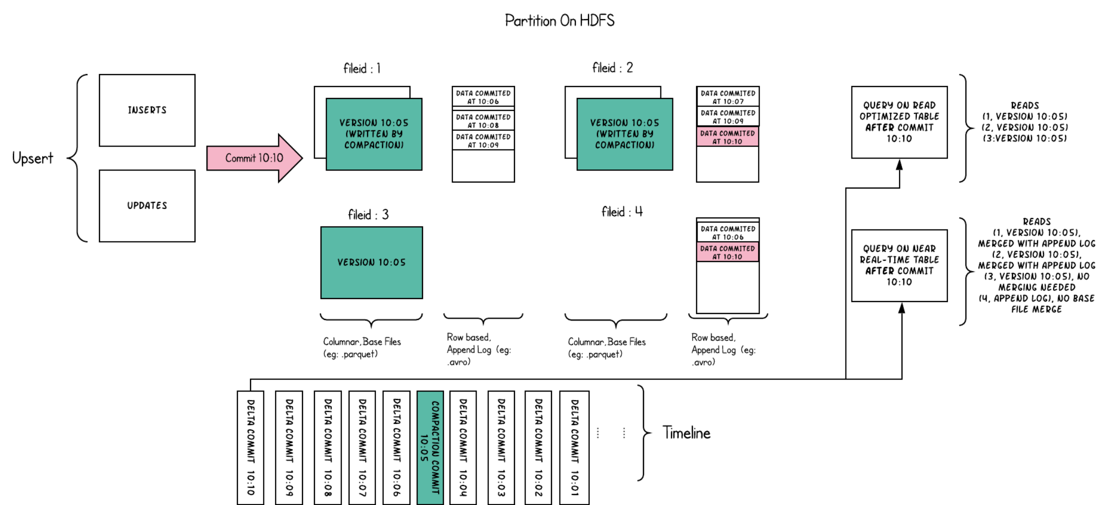
2.7.2.2 Merge On Read Tables

The Merge on Read tables use the approach of writing data updates in row based files (e.g.: Apache Avro), that store the difference (or the delta) between the current version of the data file and the next one, considering the insertions/updates/deletions that were performed in the transaction. As a result, we have a base file, stored using a columnar based format, and a sequence of row based files, that store the differences from previous versions.

As the name of the approach suggests, when the user reads this tables using a query engine, it performs a merge operation during the read phase, using the base file and the latest version of the "delta" file. This increases the read cost, since more than one file needs to be read, and the merge process needs to be performed on-the-fly (APACHE, 2022e). The benefit, though, is that, due to its nature, it enables more frequent writes, that is useful for near real-time workloads.

The figure 2.8 illustrates an example in which a new set of insert and update operations are performed over a table, generating new row based files (deltas of the previous versions of a partition's base file, illustrated in pink), registering a commit over a table. The figure also illustrates a compaction process, not cited previously, that is performed periodically, applying the modifications of a delta file to the base file, generating a new columnar based file, that is used as a new version of the dataset. This is important to improve the merge times, since the delta files become larger after each performed operation.

Figure 2.8: Hudi's Merge On Read Approach Illustration



Source: Apache (2022e)

2.7.2.3 Query Types

The Hudi's table types support different query types. They define how the table's underlying data is exposed to read operations. Each query type has its own trade-offs. A table mapping the table types to the supported query types is shown next. Then, an overview of each query type is presented.

Table 2.1: Supported Query Types by Table Type

Table Type	**Supported Query types**
Copy On Write	Snapshot Queries, Incremental Queries
Merge On Read	Snapshot Queries, Incremental Queries, Read Optimized Queries

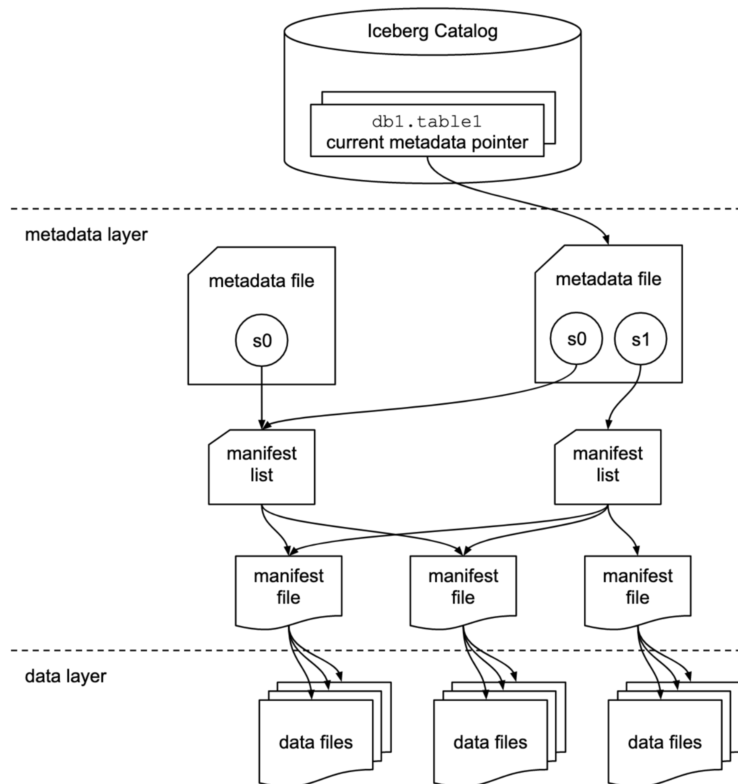
Source: Apache (2022e)

- **Snapshot Queries**: Provides the latest version of the dataset when reading/querying a table. In the case of the Merge On Read Tables, it merges the latest delta log file with the base file, and, in the case of the Copy On Write Tables, it reads the latest version of the columnar based file.
- **Incremental Queries**: Allows to query/read only the records that were modified since a commit, providing a more efficient way to only process the changed records incrementally.
- **Read Optimized Queries**: Exposes only columnar based files, generated after a commit or compaction process (used in the Merge On Read Tables). Guarantees a better query performance compared to the snapshot queries, since there's no merge process involved.

2.7.3 Apache Iceberg

As the other Data Lake Table Formats described in the previous sections, the Apache Iceberg format aims to support features like transactions, table upserts/deletions and schema evolution (APACHE, 2022a). In this section, some of the important definitions that are associated with these features are going to be described. The definitions are based on the second version of the Iceberg's specification (APACHE, 2022b), that is the one used to perform the performance experiments of this work. The definitions are based on the layers, defined by the format's specification: metadata layer and data layer (APACHE, 2022b)

Figure 2.9: Iceberg Format Overview



Source: Apache (2022a)

2.7.3.1 Metadata Layer

Similar to the other formats presented previously, the Iceberg format stores files that are important to maintain metadata information about the table. The layer is composed by *metadata files*, *manifest list files* and *manifest files*, that can be seen in the figure 2.9.

The metadata files maintain the table's state (APACHE, 2022b). They store information about the table's schema, partition information, history of snapshots, and which is the current snapshot. In this case, a snapshot represents the state of a table at a specific moment, considering its change history. Each snapshot has a manifest list file (described in the next paragraph) associated to it. If a change happens in the table, a new metadata file is created and replaces the old one atomically.

The manifest list files and manifest files maintain information about a snapshot, and are stored as Apache Avro files. The manifest list maintains a list of manifest files metadata, like their locations, the partitions of the table it belongs and some information about the data, like the lower and upper bounds of the partition column. The manifest

files, then, maintains information about the data files (that are described in the Data Layer section). These files store useful information, like statistics of a columns' data, record count and partition membership, that are used for improving query performance while reading data.

2.7.3.2 Data Layer

The data layer is where the data itself is stored. It is composed by a set of data files that can belong to more than one snapshot, for storing a table partition's data. As default, these files are stored as Apache Parquet files, but other ones, like Apache Avro or Apache ORC, can be used as well.

Like Apache Hudi, Iceberg has two possible table types: Copy on Write and Merge on Read, each one with the same trade-offs as the ones presented in the section 2.7.2. Depending on the approach, the files are arranged and referenced in the manifest files differently. In the figure 2.9, the arrows show how the files reference each other.

3 RELATED WORK

In this chapter, some work related to the main subjects of this thesis are presented. Each section presents a summary of a specific work, establishing relationships with the topics that are relevant to this thesis.

3.1 On Data Lake Architectures and Metadata Management

Sawadogo and Darmont (2021) presents an extensive review of the existent approaches for designing a Data Lake, focusing on presenting possible architectures and metadata management approaches.

Part of the work discusses possible methods for combining the implementations of Data Lakes and Data Warehouses, presenting an approach that define Data Warehouses as part of a Data Lake, based on the architecture defined by Inmon (2016), that presents the abstraction of "data ponds" for establishing regions in Data Lakes, which, when used to maintain structured data, can be viewed as Data Warehouses (INMON, 2016).

These definitions are important to this thesis, given that the purpose of the Data Lake Table Formats that were compared in this work is to satisfy this type of architecture, supporting DMBS like operations and properties in Data Lakes, which are important for the data warehousing processes.

3.2 The impact of columnar file formats on SQL-on-hadoop engine performance: A study on ORC and Parquet

Ivanov and Pergolesi (2020) propose an approach for evaluating the impact of different file formats on SQL-on-hadoop engines. More specifically, two columnar file formats are compared: Parquet and ORC, using Hive and Spark SQL as engines, and different parameter settings, such as different data compression algorithms. The work also presents some performance comparisons, using different approaches, that can be found in the literature.

The approach presented in the work uses, as well one of the experiments of this thesis (4.3.1), a TPC benchmark (TPCx-BB) as an inspiration for defining the experiments, and then, computing the load time, as well as the query time for the datasets and

queries defined in the benchmark. It uses an infrastructure that is similar to the one defined for the experiments of this thesis.

Although the work does not compare Data Lake Table Formats, like this thesis, it shows some insights on how different query engines and file formats can impact the performance of analytical workloads. Thus, given the fact that the Data Lake Table Formats compared in this thesis rely on different Big Data File Formats, and support multiple query engines, the choice on which of these formats or engines to use could impact the overall performance of the table formats.

3.3 A study of SQL-on-Hadoop systems

Chen et al. (2014) gives an overview of some SQL-on-hadoop engines, and then propose an experiment for comparing the performance of five of this systems. The experiment is based on selected workloads from the TPC-DS benchmark (that served as an inspiration for one of theset of experiments of this thesis).

Given the time is was written, the work does not use the Spark SQL module (that is used for the experiments defined in this thesis) as one of the SQL-on-hadoop systems for performing the experiments and comparisons, but it introduces some interesting aspects on the evolution of these type of systems, reinforcing the importance of having a SQL layer over the data stored in HDFS for empowering end users when analysing big volumes of data.

Some particularities of the SQL engines are also presented, like different techniques each system use for different operations, and how they impact the performance of the workloads.

3.4 Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics

Armbrust et al. (2021) overviews the evolution of the architectures used to store data in OLAP environments. It presents the concepts of Data Warehouse and Data Lake, and how the usage of these repositories changed throughout the time. It also presents some challenges imposed by these data architectures, proposing a new architecture, called "Lakehouse".

The architecture is defined as a mix of the Data Lake and the Data Warehouse architectures, ensuring support to DBMS properties and features using a low cost data storage (like Cloud Object Storages). A suggestion for implementing this type of architecture is then presented, with the definition of a transactional metadata layer on top of the data objects, stored as Parquet or ORC files, for example.

