Seminar Report

Developments in Data lakes and Data Warehousing

Sonal Lakhotia

MatrNr: 14913835

Supervisor: Aasish Kumar Sharma

Georg-August-Universität Göttingen
Institute of Computer Science

March 31, 2024

Abstract

This report focuses on the developments in data lakes and data warehousing. It highlights data lakehouse architecture's capabilities to enable reliable data management, data science tasks, and open data formats and provide state-of-the-art performance. Data Lakehouse offers solutions to the problems that data lakes and warehouses suffer from, namely, data reliability, advanced analytics, data staleness, and total cost of ownership.

Data warehouses offer easy data access and analysis but are rigid and inflexible as they do not support unstructured data. Data lakes store raw data files in an open format. Although data lakes are cost-effective, achieving data governance, reliability, and metadata management in a data lake architecture has been challenging. Data lakes and warehouses are incompatible with machine learning and data science tasks.

The metadata layer in data lakehouse architecture enables access control, auditing, and time travel. It assures data quality and reliability for advanced analytics. This report demonstrates the capabilities of a Data Lakehouse in data management. Data Lakehouse provides the benefits of cloud lakes and Relational Database Management Systems designed for Online Analytical Processing (RDBMS-OLAP) while also achieving data governance. It also enables machine learning reproducibility and data auditing.

This report also discusses the performance of Delta Lake, Hudi, and Iceberg in handling a 3 Terabyte (TB) Transaction Processing Performance Council's Decision Support benchmark (TPC-DS) workload, focusing on both data loading and the execution times of specific queries. It also evaluates a 100 Gigabyte (GB) TPC-DS benchmark that includes data loading and execution of five queries (Q3, Q9, Q34, Q42, and Q59), followed by ten iterations of merging updates into the dataset and reassessing the same queries.

Declaration on the use of ChatGPT and comparable tools in the context of examinations

In this work I have used ChatGPT or another AI as follows:

- ☐ Not at all
- ☐ During brainstorming
- ☐ When creating the outline
- ☐ To write individual passages, altogether to the extent of 0% of the entire text
- ☐ For the development of software source texts
- ☑ For optimizing or restructuring software source texts
- ☑ For proofreading or optimizing
- ☐ Further, namely: -

I hereby declare that I have stated all uses completely.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC	High-Performance Computing
RDBMS-OLAP	Relational Database Management Systems designed for Online Analytical Processing
TB	Terabyte
TPC-DS	Transaction Processing Performance Council's Decision Support benchmark
GB	Gigabyte
ORC	Optimized Row Columnar
HDFS	Hadoop Distributed File System
ETL	Extract,Transform,Load
API	Application Programming Interface
GCS	Google Cloud Storage
ADLS	Azure Data Lake Storage
S3	Amazon Simple Storage Service
BI	Business Intelligence

1 Introduction

This report explores the rise of the Data Lakehouse architecture, a fusion of the key traits from Data Warehouses and Data Lakes. Currently, data management architectures are undergoing swift evolution to accommodate the escalating demands of business intelligence and analytical workloads [Sin+22]. Data plays a crucial role in driving business decisions, prompting enterprises to seek efficient methods for managing and leveraging it in predictive or automated tasks. The proliferation of data generation has unlocked numerous use cases, necessitating a unified approach where the same dataset used for generating business reports or dashboards can seamlessly serve machine learning tasks while upholding standards of governance, reliability, and metadata management [Arm+21].

Data Lakehouse combines the benefits of high-performance transactional data warehouses and open-format, lost-cost data lakes [KW18]. It is anchored by open storage formats that can implement DBMS functionality such as indexing, transaction, and other functions on cost-efficient data lakes. The architecture serves data interoperability and scalability [MHO23].

Data warehouses revolutionized how business leaders gained analytical insights by centralizing data collection [KR11]. As a first-generation data analytics platform, they aggregated data from operational databases, serving the realms of business intelligence and decision support. Employing a schema-on-write approach, data warehouses ensured optimized utilization for BI purposes [HM10]. However, they encountered challenges, particularly in integrating on-premises storage and computing to address scalability issues [HZ22]. The surge in data volume and diversity, including audio, video, and text formats, posed formidable challenges in managing user load and data effectively while maintaining cost-efficiency. Moreover, the architecture struggled to query unstructured data types efficiently [BN21].

To tackle these challenges, organizations turned to cost-effective second-generation data analytics platforms [KW18]. These platforms embraced the practice of loading raw data into open file formats like Optimized Row Columnar (ORC) [Fou] and Apache Parquet [Voh16] using file Application Programming Interface (API)s. Data lakes emerged as a pivotal solution, facilitating direct access to machine-learning systems while leveraging technologies like Apache Hadoop's Hadoop Distributed File System (HDFS) [Had] for economical storage. Operating on a schema-on-read architecture, data lakes enabled the storage of unstructured data at a reduced cost. However, this approach also introduced complexities related to data reliability and governance. While most data would be deposited directly into the data lake, a fraction underwent Extract,Transform,Load (ETL) [Vas09] processes to feed into the data warehouse for business intelligence and reporting tasks.

