Study and Comparison of Data Lakehouse Systems

Fredrik Hellman 2101171
Master's thesis
Supervisor: Andreas Lundell
Faculty of Science and Engineering
Information Technology, Vaasa
Åbo Akademi University
2023

ABSTRACT

This thesis presents a comprehensive study and comparative analysis of three distinct data lakehouse systems: Delta Lake, Apache Iceberg, and Apache Hudi. Data lakehouse systems are an emergent concept that combines the capabilities of data warehouses and data lakes to provide a unified platform for large-scale data management and analysis.

Three experimental scenarios were conducted focusing on data ingestion, query performance, and scaling, each assessing a different aspect of the system’s capabilities. The results show that each data lakehouse system possesses unique strengths and weaknesses: Apache Iceberg demonstrated the best data ingestion speed, Delta Lake exhibited consistent performance across all testing scenarios, while Apache Hudi excelled with smaller datasets.

Furthermore, the study also considered the ease of implementation and use for each system. Apache Iceberg emerged as the most user-friendly, with comprehensive documentation. Delta Lake provided a slightly steeper learning curve, while Apache Hudi was the most challenging to implement.

This study underscores the promising potential of data lakehouses as alternatives to traditional database architectures. However, further research is necessary to solidify the positioning of data lakehouses as the new generation of database architectures.

ABSTRAKT

Denna avhandling presenterar en omfattande studie och jämförelse av tre datasjöhus-system: Delta Lake, Apache Iceberg och Apache Hudi. Datasjöhus är ett nytt koncept som kombinerar fördelarna med traditionella datalager och datasjöar, vilket skapar en samlad plattform för lagring, hantering och analys av stora mängder data.

I avhandlingen genomförs tre testscenarier med fokus på datainförsel, frågeprestanda och skalbarhet. Resultaten visar att varje system har sina specifika styrkor och svagheter: Apache Iceberg var snabbast på datainförsel, Delta Lake uppvisade jämn prestanda i alla test, och Apache Hudi presterade bäst med mindre datamängder.

Avhandlingen beaktade även användarvänligheten hos varje datasjöhus. Apache Iceberg var lättast att använda, följt av Delta Lake som var något mer komplex, medan Apache Hudi var mest utmanande att sätta upp.

Resultaten från denna studie hävdar att datasjöhus kan bli ett lovande alternativ till traditionella databasarkitekturer. Vidare forskning krävs för att fastställa huruvida datasjöhus verkligen kommer att bli den nya generationens databasarkitektur.

TABLE OF CONTENTS

1 Introduction 1
2 Background 4
2.1 Data warehouses 4
2.2 Data lakes 6
2.3 Data lakehouses 8
3 Data Lakehouse Systems 13
3.1 Delta Lake 13
3.1.1 Architecture 13
3.2 Apache Iceberg 15
3.2.1 Architecture 15
3.3 Apache Hudi 16
3.3.1 Architecture 16
3.4 Apache Spark 18
3.5 Columnar file formats 19
3.6 Databricks 20
4 Previous studies 21
4.1 Begoli et al. (2021) 21
4.2 Park et al. (2023) 22
4.3 Onehouse.ai (2023) 23
4.4 Conclusion 24
5 Data 25
6 Method 27
6.1 Data ingestion 28
6.2 Query performance 28
6.3 Scaling 29
7 Results 31
7.1 Data ingestion 31
7.2 Query performance 32
7.3 Scaling 32
7.3.1 A quarter of the dataset 33
7.3.2 Half the dataset 33
7.3.3 Full dataset and dummy data 33
7.4 Summary of the results 34
8 Conclusions 37
8.1 Limitations 39

8.2 Suggestions for further research 40
9 Svensk sammanfattning 41
9.1 Inledning 41
9.2 Datasöjhus 42
9.3 Testdata 44
9.4 Metod 44
9.5 Resultat 45
9.6 Diskussion 46

APPENDIX

Appendix 1 Delta Table conversion 51
Appendix 2 Appendix Iceberg table conversion 53
Appendix 3 Appendix Hudi table conversion 55
Appendix 4 SQL-Query 57
Appendix 5 Delta Lake Dummy rows 58
Appendix 6 Delta Lake creating scaled datasets 60

TABLES

Table 1 Key differences between data warehouses and data lakes 6
Table 2 Differences between database architectures 12
Table 3 Data lakehouse systems properties 20
Table 4 Differences between Delta Lake, Apache Iceberg, and Apache Hudi 24
Table 5 Data ingestion test result 31
Table 6 Query performance 32
Table 7 Query performance for a quarter of the dataset 33
Table 8 Query performance for ½ of the dataset 33
Table 9 Write speed 34
Table 10 Query performance for the full dataset including dummy data 34

FIGURES

Figure 1 Detailed ETL process (Reddy et al., 2010) 6
Figure 2 Data lakehouse metadata layer (Armbrust et al., 2021) 10
Figure 3 Differences between database architectures (Armbrust et al., 2021) 11
Figure 4 Data gathering setup (Morariu et al., 2020) 26
Figure 5 Data layout 27
Figure 6 Query for calculating the mean and standard deviation for each second 28
Figure 7 Query for normalizing the values and calculating the standard score 29

1 INTRODUCTION

One of the most important assets of any organization is its information. Data warehousing has traditionally been helping business leaders and companies to gain analytical insights by collecting data from multiple databases into centralized warehouses which could later be used as decision support and business intelligence (BI) (Ponniah, 2001; Kimball & Ross, 2013). Data warehouses are designed for analytical processing and provide a centralized, consistent view of the data, which makes it easier to access and analyze. The data in these warehouses was traditionally written using schema-on-write which would ensure that the model was optimized for BI (Stonebraker et al., 1998). However, due to problems with costs, user load, and not being able to store video, audio, or text, another generation of data analytics platforms began growing. Further, data warehouses can be rigid and inflexible, making it difficult to handle unstructured data or changing data requirements. This time, data analytics platforms started inserting all their unstructured, raw data into *data lakes* (Armburst et al., 2021).

Data lakes are low-cost storage systems that can hold data in open file formats using a file Application Programming Interface (API). Further, data lakes are schema-on-read architecture that enable agile storage of data at a low cost. Data lakes are designed for storing and processing large amounts of raw, unstructured data and provide a scalable and cost-effective solution for storing big data (Nikulchev et al., 2021).

A *data lakehouse* is defined as a data management system based on low-cost and directly accessible storage. A data lakehouse provides traditional analytical Database Management System (DBMS) management and performance. Data lakehouses combine the strengths of data warehouses and data lakes, providing a unified platform for storing, managing, and analyzing large-scale data (Armbrust et al., 2021). This allows for real-time analytics and decision-making, including support for batch processing. Further data lakehouses can handle structured and unstructured data, which will be providing more flexibility in terms of data types. Eventually, data lakehouses will enable a more agile approach to data management, as data will be quickly available for analysis using machine learning (ML), SQL-queries, or business intelligence (BI) (Armbrust et al., 2021). However, there are also challenges with data lakehouses. As data lakehouses are a new type of architecture, it can be complex to implement and manage. It is very important that a data lakehouse is implemented correctly, as it otherwise can lead to data inconsistencies and duplication (Begoli et al., 2021).

Data lakehouses were introduced in 2020 and are therefore still a relatively new concept. As the technology is still quite immature, it is difficult to say whether it will live up to its promises (Begoli et al., 2021). Because of the new coined concept of data lakehouses, there are not many studies comparing the performance of a data lakehouse compared to current database architectures. This thesis aims to shine more light on the concept of data lakehouses and analyze in what scenarios the performance of a data lakehouse can compete against other types of database architectures. The purpose of this study is thus to study and compare different data lakehouse systems.

This thesis aims to provide a comprehensive comparison of three different data lakehouse solutions: Delta Lake, Apache Iceberg and Apache Hudi. The thesis focuses on the architectural differences, features, and use cases of these three systems in order to understand their respective suitability for various data management and processing scenarios. Further, the thesis will evaluate the performance, scalability, readability, and ease of use of these solutions in a comparative manner. A detailed literature review, a comprehensive analysis of each system and a performance evaluation will be done, which will provide insights into the strengths and weaknesses of each data lakehouse solution. It is important to notice that this thesis is merely a comparison of three different data lakehouse systems and should not be viewed as a detailed benchmarking of the systems.

The study is limited to the comparison of Delta Lake, Apache Iceberg, and Apache Hudi as data lakehouse solutions. While there are other data management and processing systems available today, these three solutions have gained significant traction in the data engineering community and are considered to be among the frontrunners in the data lakehouse domain (Databricks, 2019; Databricks, 2021).

The thesis will be presented as follows – Chapter 2 will thoroughly go through the concepts used in the thesis. A comparison between the three different data lakehouse systems will be discussed in Chapter 3. Previous studies will be discussed in Chapter 4, and the rest of the thesis will include a test between the different data lakehouse systems. Chapter 5 will present the data used in the comparison, Chapter 6 will go through the methodology and Chapter 7 will discuss the results. The thesis concludes with a discussion chapter and a summary, including suggestions for further research.

2 BACKGROUND

2.1 Data warehouses

Since the late 1970s, relational database architectures have been implemented by most organizations to store data (Sangupamba et al., 2014). However, in today's world the needs for information storage are not the same as they used to be. Organization managers need to be properly informed to take appropriate decisions, and traditional database systems are not suitable for these requirements, as they were historically implemented to support day-to-day operations instead of data analysis and decision makings. This led to new database technologies emerging in the 1990s, namely data warehousing and Online Analytical Processing (OLAP), which include architectures, algorithms, and techniques to bring together data from heterogenous information sources into a single repository suited for analysis (Sangupamba et al., 2014; Chaudhuri & Dayal, 1997).

A data warehouse is a repository for historical and consistent data. It is usually equipped with such tools that the warehouse can easily provide the opportunity to extract reliable information to be used for the user's needs. Inmon (1992) defines data warehouse as "*a subject oriented, integrated, time-variant, and nonvolatile collection of data to support decision making.*"

Data warehousing is a concept that originally was founded from the urgent need to use large amounts of electronic data to complete tasks that go beyond routine tasks linked to daily processing. Companies used to have databases that were stored with detailed data on the tasks performed by branches, and queries could be issued to retrieve the data in question. Data warehousing includes processes that can extract relevant data from a database, transform the data, remove inconsistencies, store the data in a data warehouse and provide the user with data for data analysis or query results. Data warehouses are currently well established in many organizations, and the concept is widely known to provide users with easy access to key information that would otherwise be hard to find (Sangupamba, 2014).

However, for this process to work, database administrators must first provide an SQL-query after closely analyzing the database catalog. The processing time for a query could be up to a few hours, and such an approach was clearly not feasible because of the time and resources needed (Inmon, 1992).

Typically, a data warehouse is not maintained together with an organization's operational databases (Kimball & Ross, 2013). The reason for this is that data warehouses support OLAP, while operational databases support Online Transaction Processing (OLTP) applications. The main difference between OLTP and OLAP is that OLTP focuses on managing real-time transactional data and ensuring fast operations while OLAP is designed for efficient analysis and processing of complex queries involving aggregated, historical data. OLTP applications are used to automate the day-to-day operations of an organization. These tasks are repetitive and consist of short, isolated transactions. These transactions require detailed up-to-date data and are usually records accessed on their primary keys. This leads to operational databases increasing in size, while consistency and recoverability are the most critical part of maintaining a database.

Data warehouses, on the other hand, are targeted for decision support. Data warehouses are different from operational databases in many ways, but one way is that data warehouses support OLAP. In a data warehouse, historical, summarized data is more important than individual records. Data warehouses tend to be larger than operational databases, as data warehouses contain consolidated data from several operational databases. The workloads are query intensive with complex queries that can access millions of records. To facilitate complex analyses, the data in a data warehouse is modeled multidimensionally and hierarchically (Chaudhuri & Dayal, 1997).

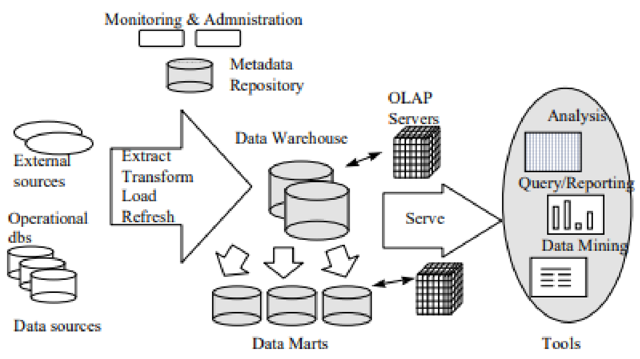
A data warehouse architecture contains historical and commutative data from multiple sources, and there are three main architectures of data warehouses: These are *single-tier architecture, two-tier architecture,* and *three-tier architecture* (Ponniah, 2001). In a single-tier architecture, the single-layer model minimizes the amount of data stored. This helps with data redundancy, but it is not frequently used in practice as it lacks a component that separates analytical and transactional processing. The two-tier architecture separates physically available sources, which makes sure that all data loaded into the warehouse is in the appropriate format. However, this architecture cannot support many end users and has problems due to network limitations. The three-tier architecture is the most widely used architecture for data warehouses and consists of a top, middle, and bottom tier. In the bottom tier, the data is cleansed. The middle tier is an OLAP server that acts as a mediator between the user and the database, and the top tier is a front-end client layer that consists of the tools and an API that is used to get data from the warehouse (Ponniah, 2001). The key differences between data warehouses and data lakes are displayed in Table 1.