This architecture relates directly to the Data Lake Table Formats compared in this thesis, given their properties and their capability of providing Data Warehouse management features in Data Lakes.

4 METHODOLOGY

In this chapter, the methodology used for the performance evaluation of the Data Lake Table Formats, presented in the Background section, is introduced. The questions posed for this work, as well as the objectives of the experiments proposed, are overviewed, explaining the motivations for choosing the methods used for evaluating the performances. Then, the experiments and the datasets used during experiments are described.

4.1 Overview

During the idealization of this work, that consists in comparing the Delta Lake, Apache Hudi and Apache Iceberg table formats (as described in the section 2.7), some questions were posed. The main ones were:

- What is the overall performance of these table formats during data load operations?
- What is the overall performance of these table formats during data update operations?
- What is the overall performance of queries made over the data stored in these table formats?
- How the these table formats compare themselves during these workloads?

Given these questions and the purpose of evaluating and comparing the overall performance of these table formats, some experiments were conducted, extracting, basically, the time a given task of the experiment took, for then, evaluating the performance using a quantitative approach. The experiments also had some parameters, that were used to vary the possible scenarios during the execution of the operations.

The architecture defined for the experiments, as well as the steps of each of them, are described in the next sections.

4.2 Architecture

The experiments of this work were conducted in a distributed environment, in the Cloud. The operations idealized for each experiment were executed using a Apache Spark

(version 3.2.0) cluster, varying the number of worker nodes used during the execution of each workload.

All the infrastructure was provisioned in AWS (Amazon Web Services), using the EMR (Elastic Map Reduce) platform. The platform, in short, provides an easier way to provision multiple virtual machines with Big Data frameworks, such as Apache Spark, already installed. Each node provisioned represented a Spark node, with one master and n worker nodes. An isolated virtual machine was also used to provision a Hive Metastore instance, that is from the Hadoop ecosystem used to define, virtually, the names of the databases and tables created by the user, as well mapping with table format and what location in the Object Storage is associated with the table.

The virtual machine specifications of all the Spark nodes were the same during the execution of the experiments. The only variation in the infrastructure was associated with the number of worker nodes. The specification of the nodes is defined in the table 4.1

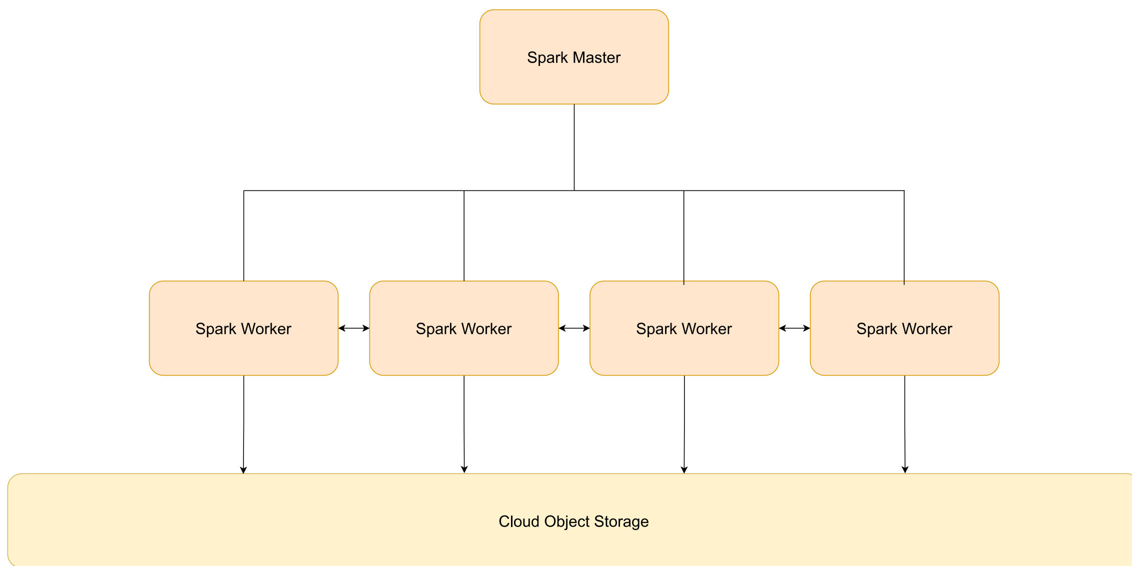
Table 4.1: Experiments' Virtual Machines Specification

Instance Type	i3.xlarge (Amazon EC2)
Number of vCPUs	4
RAM	30GB
Processor	Intel Xeon E5-2686 v4 (Broadwell) (2,3 GHz)
Architecture	x86_64
Network Bandwidth	10 Gbps
Storage	950 SSD NVMe

An illustration of the architecture, with the maximum number of nodes used in the experiments ($n = 4$), is shown in the figure 4.1. Worth noting that the illustration does not represent the physical characteristics of the nodes and the cloud storage solution. It only represents the interactions among nodes and services. Given the infrastructure was provisioned in a Cloud environment, the underlying management of the computational resources vary depending on the Cloud vendor. Techniques used for optimizing the performance of the services can be used as well, like the implementation of cache layers, for example.

For storing the data itself, the Cloud Object Storage solution chosen was the S3 (Simple Object Storage), from AWS.

Figure 4.1: Experiments' Architecture Overview



Source: The Author

4.3 Experiments

As placed earlier, the experiments conducted in this work try to obtain data for evaluating the performance of the table formats during the execution of some usual operations performed in analytical workloads over data stored in Data Lakes.

For this purpose, two main approaches were chosen: the first one, conducting the execution of experiments using as inspiration the TPC-DS benchmark (defined by the Transaction Processing Performance Council), and a second one, using a set of experiments for evaluating, in addition to load and query operations, update operations, that are not covered by the first set of experiments.

The approaches used, as well as the parameters and datasets used for each experiment, are defined in the next subsections.

4.3.1 TPC-DS Inspired Experiment

The first choice for evaluating the performance of the table formats was executing a TPC-DS inspired experiment. As a context, the TPC-DS is an industry standard benchmark for "general purpose decision support systems" (SPECIFICATION; TPC, 2000). As the specification states, the benchmark illustrates support systems that:

- Examine large volumes of data;
- Give answers to real-world business questions;
- Execute queries of various operational requirements and complexities;
- Are characterized by high CPU and IO load;
- Are periodically synchronized with source OLTP databases through database maintenance functions;
- Run on “Big Data” solutions, such as RDBMS as well as Hadoop/Spark based systems.

In order to address the range of query types that belong to OLAP workloads, ninety nine queries are defined by the benchmark, trying to emulate what the specification defines as “four broad classes of queries that characterize most decision support queries” (SPECIFICATION; TPC, 2000):

- Reporting queries
- Ad hoc queries
- Iterative OLAP queries
- Data mining queries

Like stated earlier, the experiments used in this work does not follow all the specifications defined by the Transactional Performance Council’s benchmark. The TPC-DS inspired experiment is based on the implementation of an open sourced experiment framework, used by community for evaluating the Delta Lake format performance, measuring the load performance of the data on the TPC-DS defined tables, as well as the performance of the queries made on top of data stored in these tables.

The experiment’s dataset is exactly the same as the one defined by the TPC-DS benchmark. It basically models a Data Warehouse, defining tables that are usually found in this type of environment. An overview of the dataset is placed in the next subsection. The benchmark phases are also shown next.

4.3.1.1 Dataset Overview

For performing the experiments, the TPC-DS schema defined in the specification was used. The schema is designed using an approach that is usual in data warehouses, called "dimensional modelling", that uses the concepts of facts (or measurements) and dimensions (or the descriptive contexts associated to the measurements) (KIMBALL;

ROSS, 2011), that are used to define the tables in the schema, and store data that is useful for analytical purposes. The fact tables usually have references to dimension tables. In the case of the TPC-DS's schema, the dimension tables can also have references to other dimension tables, creating a multilevel structure, that is defined as a "snowflake schema" (KIMBALL; ROSS, 2011).

The schema of the benchmark specification models sales and sales returns (when a customer or client sends a product back to the seller) of a fictitious company, that sells its products through three channels: stores, catalogs and internet. It has seven fact tables, that are defined in the table 4.2, and seventeen dimension tables.

Table 4.2: TPC-DS Tables

Table Type	Table Name
Fact	Store Sales Store Returns Catalog Sales Catalog Returns Web Sales Web Returns Inventory
Dimension	Store Call Center Catalog Page Web Site Web Page Warehouse Customer Customer Address Customer Demographics Date Dim Household Demographics Item Income Band Promotion Reason Ship Mode Time Dim

Source: (SPECIFICATION; TPC, 2000)

4.3.1.2 Experiment Phases

Like stated previous, the experiments done in this work do not follow all the specifications defined by the original TPC-DS benchmark. The TPC-DS inspired experiment,

Listing 4.1 – Data Load Pseudo-code

```
CREATE TABLE full_table_name
USING table_format_name
LOCATION 'path/to/store/table/files'
SELECT * FROM 'path/to/raw_data_file.parquet'
```

defined by this work, consists of two phases, that are defined next:

- **Load Phase**: the phase when the TPC-DS fact and dimension tables are populated. The process consists in, basically, reading the raw data from Apache Parquet files, generated according to the TPC-DS specification, and inserting the records from these files into the destination tables, that are defined using one of the table formats compared by this work (Delta Lake, Apache Hudi or Apache Iceberg). A pseudo-code of load step for each table can be seen in the listing 4.1.
- **Query Phase**: the phase when the ninety nine queries defined by the original benchmark are executed. Al the queries are defined using SQL, trying to emulate usual query operations used in decision support processes.

It is worth noting that all phases are executed using the Spark SQL module, varying the number of worker nodes used.

4.3.1.3 Implementation

The implementation of the benchmark was based on an open sourced implementation made the community of the Delta Lake format project. The original implementation was extended to support all the formats that were intended to compare and evaluate the performance. The implementation of the modified version, that supports the Apache Hudi, the Apache Iceberg and Delta Lake table formats can be found in this url: <https://github.com/rafaelvargas/delta/tree/add-other-formats-support-benchmark/benchmarks>. During the experiments, the data that was used to populate the TPC-DS tables was also the one provided by the same open source implementation. The scale of the dataset was reasonably small, having 1GB, due to cost factors associated with the Cloud Computing environments that were used to perform the experiments.

The implementation (and the generation of data) follows the requirements defined by the original TPC-DS specification, that says "the TPC-DS database must be implemented using commercially available data processing software, and its queries must be

executed via SQL interface" (SPECIFICATION; TPC, 2000). In the case of the implementation used for this work, the Spark SQL module for Spark was used, as well as the Hive Metastore for defining the organization of tables virtually (such as defining its names and associated databases).

Worth noting that, for all the steps of the experiment, the Apache Parquet format was used as the default format of the tables' "data layers", using the *snappy* compression algorithm. All the tables were defined using the Copy on Write approach (that is described in the section 2.7.2.1).

4.3.2 Load, Update and Read Performance Experiment

The second experiment for comparing the table formats was defined trying to cover an use case that was not covered by the first experiment: when the data is updated.

Given that Data Lakes are usually design based on files, this type of operation is challenging. Considering an usual Data Lake design, users typically tend to read an entire table (or partitions of them), and then, overwrite the data for updating the datasets. This process is error prone and hard maintain, thus, the use case related to it is one of the problems the table formats that are compared in this work try to tackle.