Cloud data lakes such as Google Cloud Storage (GCS) [Clo], Azure Data Lake Storage (ADLS) [Sto], and Amazon Amazon Simple Storage Service (S3) [S3] upsurged in the 2015's and started replacing HDFS [HDF]. They characterized high reliability, durability, geo-replication, cheaper and automatic archival storage eg.AWS Glacier [Gla].

The data lake and warehouse is a two-tier architecture that is very cost-efficient due to separate storage and computing and is the dominant architecture in the industry worldwide but is quite complex for users. In the traditional warehousing architecture data from operational databases was ETLed into the warehouse [Err+23]. In the current scenario, data is first ETLed into the data lake and then ETLed into the data warehouse which leads to duplicated storage costs, complexities, data staleness, and new failure modes [BLO14]. Most businesses and enterprises use machine learning for market analysis, predictions, and automation for which both the data warehouse and data lake architectures are not idea [Arm+21].

It would be accurate to conclude that modern data architectures suffer from four major issues and limitations:

- Data Staleness: The data warehouse holds stale data in comparison to the data lake, as new data ingestion is a time-consuming process. In a survey, Five Tran and Dimensional Research found that analyses used 86 percent stale data, and 62 percent were required to wait on engineering resources several times [Arm+21]. Compared to the analytics systems where operational data was available for querying immediately, this is a step-down.

- Total cost of ownership: ETL costs and duplicated storage costs due to loading data into data lakes and data warehouses are difficult for the users, and added costs for carrying multiple tasks on other platforms as the commercial warehouses cause a vendor-lock-in due to proprietary data formats [Arm+21].

- Reliability: Achieving data consistency between the data warehouse and data lakes is cumbersome, costly, and complex as the data needs to be continuous ETLed between these systems and made available for high-performance Business Intelligence (BI) and decision support [Arm+21]. Continuous ETL steps also involve risks of failures or reduced data quality due to disparities between the two systems.

- Limited or no support for machine learning: The businesses derive their decisions about products and services through predictive modeling and recommendation systems. Machine learning systems and data management confluence have not worked out in the data warehouse architecture [Arm+21]. Warehousing works well for business intelligence as it loads a small amount of data but not for complex machine learning tasks that require working with large datasets using non-SQL code. Machine learning systems, such as PyTorch and TensorFlow don't work effectively with warehousing architecture. They could run on data lakes in open formats but would not have rich management features such as indexing, ACID transactions, and data versioning [NM22].

The above limitations could be addressed if the data lakes based on open data formats, such as ORC and Parquet, could provide high performance and management features of the data warehouse along with immediate and direct I/O from machine learning workloads. As more enterprises rely on advanced analytics and operational data, Lakehouse architecture seems fitting . Figure 1, presents the differences between the three data architectures.

This report illustrates a lakehouse implementation that exhibits efficient data management capabilities, such as reliable data management on data lakes, SQL performance, and support for data science and machine learning. It provides an overview of the TPC-DC benchmark detailing leading data lakehouse systems such as Apache Hudi, Apache

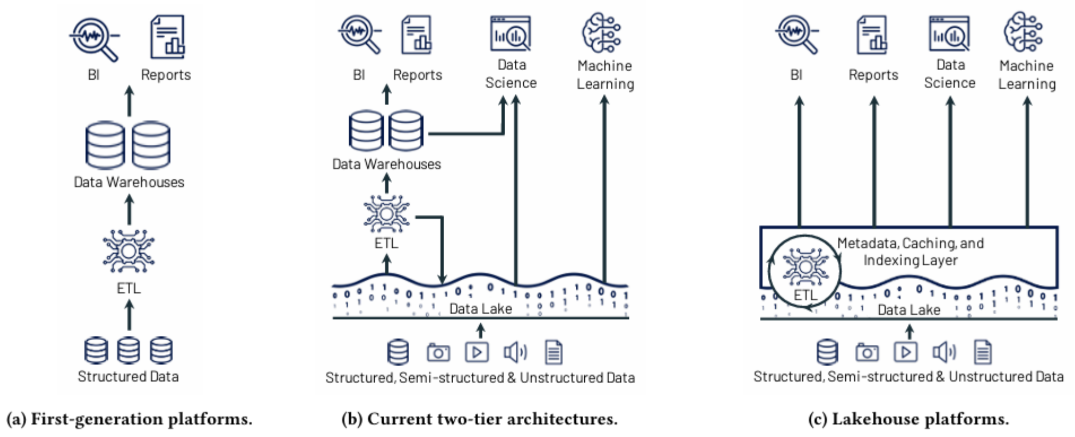
BI
Reports
Data Warehouses
ETL
Structured Data
(a) First-generation platforms.
Data Science
Machine Learning
Data Lake
Structured, Semi-structured & Unstructured Data
(b) Current two-tier architectures.
Metadata, Caching, and Indexing Layer
(c) Lakehouse platforms.

Figure 1: Differences between the data architectures [Arm+21]

Iceberg, and Delta Lake [Arm+20].

In the subsequent sections, data Lakehouse architecture, its technical components, capabilities, and diverse use cases are described, including the innovative concept of maintaining data as code. It presents a hands-on demonstration illustrating data warehousing functionalities on a data lake supporting open formats. This demonstration highlights capabilities such as machine learning reproducibility, referential integrity, data governance, and reliability for advanced analytics. Further sections present lakehouse benchmarks, future scope, and conclusion.