Table 1 Key differences between data warehouses and data lakes

Data warehouse	Data lake
Schema on write	Schema on read
Fast results	Slower results
Not Agile	Very agile
Structured	Unstructured
SQL	No SQL
ETL	ELT

Data warehouses use the Extract-Transformation-Loading (ETL) process. ETL is a data integration process that can combine data from multiple sources into a consistent data store. This is done by first extracting the data from different data sources, where each data source has its distinct set of characteristics. The ETL process needs to integrate different systems with different operating systems, different database management systems, and different communication protocols. The second step is the data transformation, where the data is cleaned and changed to output correct, and consistent data. Finally, the data is loaded into a multidimensional structure in the last step of the ETL process (Ali El-Sappagh et al., 2021). The whole ETL process is displayed in Figure 1.

Figure 1 Detailed ETL process (Reddy et al., 2010)



2.2 Data lakes

In the beginning of the 21st century, new types of diverse data were emerging all over the internet, such as real-time streaming data, web-based business transactions, and sensor data. Because of this large amount of data, the need to have better solutions for storage and analysis of large amounts of semi-structured and unstructured data became apparent. This need led to a new type of data management scheme known as *data lakes* (Nambiar & Mundra, 2021).

Data lakes are centralized storage repositories, which help the end user to make better business decisions from visualizations or dashboards using big data analysis or ML. The term data lake was first introduced by James Dixon in 2010, to address the shortcomings of data warehouses. Dixon (2010) states that "*whilst a data warehouse seems to be a bottle of water cleaned and ready for consumption, then data lake is considered as a whole lake of data in a more natural state*". In a data lake, the data is stored in its original raw form. This leads to the data being able to vary drastically in both size and structure, and they also lack any specific organizational structure, but the essence is that data would be stored in its original raw state (Nambiar & Mundra, 2021). This would lead to the data being easy to manage and data could be stored in many different formats and still be accessible for different use cases (Dixon, 2010).

Data lakes are different from data warehouses in many ways. One of the major differences is the different structure. As previously mentioned, a data warehouse stores data in a processed and filtered form, whilst a data lake stores raw or unprocessed data. In a data warehouse, data is organized into a specific single schema before being put into the warehouse, while the raw data is inserted directly into a data lake. Analysis can be performed on the cleaned data in a warehouse, while data is selected and organized when needed in a data lake (Nambiar & Mundra, 2021).

Armbrust et al. (2021) argue that data warehouses and data lakes suffer from four problems: *reliability, data staleness, limited support for advanced analytics,* and a *total cost of ownership*. It is costly for organizations to keep data lakes and warehouses reliable and consistent. Continuous engineering is required between the two systems for them to provide support for BI. Data staleness is worse for warehouses compared to data lakes, which is a step back compared to the first-generation analytic systems. Further, these architectures are not very compatible with advanced analytics such as ML. None of the leading systems within ML, such as TensorFlow and PyTorch work well together with both data warehouses and data lakes. ML systems often need to process large datasets without using complex SQL code, and with current architectures, there is no direct way to access the internal formats. Finally, Armbrust et al. (2021) name the total cost of ownership as an issue. Organizations pay double the storage cost if the data needs to be copied to a warehouse, suffer from continuous ETL costs, and are locked into proprietary formats, as it might be challenging to migrate data to other systems.

Khine & Wang (2018) argue that in contrast to the traditional idea of ETL of data warehouses, data lakes change the order in which it processes the data. The data is stored in its original form, and the preprocessing step will not be done until the data is required by the application. Therefore, the original ETL rules are broken down and changed to ELT. Data lakes have no predefined data schema, so when data is extracted from the source, the required metadata is added. Approaching it in this manner enables the data lake to handle both a write-heavy and a read-heavy workload. The data is not transformed until the application calls from it, but only when the data is required and called for query is it transformed into the appropriate form using the metadata added earlier on. This avoids expensive data pre-transformation in data lakes, which is called the schema-on-read approach.

There are also disadvantages with data lakes, mostly related to metadata management. Data lakes contain substantial integration of various data sources, and metadata management requirements are therefore necessary, especially for raw data. As data lakes may contain many different types of data, it has been challenging to develop a metadata catalog to guarantee the respect of data governance and properly manage metadata (Madera & Laurent, 2016). Further, security controls have raised issues for data lakes. Because of the many types of data stored in a data lake, it is difficult to implement the proper data security for each data format when there is no way of knowing which data types will be stored in the data lake. Mehmood et al. (2019) state that the only way of securing a data lake is by providing access only to whitelisted IP addresses. In that way, there would always be some security control in place and who could access the data lake would be controlled. However, there is no way of controlling that data stored in the data lake does not contain low quality or unreliable data. If the data is not properly managed, the data runs a risk of becoming corrupt, or the data lake being accessed by unauthorized users, which further puts quality and reliability at risk.

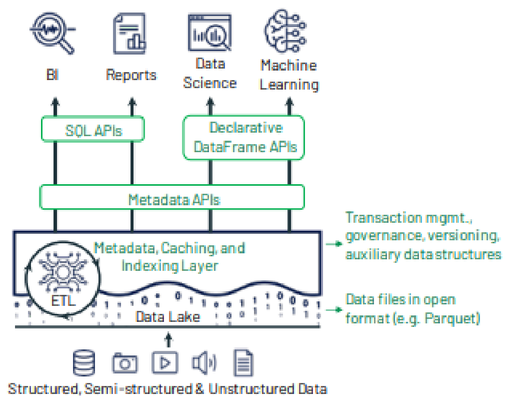
2.3 Data lakehouses

Data warehouses can be seen as the first-generation data analytics platform. However, around a decade ago these systems started to face challenges related to storage and costs, and because datasets were becoming more unstructured, data warehouses could not store and query the data at all. To solve these problems, data lakes were introduced as the second-generation data analytics platform. This was very helpful as raw data could be inserted directly into the data lakes. The data lake was a schema-on-read architecture that gave the user the opportunity to store data at low cost, and at the same time eliminating the problems with quality and governance. Data lakes eventually grew into cloud data lakes, which are used in many organizations today. However, in today's world organizations face challenges with both architectures as new enterprise use cases include new types of analytics, such as machine learning (ML). A data lakehouse combines the key benefits of data warehouses and data lakes, whilst providing the user with a more flexible, cost-effective and scalable solution for managing and analyzing large amounts of data. A data lakehouse offers better data governance, improved performance, and the ability to handle both structured and unstructured data. It is based on low-cost and directly accessible storage, and provides traditional analytical DBMS management and performance features such as atomicity, consistency, isolation, durability (ACID) transactions, data versioning, auditing, and query optimization, to name a few. Armburst et al. (2021) argue that data lakehouses are an especially good fit for cloud environments with separate compute and storage: different applications can run on-demand on separate computing nodes, whilst accessing the same storage data.

A data lakehouse's metadata layer is an essential component for enabling a lakehouse. The metadata layer provides management features such as transactions, time travel, and data quality enforcement, which makes them useful for organizations that are already using a data lake. Further, metadata layers naturally implement governance features such as access control and audit logging. The metadata layer is displayed in Figure 2.

However, there are still several directions for future development in this area. One potential direction for future development is to improve scalability of the metadata layer. While Apache Iceberg and Delta Lake have demonstrated impressive scalability, there is still room for improvement. As organizations generate more and more data, metadata layers must be able to handle even larger data sets (Armbrust et al., 2021).

Figure 2 Data lakehouse metadata layer (Armbrust et al., 2021)


BI
Reports
Data Science
Machine Learning
SQL APIs
Declarative DataFrame APIs
Metadata APIs
Metadata, Caching, and Indexing Layer
ETL
Data Lake
Transaction mgmt., governance, versioning, auxiliary data structures
Data files in open format (e.g. Parquet)
Structured, Semi-structured & Unstructured Data

Further, Armbrust et al. (2021) argue that the time for the data lakehouse has come mostly to address a few key problems: *reliable data management, support for machine learning*, and *SQL-performance*. Firstly, the lakehouse needs to store raw data while at the same time supporting both ETL and ELT processes. Data lakes have historically managed data in a semi-structured format, which causes problems regarding key management features that simplify ELT in data warehouses, such as transactions and table rollbacks. New systems such as Delta Lake and Apache Iceberg have been created to address these issues by providing transactional views of a data lake while enabling for these key features.

Secondly, previous database architectures do not provide good support for ML. However, as ML systems have adopted data frames as the main way of manipulating and handling data, the declarative data frame APIs that enable query optimizations will enable ML workloads to directly be able to use data lakehouses. Finally, in today's world data warehouses are critical for most business processes, but are often associated with problems regarding cost, staleness, and incorrect data. This is mostly due to the way enterprise data platforms are designed and could be eliminated by implementing a data lakehouse architecture (Armbrust et al., 2021).

Kutay (2021) argues that data lakehouses will reduce the level of data redundancy. Current database architectures can incorrectly provide redundant data because of deduplication or by only storing relevant data. The data lakehouse will reduce data duplications by providing a single repository, and by using a central all-purpose platform to handle all the data demands by the organization, duplicates can be removed. Neither will the storage of relevant data be a problem in data lakehouses. Different from a data lake, where there is no filter on the type of data that can be stored, and a data warehouse where there is a heavy filter on what data can be stored, the data lakehouse contains a restrictive filter that will remove data deduplication. The result is a database that is only storing relevant data and at the same time freeing up space by removing unneeded duplicates.

Kutay (2021) mentions that the whole concept of data lakehouses is still relatively new as it was introduced in 2020. Thus, the technology is still relatively immature, and it is still unclear whether it will live up to its promises. It might still be a few years before data lakehouses can compete with current big-data storage solutions. However, with the innovation pace we have been experiencing during the last decades, it is difficult to predict whether a new data storage solution could overtake current solutions. Table 1 displays a summary of the key differences between a data warehouse, a data lake, and a data lakehouse. The main architecture differences between data warehouses, data lakes, and data lakehouses is visualized in Figure 3.

Figure 3 Differences between database architectures (Armbrust et al., 2021)

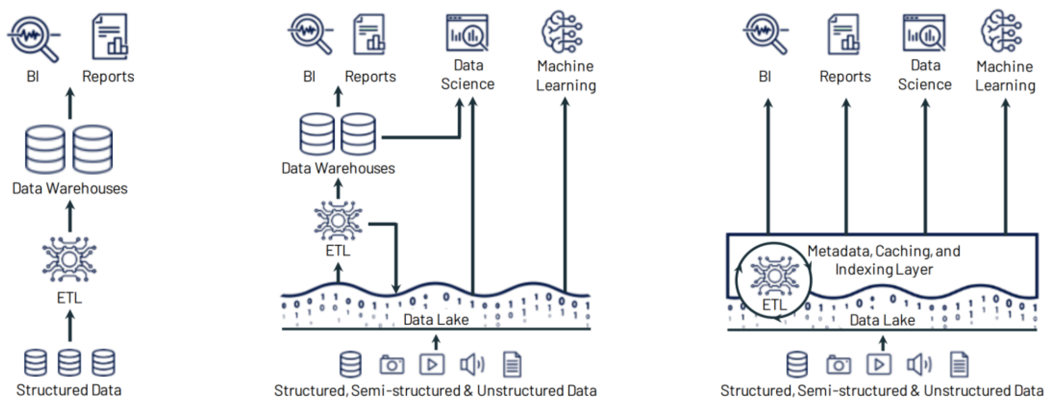


Table 2 Differences between database architectures

	Data Warehouse	**Data Lake**	**Data Lakehouse**
Data	Structured	Structured, Semi-structured, Unstructured	Structured, Semi-structured, Unstructured
Processing	OLAP (Online Analytical Processing)	Batch processing, Real-time processing	Batch processing, Real-time processing
Storage	Optimized for storage of structured data	Optimized for storage of raw data	Optimized for both storage of raw data and structured data
Schema	Schema-on-write	Schema-on-read	Hybrid approach with schema-on-read and schema-on-write options
Integration	Extract, Transform, Load (ETL)	Extract, Load, Transform (ELT)	Extract, Load, Transform (ELT)
Use Cases	Business intelligence, reporting, analytics	Data exploration, machine learning, analytics, data science	Hybrid use cases of data warehouse and data lake

3 DATA LAKEHOUSE SYSTEMS

3.1 Delta Lake

A data lakehouse typically contains various types of data in a data lake before converting it to Delta Lake format (Kutay, 2021). Delta Lake is an open-source ACID table storage layer over cloud object stores initially developed at Databricks. It uses a transaction log that is compacted into Apache Parquet format to provide ACID properties, ensuring data consistency and reliability. Built on popular cloud objects stores like Amazon S3 and Azure Data Lake Storage, Delta Lake integrates seamlessly with existing cloud infrastructures, enabling significantly faster metadata operations for large tabular datasets. It also offers automatic data layout optimization, upserts, caching, and audit logs, simplifying data management.

Delta Lake is currently in active use at thousands of organizations, where it processes exabytes of data every day. The typical data type stored in a Delta Lake includes change data capture (CDC) logs from OLTP systems, application logs, time series data, and graphs. The applications that are running over the data include SQL-workloads, BI, streaming, data science, and ML. Delta Lake is considered a good fit for most data lake applications that otherwise would have used file formats such as Parquet as structured storage formats, as well as many traditional data warehousing workloads. (Armbrust et al., 2020)

3.1.1 Architecture

Delta Lake is built on top of Apache Spark and utilizes a transaction log along with Apache Parquet files to provide ACID properties, ensuring users with access to consistent data. The architecture includes the following components:

1. **Table metadata**: Delta Lake stores metadata information about the table, such as its schema, partitioning information, and properties.