The experiment designed for comparing the performance of the table formats when submitted to this use case consists of three phases: the load phase, the update phase and the query phase, that are going to be described in the section 4.3.2.2. To perform the experiments, a synthetic dataset was generated to populate the tables that were defined to execute the experiments. An overview of the dataset and the schema is shown in the section 4.3.2.1. At the end, in section 4.3.2.3, an overview of the implementation of experiment is given.

4.3.2.1 Dataset Overview

The dataset used for this experiment models tables that are used to maintain daily usage data of a fictitious music streaming app. The schema consists of one fact table and four dimension tables. The fact table keeps track of daily measures related to the user's behavior in the app. The dimension tables have the contexts that are referenced by each measurement. The list of tables is given next.

The fact table maintains columns with IDs that reference all the dimensions, as

Table 4.3: Load, Update and Read Performance Experiment Tables

Table Type	**Table Name**
Fact	FactDailyUsageByUser
Dimension	DimUser DimPlan DimPlatform DimCountry DimSoftwareVersion

Source: The Author

well as the date associated with the fact table measurements. The primary key of the table is composed of six columns: `Date`, `UserID`, `PlanID`, `PlatformID`, `CountryID` and `SoftwareVersionID`. Each row maintains the following measurements: number of logins, number of songs played and the usage duration in seconds. The measurement is associated with each combination of values of the columns of the primary key that are presented in the table. The table is partitioned by the `Date` column.

The dataset generation for the fact table consisted in generating random records for a month. For the update operations, portions of the dataset measurements were regenerated randomly, for then, performing the updates.

4.3.2.2 Experiment Phases

The phases of this experiment, as stated earlier, were designed considering three operations. One of them, focusing on the use case that was not covered by the TPC-DS Inspired Experiment: the data update phase. Each phase is described next. The variable factors associated with each phase are defined as well.

1. **Data Load Phase**: the phase when the generated datasets are loaded in the fact and dimension tables. The synthetically generated data is read from Apache Parquet files and loaded into the tables (like the load phase in the TPC-DS Inspired Experiment, that has its pseudo code defined in the listing 4.1).
2. **Data Update Phase**: the phase when the fact table data is updated, by randomly modifying data of a portion of measurements. The percentage of rows of the fact table updated during this phase is 16%. A pseudo-code of the update phase is shown in the listing 4.2.
3. **Query Phase**: the phase when six queries are executed. The queries are identified by indexes from one to six. They extract the following metrics, respectively: the number of daily active users for all the period; the total number of active users for

Listing 4.2 – Data Update Pseudo-code

```
MERGE INTO fact_daily_usage_by_user t
USING (SELECT * FROM parquet.`updated_measurements`) s
    ON t.date = s.date
    AND t.user_id = s.user_id
    AND t.plan_id = s.plan_id
    AND t.software_version_id = s.software_version_id
    AND t.country_id = s.country_id
    AND t.platform_id = s.platform_id
WHEN MATCHED THEN UPDATE SET
    t.duration_in_seconds = s.duration_in_seconds,
    t.number_of_sessions = s.number_of_sessions,
    t.number_of_songs_played = s.number_of_songs_played;
```

all the period; the average number of daily active users by plan, for all the period; the total number of sessions by each software version, for all the period; the top five countries with the greatest number of active users, for all the period; and the platform with the greatest number of active users, by each plan, for all the period.

Each of the phases is executed varying the number of nodes used in the Spark cluster. All the phases were executed using the Spark SQL module.

4.3.2.3 Implementation

The implementation of the experiment used Python as a programming language, using the PySpark (an interface for Apache Spark in Python). Python was also used to implement the scripts to submit each PySpark application to the Spark cluster, considering the parameters and variables that were defined in the experiment by each variation of the experiments. The variable factors of the experiment were the following:

- Number of Worker Nodes: 1, 2, 3 and 4
- Dataset Scale: 1GB, 2GB, 4GB and 8GB
- Portion of the fact table updated during the update phase: 16%

All the queries and steps were executed via SQL interface. The execution time for each step was then measured and collected to analyse the results. The Hive Metastore was used for defining the tables virtually. The implementation of the experiment can be found in the following url: <https://github.com/rafaelvargas/data-lake-table-formats>.

Like in the previous experiment, the Apache Parquet format was used as the de-

fault format of the tables' "data layers", using the *snappy* compression algorithm. All the tables were defined using the Copy on Write approach.

4.4 Metrics and Variations

As placed in the previous sections, each experiment considered specific variations, that were defined in order to analyse and compare the performance of the different Data Lake Table Formats during the execution of workloads that are usually associated with decision support processes.

In the table 4.4, an overview of the variations of the experiments, as well as the metrics computed for each experiment, is presented.

Table 4.4: Experiments' Metrics and Variations Overview

	TPC-DS Inspired Experiment	**Load, Update and Read Experiment**
Data files format	Parquet	Parquet
Compression	Snappy	Snappy
Table Type	Copy on Write	Copy on Write
Operation Types	Insert and Read	Insert, Update and Read
Number of Workers	1 to 4	1 to 4
Scale	1GB	1GB, 2GB, 4GB, 8GB
Number of Tables	24	6
Number of Queries	99	6
Update portion	-	16%
Iterations by Operation	5	5
Metrics	Execution time and Speedup	Execution time and Speedup

Source: The Author

To analyse the results, the average of the execution times obtained after the execution of the five iterations planned for each operation was computed. The Speedup, which aims to show the evolution of the execution time as the number of workers grow, was also computed. The metric is defined as the division of a base time, which is the execution time of an operation using one worker ($T(1)$), by the execution time using n workers ($T(n)$). The formula is shown in the equation .

$$Speedup(n) = \frac{T(1)}{T(n)} \tag{4.1}$$

The results, using the metrics defined in this section, are presented in the next chapter.

5 RESULTS

This chapter presents the results obtained from the executions of the experiments defined in the Methodology section. The results obtained for each table format are, then, used for comparison purposes.

5.1 TPC-DS Inspired Experiment

In this subsection, the results related to the TPC-DS Inspired Experiment are going to be presented. First, the results of the load phase are presented and compared, and then, the results for the query phase are presented and compared.

5.1.1 Load Phase

Like described earlier, the load phase consisted of loading data into the fact and dimension tables that are defined in the TPC-DS specification. For each table format compared, and each table that was populated, the execution time of load operation was measured, repeating it five times, and computing the average of the five executions. The process, then, was repeated, varying the number of worker nodes in the Spark cluster. The number of nodes varied from 1 to 4.

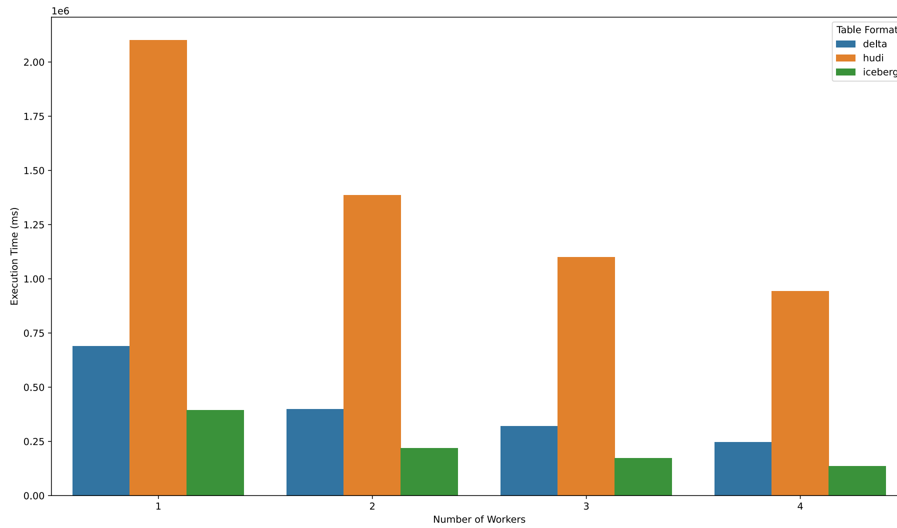
The plot, shown in the figure 5.1, presents the sum of the average load execution time of each table (represented in the Y axis, in milliseconds), for each table format (represented using different colors, defined in the legend), and each number of worker nodes used during the process (represented in the X axis). The shorter the execution time, the better.

As can be seen in the figure 5.1, the Apache Iceberg format outperformed the other formats during the load phase, for all the possible number of worker nodes. The Apache Hudi format had the worst performance, having an execution time greater than the double of the execution time of the second best format (Delta Lake).

For each table format, the Speedup for the load phase was also computed. The definition used for computing the metric is defined in the equation 4.1.

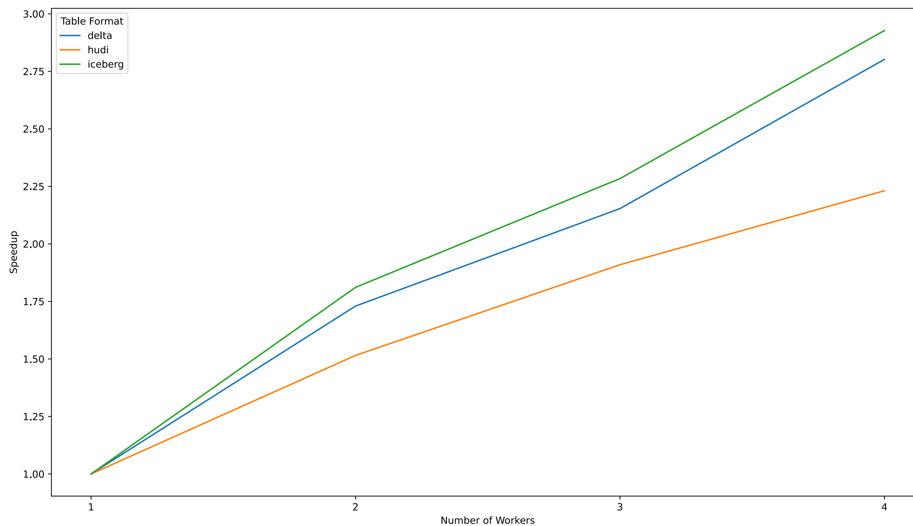
The figure 5.2, plots the speedup for each number of worker nodes used in the experiments. As can be seen in the results, the Apache Iceberg format also outperformed

Figure 5.1: Total Load Phase Duration



the other formats, executing the load phase almost 3 times faster using four worker nodes (when comparing the execution time to its base time, using one worker node).