2 Data Lakehouse Architecture

Data Lakehouse systems seamlessly integrate the core benefits of both data warehouses and data lakes. They offer cost-effective, open storage coupled with robust data management and performance functionalities, including auditing, caching, ACID transactions, data versioning, and query optimization [Sch+23]. This architecture excels particularly in cloud environments, where storage and computing are decoupled, allowing diverse computing applications to operate on different computing nodes while directly accessing shared storage data [MHO23]. The data lakehouse characterizes the following:

- ACID Transactions: Data lakehouses ensure reliability and consistency in every transaction, such as INSERT or UPDATE, like a RDBMS-OLAP [Har+18] system. It implies that the system performs concurrent reads and writes without compromising data integrity.

- Open file format: At the core of a data lakehouse lies the storage of data in open file formats like Apache Parquet, ORC, etc., as well as table formats such as Apache Iceberg, Apache Hudi, and Delta Lake. This approach allows various analytical workloads to be performed by different engines, often from different vendors or providers, all on the same dataset. Importantly, it prevents the data from being locked into a proprietary format, ensuring flexibility and interoperability across different tools and systems [OH21].

- Data governance and quality: In a lakehouse system, there's a strong emphasis on data governance and quality. It achieves this by integrating proven best practices from the data warehousing domain [MHO23]. This includes implementing robust access control mechanisms and adhering to regulatory standards and requirements.
- Schema Management: In data lakehouses, there's assurance that a predefined schema is followed when adding new data. This ensures consistency and organization [MHO23]. Additionally, it allows for the schema to evolve gradually over time without requiring the entire dataset to be rewritten, thus minimizing disruption and cost.
- No copy architecture: A data lakehouse minimizes data duplication as the compute engine can access data directly from the source. The table format layer on the data lake architecture adds a structured model and ensures reliability and governance, enhancing data organization and management [MHO23].
- Scalability: A data lakehouse is structured around the concept of separating storage and compute functions. It leverages the cost-effective storage capabilities of a data lake, enabling the storage of various types of data (structured, loosely structured, etc.) and massive volumes, often reaching petabytes, in open file formats like Apache Parquet. This setup allows different analytical workloads, including batch processing, ad hoc SQL queries, and machine learning tasks, to operate on the stored data [Sin+22]. Importantly, each workload can be scaled independently according to specific needs and requirements.

3 Requirements for Data Lakehouse

A data lakehouse is an open file and table format that makes data accessible to several computing engines that support the format and enable agility. The lakehouse architecture breaks down associated technical components into distinct ones. In the following sections, the technical components and capabilities are described.

3.1 Technical Components

The data lakehouse architecture comprises several components, such as a file, table formats, compute engines, storage, and catalogs. An overview of the system is presented in Figure 2.

- Data Storage: In an open lakehouse architecture, the first component is storage. Here, data files arrive after being collected from different operational systems through ETL processes [MHO23]. Cloud object stores like Amazon S3, Azure Storage, and Google Cloud Storage are used for this purpose. They can store any type of data and can scale to handle extremely large volumes. Additionally, these systems are cost-effective compared to traditional data warehousing storage costs, making them a popular choice.
- Storage engine: The storage component is responsible for data management tasks, such as indexing, repartitioning, and compaction, which optimizes the data organization in cloud object storage and enhances the query performance [Iva].

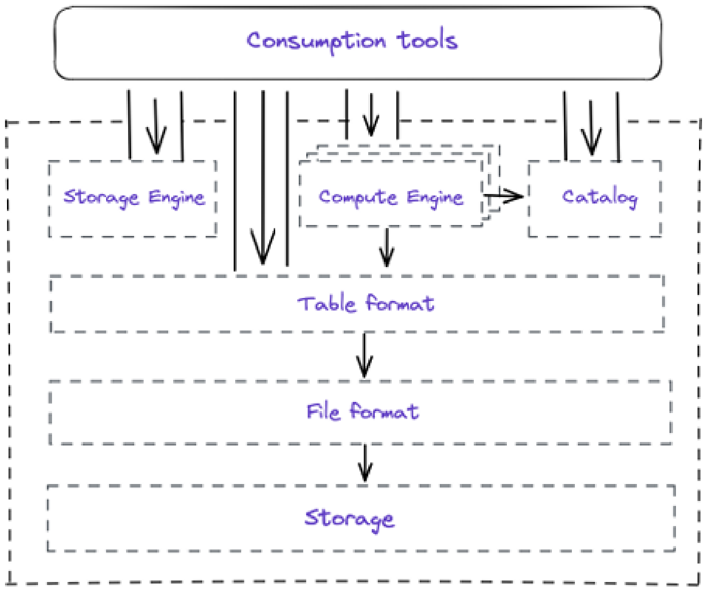

Figure 2: Data Lakehouse system technical components [MHO23]