2. **Data files**: Data files store the actual table data in Apache Parquet format.

3. **Partitioning scheme**: Delta Lake uses a partitioning scheme to define how data is partitioned within the table. Partitioning improves query performance by enabling predicate pushdowns and minimizing the amount of data that needs to be read.

4. **Transaction log** (Delta Log): Delta Log keeps track of changes to the table, including new data insertions, schema updates, and table modifications. The log is stored as a series of JSON files, which are used for time-travel capabilities.

5. **Checkpoints**: Periodically, Delta Lake generates checkpoint files that summarize the current state of the transaction log. This helps with improving the performance of metadata operations and simplifies recovery.

6. **Tombstones**: When data files are deleted during operations, Delta Lake records the deletion in the transaction log as a "tombstone" entry.

Currently, Delta Lake is used by thousands of organizations to process exabytes of data daily. It replaces more complex architectures involving multiple data management systems and is accessible with an Apache 2 license. Consequently, Delta Lake presents a scalable, cost-effective, and reliable solution for storing and processing large data volumes, especially in the cloud. Its support for ACID transactions, data versioning, schema enforcement, and data integrity makes it a powerful tool for data-driven applications across various industries (Databricks, 2019;2021).

However, querying large-scale datasets in a lakehouse system also raises challenges. A data lakehouse system requires fast and efficient query performance while also supporting a wide range of SQL operations and integrations with other tools. Photon, for example, is a fast query engine for lakehouse systems, which is built on top of Apache Spark and operates on a columnar storage model, with an innovative indexing technique called Adaptive Indexing. Adaptive Indexing allows for faster query performance and improves scalability, while dynamically determining the most efficient index based on the query workload. Photon has many features to help with lakehouse queries, such as query optimization, predicate pushdown, and projection pruning, to name a few. Photon is designed to be easily integrated with Delta Lake, for a complete end-to-end data and analytics solution (Behm, 2021). Even though Photon would be a suitable query engine for testing data lakehouses, this thesis will be using the built-in notebook feature available in Databricks, which is a data analytics platform that provides a unified and collaborative workspace for data science and engineering teams (Databricks, 2021).

3.2 Apache Iceberg

Like Delta Lake, Apache Iceberg is also an open-source table format system that enables the user to create a lakehouse architecture. Iceberg was originally created by Netflix (Machado et al., 2022) to address the previous Apache Hives' limitations and consistency issues when handling large-scale tables. Apache Iceberg supports transactional and consistent data through the metadata collection in manifest lists and manifest files related to snapshots of tables. Apache Iceberg is a tabular format that is suitable for storing tables larger than a petabyte, it was originally designed from the ground up to be used in the cloud, and the main goal was to address the various data consistency and performance issues that Hive suffers from. Apache Iceberg can be used to manage large analytic spreadsheets using immutable file formats such as Parquet (Apache Iceberg, n/d).

3.2.1 Architecture

The architecture of Apache Iceberg centers on its table format, which consists of snapshot metadata files, manifest lists, manifest files, and data files. The components of an Iceberg table include:

1. **Table metadata**: Iceberg stores metadata about the table, such as schema, partitioning information, and properties in JSON format.

2. **Snapshots**: Snapshots represent the current state of an Iceberg table at a particular point in time. Each snapshot contains a set of manifest files that describe the table data.

3. **Manifest files**: A manifest file is a metadata file that describes a collection of data files in the table. Information regarding file paths, partition data, and file-level statistics is stored here.

4. **Data files**: Data files store the actual table data which can be in formats such as Apache Parquet. Data files are organized into partitions based on the table's partitioning specification.

5. **Transaction log**: Iceberg maintains a transaction log to keep track of changes to the table's metadata, snapshots, schema updates, and other table modifications.

Using Apache Iceberg, all information is stored in different files. An Iceberg table consists of a snapshot metadata file, a manifest list, a manifest file, and a data file. The use of a snapshot guarantees isolated reading and writing, which leads to the readers always being able to see a consistent version of the data as writers work in isolation without affecting the table. Apache Iceberg also ensures that schema changes are independent and have no side effects, and by implementing hidden partitioning, the system can propose an evolution in the partitioning specification. This leads to the ability to change the column that is split without breaking the table in question. Finally, Apache Iceberg is supported by Apache Spark, which means that data can be read and written using Spark DataFrames, which will be discussed later in this chapter.

3.3 Apache Hudi

Apache Hudi (Hadoop Upserts Deleted Incrementals) is a framework designed to facilitate fast upserts and deletes as well as incremental processing on top of data file systems, just like Delta Lake and Apache Iceberg. Hudi was mainly developed to manage big data storage in distributed file systems such as cloud storage and HDFS (Belov and Nikulchev, 2021). Apache Hudi prioritizes the optimization of streaming data ingestion and the capture of data changes to accelerate the ingestion of streaming data and analysis in situations where only data ingested over a certain period is required. The incremental processing feature increases query performance by only processing new data and avoiding re-processing old data. Apache Hudi adopts a directory-based data management structure, where each table is stored in a directory that comprises partitions represented by folders. These partitions are further divided into file groups, which are sliced, with each slice containing data in the Parquet format. Apache Hudi offers two types of table storage, namely copy-on-write and merge-on-read, depending on the nature of the workload. It provides several features to support the lakehouse architecture design, including fast upsert and delete functionality enabled by its index, ACID compliance, and optimistic concurrency control to handle concurrent writes. Apache Hudi also offers data layout optimization through file-level stats, rollback support, and lineage tracking (Apache Hudi, n/d).

3.3.1 Architecture

Apache Hudi organizes its tables through a directory-based structure, where each table is stored in a directory comprising partitions represented by folders. These partitions are divided into file groups that are sliced, with each slice containing data in the parquet format. The components of a Hudi table are:

1. **Table-based path**: The base directory in the distributed file system where the Hudi table is stored. The base path acts as the root folder for all the metadata associated with the table.

2. **Partitions**: A Hudi table is divided into partitions, which are logical divisions of the data based on certain column values, such as a timestamp. Partitions are represented as subdirectories within the table base path.

3. **File groups**: Partitions are further divided into file groups, where each group consists of a base file and a set of log files.

4. **Data files**: Data files store the actual table data in Parquet format.

5. **Log files**: These files store the changes made to the data, such as inserts, updates, and deletes.

6. **Timeline**: Hudi tables store metadata about each commit, which enables time-travel queries, incremental data processing, and rollback support.

7. **Index**: Hudi uses an index to enable efficient record-level operations, such as upserts and deletes.

Changes to a Apache Hudi table are achieved by either copy on write, or merge on read. Copy on write means that the data is stored in a Parquet file format, and each update crates a new version of the file. Merge on read means that the data is stored as a combination of the Parquet and Avro file formats, and updates are logged in delta files. Copy-on-write is better used for read-intensive batch downloads, while merge-on-read is better used for streaming write-intensive workloads (Jain et al., 2023).

To summarize, Delta Lake, Apache Hudi, and Apache Iceberg all extend data lakes with ACID guarantees by using Apache Spark for queries and updates. These platforms store both version and temporal information about updates to tables in a commit log, which contains the row-level changes to tables. Because of this, the user can issue time-travel queries, which is not traditionally possible in a data warehouse that typically only store the latest values for rows in tables (Jain et al., 2023).

In addition to their technical advantages, commercial support for these systems also plays a role in adoption for enterprise-level use. Delta Lake has robust commercial support by Databricks, that not only provides managed cloud services but also offers an enterprise version of Delta Lake (Databricks, 2021). Apache Iceberg enjoys contributions from technology giants like Netflix and Alibaba, while Apache Hudi can easily be integrated into Amazon EMR, for example.

3.4 Apache Spark

Apache Spark is a cluster computing system that can run batch and streaming analysis jobs on data distributed on the cluster. It is an open-source platform for large-scale data processing that contains some high-level APIs in Java, Scala, Python and R. Further, Spark provides an optimized engine that can support general execution graphs, as well as a rich set of higher-level tools such as Spark SQL, Pandas API on Spark, MLib, and GraphX. Spark was originally founded in the UC Berkeley AMPLab and open sourced in 2009. The goal was to find a solution for efficient iterative computation and since its first releases it has been packaged with ML algorithms.

Spark has emerged as the standard for big data analytics after Hadoop's MapReduce. It combines a core engine for distributed computing together with an advanced programming model. Even though it is similar to MapReduce in terms of scalability and fault tolerance, Apache Spark is much faster and easier to use. Further, Apache Spark can leverage the memory of a computing cluster to reduce the dependency on the underlying distributed file system. This leads to massive gains in performance compared to Hadoop's MapReduce. Today, Apache Spark is considered as a general-purpose engine that goes beyond batch applications to combine different types of computations, that previously would require different separated distributed systems (Salloum et al., 2016).

Apache Spark is built upon the basic RDD (Resilient Distributed Datasets) API (Application Programming Interface), which provides the system with efficient data sharing between computations. RDD is a read-only, partitioned collection of records that provide fault-tolerant, parallel data structures. Beyond this, Apache Spark has introduced several improvements for its data abstraction. One is the DataFrame API, which is in theory equivalent to a table in relational databases. The Spark DataFrame is a distributed collection of data, but organized into named columns, which provides Spark with more useful information about the data structure, which yields for extra optimizations (Salloum et al., 2016).

On a fundamental level, Apache Spark applications consists of two types of components: a driver and executors. The driver converts the code entered by the user into multiple tasks that can be distributed among the worker nodes. The executors run on those nodes, and execute the tasks assigned to them. Apache Spark can run on a standalone cluster, which only requires the Apache Spark framework and a Java Virtual Machine on each machine in the cluster (Salloum et al., 2016).

The experiments performed in this thesis uses the Databricks Unified Analytics Platform, a comprehensive managed service that offers Apache Spark clusters over a standard Apache Spark distribution. The different lakehouse systems are all built in a separate cluster in Databricks, and a separate blob container in Microsoft Azure.

3.5 Columnar file formats

Apache ORC and Apache Parquet are arguably the most popular file formats for Big Data analytics. ORC (Optimized Record Columnar File) is a self-describing, type-aware columnar file format that was originally designed for Hadoop workloads but has later been used as a general-purpose storage format. It is useful for large streaming reads, and ORC supports the complete set of data types available in Hive. The metadata in ORC is stored at the end of the file, which allows for adding new fields to the table schema without breaking the current readers. Additionally, ORC files offer other benefits such as efficient compression, reduced storage costs, improved query performance, and lightweight metadata. The format uses a lightweight compression algorithm that allows for faster decompression, which makes it suitable for real-time analytics use cases. Further, ORC files support predicate pushdown and column pruning, enabling efficient query execution by only reading the relevant columns (Ivanov & Pergolesi, 2019).

Apache Parquet, on the other hand, is an open-source columnar storage format that uses complex nested data structures. Like ORC, it is a general-purpose storage format that can be used or integrated with any data processing framework. Parquet has been shown to support efficient compression and encoding schemas. In contrast to ORC, Parquet row groups are not explicitly separated from each other in Parquet. The default size of each data page in Parquet is 1 MB, which is 4 times larger than the compression chunks in ORC (Ivanov & Pergolesi, 2019).

Parquet also supports various encoding techniques, such as dictionary encoding, run length encoding, and bit packing, which can help reduce storage costs and improve query performance. Similar to ORC, Parquet also supports predicate pushdown and column pruning, which leads to efficient query execution.

Both Apache ORC and Apache Parquet are powerful columnar file formats suitable for working with big data analytics. While ORC offers advantages in large streaming reads and schema evolution, Parquet works better with efficiency and handling complex data structures. The choice between the two file formats depends on the specific use case, but in this thesis, Apache Parquet will be used as it is the most suitable file format for the three data lakehouse systems used.

3.6 Databricks

The Databricks Lakehouse Platform provides a unified set of tools for building, deploying, sharing, and maintaining data solutions. Databricks integrates directly with cloud storage and security in Azure and manages and deploys infrastructure based on the user's needs. Databricks provides user-friendly UIs and scalable storage to provide a platform for running analytic queries (Databricks, 2023).

Databricks is available on Microsoft Azure along with other cloud environments, such as Amazon Web Services and Google Cloud. The purpose of Databricks is to serve as a unified data and analytics platform that supports traditional data warehousing within a data lake as well as real-time streaming analytics. Databricks have many different use cases, such as ML, data science, SQL analytics, and data engineering.

As organizations are adopting the data lakehouse paradigm, it is critical to have similar capabilities of traditional data and BI systems applying SQL-based analysis to their Lakehouse data (Armbrust et al., 2020). All in all, Databricks serves as a versatile and scalable platform for working with data in a data lakehouse architecture. The main differences and similarities between Delta Lake, Apache Iceberg, and Apache Hudi can be seen in Table 3.

Table 3 Data lakehouse systems properties

	Delta Lake	**Apache Iceberg**	**Apache Hudi**
ACID Compliance	Yes	Yes	Yes
Concurrency control	Optimistic	Optimistic	Optimistic
Storage formats	Parquet	Parquet, ORC	Parquet, AVRO
Compaction	Yes	No	Yes
Data skipping	Yes	Yes	Yes
Upsert/delete	Yes	No	Yes
Caching	Yes	No	No
Time Travel	Yes	Yes	Yes
Schema Evolution	Yes	Yes	Yes
Object storage support	Yes	Yes	No

4 PREVIOUS STUDIES

This chapter presents a review of relevant previous studies on data lakehouse systems. The goal of this chapter is to provide an overview of the current research landscape and discuss the key findings from comparative studies of various data lakehouse frameworks. Both comparative studies, as well as real-world implementations of data lakehouse systems, will be presented.