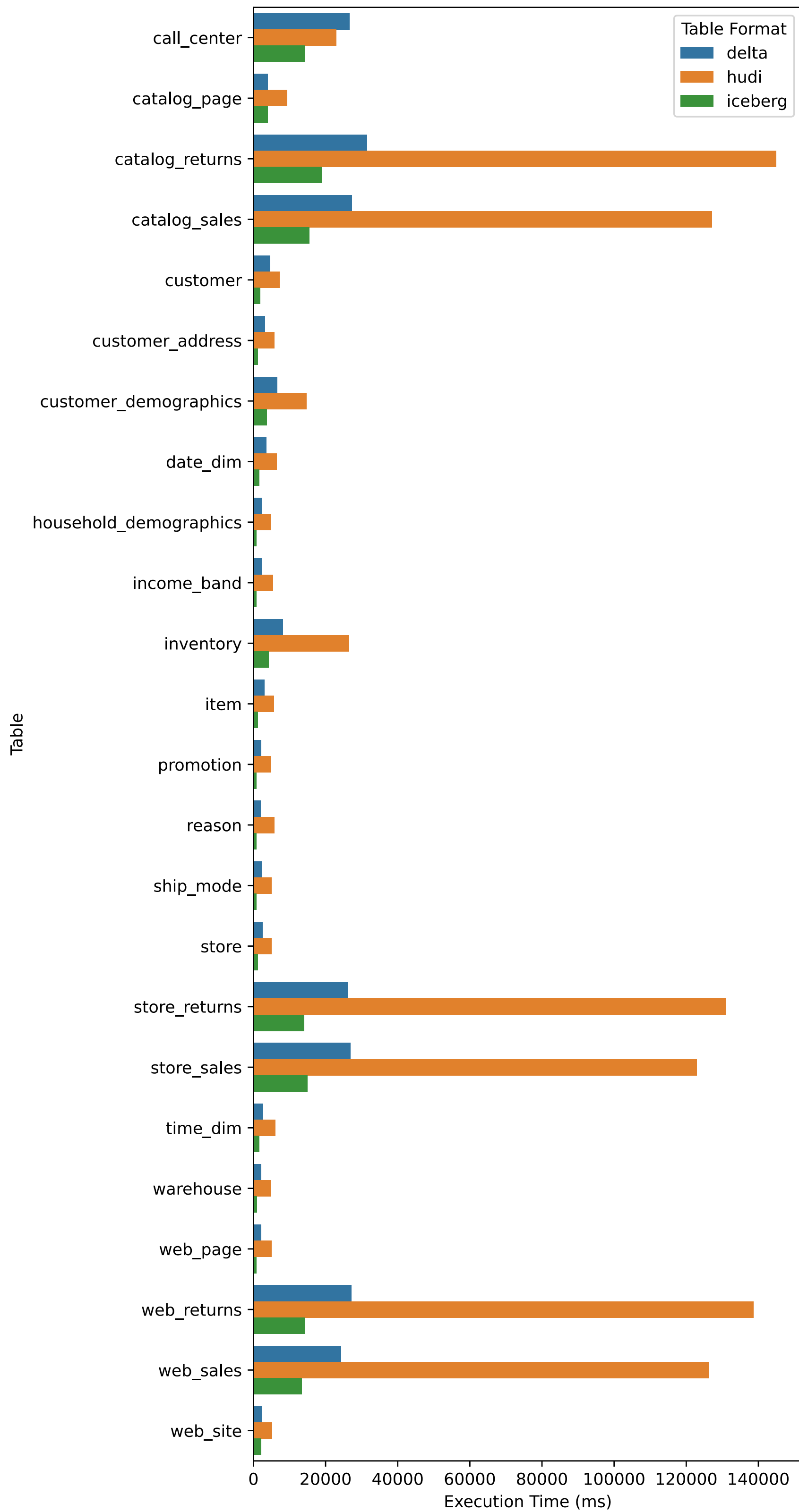
Figure 5.2: Speedup - Load Phase



When comparing the average execution time for loading each table, using 4 worker nodes, the Apache Iceberg format also outperforms the other formats for all the tables. The Apache Hudi format, was outperformed by the Delta Lake format in almost all the tables, except one: the *Call Center* table.

The plots of average execution times, for each table and table format can be seen in the figure 5.3. The plots for the results considering different numbers of worker nodes can be found in the appendix A.

Figure 5.3: Average Load Duration for each table (using 4 worker nodes)



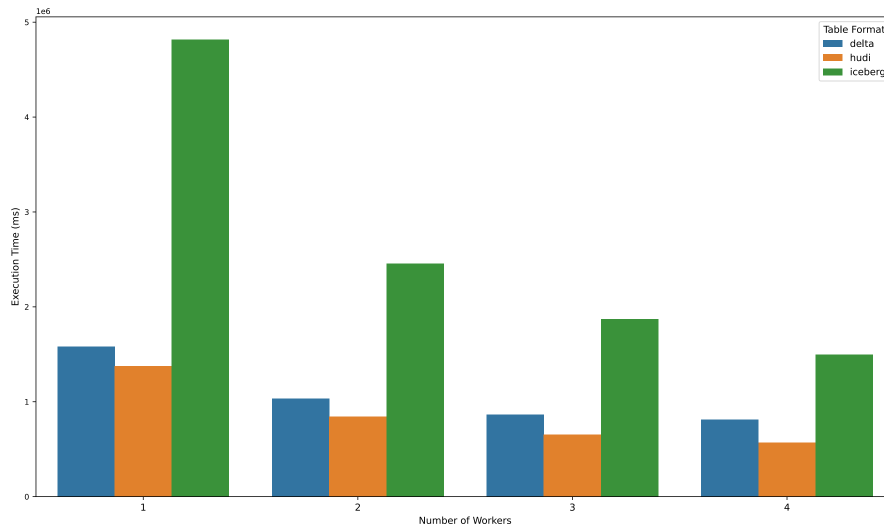
5.1.2 Query Phase

The query phase was executed after the load phase. It consisted of executing ninety nine queries defined by the TPC-DS benchmark, computing the execution time for each query.

Like the load phase, each step (or query) of the phase was executed five times. Then, the average of each execution time was computed. These averages were then used to compute the metrics that are shown next. Also worth noting that the execution of the phase was repeated varying the number of worker nodes in the Spark cluster. The number of nodes varied from 1 to 4 as well.

In the figure 5.4, a plot shows the sum of all the average query execution times of each query (represented in the Y axis, in milliseconds), for each table format (represented using different colors, defined in the legend) and each number of worker nodes used in the experiments (represented in the X axis).

Figure 5.4: Total Query Phase Duration

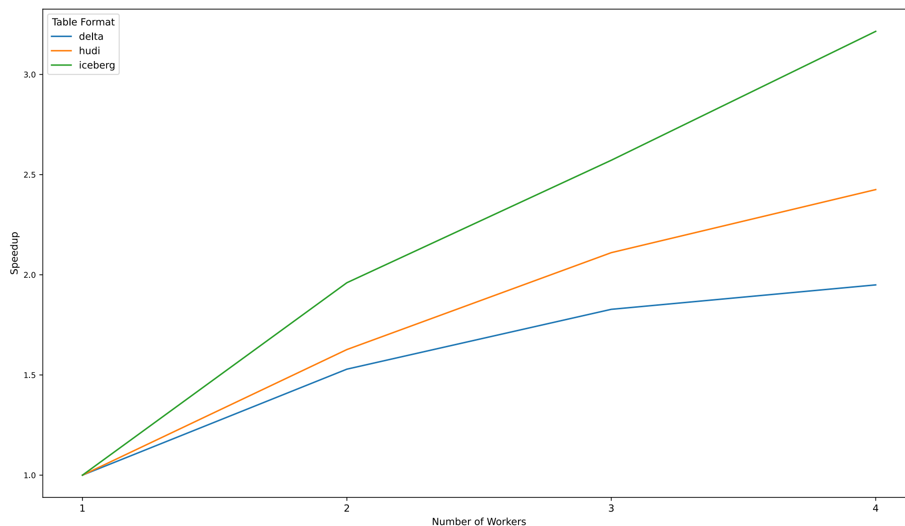


The results (plotted in the figure 5.4), show that the Apache Hudi format outperformed all the other formats during the query phase, for all the possible number of worker nodes. Using 4 worker nodes, the difference between the Apache Hudi format and the Delta Lake format execution time was, approximately, four minutes, while the difference of execution time between the Apache Hudi and the Apache Iceberg format was, approximately, fifteen minutes.

The speedup, as defined in the equation 4.1, was also computed for the query phase. The results, considering the sum of all the average execution times by query are

presented in the figure 5.5.

Figure 5.5: Speedup - Query Phase

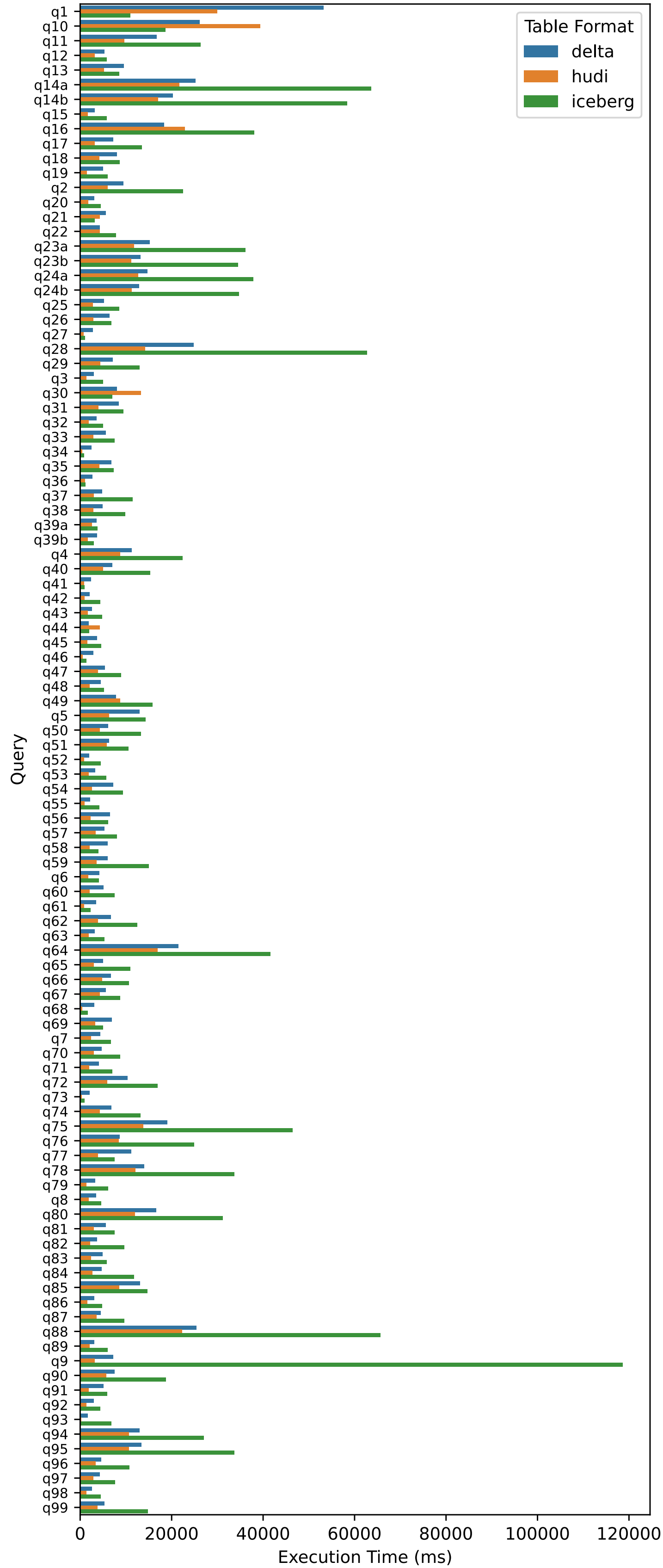


As can be seen, the Apache Iceberg format was the one that demonstrated the greatest improvement when the number of workers grew, followed by the Apache Hudi format, which had the second best performance in terms of speedup. The Delta Lake format was the one that improved the least.

Worth noting that the Apache Hudi format was the one that had the best performance in terms of execution time and the second best performance in terms of speedup, which may indicate that the format would perform even better than the others as new worker nodes were added to the cluster.

The average execution time for each query, using 4 worker nodes, can be seen in the figure 5.6. The results show that the Apache Hudi format outperformed the other table formats in almost all the queries, only being outperformed in five of them. There were only three queries that the Apache Hudi format had the worst performance among all the formats.