- File format: In a data lakehouse setup, file formats play a crucial role as they store the raw data physically on object storage [Arm+20]. These formats, like Apache Parquet or JSON, are open and flexible, allowing data to be used by different engines for different tasks. For example, one engine might be great for business intelligence while another excels in machine learning. These file formats are usually column-oriented, which makes it easier to read and share data across various systems [Iva].
- Table format: A table format is the most essential component of the data lakehouse architecture. It is the metadata layer that facilitates management and file organization in the data lake thereby abstracting the complexity of the architecture [MHO23]. Different engines can simultaneously read and write on the same dataset using the APIs provided by table format enabling ACID transactions with consistency and atomicity. Table formats like Apache Hudi, Apache Iceberg, and Delta Lake enable time travel, partitioning, and schema evolution. As presented in Figure 3, in a Hive table format [For], all the files in the table's directories form the table's content, which makes it inefficient for updates, data in multiple partitions aren't changed safely, and running multiple jobs on the same dataset is not possible. It requires the users to be aware of the physical state of the table and the table statistics could be stale [Arm+21]. Apache Iceberg tacks all the files at the file level, which helps to maintain table consistency, faster querying, table evolution, and scalability [Arm+21].
- Catalog: The metastore or catalog is a crucial component in the lakehouse architecture. It enables efficient discovery and search within the lakehouse, tracks all the tables, and maintains the metadata information [MHO23].
- Compute Engine: This component in a data lakehouse architecture processes data and ensures read and write performance. Depending on the workload, compute engines could vary. Apache Spark [Wik] is suited for machine learning tasks, Apache Flink [Dat] is for streaming, and low-latency ad hoc SQL is for Dremio Sonar [Son].

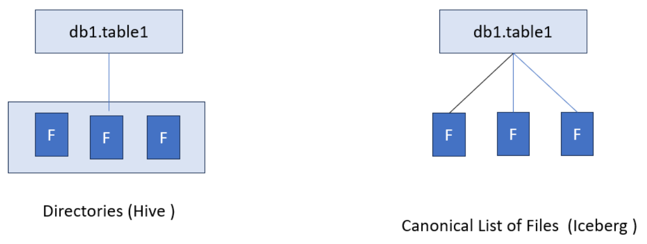

Figure 3: Hive (old) and Iceberg (new) table format

3.2 Capabilities of Data Lakehouse

The technical capabilities that ensure performant data warehousing features in a data lakehouse are as follows:

- Data governance and security: Data lakehouses store extensive data in raw formats in object storage. It is essential to ensure proper access and data governance and sensitive data information [Hel23]. The data catalog component in the lakehouse architecture ensures auditing, access control, and regulatory complaints.

- Low query latency: Compute engines use optimization techniques like partitioning, compaction, clustering, and indexing aided by table formats and query acceleration technologies to speed up queries in lakehouse architecture [MHO23]. It makes creating and scaling BI dashboards on cloud data lakes easier.

- ACID transactions: Lakehouse compute engines support multi-statement transactions, allowing operations like inserting or deleting rows across multiple tables. Formats like Apache Iceberg ensure ACID-compliant support, guaranteeing atomicity and consistency. Solutions like Project Nessie and lakeFS [lak] act as version controls for table formats and object stores, facilitating isolated and atomic changes. Lakehouse table formats like Apache Iceberg maintain a historical lineage of table states via snapshots, allowing querying prior table states and enabling transaction rollbacks to correct data when needed [Ice]. Similar to data warehousing systems, lakehouse table formats ensure safe concurrent writes to the same table by multiple users or engines. Concurrency models like optimistic concurrency control provide transactional guarantees across table formats such as Apache Iceberg, Hudi, and Delta Lake.

- High concurrency: Lakehouse platforms enable flexible scaling of workloads and management of concurrency levels while controlling compute resources to manage costs effectively [OH21]. In a lakehouse architecture, table formats like Apache Iceberg facilitate simultaneous reading and writing through mechanisms like optimistic concurrency control. Additionally, these table formats enforce ACID compliance, ensuring downstream applications remain unaffected by any errors from concurrent writes [MHO23].

- Workload management: Distinct engines in the lakehouse architecture are specialized for different tasks like creating interactive dashboards, machine learning, or ensuring physical isolation. Workload isolation within each engine is achieved through queue configuration, specifying attributes like memory limits and CPU priority [MHO23]. This setup optimizes workload management and resource allocation, allowing for efficient utilization of cluster resources with isolated workloads for different users and groups.

- Ad hoc queries: Compute engines allow users to run ad hoc queries directly on the data stored in the data lake and provide access to the entire dataset in the lake. This access helps make a comprehensive analysis [MHO23]. Some data lakehouse compute engines connect directly with popular BI tools like Tableau and Power BI, making it easy to ask ad hoc questions and conduct exploratory analysis. They also support raw SQL queries for flexibility.

- Separate storage and compute: The separation of storage and computing is a foundational principle of data lakehouse architecture. As an extensive repository for diverse workloads such as ad hoc SQL, ETL, and machine learning, a data lakehouse enables independent scaling of its storage and compute components [Arm+21].

- Schema and physical layout evolution: Modification of the internal structure of an existing table without excessive overhead or rewrites is a crucial capability of lakehouses [Arm+21]. Table formats like Apache Hudi and Apache Iceberg enable this by in-place schema evolution capabilities.

4 Implementing a Lakehouse

The motivation and the demonstration for a lakehouse are discussed in this section. Code samples for implementation are added in the Appendix. In the Practical demonstration section, the listings show a small snippet of the code.