4.1 Begoli et al. (2021)

Begoli, Goethert and Knight analyzed a lakehouse architecture for the management and analysis of heterogeneous data for biomedical research and mega-biobanks. The study involved the integration of data from various source, and also involved the development and deployment of data processing and analysis pipelines for various biomedical research tasks, such as identifying disease biomarkers and predicting treatment outcomes. In their paper, the authors presented a case study on a lakehouse architecture that combines the strengths of both data warehouses and data lakes to support complex biomedical research projects. The proposed lakehouse architecture in this paper consisted of three layers: (1) a storage layer that stores structured and unstructured data, (2) a compute layer that provided efficient data processing and analysis capabilities, and (3) a metadata layer that provided a unified view of the data while supporting data discovery and governance. The architecture was designed to handle diverse types of data, such as genomic data, imaging data, and clinical data to support various data analysis tasks, such as statistical analysis and ML.

The authors used real-world data from the Biobanking and BioMolecular Resource Research Infrastructure (BBMRI) project, which is a large biobank project that collects and stores biological samples and associated data from various sources across Europe. The results suggested that the proposed data lakehouse architecture provides a scalable and flexible solution for managing and analyzing large, diverse biomedical data sets. The architecture was able to handle complex data management and analysis tasks efficiently while addressing the limitations of data warehouses and data lakes. Finally, the study demonstrated the potential of the proposed architecture to support biomedical research projects and mega-biobanks while also enabling new discoveries and insights in the field of biomedicine.

4.2 Park et al. (2023)

The paper *Design of Vessel Data Lakehouse with Big Data and AI Analysis Technology for Vessel Monitoring System* proposes a design for a data lakehouse infrastructure that can improve the monitoring and safety of vessels in the maritime industry. As maritime data is growing at an unprecedented rate, the authors propose a vessel data lakehouse architecture consisting of four layers: an ETL layer, a vessel data lake layer, a vessel data warehouse layer, and finally a vessel application service layer that supports marine application services. The proposed data lakehouse architecture can efficiently manage heterogeneous vessel related data and can easily be integrated managed at low cost by structuring various types of heterogeneous data using an open-source big data framework. Further, various types of vessel big data stored in data lakehouses can be directly utilized in different types of vessel analysis services.

The Vessel Data Lakehouse consists of four layers:

1. **Extraction and ingestion layer**: This layer imports vessel-related data from external sources in a specific format. Data is extracted using a push or pull approach based on Message Queuing technology.

2. **Vessel Data Layer**: Implemented using a Hadoop cluster, this layer enable storage and processing of large volumes of vessel data. It includes HDFS, Hadoop YARN, and Hadoop MapReduce.

3. **Vessel Data Warehouse layer**: This layer consists of the Vessel Big Data layer and the Vessel AI layer. The big data layer supports AI models and is responsible for data collection, purification, and modeling. The AI layer includes the AI module, and a ML Module.

4. **Vessel Application Services**: This layer consists of visualization services and operation services.

The Vessel Data Lakehouse employs open-source software to build a comprehensive system that can manage, process, and analyze large-scale vessel data. The implemented system can be used for various applications, such as determining ship distribution and predict fishing activity. By using AI techniques, the Vessel Data Lakehouse supports an array of applications and services for VMS, enabling efficient and effective data-driven decision-making.

A test was performed by comparing the query processing results of PostgreSQL, a relational database, and Impala, the query engine of the data lake. The main measure of performance was the time taken to process a count query that counts the number of rows in the dataset. The comparison was made between a Data Lake cluster with 4 nodes and a PostgreSQL with 1 node.

The results showed that the Data Lakehouse performed significantly better than relational databases in query response rates. The Vessel AI layer showed good performance in predicting fishing activities and vessel types. The system's efficiency was proven with an average query response rate of 92.95% higher than the relational database.

4.3 Onehouse.ai (2023)

In another recent comparative study by Onehouse.ai (2023), they examined the key differences between Apache Hudi, Delta Lake, and Apache Iceberg, with a main focus on data models, storage formats, upsert supports, time travel capabilities, and compatibility with storage systems. The results showed that while Hudi and Delta Lake both work with log-based data models, Apache Iceberg works better as a table-based data model. Apache Hudi and Apache Iceberg both support multiple storage formats, for example, Parquet and Avro, whereas Delta Lake primarily depends on Parquet. Further, in terms of upsert capabilities, Hudi offers advanced support, while Delta Lake and Apache Iceberg only offer basic support.

Regarding the time-traveling capabilities, Delta Lake and Iceberg allow the users to query older data versions, while Hudi provides incremental query support. All three systems support streaming, but Apache Hudi and Delta Lake showed better integration with various data processing engines. Finally, Delta Lake and Apache Iceberg are compatible with multiple storage systems, while Apache Hudi is more reliant on Apache Hadoop.

The authors conclude that the choice of a data lakehouse framework should be determined by the project's specific requirements and priorities. For a project that needs advanced upserts support, Apache Hudi would be a suitable choice. Delta Lake and Apache Iceberg might be more suitable for those who seek time travel capabilities and compatibility with multiple storage systems.

4.4 Conclusion

Summary of the previous studies, data lakehouse architectures are a promising solution to overcome the current challenges and limitations of data warehouses and data lakes by combining the strengths of them both. Data lakehouses provide scalable and flexible solutions for managing and analyzing large, diverse datasets while supporting various data analysis tasks. As the field of big data analytics continues to evolve, it is likely that data lakehouse architectures will play an increasingly important role. However, the technology is quite immature, and further research is needed before we can say whether it will live up to its promises. Table 4 displays the differences between the three data lakehouse systems.

Table 4 Differences between Delta Lake, Apache Iceberg, and Apache Hudi

Feature	**Apache Hudi**	**Delta Lake**	**Apache Iceberg**
Data Model	Log-based	Log-based	Table-based
Storage and Format	Parquet, Avro	Predominantly Parquet	Parquet, Avro
Upsert Support	Advanced	Basic	Basic
Time Travel	Incremental Query Support	Full Time Travel Support	Full Time Travel Support
Streaming and Integration	Good Integration	Good Integration	Limited Integration
Compatibility	Best suited for Apache Hadoop	Multiple Storage Systems	Multiple Storage Systems

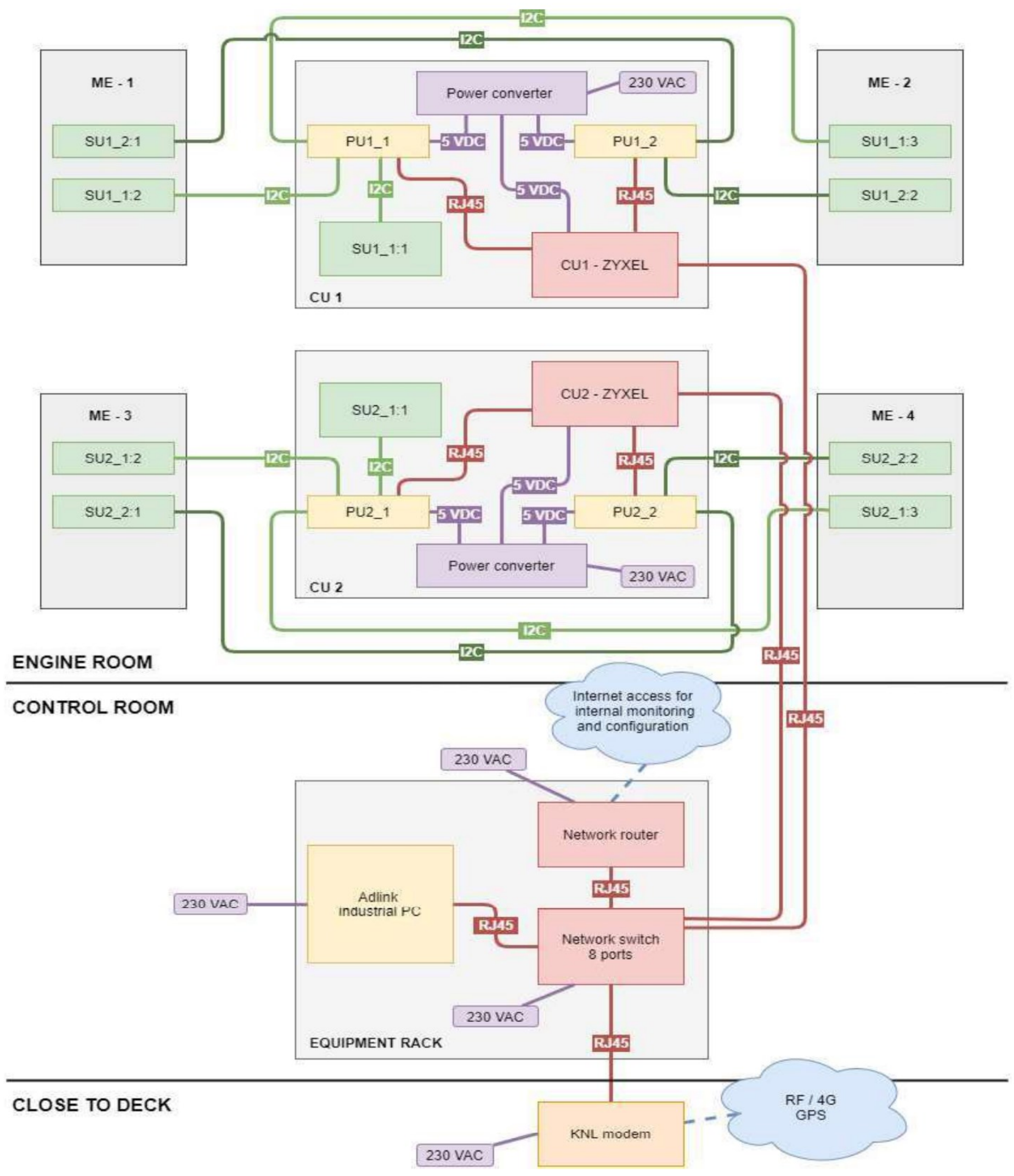
5 DATA

The data that will be used in this study was gained from an experiment done by Morariu et al. (2020) where the design, installation, and data analysis of an edge-based vibration monitoring system installed on the main engines of a cruise ferry traveling between Finland and Sweden was described. The objective of the study was to analyze the reliability of non-expensive sensors and computing systems, while also performing data analysis close to the sensors. The goal was to enable unmanned monitoring and increase situation awareness in the engine room. The data collected would be important for identifying trends in engine vibration patterns and improving maintenance practices. The long-time objective is to use edge sensors in order to achieve increased situational awareness based on ML and AI, driven by edge computing equipment installed close to the sensors themselves Morariu et al. (2020).

The data was collected by mounting SPB sensors using edge analytics for monitoring the main engines of a cruise ferry. The sensor units were connected to Raspberry Pi 4 computers that were acting as power units. These were placed in respective control unit boxes, with two Raspberry Pis per control unit box. Further, sensor units were connected externally to each Raspberry Pi. The control unit boxes were connected to an equipment rack in the control room that holds an industrial Adlink computer. SBP sensors were used for temperature, accelerometer, and gyroscope readings by being placed in a metallic box to be protected from temperature and humidity. Morariu et al. (2020)

The data was gathered under a 102-day period, with a total uptime of 73% during the whole experiment. A total of 3.67 GB of data (compressed text) was generated each day, with a total of 375 GB of data gathered during the whole experiment. The daily data contains data related to the ship location, as well as information about vibration energy, histograms, temperature readings, and spectrograms from the sensors. The final setup is displayed in Figure 4.

Figure 4 Data gathering setup (Morariu et al., 2020)



The data used in this study is gathered during a four-month period within the time window used by Morariu et al. (2020), starting from June 2020 to September 2020. Due to the prospect of this thesis, only data for one CU, PU and MU is included. Beyond this, the data from the Adlink computer was also included in the dataset.

6 METHOD

This chapter will go through the method for performing the analysis. The analysis was done by running three different tests on the different data lakehouse systems: data ingestion, query performance, and security.

After the data cleaning and preparation, it was uploaded to different Azure blob containers. One container for each data lakehouse system was created, and the data was uploaded to each container. This setup ensured that each data lakehouse had access to the same data, but in a different container for a fair comparison. The data layout in each blob container is shown in Figure 5.

Figure 5 Data layout

```
lakehouses
├── delta-lake
│   ├── adlink_csv
│   │   ├── file1.csv
│   │   ├── file2.csv..
│   ├── CU1_SU_csv
│   │   ├── file1.csv
│   │   ├── file2.csv...
│   ├── ME1_SU1_csv
│   │   ├── file1.csv
│   │   ├── file2.csv
│   └── PU1_1_csv
│      ├── file1.csv
│      ├── file2.csv...
├── iceberg
│   ├── ...
└── hudi
│   ├── ...
```

A separate workspace and cluster were set up for each data lakehouse system in Databricks. The cluster used for each solution was a single-node Databricks cluster with Databricks Runtime version 12.2, which includes Apache Spark 3.3.2 and Scala 2.12. The node type used is of type standard with 14 GB memory and 4 cores, which is the basic performance package offered by Databricks.