Figure 5.6: Average Query Duration (using 4 worker nodes)



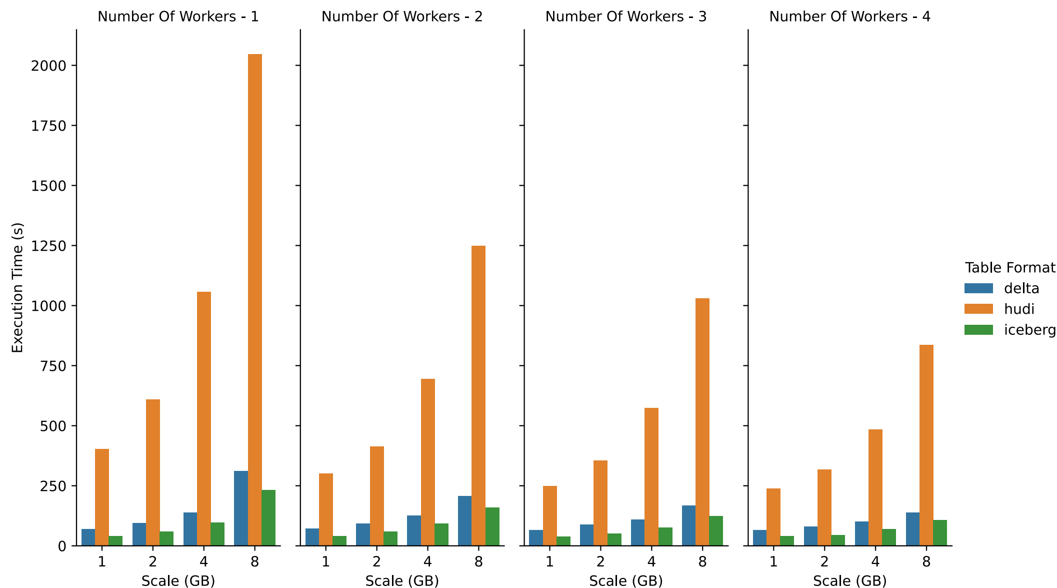
5.2 Load, Update and Read Performance Experiment

In this subsection, the results of the second experiment defined for this work are presented. The results are presented in the following order: load phase, update phase and query phase, each one presented in its own subsection.

5.2.1 Load Phase

In the load phase, as placed in the section 4.3.2.1, data was loaded into six tables: one fact table and five dimension tables. The execution time of each operation was computed. Each plot, shown in the figure 5.7, presents the sum of the average load duration for all the tables, (represented in the Y axis, in seconds), for each scale (represented in the X axis, in Gigabytes), considering each variation of the number of worker nodes, all grouped by the table formats (represented using different colors, defined in the legend).

Figure 5.7: Total Load Phase Duration by Scale



As can be seen in the results shown in the figure 5.7, the load phase showed similar performance results when comparing with the ones obtained in the TPC-DS Inspired Experiment. The Apache Iceberg format performed the best, for all the possible variations of number of worker nodes, followed by the Delta Lake format. The Apache Hudi showed the worst performance, like in the TPC-DS Inspired Experiment.

5.2.2 Update Phase

During the update phase, 16% of the measurements of fact table data were updated. This process was executed for each variation of the dataset scale, number of worker nodes and table format. The plots, shown in the figure 5.8, uses the same axes and legends definitions of the figure 5.7.

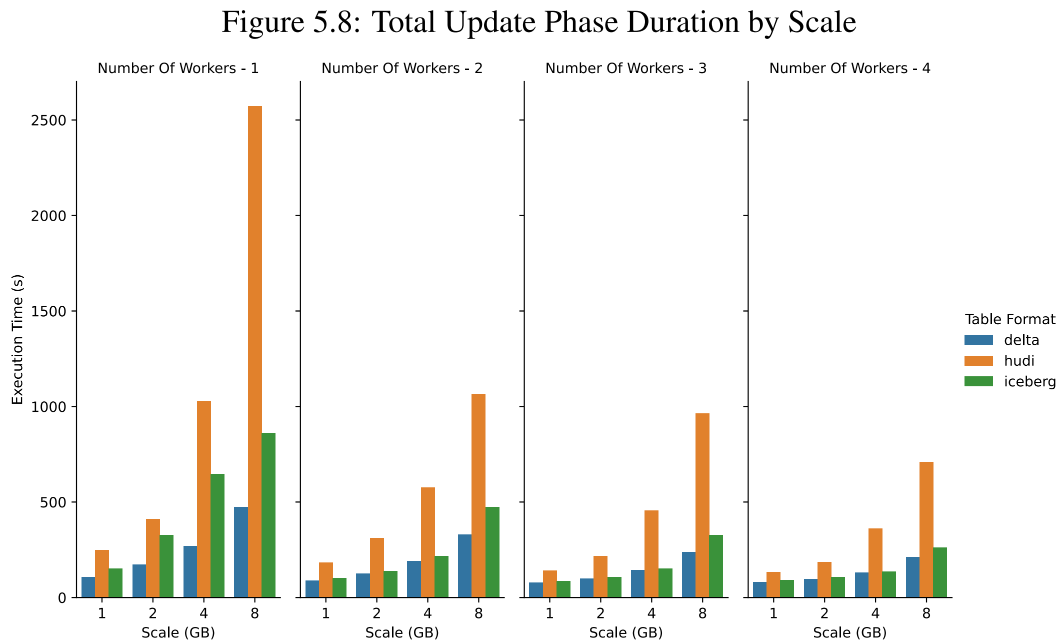


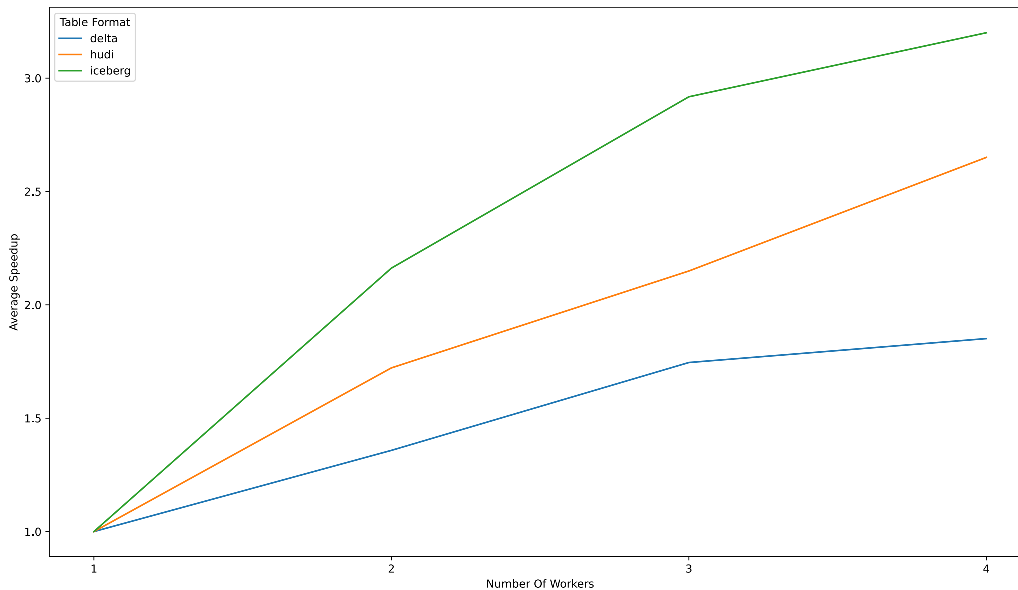
Figure 5.8: Total Update Phase Duration by Scale

The results presented in the figure 5.8 show that the Delta Lake format outperformed the other formats, executing the update phase faster in all the possible variations of the experiment. The Apache Iceberg format had the second best performance, and the Apache Hudi, the worst (as well as in the Load Phase).

After obtaining the results, the Speedup for each dataset scale was computed for each table format. Then, the average of the Speedups obtained for each table format was computed. The results are presented in the figure 5.9.

The results (plotted in the figure 5.9) show that the Apache Iceberg format performed the best as the number of worker nodes grew. The Delta Lake format, despite having the best execution times for all the possible variations of the update phase, had the worst Speedup results.

Figure 5.9: Average Speedup - Update Phase



5.2.2.1 Query Phase

The query phase, as placed in the section 4.3.2.2, consisted in executing six queries using the fact table and the dimension tables defined for the experiment. The results are shown next.

Considering the sum of the execution times for all the queries, the figure 5.10 shows the overall performance of each table format, using four workers nodes for executing the queries. The data plotted in the figure 5.10 shows that the Apache Hudi format outperformed the other table formats. The results are similar to those of the TPC-DS Inspired Experiment, in which the Delta Lake format had the second best performance, and the Apache Iceberg format, the worst. This behavior can be seen analysing the execution times by query, as well.

The results obtained for each query, using four worker nodes, are shown in the figure 5.11. For all the queries, the Apache Hudi format performed the best. For all the formats, the query that computes the number of daily active users (q1) has the worst performance. The second worst performance was obtained for the query that computed the average number of daily active users by plan (q3). The query that computed the software version with the greatest number of sessions performed the best for all the table formats. The results obtained using different numbers of worker nodes can be found in the appendix B, in which can be seen that the Apache Iceberg format performed significantly worse than the other formats when executing the queries using only one worker node.

Figure 5.10: Total Query Phase Duration by Scale (using 4 worker nodes)

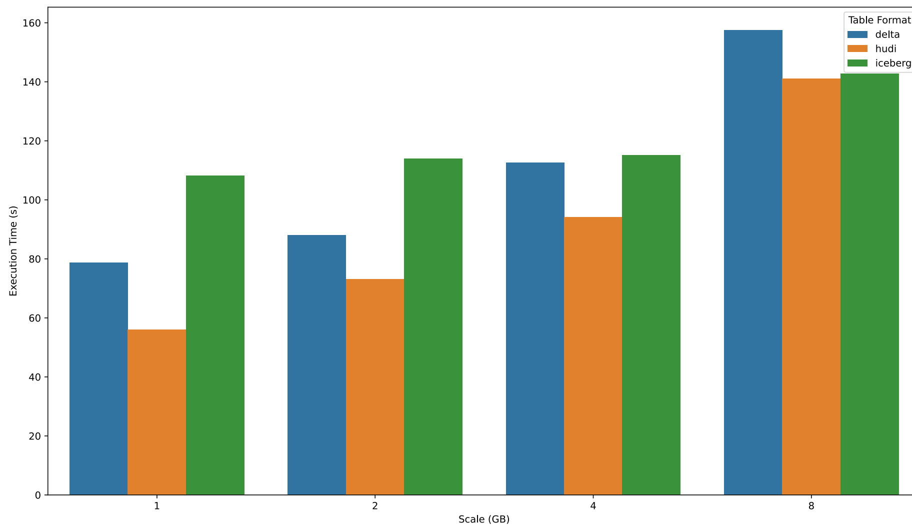
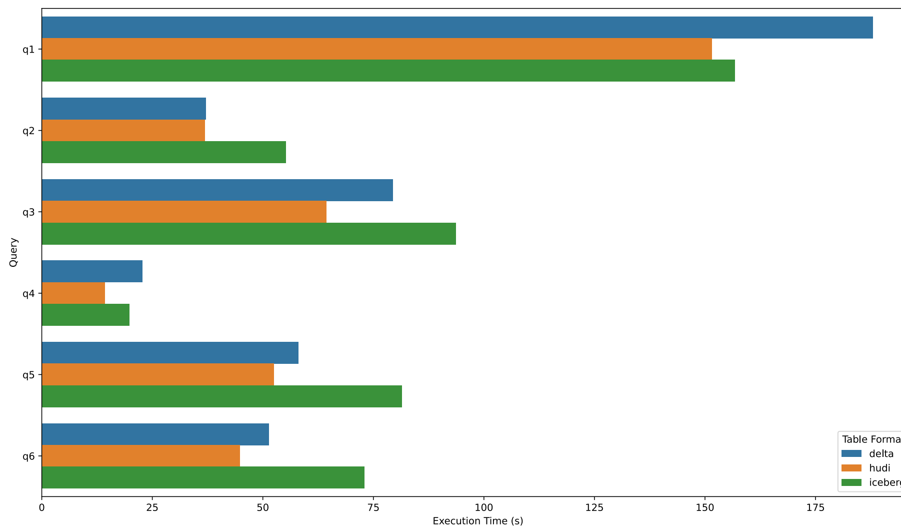


Figure 5.11: Average Execution Duration by Query (using 4 worker nodes)



5.3 Overall Analysis

As presented in the previous sections, each set of experiments provided results that can be used to better understand the behavior of each table format when submitted to read, insert and update operations. Thus, a brief analysis is presented, reviewing the results of both set of experiments.