4.1 Motivation: Addressing issues in data lakes and warehousing

The data lakehouse architecture revolutionizes data management by embracing an open data architecture. This approach stores data in open formats like Parquet and leverages table formats like Apache Iceberg within cloud storage, allowing diverse analytical engines to access and process data simultaneously. It is a flexible open architecture, supports existing compute engines optimized for specific tasks, and is easy to integrate with emerging technologies without vendor lock-in [Arm+21]. In traditional data management, data travels from source systems to a data lake. It often involves moving data into specialized systems like RDBMS-OLAP for specific tasks, which can be inefficient and resource-intensive. Transformations are made to the data to become usable for analysis. The lakehouse architecture streamlines this process by using advanced query acceleration and supporting direct querying on the data lake, drastically reducing the need for data duplication. Moreover, features like incremental materialized views further minimize unnecessary data replication [MHO23].

The challenges of managing multiple data copies, such as complex ETL processes, governance issues, and data drift, are significantly mitigated in a lakehouse architecture. It ensures data remains in open, accessible formats, ready for workloads from BI to machine learning, enhancing governance and reducing overhead. Furthermore, the data lakehouse introduces a code-based data management paradigm. An example: Project Nessie [Nes], used with table formats like Apache Iceberg, brings version control to data tables, offering remarkable control and visibility. This approach enables isolated data environments for testing or experimentation without impacting production data. The data lakehouse architecture offers a streamlined, flexible, and efficient framework for data management and allows enterprises to leverage their data assets effectively.

4.2 Practical Implementation of a Lakehouse

This section demonstrates the data management capabilities of Data Lakehouse and the novel use case that catalogs, like Project Nessie, maintain data as code to promote machine learning experimentation and reproducibility. The zero-copy architecture allows the same dataset to be used for machine learning and reporting tasks.

A local laptop lakehouse consists of the following components:

- Apache Spark: Used for streaming or ingesting data [Wik].
- Minio: S3 compatible storage layer [Min].
- Nessie Catalog: Enables git-like features for experimentation, data quality, and rollbacks [Nes]. It allows the dataset to be accessible across tools.
- Apache Iceberg: Table format for the data lakehouse. [Ice]

Docker Compose is used to orchestrate multiple services on the laptop. The docker-compose.yml and a .env file are used to define and run multi-container docker applications as in Listing 1. As this implementation is carried out on a local machine, the environ-

```yaml
version: "3.9"

services:
  minioserver:
    image: minio/minio
    ports:
      - 9000:9000
      - 9001:9001
```

Listing 1: Docker file to orchestrate services

ment variables are listed. In production environments, these variables would have to be encrypted. Listing 2 shows how to leverage Spark with Apache Iceberg and Nessie for managing and querying data within a lakehouse architecture.

After running the code block in Listing 2, the Spark Application setup can be verified when the output is 'Spark Running'. An iceberg table is created and minio stores data and metadata. For demonstration purposes, the sales dataset is used and the operations are performed on it. In the warehouse bucket, sales data, and metadata are created and various operations are carried on it. Nessie catalog allows git-like capabilities like maintaining different branches for updating, deleting, faster querying, and transactions. The data in these branches remain completely unaffected by the operations being carried on

```
import pyspark
from pyspark.sql import SparkSession
conf = (
    pyspark.SparkConf()
    .setAppName('app_name')
    .set('spark.jars.packages',
    'org.apache.iceberg:iceberg-spark-runtime-3.3_2.12:1.3.1,'
    'org.projectnessie.nessie-integrations:nessie-spark-extensions-3.3_2.12:0.67.0,'
    'software.amazon.awssdk:bundle:2.17.178,'
    'software.amazon.awssdk:url-connection-client:2.17.178')
    .set('spark.sql.extensions',
'org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions,'
'org.projectnessie.spark.extensions.NessieSparkSessionExtensions')
    .set('spark.sql.catalog.nessie', 'org.apache.iceberg.spark.SparkCatalog')
    .set('spark.sql.catalog.nessie.uri', NESSIE_URI)
    .set('spark.sql.catalog.nessie.ref', 'main')
    .set('spark.sql.catalog.nessie.authentication.type', 'NONE')
    .set('spark.sql.catalog.nessie.catalog-impl',
    'org.apache.iceberg.nessie.NessieCatalog')
    .set('spark.sql.catalog.nessie.s3.endpoint', AWS_S3_ENDPOINT)
    .set('spark.sql.catalog.nessie.warehouse', WAREHOUSE)
    .set('spark.sql.catalog.nessie.io-impl', 'org.apache.iceberg.aws.s3.S3FileIO')
    .set('spark.hadoop.fs.s3a.access.key', AWS_ACCESS_KEY)
    .set('spark.hadoop.fs.s3a.secret.key', AWS_SECRET_KEY)
)
```

Listing 2: Setup for efficient and scalable data management and analysis on a distributed computing platform like Apache Spark

them. Listing 3 demonstrates the operations.