Three test scenarios were designed to evaluate the data lakehouse systems: data ingestion, query performance, and scaling. Each test scenario focused on a specific aspect of the data lakehouse systems and aimed to determine their performance, efficiency, and robustness under different conditions. The tests were completed on a separate cluster and in a separate container to avoid performance bias. Performance bias might occur when the performance of one technology is unfairly influenced by the environment in which it is begin tested. By using separate clusters and blob storages, the risk of skewed performance results can be mitigated.

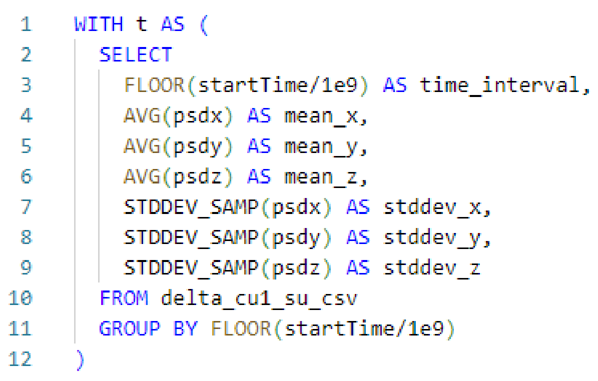
6.1 Data ingestion

The data ingestion test evaluated the speed and efficiency with which each data lakehouse system ingested and converted data from the Azure blob container. This test allows for an assessment of each system's ability to handle data, and the time required to make the data available for querying by converting it to the correct format. The code used for this test can be found in Appendix 1.

6.2 Query performance

Query performance is a critical aspect of any database as it determines how efficiently users can access the data stored in the system. The query performance is measured by calculating the standard score value for each second in the psdx, psdy, and psdz values from the CU1 unit. First, the mean and standard deviation for each variable for each second is calculated using the query displayed in Figure 6.

Figure 6 Query for calculating the mean and standard deviation for each second



```sql
WITH t AS (
  SELECT
    FLOOR(startTime/1e9) AS time_interval,
    AVG(psdx) AS mean_x,
    AVG(psdy) AS mean_y,
    AVG(psdz) AS mean_z,
    STDDEV_SAMP(psdx) AS stddev_x,
    STDDEV_SAMP(psdy) AS stddev_y,
    STDDEV_SAMP(psdz) AS stddev_z
  FROM delta_cu1_su_csv
  GROUP BY FLOOR(startTime/1e9)
)
```

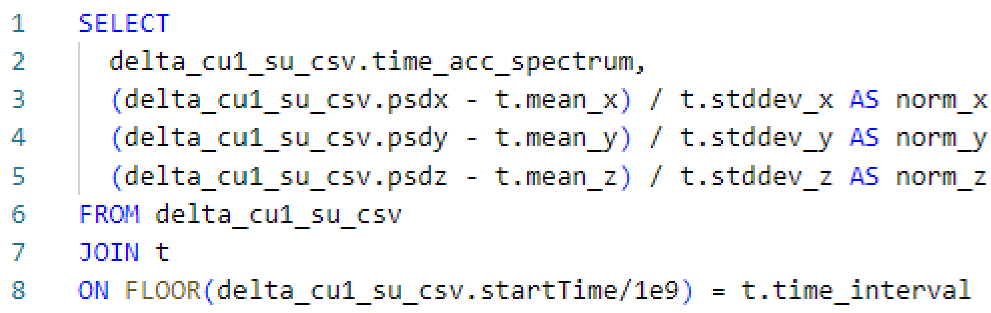
After this, the result from the query is used to calculate the standard score using the formula:

$$w = (x - \mu) / \sigma,$$

where w represents the standard score, x is the raw score value for the variable (psdx, psdy, psdz), μ is the mean value for the variable, and σ is the standard deviation.

Finally, the query shown in Figure 7 is used to normalize the values and calculate the standard score. The performance is based on the time it takes for each system to process the data and return the query result.

Figure 7 Query for normalizing the values and calculating the standard score



```sql
SELECT
  delta_cu1_su_csv.time_acc_spectrum,
  (delta_cu1_su_csv.psdx - t.mean_x) / t.stddev_x AS norm_x,
  (delta_cu1_su_csv.psdy - t.mean_y) / t.stddev_y AS norm_y,
  (delta_cu1_su_csv.psdz - t.mean_z) / t.stddev_z AS norm_z
FROM delta_cu1_su_csv
JOIN t
ON FLOOR(delta_cu1_su_csv.startTime/1e9) = t.time_interval
```

6.3 Scaling

Beyond data ingestion and query performance, scaling is an essential factor to consider when evaluating the different data lakehouse systems. To test scaling for the three different data lakehouse systems, the same query used in the query performance test will be employed in the scaling test, but with different amounts of data to evaluate how each system handles increased data loads. Three scenarios will be designed for this test. Further, to display each system's ability for time travelling, a million dummy rows will be generated, and the same query will be run on the whole dataset with an additional million rows of dummy data.

1. A quarter of the original dataset (3,914,861 rows)
2. Half the dataset (7,829,722 rows)
3. The full dataset including dummy data (15,659,444 rows)

To create the dummy data, the minimum and maximum values for each variable in the original dataset are identified. Based on these values, random values are generated for each variable that falls within the respective minimum-maximum range, which maintains a similar distribution to the original dataset. Since the original dataset contains 500 observations per second, the dummy data will follow the same pattern.

1. Identifying the minimum and maximum values for each variable.

2. Initialize a random number generator with a specific seed value to ensure reproducibility of the results.

3. For each variable, generate a million random values that fall within the respective minimum-maximum range.

4. Extending the timestamp column for the dummy data by calculating the maximum and minimum timestamp value from the original dataset and incrementing it by one second for each new row of dummy data, maintaining 500 observations per second.

5. Append the dummy data to the original dataset, resulting in a total of 16,659,444 rows (15,659,444 original rows + 1,000,000 dummy rows).

By generating dummy data based on the original data's value ranges, we make sure that the scaling test accurately represents how each data lakehouse system will handle increased data loads in a real-world scenario, as well as the time is taken to calculate and write new data. Furthermore, this approach allows for evaluating each s system's time-traveling capabilities, as the systems need to handle and process the additional million rows of data.

7 RESULTS

7.1 Data ingestion

To ingest data into a data lakehouse, it needs to be converted to an appropriate format. As previously mentioned, this study will be using the Parquet format. A columnar storage format is well-suited for big data workloads, especially when only tabular data is included. To convert the data to parquet format, Scala and Spark were used. First, all the necessary packages required for the conversion process are imported and installed in the Spark configuration. This includes packages for reading and writing data to parquet format, as well as packages for analyzing the data frames and configuration for the Spark session.

Once the necessary libraries were installed, a Spark session was created to begin the conversion process. The data was initially stored in an Azure blob container, and the same approach was used for all three data lakehouse systems, with only minor modifications required to account for differences in the specific APIs and data ingestion tools used by each system. The code for each system can be found in Appendix B.

The results show that the data ingestion process was quite efficient for all data lakehouse systems, but in terms of percentage differences, the results vary. Table 5 shows that Iceberg performed the fastest data conversion at 5.14 minutes, followed by Delta Lake at 6.48 minutes. Apache Hudi was able to convert the data in 8.18 minutes.

Table 5 Data ingestion test result

Data lakehouse	Performance (in minutes)
Delta Lake	6.48
Iceberg	5.14
Hudi	8.18

Despite the differences in performance, all data lakehouse systems were able to convert and ingest data in a relatively short amount of time, which demonstrates the efficiency and scalability of these systems for big data workloads. However, it is important to note that the performance may vary depending on different factors, such as the size and complexity of the data, the hardware and infrastructure, and the specific configuration of each system. These results should therefore be interpreted in the context of the specific data and use case studied.

7.2 Query performance

In this subchapter, the results of the query performance test carried out on the three different data lakehouses systems are presented. The query is simulating a real-world analytical scenario that requires calculations of the mean, standard deviation, and standard score for the psdx, psdy and psdz values from the CU1 unit. The dataset used for this query contained a total of 15,659,444 rows with 500 observations per second, which is data for a total of 8.70 hours, or 521.97 minutes.

The performance of each data lakehouse system is compared based on the time taken to process the data and return the query result. The performance results are presented in Table 6.

Table 6 Query performance

Data lakehouse	Performance (in seconds)
Delta Lake	38.69
Iceberg	34.67
Hudi	36.67

The results show that Apache Iceberg delivered the fastest query performance by completing the calculation and returning the query result in 34.67 seconds. Apache Hudi was a close second with a slightly slower response time of 36.67 seconds, and finally, Delta Lake exhibited a somewhat longer processing time of 38.69 seconds.

Although there are differences in the performance on the original dataset, all three data lakehouse systems performed and executed the analytical query efficiently, highlighting their suitability for handling large-scale data analysis tasks.

7.3 Scaling

In this subchapter, the results of the scaling tests carried out on the three different data lakehouses systems are presented. The three systems were evaluated based on three scenarios: a quarter of the original data, half the data, and the full dataset including dummy data. Further, the time-traveling capability of the data lakehouse systems and the further scaling ability of each system were tested using the full dataset with an additional million rows of dummy data. The code for this experiment can be found in Appendix 4.

7.3.1 A quarter of the dataset

In the first scenario, the dataset was reduced to a quarter of its original size, containing 3,914,861 rows. The results in Table 7 show that Apache Hudi performed the best, with a query execution time of 11.17 seconds. Delta Lake came in second, with 15.89 seconds, followed by Apache Iceberg which had the slowest query performance for this scenario, with a time of 23.20 seconds.

Table 7 Query performance for a quarter of the dataset

Data lakehouse	Performance (in seconds)
Delta Lake	15.89
Iceberg	23.30
Hudi	11.17

7.3.2 Half the dataset

When reducing the dataset to half its size, the total amount of rows was 7,829,722. Table 8 displays the query performance results for each data lakehouse system. Again, Apache Hudi performed the best with a query execution time of 16.42 seconds. Delta Lake performed the query in 20.58 seconds, and Apache Iceberg was the slowest of the three again with a query performance time of 27.84 seconds.

Table 8 Query performance for ½ of the dataset

Data lakehouse	Performance (in seconds)
Delta Lake	24.04
Iceberg	25.37
Hudi	17.71

7.3.3 Full dataset and dummy data

Finally, the full dataset with an additional million rows of dummy data was used, resulting in a total of 16,659,444 rows. Table 9 displays the writing time required for each data lakehouse system. This includes calculating the new values and writing them to the existing dataset. The results show that Apache Iceberg performed the best with a writing time of 1.28 minutes, closely followed by Delta Lake with 1.55 minutes. Apache Hudi showed a significantly slower write speed compared to the other two systems, taking 5.11 minutes to calculate and insert the additional million rows.

Table 9 Write speed

Data lakehouse	Write speed (1,000,000 rows)
Delta Lake	1.55
Iceberg	1.28
Hudi	5.11

When running the query on the full dataset with an additional million rows of dummy data, the results show that Apache Iceberg had the best time of 41.12 seconds, followed by Delta Lake at 41.45 seconds. Surprisingly, Apache Hudi came in last with a query execution time of 43.2 seconds. The result from the final test is displayed in Table 10.

Table 10 Query performance for the full dataset including dummy data

Data lakehouse	Performance (in seconds)
Delta Lake	41.45
Iceberg	41.12
Hudi	43.2

7.4 Summary of the results

Several conclusions can be drawn from the results of the scaling performance test for each data lakehouse system. Regarding the data ingestion test, all three data lakehouse systems managed to convert and ingest the maritime data within a relatively short amount of time, but there were some noticeable differences in performance. Apache Iceberg exhibited the fastest data ingestion performance, taking only 5.14 minutes to convert and ingest all CSV files in all folders in the original dataset. Delta Lake showed a slightly slower performance than Iceberg with 6.48 minutes to complete the data ingestion process, while Apache Hudi showed the slowest data ingestion performance among the three systems, requiring 8.18 minutes to complete the process. While there are differences in the writing speed of the systems, they all managed to complete the data ingestion process in a relatively short amount of time.

Delta Lake demonstrated relatively stable and consistent performance across all scaling scenarios. It was not the fastest in any specific scenario but maintained a performance level that was highly competitive against Iceberg and Hudi. Delta Lake's write speed for inserting the dummy data was also fast compared to the other systems and is overall a well-rounded option for handling various amounts of data.

Apache Iceberg consistently performed well across all scaling scenarios, and also in the original query performance test. Even though Apache Iceberg had slower query performance results with the smaller datasets compared to the other data lakehouse systems, it was the fastest when handling the full dataset, including dummy data. Apache Iceberg also exhibited the fastest write speed for calculating and inserting the additional one million rows of dummy data. These results suggest that Apache Iceberg is a solid and scalable option that can efficiently manage both small and large datasets.

Apache Hudi demonstrated strong performance when dealing with smaller datasets, such as a quarter and half of the original dataset. Apache Hudi outperformed both Delta Lake and Apache Iceberg drastically with the smaller datasets. However, with an increase in dataset size to the full dataset, and also with an additional million rows of dummy data, Apache Hudi's performance was slower compared to Apache Iceberg and Delta Lake. This suggests that Apache Hudi might be more suitable for handling smaller or moderate-sized datasets, as the performance may decline as the size of the dataset increase. However, Apache Hudi still showed good performance compared to the other data lakehouse systems, and the choice of data lakehouse provider should not only be based on scaling.