The phases of each set of experiments contemplated different types of operations. For each type of operation, a brief summary of the results is shown next.

- **Insert**: the results obtained for both set of experiments were consistent. The Apache Iceberg stood out, presenting the best performance when comparing it to

the results of other table formats.

- **Update**: for different variations of the second set of experiments during the update phase, the Delta Lake format showed the best results.
- **Read**: the results obtained for both set of experiments were also consistent. The Apache Hudi stood out, presenting the best performance when comparing it to the results of other table formats.

Worth noting that, on average, the Delta Lake format was the most consistent one, in terms of performance, for all the possible variations of the experiments. The results of performance evolution as the number of workers grew, however, showed that the table formats could behave more similarly, in terms of performance, with a larger number of workers. This is discussed, as an improvement of this work, in the chapter 6.

Given the characteristics of analytical workloads, that involve more frequent data reads, the format Apache Iceberg showed the worst performance for this scenario. The costs for insert and update operations, however, cannot be neglected, given these type of operations are important to maintain data freshness (with more frequent writes) and consistency/quality (with more frequent updates).

6 CONCLUSION

This work provided a set of results based on the experiments idealized for comparing the performance of Data Lake Table Formats, when some usual operations over these tables are performed. The results obtained, thus, can be used for getting insights on which one is the most appropriate format for a specific set of requirements. The results of this work could also be used for identifying points of improvements, that could be implemented in new versions of these formats.

Given the objectives of this work, that was centered around the comparison of the table formats when submitted to insert, update and query operations, it was identified that each table format outperformed the others when submitted to one of these operations. The Iceberg format surpassed the others in the insert phases, the Delta Lake format, in the update phase, and the Hudi format, in the query phases. Thus, based on the type of workload that is most frequent in the user's data environment, one format can be chosen over another based on these results. Given that, usually, the analytical workloads are composed by read heavy operations, the natural choice would be choosing the one that has the best read performance, but, if there are some peculiarities in the workloads, that require more frequent writes or updates, with the objective of maintaining the data consistent or updated, for example, other format could be chosen over the one with the best read performance.

The available infrastructure is also a constraint that could impact the performance of one format over another, specially, when the number of worker nodes vary. The results showed that the speedup during some operations were significantly better for one format over another, what could indicate that, if the where more workers, a format could surpass another in terms of performance.

Although this work defined a set of experiments that were idealized based on the characteristics of usual scenarios related to the usage of data for analytical purposes, there are variations and improvements that still can be explored in future work. One of the possible variations for the experiments could be related to which query engine to use, for example. Engines such as Presto or Trino could be used, analyzing what is the impact of using different implementations of the APIs that manipulate the table formats on different engines, obtaining more information on how they behave in terms of performance.

Another possible variation could be related to volume of data that was used in the experiments. Once the scale of the datasets grows significantly, some formats could

demonstrate a different behavior when comparing the performance. The same applies to the number of workers used in the experiments, as stated earlier.

The evaluation of performance using different Cloud Object Storage could be a possible improvement as well, given that the object storage features of one cloud vendor could benefit one table format over another, in terms of performance.

Variations and adjustments on the table formats' properties and configurations are also relevant, given that this work only compares the performance of tables defined using the copy on write approach, and Apache Parquet files in the "data layer". The merge on read table types could be appropriate for streaming ingestion workloads, for example, which could be another possible scenario to replicate in an experiment.

There still is a lot to explore when comparing the performance of these table formats, which could benefit from robust analysis for implementing improvements. New work could also be important for the idealization of new formats, that could implement new approaches for improving the performance of the existent ones.

REFERENCES

ABADI, D. J.; BONCZ, P. A.; HARIZOPOULOS, S. **Column-Oriented Database Systems**. 2009. Available from Internet: <https://www.cs.umd.edu/~abadi/talks/Column_Store_Tutorial_VLDB09.pdf>.

ACKERMAN, H.; KING, J. **Operationalizing the Data Lake**. [S.l.]: O'Reilly Media, Inc., 2019.

APACHE. **Apache Iceberg Documentation**. Apache Software Foundation, 2022. Available from Internet: <https://iceberg.apache.org/docs/>.

APACHE. **Apache Iceberg Specification**. Apache Software Foundation, 2022. Available from Internet: <https://iceberg.apache.org/spec/>.

APACHE. **Apache ORC Specification**. Apache Software Foundation, 2022. Available from Internet: <https://orc.apache.org/docs/index.html>.

APACHE. **Overview: Apache hudi**. Apache Software Foundation, 2022. Available from Internet: <https://hudi.apache.org/docs/overview/>.

APACHE. **Table & Query Types | Apache Hudi**. Apache Software Foundation, 2022. Available from Internet: <https://hudi.apache.org/docs/next/table_types/>.

ARMBRUST, M. et al. Delta lake: high-performance acid table storage over cloud object stores. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 13, n. 12, p. 3411–3424, 2020.

ARMBRUST, M. et al. Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In: **Proceedings of CIDR**. [S.l.: s.n.], 2021.

ARMBRUST, M. et al. Spark sql: Relational data processing in spark. In: **Proceedings of the 2015 ACM SIGMOD international conference on management of data**. [S.l.: s.n.], 2015. p. 1383–1394.

CHEN, Y. et al. A study of sql-on-hadoop systems. In: SPRINGER. **Workshop on big data benchmarks, performance optimization, and emerging hardware**. [S.l.], 2014. p. 154–166.

DATABRICKS. **What is SPARK SQL?** Databricks, 2022. Available from Internet: <https://databricks.com/glossary/what-is-spark-sql>.

DEVLIN, B. A.; MURPHY, P. T. An architecture for a business and information system. **IBM systems Journal**, IBM, v. 27, n. 1, p. 60–80, 1988.

DIXON, J. **Pentaho, Hadoop, and Data Lakes**. 2010. Available from Internet: <https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/>.

FACTOR, M. et al. Object storage: The future building block for storage systems. In: IEEE. **2005 IEEE International Symposium on Mass Storage Systems and Technology**. [S.l.], 2005. p. 119–123.

FLORATOU, A.; MINHAS, U. F.; ÖZCAN, F. Sql-on-hadoop: Full circle back to shared-nothing database architectures. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 7, n. 12, p. 1295–1306, 2014.

INMON, B. **Data Lake Architecture: Designing the Data Lake and avoiding the garbage dump**. [S.l.]: Technics publications, 2016.

INMON, W. H. **Building the data warehouse**. [S.l.: s.n.], 1990.

IVANOV, T.; 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**, Wiley Online Library, v. 32, n. 5, p. e5523, 2020.

KIMBALL, R. **The Data Warehouse Toolkit (1st edition)**. [S.l.: s.n.], 1996.

KIMBALL, R.; ROSS, M. **The data warehouse toolkit: the complete guide to dimensional modeling**. [S.l.]: John Wiley & Sons, 2011.

MELL, P.; GRANCE, T. et al. The nist definition of cloud computing. Computer Security Division, Information Technology Laboratory, National . . . , 2011.

SAWADOGO, P.; DARMONT, J. On data lake architectures and metadata management. **Journal of Intelligent Information Systems**, Springer, v. 56, n. 1, p. 97–120, 2021.

SETHI, R. et al. Presto: Sql on everything. In: IEEE. **2019 IEEE 35th International Conference on Data Engineering (ICDE)**. [S.l.], 2019. p. 1802–1813.

SPECIFICATION, S.; TPC, T. P. P. C. Tpc benchmark™ ds. 2000.

VERTICA. **Improving Query Performance**. Micro Focus, 2022. Available from Internet: <https://www.vertica.com/docs/10.0.x/HTML/Content/Authoring/ExternalTables/QueryPerformance.htm>.

VOHRA, D. **Practical Hadoop ecosystem: a definitive guide to hadoop-related frameworks and tools**. [S.l.]: Apress, 2016.

APPENDIX A — RESULT PLOTS - TPC-DS INSPIRED

Figure A.1: Average Load Duration for each table (using 1 worker node)

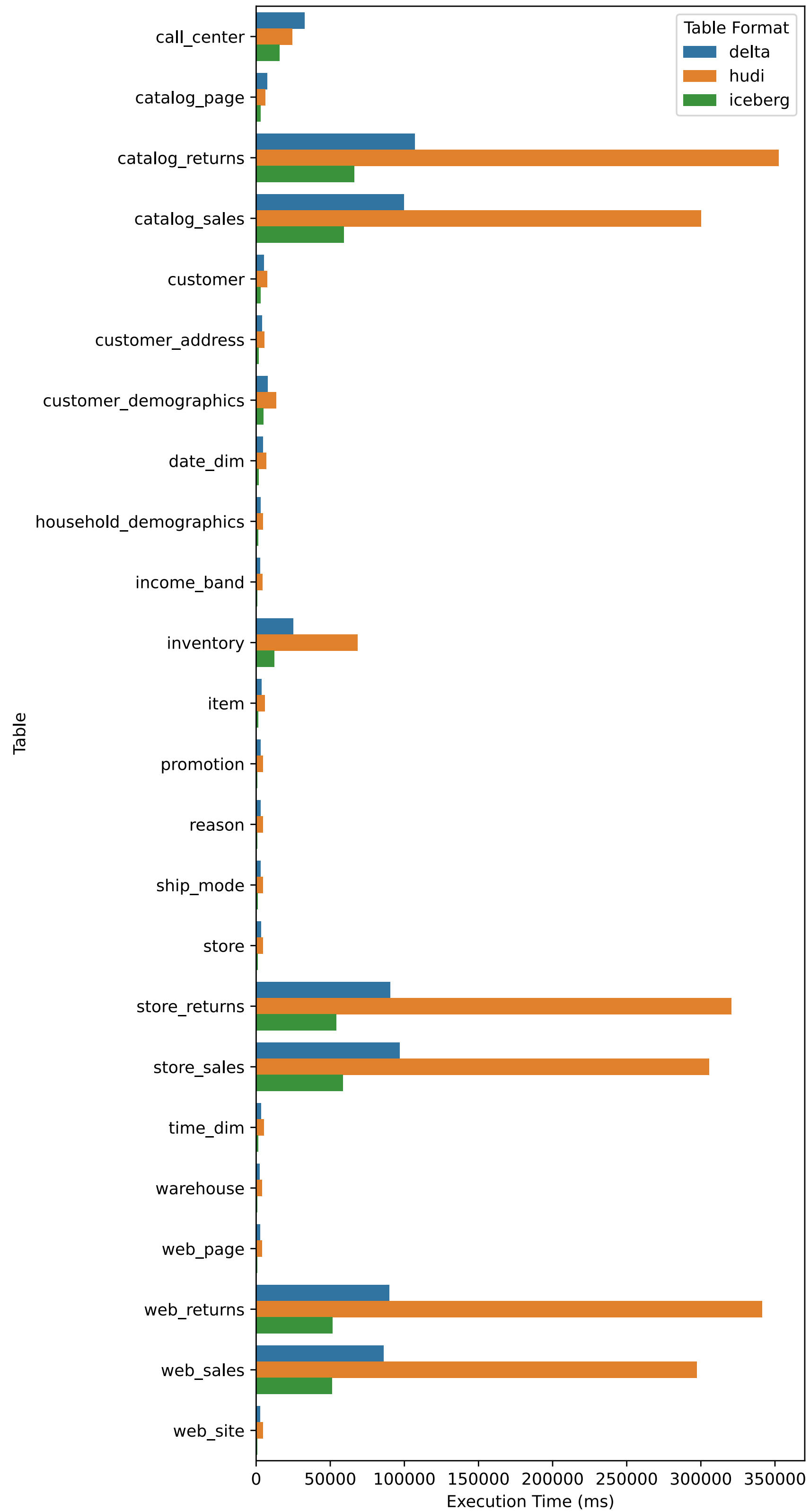


Figure A.2: Average Load Duration for each table (using 2 worker nodes)