An experiment to demonstrate data as code is carried out such that the same dataset can be used for decision-making and machine learning tasks while maintaining the dataset as two separate codebases and performing experimentation machine learning or other reporting or data analysis and visualization tasks. This approach helps teams work together smoothly, makes projects more reliable, and ensures that data handling is as transparent and error-free as possible. Essentially, Data as Code makes managing data in ML projects more systematic and efficient, leading to better outcomes and easier maintenance. Machine learning experiments for predictions, and novel use cases and be performed without maintaining any extra storage and costs. Listing 4 shows Machine Learning experimentation to understand market trends.

Various other data scientists or data analysts could branch out from the ml_branch and use the same dataset or modify it for different results. Both ml_branch and the bi_branch originate from the main branch. This demonstrates zero-copy data architecture to be used for multiple tasks and to reproduce the same task again if needed, a use case, which is inevitable in machine learning. Machine learning reproducibility enables the developers to understand data patterns and correlations effectively to employ a better or

```python
## CREATE AN ICEBERG TABLE FROM THE SQL VIEW
spark.sql("CREATE TABLE nessie.sales
"USING iceberg AS SELECT * FROM sales_data;").show()
## CREATE A BRANCH WITH NESSIE
spark.sql("CREATE BRANCH IF NOT EXISTS demo IN nessie")
## DELETE ALL RECORDS WHERE countryOfOriginCode = 'FR'
spark.sql("DELETE FROM nessie.sales WHERE COUNTRY = 'France'")
## Table History
spark.sql("SELECT * FROM nessie.sales.history").show()
## Listing manifest files
spark.sql("SELECT * FROM nessie.sales.files").show()
## Listing snapshots
spark.sql("SELECT * FROM nessie.sales.snapshots").show()
## Listing records at a particular timestamp (time travel)
spark.sql("SELECT * FROM nessie.sales TIMESTAMP AS OF '2024-02-05 06:10' ").show()
```

Listing 3: Data Management Operations in an Iceberg Table in a Lakehouse Architecture, Timetravel, Snapshots, Listing Manifest files and table history

new model while maintaining data quality and governance.

A group of developers, researchers, or scientists could work on the same dataset while only maintaining different branches and producing unique work that could be merged into the main branch if needed. The dataset can be modified and made suitable for decision-making tasks or any visualization tasks for the business to make effective decisions as presented in Listing 5.

As demonstrated in Figure 4, an ML experimentation task and a sales report for states in the USA can be developed by different teams using the same data without copying them again or storing another version of it as per their use case. Another set of researchers might be interested in finding out which features contributed to the principal components and performing more analysis as presented in Listing **??**, further branching from ml_branch and performing analysis is possible without copying data or performing the experiment again. This enables collaborative development without adding to extra costs of storage or quality maintenance.

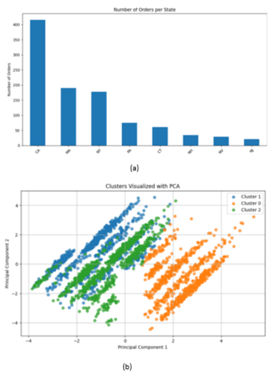

Figure 4: A sales report (a) and results of ML experimentation task (b)

```python
# Switching to ml_branch
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
spark.sql("USE REFERENCE ml_branch IN nessie")
ml_df = spark.read.format("iceberg").load("nessie.sales")
ml_df_pandas = ml_df.toPandas()
# Load the dataset
data = ml_df_pandas
# Preprocess the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(numeric_data)
# Perform K-means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(scaled_data)
```

Listing 4: Machine Learning Experimentation in the ml_branch: Employing K-means clustering to understand the sales of automobiles

```python
# Switching to bi_branch
spark.sql("USE REFERENCE bi_branch IN nessie")

bi_df = spark.read.format("iceberg").load("nessie.sales_filtered")
bi_stat = bi_df.toPandas()

# Count the number of orders per state
orders_per_state = bi_stat['STATE'].value_counts()
```

Listing 5: Create a report that demonstrates the sales in various states in the USA, in bi_branch

5 Benchmarking Lakehouses

This section evaluates three prominent lakehouse storage systems (Apache Hudi, Apache Iceberg, and Delta Lake) using the benchmarking toolkit LHBench and focuses on three critical aspects: data ingestion, efficiency, overall performance, and metadata access during query planning.

5.1 Setup

The experiments were conducted using Apache Spark on AWS EMR 6.9.0, with data stored in AWS S3 utilizing Delta Lake 2.2.0, Apache Hudi 0.12.0, and Apache Iceberg 1.1.0. The lakehouse systems and EMR were used in their default configurations, with the only adjustment being an increase in the Spark driver JVM memory to 4GB to prevent memory issues. It reflects the aim to evaluate lakehouse systems' readiness for a variety of workloads, from batch processing to business intelligence, without the need for specialized configuration. The tests utilized 16 AWS i3.2xlarge instances, each equipped with 8 vCPUs and 61 GiB of RAM.

5.2 Performance: Load and Query

Authors in [Jai+23] began by assessing the impact of the different lakehouse storage formats on data loading times and query performance, employing the TPC-DS benchmark with 3 TB of data loaded into Delta Lake, Hudi, and Iceberg. Subsequent query tests were conducted three times, reporting the median runtime for accuracy.