In conclusion, each data lakehouse system has its strengths and weaknesses when it comes to data ingestion, query performance, and scaling. Apache Iceberg demonstrated the fastest data ingestion and strong performance in both small and large versions of the dataset, making it a solid choice for various data workloads. Delta Lake offered consistent and competitive performance across all scaling scenarios, making itself a well-rounded option as well. Apache Hudi excelled in handling smaller datasets but experienced a decline in performance as the dataset size increased, indicating that it might be more suitable for smaller or moderate-sized datasets.

Regarding the ease of use for each data lakehouse system, it is important to notice that ease of use is subjective and might vary depending on the individual's familiarity and experience with different tools and programming languages. However, based on the experience gained from these tests, a general overview of the ease of use based on the steps required for implementation, and the level of documentation available, a short summary will be discussed regarding the author's experience.

In this study, Apache Iceberg was found to be the easiest one to use, with a straightforward implementation process and clear documentation. This allowed for a quick setup and integration, which may be an advantage for organizations looking to implement a data lakehouse architecture.

As Delta Lake is developed by Databricks, one could argue that it can be more easily integrated with the Databricks platform, and could be beneficial for organizations already familiar with Databricks. The ease of use for Delta Lake was considered to be slightly more challenging compared to Apache Iceberg, but still relatively user-friendly and should be considered a good choice for a data lakehouse system.

Finally, Apache Hudi was considered to be the most challenging in terms of ease of use among the three systems. Apache Hudi required more configuration and a deeper understanding of its numerous parameters. This could potentially lead to a steeper learning curve for new users, but for those with a good understanding of the requirements, Apache Hudi can still be a valuable choice.

8 CONCLUSIONS

This thesis compared and analyzed three different data lakehouse systems: Delta Lake, Apache Iceberg, and Apache Hudi. The thesis focused on architectural differences, features, and use cases of these three systems in order to understand their respective suitability for various data management and processing scenarios. The thesis evaluated the performance, scalability, readability, and ease of use of these solutions in a comparative manner. The purpose of the study was to study and compare different data lakehouse systems.

The main differences between Delta Lake, Apache Iceberg, and Apache Hudi include their supported data file formats, transaction log handling, indexing, and storage types. While all three systems support ACID compliance, snapshot isolation, and time travel, they differ in query engine integration and schema evolution capabilities. Delta Lake was originally developed by Databricks, Apache Iceberg by Netflix, and Apache Hudi by Uber.

The study uses data from an experiment performed by Morariu et al. (2020) which analyzed the reliability of sensors and computing systems in monitoring vibrations on the main engines of a cruise ferry traveling between Finland and Sweden. The data was collected using SBP sensors mounted on Raspberry Pi 4 computers for 102 days, generating 3.67 GB of compressed text data daily. This thesis focuses on data from a four-month period between June and September 2020, using data for one control unit, one power unit, and a measurement unit along with the data from the Adlink computer.

The purpose of this thesis was to compare different data lakehouse systems, specifically Delta Lake, Apache Iceberg, and Apache Hudi. The results showed differences in the data ingestion performance among the three systems. Apache Iceberg demonstrated the fastest data ingestion, followed by Delta Lake, while Apache Hudi had the slowest performance. This suggests that organizations with larger volumes of data to ingest should consider Apache Iceberg or Delta Lake as a suitable option for efficient data ingestion.

The test performed in this thesis evaluates three data lakehouse systems using three test scenarios: data ingestion, query performance, and scaling. The data was uploaded to separate Azure Blob Storage Containers, and separate workspaces and clusters were set up for each system in Databricks. The data ingestion test measured the speed and efficiency of data ingestion and conversion to the respective table, while the query performance test evaluated the efficiency of data calculation and access. The scaling test examined how each data lakehouse system handled increased data loads by processing different amounts of data, including an additional million rows of dummy data to test writing speed and time-traveling capabilities. This methodology ensures a fair comparison of the data lakehouse systems' performance, efficiency, and robustness under different conditions.

The results showed that each data lakehouse system has its strengths and weaknesses. Apache Iceberg demonstrated the fastest data ingestion and strong performance across scaling scenarios, making it a good choice for different data workloads. Delta Lake showed consistent and competitive performance, making it a well-rounded option for handling different dataset sizes. Apache Hudi excelled in handling smaller datasets, but its performance declined as the dataset size increased.

In terms of ease of use, Apache Iceberg was found to be the easiest to implement and use, with clear documentation. Delta Lake was slightly more challenging, but still relatively user-friendly, especially for those familiar with Microsoft Azure and the Databricks platform. Apache Hudi was the most challenging to set up, requiring more configuration and understanding of its numerous parameters.

While the results in this study provide insights into the performance of the three data lakehouse systems, it is important to consider other factors such as integrations, cost, and support for specific use cases when deciding for a data lakehouse architecture.

The comparison and results from this study suggest that data lakehouses are a competitive alternative to current database architectures. However, it is important to notice that the concept of Data Lakehouses is till relatively new, and it is difficult to say whether it will live up to its expectations. Compared to current database architectures, the results look promising. Further research is needed on Data Lakehouses before it can be concluded that this is the new generation of database architectures.

Regarding query performance and scaling, each system showed unique strengths and weaknesses. Again, Apache Iceberg consistently performed well across all scaling scenarios, exhibiting its ability to handle datasets of different sizes efficiently. Delta Lake also showed consistent performance across all scenarios and should be considered a well-rounded option for different data workloads. Delta Lake should especially be considered by organizations already familiar with Microsoft Azure or Databricks, as Delta Lake is originally developed by Databricks. In contrast, Apache Hudi performed exceptionally well with smaller datasets but experienced a decline in performance as the dataset size increased. This suggests that Apache Hudi might be more suitable for smaller datasets. Apache Hudi also had a longer data ingestion time compared to Delta Lake and Apache Iceberg.

The findings from this study align with previous research in some respects and diverge in others. For example, the study by Jain et al. (2023) found that Delta Lake outperformed Apache Hudi in terms of query execution and handling incremental data processing and real-time data updates. This aligns with the results from this study, as Delta Lake demonstrated consistent performance across all scaling scenarios, while Apache Hudi performed well with smaller datasets, but the performance declined with larger datasets.

Regarding data ingestion time, this study found that Apache Iceberg performed the fastest data ingestion and conversion, which was not discussed in previous studies. These findings are relevant to organizations with large amounts of data considering a data lakehouse architecture. Further, the author's preference in terms of ease of use falls on Apache Iceberg. Compared to Delta Lake and Apache Hudi, Apache Iceberg was the easiest to implement and had the best documentation.

8.1 Limitations

This thesis has some limitations that should be considered. First, the data used in this study is not representative of all types of data and use cases. The dataset used was also quite small, and it only included structured data. Further, the thesis only evaluated three data lakehouse systems, leaving the possibility that other systems could outperform the systems included in this study.

Additionally, this thesis focused mainly on data ingestion, query performance, and scaling. These metrics might not be enough for a definite result on which data lakehouse system performs the best, but it should be considered a start. The thesis did not either consider the impact of hardware and infrastructure differences on the performance, as all data lakehouse systems included in this study used the same hardware on Databricks and Microsoft Azure.

Finally, the study did not explore the potential impact of different and more specific configurations for each data lakehouse system, which potentially could affect the performance.

8.2 Suggestions for further research

There are constantly being new studies published around data lakehouses, and the performance can change rapidly as new research is published as well as new data lakehouse systems being developed. Further research should consider the use of diverse datasets and use cases, including structured and semi-structured data. This would provide a more comprehensive understanding of the strengths and weaknesses of each system in different scenarios.

Security and privacy features are important aspects of data lakehouse systems, and areas such as encryption, access control, and data governance would all be interesting research topics. A more extensive comparison of data lakehouse systems could also be beneficial. As previously mentioned, there are several data lakehouse systems available, and a more comprehensive study of these systems could help in identifying additional data lakehouse systems that might outperform the systems included in this thesis.

Different hardware and infrastructure configurations should also be taken into consideration in future studies. The study in this thesis was done on the basic infrastructure offered by Databricks, and exploring different options could be beneficial. Finally, further research could analyze the potential impact of more specific configurations of each data lakehouse system. Especially Apache Hudi offers a lot of different configuration possibilities, and it should be taken into consideration to determine overall performance.

9 SVENSK SAMMANFATTNING

9.1 Inledning

Data är i många fall varje företags viktigaste tillgång. Traditionellt sett så har datalager (data warehouse) hjälp företagsledare och organisationer att få en analytisk överblick genom att samla in data från flera olika databaser till centraliserade datalager som i sin tur kan användas som grund till olika beslut. Datalager är designade för analytisk processering, och ger en centraliserad och konsistent överblick över data, vilket gör det lättare att analysera (Ponniah, 2001; Kimball & Ross, 2013).

I dagens värld är behovet av informationslagring inte längre densamma som det en gång var. Beslutsfattare måste vara välinformerade för att kunna ta lämpliga beslut, och traditionella databassystem är inte längre lämpliga för dessa krav eftersom de historiskt implementerades för att stödja dagliga operationer istället för dataanalys och beslutsfattande.

Traditionella datalager är i en eller annan form i användning i de flesta organisationer i dag, men under de senaste åren så har problem relaterade till kostnader, användarbelastning samt att sakna förmågan att kunna lagra video, ljud eller text lett till att en ny generation av dataanalysplattform har växt fram. Dessutom så kan traditionella datalager vara rigida och oflexibla, vilket gör det svårt att hantera ostrukturerad data eller förändrade datakrav. På grund av detta så har allt fler företag och organisationer börjat infoga all sin ostrukturerade, råa data i datasjöar (data lakes) (Sabbouh & Seufert, 2016).

En datasjö är ett lågkostnadslagringssystem som kan spara data i ett öppet filformat med hjälp av ett fil-API (Thi et al., 2018). Datasjöar har en schema-on-read arkitektur som möjliggör agil lagring av data till en låg kostnad. Datasjöar är utformade för att lagra och bearbeta stora mängder rå, ostrukturerad data och ger en skalbar och kostnadseffektiv lösning för att laga stora mängder data (Nikulchev et al., 2021).

Även om datasjöar har många fördelar när det gäller att lagra och analysera stora mängder data, speciellt i jämförelse med datalager, finns det också flera utmaningar och problem med dem. Några av de vanligaste problemen är datakvalitet, sekretess och integration. Som svar på dessa problem myntades ett nytt koncept som kallas för datasjöhus (data lakehouse).

Ett datasjöhus kan definieras som en utveckling av en datasjö. Ett data lakehouse har ofta ännu lägre kostnader än en datasjö, och kombinerar styrkorna med ett vanligt datalager och en datasjö. Genom att kombinera fördelarna med traditionella databasarkitekturer så blir datasjöhus en enhetlig plattform för att lagra, hantera och analysera stora mängder data, samt möjliggör realtidsanalys och beslutsfattande. Vidare kan ett datasjöhus hantera både strukturerad och ostrukturerad data, vilket ger stor flexibilitet när det gäller datatyper. Datasjöhus ger en ännu mer agil approach till datahantering, eftersom data är direkt tillgängligt för maskinlärning, SQL-analyser eller artificiell intelligens (Armbrust et al., 2020).

Datasjöhus introducerades år 2020 och är därför ett relativt nytt koncept. Det är ett område där det inte ännu finns så mycket forskning, och det är därför svårt att säga huruvida teknologin kommer att leva upp till sina förväntningar (Jain et al., 2020). Syftet med denna avhandling har varit att belysa konceptet datasjöhus mer och analysera i vilka scenarier prestandan för ett datasjöhus kan utmana andra, traditionella typer av databasarkitekturer.

Denna avhandling presenterar en omfattande jämförelse av tre olika datasjöhus-lösningar: Delta Lake, Apache Iceberg och Apache Hudi. Avhandlingen fokuserar främst på de arkitektoniska skillnaderna, funktionerna och användningsområdena för dessa tre system för att bättre förstå deras lämplighet för olika datahanteringsscenarion. Avhandlingen utvärderar prestanda, skalbarhet, läsbarhet och användarvänlighet hos dessa lösningar på ett jämförande sätt. En detaljerad litteraturöversikt, en omfattande analys av varje system och ett prestandatest görs i avhandlingen, vilket ger insikter i både styrkor och svagheter hos varje datasjöhus-lösning. Avhandlingen avgränsas till dessa tre datasjöhus-system och bör inte ses som en detaljerad jämförelse av datasjöhus som koncept. Det finns många andra data lakehouse-lösningar tillgängliga idag, men denna avhandling har valt att fokusera på endast Delta Lake, Apache Iceberg och Apache Hudi eftersom dessa tre lösningar har fått mycket uppmärksamhet och anses vara bland frontlöparna inom teknologin (Databricks, 2019;2021).

9.2 Datasöjhus

Delta Lake är utvecklat av Databricks och är ett av de första datasjöhus-system som implementerades. Delta Lake använder en transaktionslogg som komprimeras till Apache Parquet-format för att ge ACID-egenskaper, vilket garanterar datakonsistens och tillförlitlighet. Genom att bygga på populära och kända molnobjektlager som Amazon S3 och Azure Data Lake Storage, integreras Delta Lake sömlöst med befintliga molninfrastrukturer. Detta leder till betydligt snabbare metadataoperationer för stora, tabulära dataset. Den erbjuder också automatisk datalayoutoptimering, uppsättningar, caching och revisionsloggning, vilket förenklar datahanteringen.