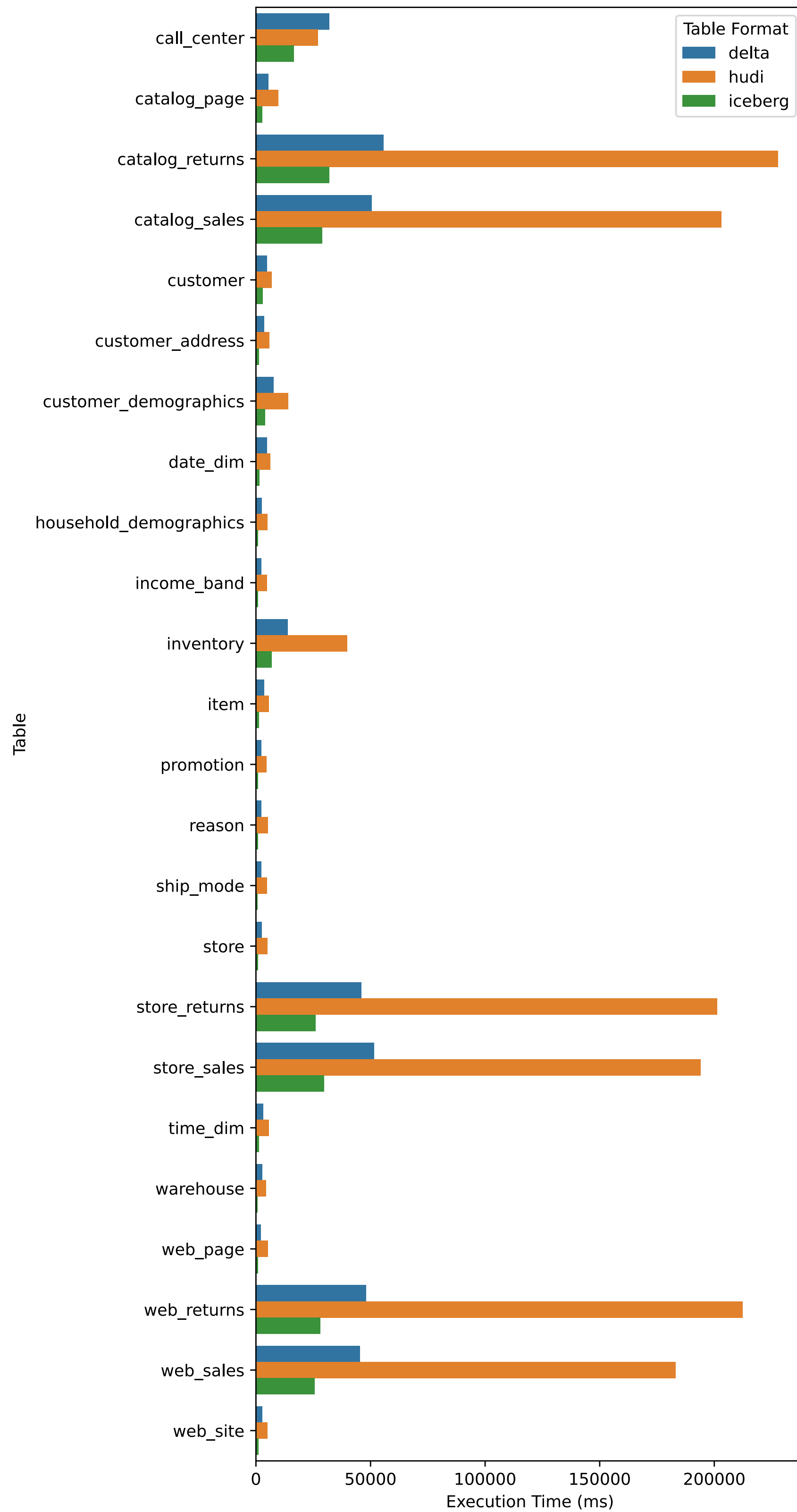


Figure A.3: Average Load Duration for each table (using 3 worker nodes)

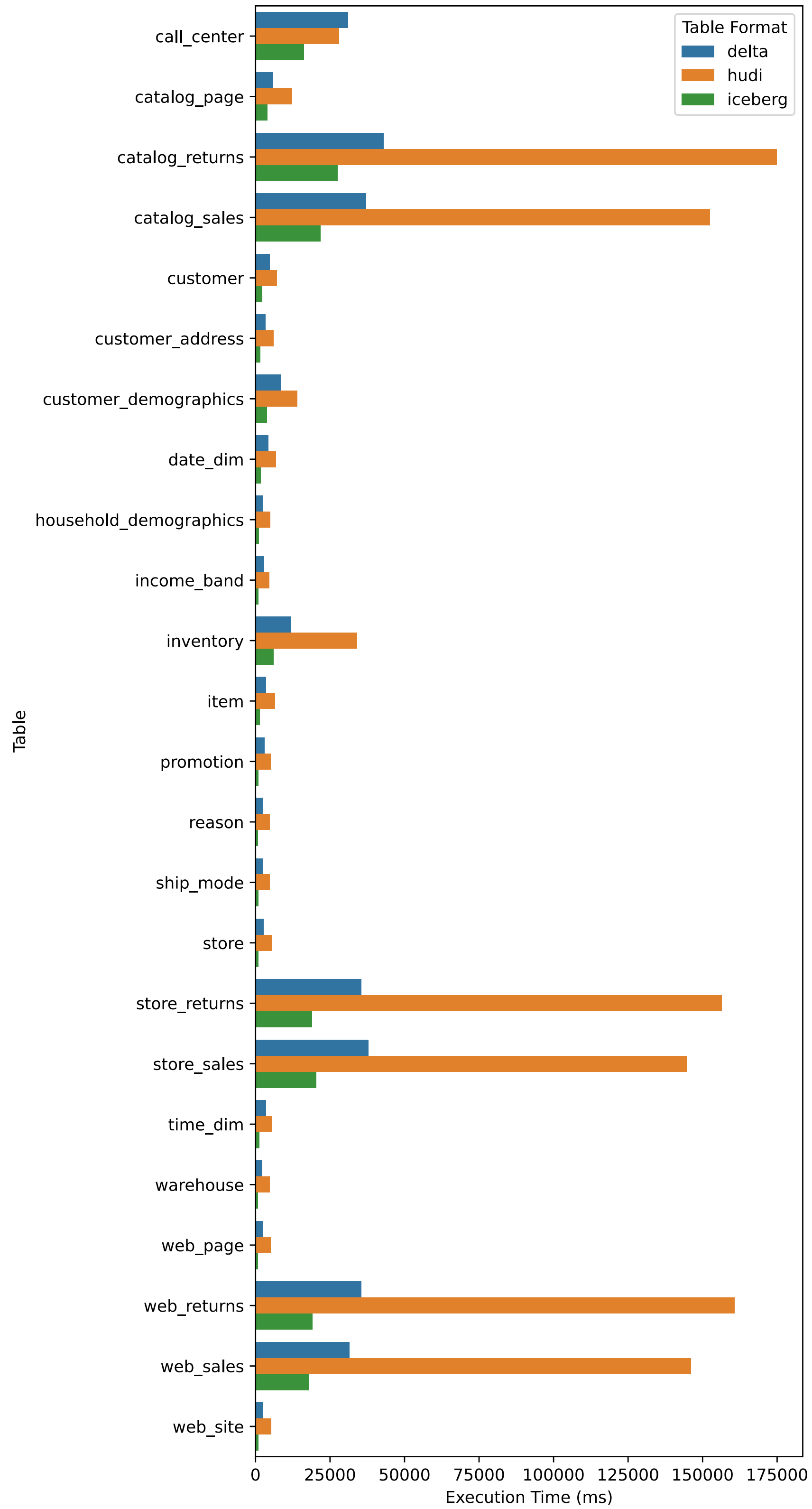


Figure A.4: Average Query Duration (using 1 worker node)

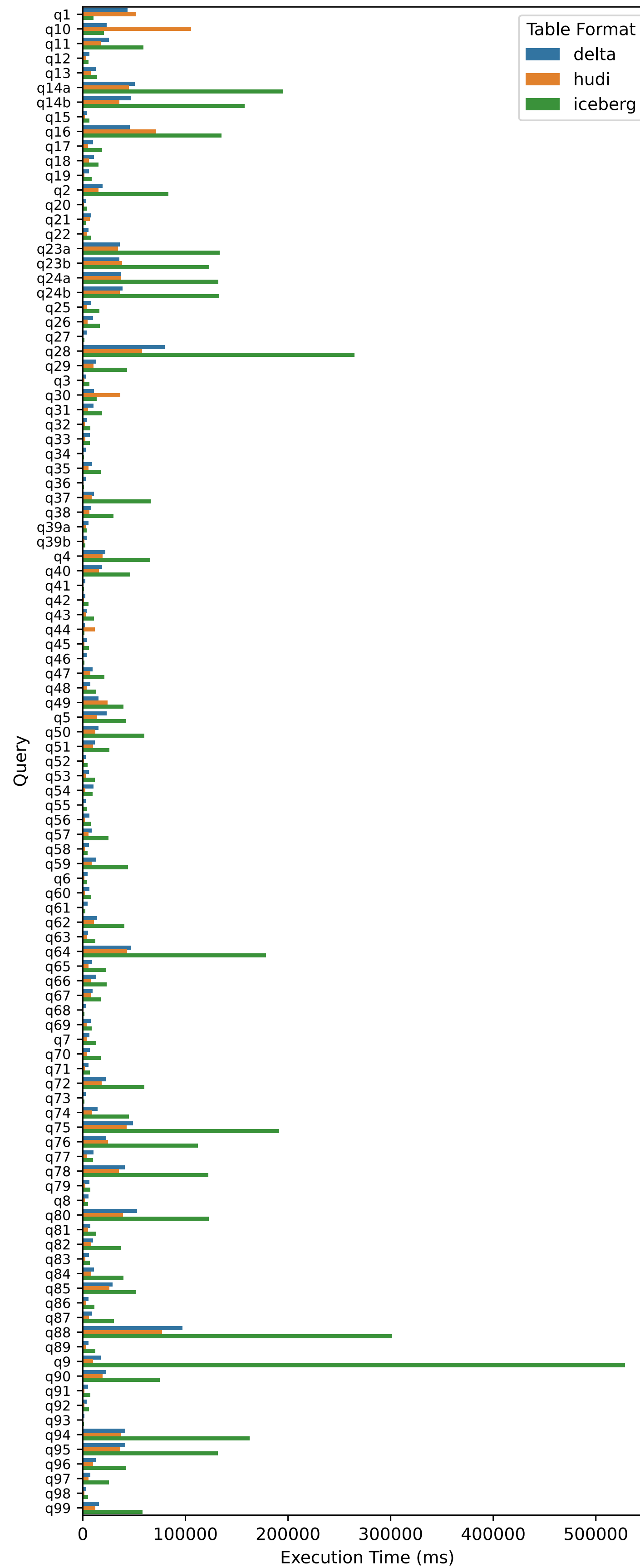


Figure A.5: Average Query Duration (using 2 worker nodes)