The findings revealed that while Delta and Iceberg had comparable loading times, Hudi's process was nearly ten times slower. Hudi's slower performance is due to its focus on keyed upserts, which require extra pre-load processing like checking key uniqueness and redistributing data. Query performance varied, with TPC-DS queries running 1.4 times faster on Delta Lake compared to Hudi, and 1.7 times faster than Iceberg [Jai+23]. A closer look at individual queries, such as Q90 and Q67 (which together account for a significant portion of the total runtime), showed that all three formats generated identical query plans, indicating that the performance differences were primarily due to data reading speeds [Jai+23]. For instance, Delta Lake significantly outpaced Hudi and Iceberg in reading large tables due to its more efficient file size management and columnar compression, which reduces overhead for large-scale data scans. However, there were exceptions. For smaller queries, such as Q68, metadata operations became the bottleneck. Hudi performed better in these instances due to its caching of query plans. Moreover, Iceberg's use of Spark's DataSource v2 API, as opposed to the v1 API used by Delta Lake and Hudi, led to occasional differences in query plan efficiency due to the API's immaturity and lack of certain optimization metrics, affecting performance negatively in some cases [Jai+23]. This comprehensive evaluation as shown in Figure 5 and Figure 6 aims to provide a clear understanding of the strengths and limitations of each lakehouse system, guiding users in selecting the most suitable solution for their specific needs.

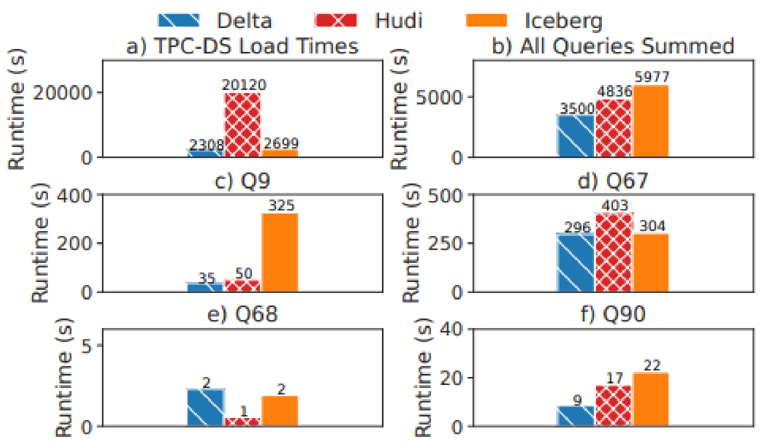

Figure 5: A comprehensive comparison of Delta Lake, Hudi, and Iceberg, focusing on their performance in loading and querying 3TB of TPC-DS data. The analysis encompasses various load and query times, highlighting significant disparities across four specific queries [Jai+23]

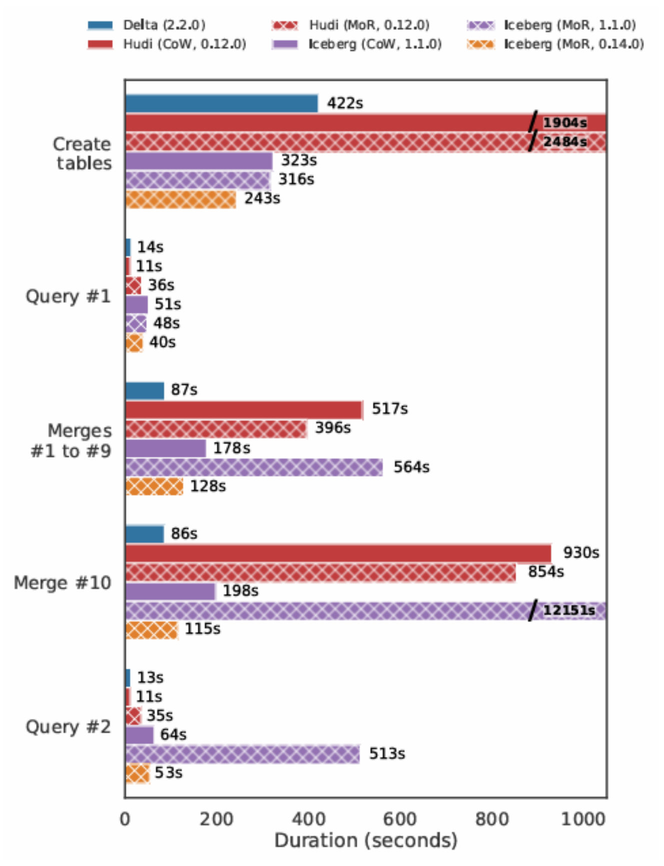

Figure 6: The performance of the 100GB TPC-DS incremental refresh benchmark. This benchmark involved loading data, executing five queries (Q3, Q9, Q34, Q42, and Q59), merging changes into the table ten times, and rerunning the queries. Notably, Hudi ran compaction in the tenth merge iteration, which is reported separately from iterations 1–9. Iceberg's results required a higher S3 connection pool size to mitigate timeout errors, while Delta and Hudi were tested with the default EMR configuration [Jai+23]

6 Future Scope

Developing efficient and easily accessible lakehouse systems and creating new data lake storage formats is a rich area for future work. These formats could offer enhanced flexibility, allowing lakehouse systems to implement data layout optimizations tailored to modern hardware capabilities [Arm+21]. In parallel to exploring new data formats, there are other encouraging areas for improving lakehouse systems. These include investigating various caching techniques, auxiliary data structures, and data arrangement strategies. Such innovations could significantly enhance how lakehouses handle large datasets, especially in cloud environments [NM22]. Another stimulating research direction focuses on leveraging serverless computing architectures for query processing. Optimizing lakehouse components, including storage, the metadata layer, and the query engine, for serverless environments could substantially reduce latency, offering a more efficient way to process queries [Hel23]. The evolution of data access interfaces, particularly for machine learning (ML) applications, represents another potential area for innovation [MHO23].