Delta Lake används redan av tusentals organisationer, där den hanterar exabyte med data varje dag. Delta Lake anses vara lämpligt för de flesta datasjö-applikationer som annars skulle ha använt Parquet eller ORC som strukturerade lagringsformat. Delta Lake är lätt att använda och lättillgängligt tack vare dess öppna källkod. Dess stöd för ACID-transaktioner, dataversionering och dataintegritet gör det till ett kraftfullt verktyg för datadrivna applikationer inom många olika branscher.

Liksom Delta Lake är Apache Iceberg också ett tabellformatssystem som möjliggör skapandet av en datasjöhus-arkitektur. Apache Iceberg skapades ursprungligen av Netflix för att hantera tidigare begränsningar hos Apache Hive när det gäller stora tabeller. Apache Iceberg stöder transaktionell och konsistent data genom metadatainsamling i manifestlistor och manifestfiler relaterade till tabellernas snapshot-tider. Apache Iceberg är ett tabellformat som är lämpligt för att lagra tabeller större än en petabyte, och det var ursprungligen utformat från grunden för att användas via molntjänster. Apache Iceberg kan användas för att hantera stora analytiska kalkylblad med hjälp av oföränderliga filformat som Parquet, Arvo och ORC.

Apache Hudi är ursprungligen skapat av Uber, och möjliggör snabba uppdateringar och raderingar samt inkrementell bearbetning av stora datamängder som lagras i distribuerade filsystem, likt Delta Lake och Apache Iceberg. Apache Hudi utvecklades i huvudsak för att hantera storskalig datalagring i distribuerade filsystem som molnlagring och HDFS. Apache Hudi prioriterar optimering av strömmande inläsning av data och insamling av dataförändringar för att snabba upp datainsamlingen och analysen där endast data som samlats in under en viss period krävs. Den inkrementella bearbetningsfunktionen ökar prestanda genom att endast bearbeta ny data och undvika återbearbetning av gammal data.

Apache Hudi använder en katalogbaserad struktur för datalagring, där varje tabell lagras i en katalog som består av partitioner i form av mappar. Dessa partitioner delas upp i grupper som är skivade där varje skiva innehåller data i Parquet-format.

Delta Lake, Apache Iceberg och Apache Hudi utökar alla datasjöar med ACID-garantier genom att använda Apache Spark för SQL-queries och uppdateringar. Dessa plattformar lagrar både versions- och tidsinformation om uppdateringar till tabeller i en logg, som innehåller radändringar till tabellerna. Tack vare detta kan användaren göra tidsresor i databasen, vilket inte har varit möjligt i traditionell databasarkitektur.

9.3 Testdata

Den data som används i denna avhandling har samlats in från en experimentell installation av en vibrationsövervakning på huvudmotorerna på en kryssningsfärja. Datainsamlingen, eller experimentet, gjordes av Morariu et al. (2020) med målet att analysera tillförlitligheten hos billigare sensorer och datorsystem som också används nära sensorerna för dataanalys. Syftet var att möjliggöra övervakning utan bemanning och öka situationsmedvetenhet i maskinrummet. Datainsamlingen var viktig för att identifiera trender i motorvibrationsmönster och förbättra underhållspraxis.

Sensordatan samlades in med hjälp av Raspberry Pi-datorer. Data samlades in under en period på 102 dagar, och rapporter skapades på all insamlad data. Totalt genererades 3,67 GB data (komprimerad text) per dag, och totalt 375 GB data under hela experimentet. I denna studie inkluderas endast data för en kontrollenhet, en strömförsöjningsenhet, en mätenhet samt data från en Adlink-dator som användes i experimentet.

9.4 Metod

Analysen inkluderar tre olika test: datakonvertering, frågeprestanda och skalning. Efter förprocessering av datan laddades den upp till en Azure blob container. En container för varje data lakehouse-system skapades och datan laddades upp till respektive container. Denna uppsättning säkerställer att varje data lakehouse har tillgång till samma data, men i en annan container för rättvis jämförelse.

En separat arbetsyta i Databricks och kluster skapades också för varje data lakehouse-system. Klustret som används för varje system är en enda Databricks-nod med Databricks runtime version 12.2, vilket inkluderar Apache Spark 3.3.2 och Scala 2.12. Nodtypen som används är av typ standard med 14 GB minne och 45 kärnor, vilket är det grundläggande prestandapaket som erbjuds av Databricks.

Tre testscenarier skapas för att utvärdera de tre data lakehouse-systemen. Varje testscenario fokuserar på en specifik aspekt av data lakehouse-systemen och görs för att bestämma deras prestanda, effektivitet och robusthet under olika förhållanden.

Datainsamlingstestet utvärderar hastigheten och effektiviteten med vilken varje data lakehouse-system läste och konverterade data från sin Azure blob container. Detta bedömer varje systems förmåga att hantera data och tiden som krävs för att göra datan tillgänglig för SQL-frågor, samt konvertera den till rätt format.

Frågeprestanda illustrerar hur effektivt användaren kan få åtkomst till den data som lagras i systemet. Frågeprestandan mäts i denna avhandling genom att beräkna det normaliserade värdet för varje sekund i psdx-, psdy- och psdz-värdena från CU1-enheten. Först beräknas medelvärdet och standardavvikelsen för varje variabel för varje sekund, och efter detta används resultatet för att beräkna det normaliserade värdet.

Utöver datainsamling och frågeprestanda testas också skalning genom att använda samma fråga som i frågeprestandatestet men med olika mängder data för att utvärdera hur varje system hanterar ökade datalaster. Tre scenarier designades för detta test: ¼ av det totala observationerna, hälften av de totala observationerna samt alla observationer inklusive dummy data.

För varje datasjöhus-system genereras dummy-data baserat på orginaldatans minimum- och maximumvärden. Detta säkerställer att skalningstestet återspeglar hur varje system hanterar ökade datalaster, samt tiden som krävs för att beräkna och skriva ny data.

9.5 Resultat

Resultaten visar att datainsamlingsprocessen var ganska effektiv för alla datasjöhus-system, men procentuellt sett så var resultaten varierande. Apache Iceberg hade den snabbast datakonverteringen på 5,14 minuter, följt av Delta Lake på 6,48 minuter. Apache Hudi konverterade datan på 8,18 minuter.

Trots skillnaderna i prestanda så kunde alla datasjöhus-system läsa in och konvertera datan på en relativt kort tid, vilket illustrerar effektiviteten och skalbarheten hos dessa system för stora mängder data. Det är dock viktigt att notera att prestandan kan varierar beroende på olika faktorer, såsom datamängd och komplexitet, maskinvara och infrastruktur. Dessa resultat bör därför tolkas i kontexten av den specifika data och användingsfall som studerats.

Frågeprestandatestet visade att Apache Iceberg snabbast kunde beräkna och returnera queryn, Apache Hudi var näst bäst, och Delta Lake visade en lite längre bearbetningstid. Alla tre data lakehouse-system klarade av beräkning och att returnera resultatet effektivt.

I skalningstestet visade Apache Hudi den bästa prestandan när datamängden minskades till en fjärdedel av dess ursprungliga storlek, medan Delta Lake hade en mera stabil prestanda i alla skalningsscenarion. Apache Iceberg visade god prestanda i alla skalningsscenarier.

Resultatet visar att alla datasjöhus-system har både styrkor och svagheter när det gäller datainläsning, frågeprestanda och skalning. Enligt författarens egna preferenser så var Apache Iceberg bäst av de tre datasjöhus-systemen när det gäller användbarhet. Delta Lake var lite mera utmanande men fortfarande relativt användarvänligt, medan Apache Hudi ansågs vara det mest utmanande systemet.

9.6 Diskussion

Syftet med denna avhandling var att jämföra tre olika datasjöhus-system: Delta Lake, Apache Iceberg och Apache Hudi. Resultaten visade skillnader mellan de tre olika systemen i både konvertering av data och skalning, medan alla tre system hade liknande prestationer i frågeprestanda. Apache Iceberg hade snabbaste datainläsning, följt av Delta Lake medan Apache Hudi hade den långsammaste prestandan. Resultaten tyder på att användare som vill konvertera stora datamängder bör överväga Apache Iceberg eller Delta Lake som ett lämpligt alternativ.

När det gäller frågeprestanda och skalning visade varje system olika styrkor. Apache Iceberg presterade konsekvent i alla scenarier, liksom Delta Lake, medan Apache Hudi presterade bättre än båda de andra systemen i mindre datamängder men stötte på problem när datamängden ökade.

Datan som användes i denna studie är inte representativ för alla typer av data och användningsfall. I studien användes inte en så stor datamängd att man kan dra konkreta slutsatser. Avhandlingen utvärderade endast tre datasjöhus-system, vilket lämnar möjligheten av andra system kan prestera bättre än de system som ingår i denna studie. Vidare forskning bör överväga användningen av olika datamängder och användningsfall, och även använda sig av både strukturerad och ostrukturerad data. Resultatet från denna avhandling bör därför endast ses som riktgivande och ett bidrag till forskningen inom data lakehouse-system.

REFERENCES

El-Sappagh, S. H. A., Hendawi, A. M. A., & El Bastawissy, A. H. (2011). A proposed model for data warehouse ETL processes. *Journal of King Saud University-Computer and Information Sciences*, *23*(2), 91-104.

Armbrust, M., Das, T., Sun, L., Yavuz, B., Zhu, S., Murthy, M., ... & Zaharia, M. (2020). Delta lake: high-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, *13*(12), 3411-3424.

Armbrust, M., Xin, R., Lian, C., Huai, Y., Liu, D., Bradley, J. K., ... & Zaharia, M. (2020). Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. Proceedings of CIDR 2021.

Apache Hudi. (N.d). Apache Hudi Documentation. Retrieved from: https://hudi.apache.org/docs/overview/

Apache Iceberg. (N.d). Apache Iceberg Documentation. Retrieved from: https://iceberg.apache.org/docs/latest/

Begoli, E., Goethert, I., & Knight, K. (2021, December). A lakehouse architecture for the management and analysis of heterogeneous data for biomedical research and mega-biobanks. In *2021 IEEE International Conference on Big Data (Big Data)* (pp. 4643-4651). IEEE.

Behm, A. (2021). Photon: A High-Performance Query Engine for the Lakehouse. Databricks Inc. Retrieved from: https://www.cidrdb.org/cidr2022/papers/a100-behm.pdf

Belov, V., & Nikulchev, E. (N.d). Analysis of Big Data Storage Tools for Data Lakes based on Apache Hadoop Platform. MIREA—Russian Technological University, Moscow, Russia.

Chaudhuri, S., & Dayal, U. (1997). An Overview of Data Warehousing and OLAP Technology. Microsoft Research, Redmond; Hewlett-Packard Labs, Palo Alto. Retrieved from: https://dl.acm.org/doi/pdf/10.1145/248603.248616

Databricks. (2019). Introducing Delta Lake: An Open Source Storage Layer for Massive Scale Data Lakes. Retrieved from: https://databricks.com/blog/2019/04/24/introducing-delta-lake.html

Databricks. (2021). Delta Lake: Reliable Data Lakes at Scale. Retrieved from: https://databricks.com/product/delta-lake-on-databricks

Databricks. (2023). Databricks Documentation. Retrieved from: https://docs.databricks.com/

Dixon, J. Pentaho, Hadoop, and Data Lakes. (2010). Available online: https://jamesdixon.wordpress.com/2010/10/14/pentaho-hadoop-and-data-lakes/

Inmon, W. H. (1992). *Data architecture: the information paradigm*. QED Information Sciences, Inc..

Ivanov, T., & Pergolesi, M. (2020). The impact of columnar file formats on SQL‐on‐hadoop engine performance: A study on ORC and Parquet. *Concurrency and Computation: Practice and Experience*, *32*(5), e5523.

Jain, P., Kraft, P., Power, C., Das, T., Stoica, I., & Zaharia, M. (2023). Analyzing and Comparing Lakehouse Storage Systems. CIDR.

Khine, P. P., & Wang, Z. S. (2018). Data lake: a new ideology in big data era. In *ITM web of conferences* (Vol. 17, p. 03025). EDP Sciences.

Kimball, R., & Ross, M. (2013). The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling (3rd ed.). Wiley.

Kutay, S. (2021). Data Warehouse vs. Data Lake vs. Data Lakehouse: An Overview. Available online: https://www.striim.com/blog/data-warehouse-vs-data-lake-vs-data-lakehouse-an-overview/

Machado, I. A., Costa, C., & Santos, M. Y. (2022). Data Mesh: Concepts and Principles of a Paradigm Shift in Data Architectures. Procedia Computer Science, 196, 263-271.

Madera, C., & Laurent, A. (2016, November). The next information architecture evolution: the data lake wave. In *Proceedings of the 8th international conference on management of digital ecosystems* (pp. 174-180).

Morariu, A. R., Ashraf, A., & Björkqvist, J. (2021, September). A Systematic Mapping Study on Edge Computing Approaches for Maritime Applications. In *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 37-44). IEEE.

Nambiar, A., & Mundra, D. (2021). An Overview of Data Warehouse and Data Lake in Modern Enterprise Data Management. Journal of Big Data Analytics in Transportation, 3(1), 45-56.

Belov, V., & Nikulchev, E. (2021). Analysis of Big Data Storage Tools for Data Lakes based on Apache Hadoop Platform. *International Journal of Advanced Computer Science and Applications*, *12*(8).