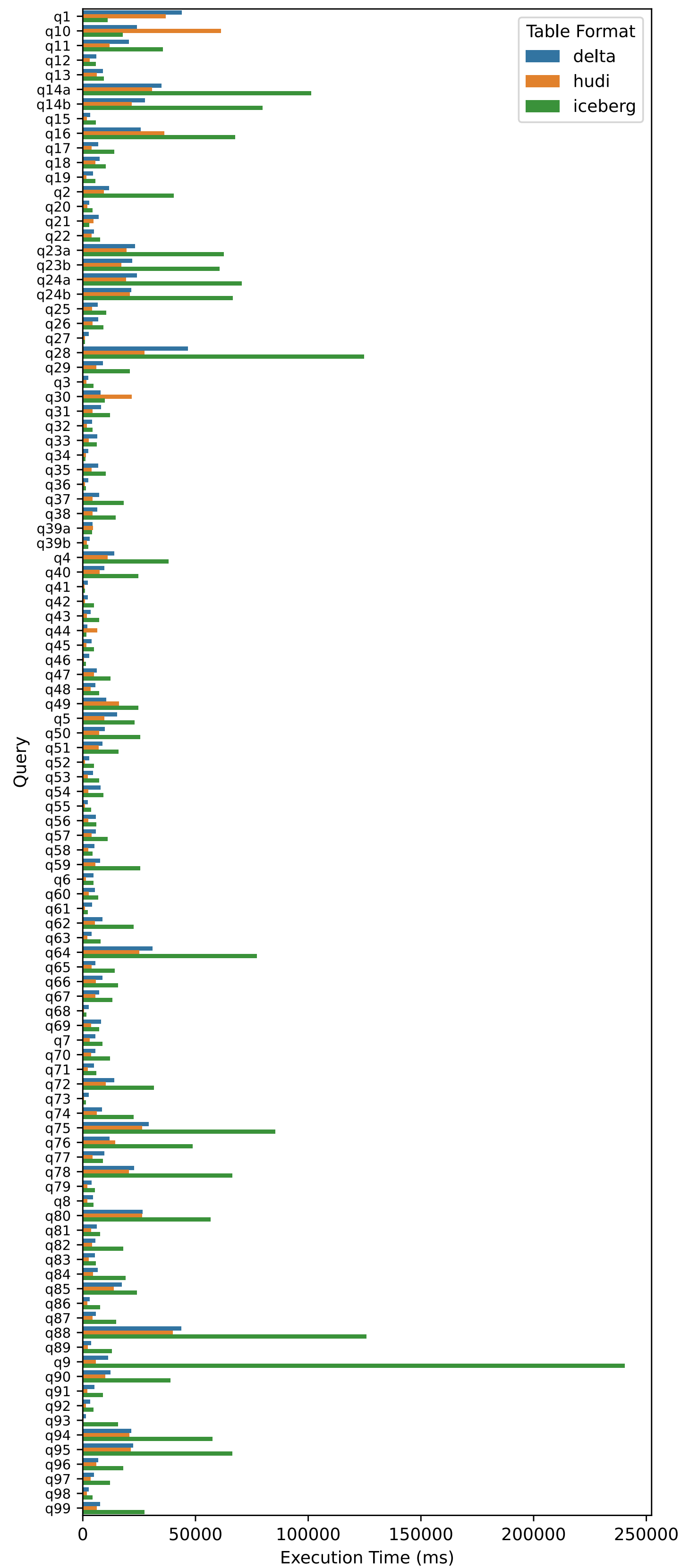
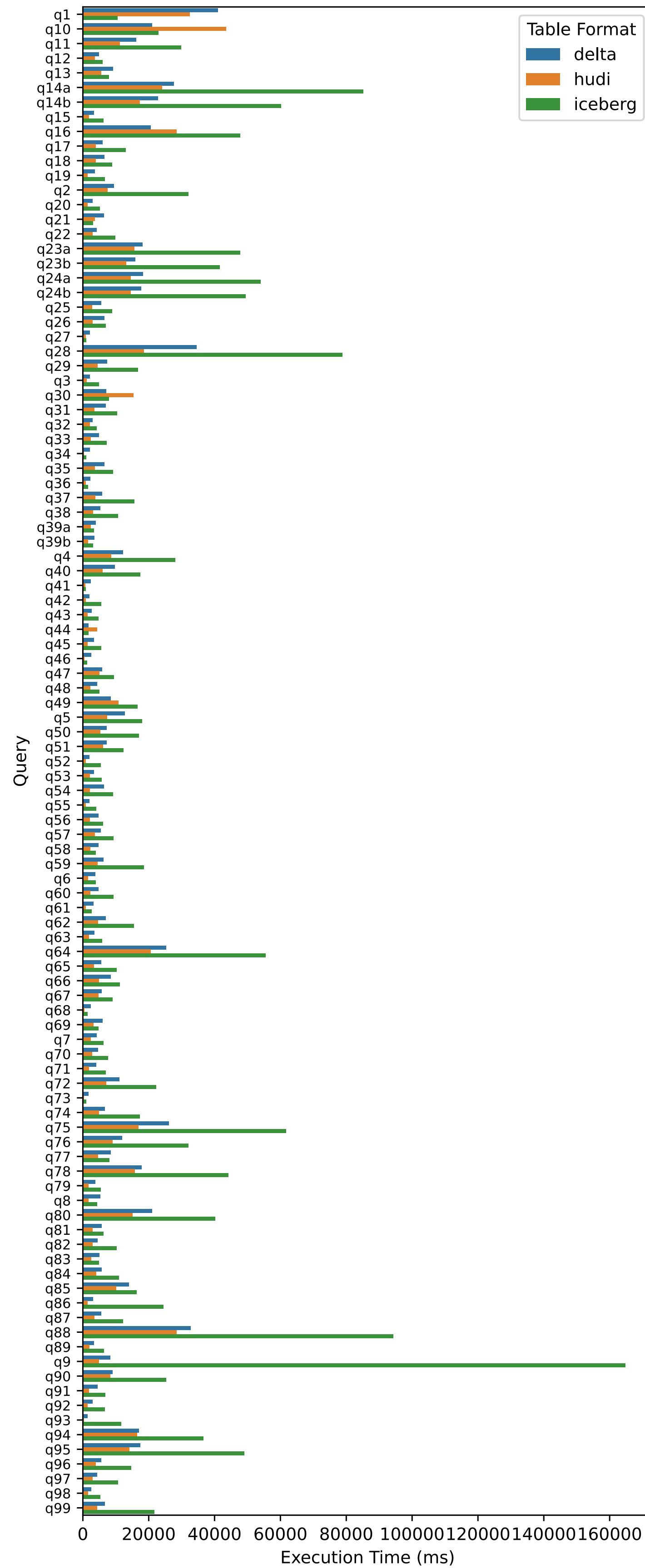


Figure A.6: Average Query Duration (using 3 worker nodes)



APPENDIX B — RESULT PLOTS - LOAD, UPDATE AND READ

Figure B.1: Average Execution Duration by Query (using 1 worker node)

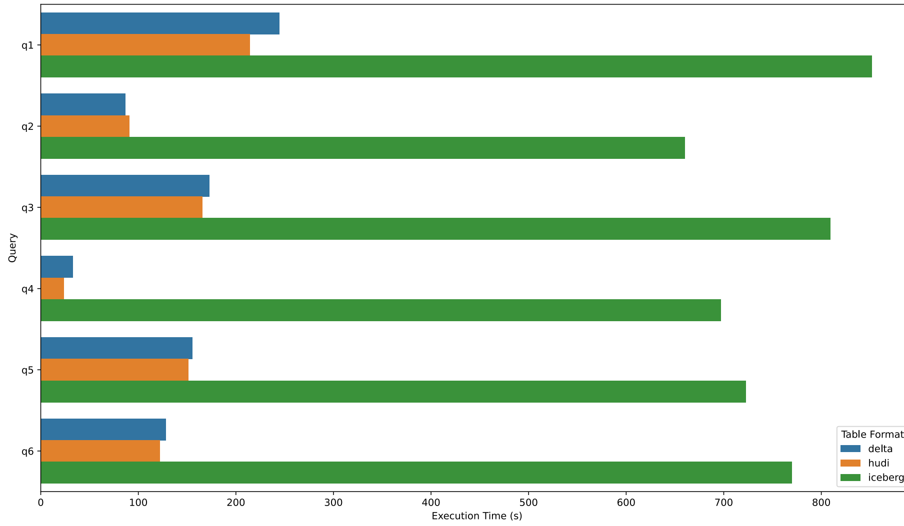


Figure B.2: Average Execution Duration by Query (using 2 worker nodes)

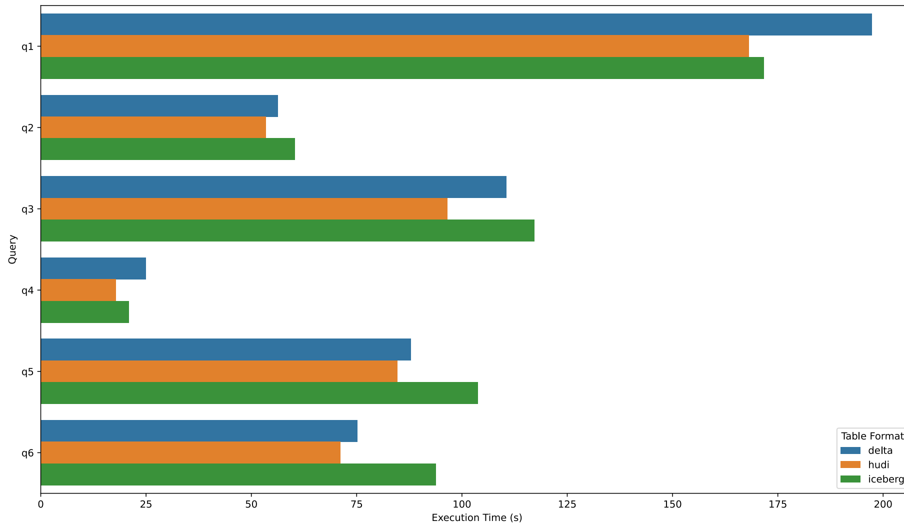


Figure B.3: Average Execution Duration by Query (using 3 worker nodes)