Onehouse.ai. (2023). Apache Hudi vs. Delta Lake vs. Apache Iceberg: Lakehouse Feature Comparison. Available online: https://www.onehouse.ai/blog/apache-hudi-vs-delta-lake-vs-apache-iceberg-lakehouse-feature-comparison

Park, S., Yang, C. S., & Kim, J. (2023). Design of Vessel Data Lakehouse with Big Data and AI Analysis Technology for Vessel Monitoring System. *Electronics*, *12*(8), 1943.

Ponniah, P. (2001). Data Warehousing Fundamentals: A Comprehensive Guide for IT Professionals. John Wiley & Sons.

Sabbouh, M., & Seufert, J. (2016). A Unified Data Analytics Platform: Data Lake and Data Warehouse Combined. Journal of Big Data, 3(1), 1-12.

Salloum, S., Dautov, R., Chen, X., Peng, P. X., & Huang, J. Z. (2016). Big data analytics on Apache Spark. *International Journal of Data Science and Analytics*, *1*, 145-164.

Sangupamba, O. M., Prat, N., & Comyn-Wattiau, I. (2014). Business intelligence and big data in the cloud: opportunities for design-science researchers. In *Advances in Conceptual Modeling: ER 2014 Workshops, ENMO, MoBiD, MReBA, QMMQ, SeCoGIS, WISM, and ER Demos, Atlanta, GA, USA, October 27-29, 2014. Proceedings 33* (pp. 75-84). Springer International Publishing.

Stonebraker, M., Cattell, R., & Skeen, J. (1992). Object-Relational DBMSs, Second Edition. Morgan Kaufmann.

Thi, N. N., Tho, N. D., Thinh, N. H., & Huynh, D. D. (2018). Comparison of Data Lake and Data Warehouse in Big Data Architecture. ITM Web of Conferences.

APPENDIX 1 DELTA TABLE CONVERSION

```
import org.apache.spark.sql.SparkSession
import io.delta.tables._
import org.apache.spark.sql.delta._
import com.microsoft.azure.storage.CloudStorageAccount
import com.microsoft.azure.storage.blob.CloudBlobClient
import com.microsoft.azure.storage.blob.ListBlobItem
import scala.collection.JavaConverters._
import org.apache.spark.sql.functions._

val spark = SparkSession.builder()
      .appName("Delta conversion")
      .config("spark.databricks.delta.schema.autoMerge.enabled", "true")
      .getOrCreate()

// Set the variables
val storageAccountName = ""
val storageAccountAccessKey = "
val containerName = ""
val deltaTablePath = s""

spark.conf.set(s"spark.hadoop.fs.azure.account.key.$storageAccountName.blob.co
re.windows.net", storageAccountAccessKey)
spark.conf.set("spark.hadoop.fs.azure",
"org.apache.hadoop.fs.azure.NativeAzureFileSystem")
spark.conf.set(s"fs.azure.account.key.$storageAccountName.blob.core.windows.ne
t", storageAccountAccessKey)
spark.conf.set("fs.azure.account.key.provider",
s"org.apache.hadoop.fs.azure.SimpleKeyProvider")

// Set up the Azure Blob Storage configuration
val storageConnectionString =
s"DefaultEndpointsProtocol=https;AccountName=$storageAccountName;AccountKey=$s
torageAccountAccessKey;EndpointSuffix=core.windows.net"
val storageAccount = CloudStorageAccount.parse(storageConnectionString)
val blobClient = storageAccount.createCloudBlobClient()

// Get a list of all the blobs and virtual folders in the container
val container = blobClient.getContainerReference(containerName)
val blobList = container.listBlobs().asScala
blobList.foreach(item => println(item.getClass.getName))

val folderList = blobList.collect {
  case d: com.microsoft.azure.storage.blob.CloudBlobDirectory => d.getPrefix()
}

println(folderList)

// Read data from Azure Blob Storage
val sourceData = spark.read
      .option("header", "true")
      .option("inferSchema", "true")

.csv(s"wasbs://$containerName@$storageAccountName.blob.core.windows.net/source
-data-folder")

// Write data to Delta table
sourceData.write
      .format("delta")
      .mode("overwrite")
      .save(deltaTablePath)
```

APPENDIX 2 APPENDIX ICEBERG TABLE CONVERSION

```
// Import the necessary libraries
import org.apache.spark.sql._
import org.apache.spark.sql.functions._
import org.apache.iceberg.spark.SparkSessionCatalog
import org.apache.iceberg.types.Types
import org.apache.iceberg.Schema
import scala.collection.JavaConverters._

// Define the Azure Blob Storage configurations
val containerName = ""
val storageAccountName = ""
val storageAccountKey = ‘’
val endpoint = s""

// Set up the necessary configurations
spark.conf.set(s"spark.hadoop.${endpoint}", storageAccountKey)

// Define the base path to your Azure Blob Storage folders
val basePath =
s"wasbs://${containerName}@${storageAccountName}.blob.core.windows.net/"

// List your folder name
val folderNames = Seq("adlink_csv", "CU1_SU_csv", "ME1_SU1_csv", "PU1_1_csv",
"ship_csv")

// Iterate over the folders and convert each CSV file to Apache Iceberg format
for (folder <- folderNames) {
  val inputPath = basePath + folder

// Read the CSV files from the Azure Blob Storage folder
val df = spark.read
.option("header", "true")
.option("inferSchema", "true")
.csv(inputPath)

// Define the Iceberg table name and path
val icebergTableName = s"iceberg_${folder}"
val icebergTablePath = s"${basePath}iceberg_tables/${icebergTableName}"

// Configure the Iceberg catalog
spark.conf.set("spark.sql.catalog.spark_catalog",
"org.apache.iceberg.spark.SparkSessionCatalog")
spark.conf.set("spark.sql.catalog.spark_catalog.type", "hive")
spark.conf.set("spark.sql.catalog.spark_catalog.warehouse",
"iceberg_warehouse")

// Create an Iceberg table from the DataFrame
val firstColumnName = df.schema.head.name
val schema = SparkSchemaUtil.convert(df.schema)

// Create the Iceberg table using SQL
df.createOrReplaceTempView("tempView")
spark.sql(s"CREATE TABLE ${icebergTableName} USING iceberg
OPTIONS('format'='orc', 'write.object-storage.enabled'='true') AS SELECT *
FROM tempView")


println(s"Converted CSV files in folder '$folder' to Apache Iceberg format and
saved them at '$icebergTablePath'")
}
```

APPENDIX 3 APPENDIX HUDI TABLE CONVERSION

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession
  .builder()
  .appName("HudiExample")
  .config("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
  .getOrCreate()

// Set up the necessary configurations
val containerName = ""
val storageAccountName = ""
val storageAccountKey = ‘’
val config = ""

dbutils.fs.unmount("/mnt/hudi")
dbutils.fs.mount(
  source = s"",
  mountPoint = "/mnt/hudi",
  extraConfigs = Map(config -> storageAccountKey)
)


// Define the base path to Azure Blob Storage folders
val basePath = "/mnt/hudi/"

val folderNames = Seq("adlink_csv", "CU1_SU_csv", "ME1_SU1_csv", "PU1-1_csv",
"ship_csv")

// Iterate over the folders and convert each CSV file to Apache Hudi format
for (folder <- folderNames) {
  val inputPath = basePath + folder

  val df = spark.read
    .option("header", "true")
    .option("inferSchema", "true")
    .csv(inputPath)

  // Write the data to a Hudi table
  val hudiTableName = s"hudi_${folder}"
  val hudiTablePath = s"${basePath}hudi_tables/${hudiTableName}"

  val hudiOptions = Map[String, String](
    "hoodie.table.name" -> hudiTableName,
    "hoodie.datasource.write.recordkey.field" -> df.columns.head, // Assuming
the key column is the first column
    "hoodie.datasource.write.table.name" -> hudiTableName,
    "hoodie.datasource.write.operation" -> "upsert",
    "hoodie.datasource.write.precombine.field" -> df.columns.head,
    "hoodie.datasource.hive_sync.enable" -> "false",
    "hoodie.datasource.write.hive_style_partitioning" -> "false"
  )
  df.write
    .format("hudi")
    .options(hudiOptions)
    .mode("Overwrite")
    .save(hudiTablePath)


  println(s"Converted CSV files in $folder to Hudi format")
}
```

APPENDIX 4 SQL-QUERY

```
%sql
-- Calculate mean and standard deviation for each second
WITH t AS (
  SELECT
    FLOOR(startTime/1e9) AS time_interval,
    AVG(psdx) AS mean_x,
    AVG(psdy) AS mean_y,
    AVG(psdz) AS mean_z,
    STDDEV_SAMP(psdx) AS stddev_x,
    STDDEV_SAMP(psdy) AS stddev_y,
    STDDEV_SAMP(psdz) AS stddev_z
  FROM delta_cu1_su_csv -- VERSION AS OF 1 (time travel for non dummy rows)
  GROUP BY FLOOR(startTime/1e9)
)
-- Normalize values
SELECT
  delta_cu1_su_csv.time_acc_spectrum,
  (delta_cu1_su_csv.psdx - t.mean_x) / t.stddev_x AS norm_x,
  (delta_cu1_su_csv.psdy - t.mean_y) / t.stddev_y AS norm_y,
  (delta_cu1_su_csv.psdz - t.mean_z) / t.stddev_z AS norm_z
FROM delta_cu1_su_csv
JOIN t
ON FLOOR(delta_cu1_su_csv.startTime/1e9) = t.time_interval
```

APPENDIX 5 DELTA LAKE DUMMY ROWS

```
import org.apache.spark.sql.functions._
import scala.util.Random
import spark.implicits._

case class DataLayout(
    time_acc_spectrum: Long,
    psdx: Double,
    psdy: Double,
    psdz: Double,
    freq: Double,
    startTime: Double,
    stopTime: Double,
    time_diff: Double
)

val existingDeltaTablePath =
s"wasbs://${containerName}@${storageAccountName}.blob.core.windows.net/delta-
table/CU1_SU_csv/delta"
val existingData = spark.read.format("delta")
                              .option("versionAsOf", 2)
                              .load(existingDeltaTablePath)

val (minPsdx, maxPsdx) =
existingData.select(col("psdx").cast("double")).agg(min("psdx"),
max("psdx")).as[(Double, Double)].head
val (minPsdy, maxPsdy) =
existingData.select(col("psdy").cast("double")).agg(min("psdy"),
max("psdy")).as[(Double, Double)].head
val (minPsdz, maxPsdz) =
existingData.select(col("psdz").cast("double")).agg(min("psdz"),
max("psdz")).as[(Double, Double)].head
val (minFreq, maxFreq) =
existingData.select(col("freq").cast("double")).agg(min("freq"),
max("freq")).as[(Double, Double)].head
val (minStartTime, maxStartTime) =
existingData.select(col("startTime").cast("double")).agg(min("startTime"),
max("startTime")).as[(Double, Double)].head
val (minStopTime, maxStopTime) =
existingData.select(col("stopTime").cast("double")).agg(min("stopTime"),
max("stopTime")).as[(Double, Double)].head
```

```
val (minTimeDiff, maxTimeDiff) =
existingData.select(col("time_diff").cast("double")).agg(min("time_diff"),
max("time_diff")).as[(Double, Double)].head


var lastTimeAccSpectrum: Long =
existingData.select(col("time_acc_spectrum").cast("long")).agg(max("time_acc_s
pectrum")).as[Long].head

def generateDummyData(): DataLayout = {
  lastTimeAccSpectrum += 2000000
  DataLayout(
    time_acc_spectrum = lastTimeAccSpectrum,
    psdx = minPsdx + (maxPsdx - minPsdx) * Random.nextDouble(),
    psdy = minPsdy + (maxPsdy - minPsdy) * Random.nextDouble(),
    psdz = minPsdz + (maxPsdz - minPsdz) * Random.nextDouble(),
    freq = minFreq + (maxFreq - minFreq) * Random.nextDouble(),
    startTime = minStartTime + (maxStartTime - minStartTime) *
Random.nextDouble(),
    stopTime = minStopTime + (maxStopTime - minStopTime) *
Random.nextDouble(),
    time_diff = minTimeDiff + (maxTimeDiff - minTimeDiff) *
Random.nextDouble()
  )
}

val numDummyRows = 1000000

val dummyData = (1 to numDummyRows).map(_ => generateDummyData()).toDF()
val combinedData = existingData.union(dummyData)

val scaledDeltaTablePath =
s"wasbs://$containerName@$storageAccountName.blob.core.windows.net/delta-
table/CU1_SU_csv/delta"
combinedData.write.format("delta").mode("overwrite").save(scaledDeltaTablePath
)
```

APPENDIX 6 DELTA LAKE CREATING SCALED DATASETS

```
import org.apache.spark.sql.functions._

val numbered_rows_df = spark.table("numbered_rows")
val total_rows =
numbered_rows_df.select("total_rows").first()(0).asInstanceOf[Long]

val subset_25 = numbered_rows_df.filter(col("row_number") <= (total_rows *
0.25))
val subset_50 = numbered_rows_df.filter(col("row_number") <= (total_rows *
0.50))
```

```
import org.apache.spark.sql.functions._

val numbered_rows_df = spark.table("numbered_rows")
val total_rows =
numbered_rows_df.select("total_rows").first()(0).asInstanceOf[Long]

val subset_25 = numbered_rows_df.filter(col("row_number") <= (total_rows *
0.25))
val subset_50 = numbered_rows_df.filter(col("row_number") <= (total_rows *
0.50))
```