7 Conclusion

This report discusses the latest developments in data management architectures, specifically data lakes and data warehouses. Data lakehouse architectures integrate and improve upon the key features of traditional data warehouses, while also introducing new benefits. By utilizing cloud-based storage solutions, lakehouses can handle extensive and diverse datasets, ranging from structured to semi-structured, without incurring the high costs associated with cloud data warehouses. This innovative architecture combines robust data warehousing capabilities with the flexibility of open data lake formats, resulting in cutting-edge performance. This approach not only addresses several limitations experienced by data warehouse users but also creates a streamlined, unified data management platform. Given the growing amount of data stored in data lakes and the potential to simplify complex enterprise data ecosystems, it is becoming increasingly clear that lakehouse models will become the standard for businesses.

References

[Arm+20] Michael Armbrust et al. "Delta lake: high-performance ACID table storage over cloud object stores". In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3411–3424.

[Arm+21] Michael Armbrust et al. "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics". In: *Proceedings of CIDR*. Vol. 8. 2021, p. 28.

[BLO14] BLOG@CACM. "Why the 'data lake' is really a 'data swamp'." In: (2014).

[BN21] Vladimir Belov and Evgeny Nikulchev. "Analysis of Big Data Storage Tools for Data Lakes based on Apache Hadoop Platform". In: *International Journal of Advanced Computer Science and Applications* 12.8 (2021).

[Clo] Google Cloud. In: (). URL: `https://cloud.google.com/storage?hl=en`.

[Dat] Apache Flink-Stateful Computations over Data Streams. In: (). URL: `https://flink.apache.org/`.

[Err+23] Soukaina Ait Errami et al. "Spatial big data architecture: from data warehouses and data lakes to the Lakehouse". In: *Journal of Parallel and Distributed Computing* 176 (2023), pp. 70–79.

[For] Create Hive Table Format. In: (). URL: `https://spark.apache.org/docs/latest/sql-ref-syntax-ddl-create-table-hiveformat.html`.

[Fou] The Apache Software Foundation. "LanguageManual ORC". In: (). URL: `https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=31818911`.

[Gla] AWS Glacier. In: (). URL: `https://en.wikipedia.org/wiki/Amazon_S3_Glacier`.

[Had] Apache Hadoop. In: (). URL: `https://hadoop.apache.org/`.

[Har+18] Stavros Harizopoulos et al. "OLTP through the looking glass, and what we found there". In: *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 2018, pp. 409–439.

[HDF] Apache Hadoop HDFS. In: (). URL: `https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html`.

[Hel23] Fredrik Hellman. "Study and Comparsion of Data Lakehouse Systems". In: (2023).

[HM10] Terry Halpin and Tony Morgan. *Information modeling and relational databases*. Morgan Kaufmann, 2010.

[HZ22] Ahmed A Harby and Farhana Zulkernine. "From data warehouse to lakehouse: A comparative review". In: *2022 IEEE International Conference on Big Data (Big Data)*. IEEE. 2022, pp. 389–395.

[Ice] Apache Iceberg. In: (). URL: `https://iceberg.apache.org`.

[Iva] Raido Ivalo. "Data Lakehouse Architecture for Big Data with Apache Hudi". In: ().

[Jai+23] Paras Jain et al. “Analyzing and comparing lakehouse storage systems”. In: *Proc. Conf. Innov. Data Syst. Res.* 2023.

[KR11] Ralph Kimball and Margy Ross. *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, 2011.

[KW18] Pwint Phyu Khine and Zhao Shun Wang. “Data lake: a new ideology in big data era”. In: *ITM web of conferences*. Vol. 17. EDP Sciences. 2018, p. 03025.

[lak] lakeFS. “Scalable Data Version Control”. In: (). URL: `https://lakefs.io/`.

[MHO23] Dipankar Mazumdar, Jason Hughes, and JB Onofre. “The Data Lakehouse: Data Warehousing and More”. In: *arXiv preprint arXiv:2310.08697* (2023).

[Min] Minio. In: (). URL: `https://min.io/`.

[Nes] Project Nesssie. “Transactional Catalog for Data Lakes”. In: (). URL: `https://projectnessie.org/`.

[NM22] Athira Nambiar and Divyansh Mundra. “An overview of data warehouse and data lake in modern enterprise data management”. In: *Big data and cognitive computing* 6.4 (2022), p. 132.

[OH21] Dražen Oreščanin and Tomislav Hlupić. “Data lakehouse-a novel step in analytics architecture”. In: *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE. 2021, pp. 1242–1246.

[S3] Amazon S3. In: (). URL: `https://aws.amazon.com/s3/`.

[Sch+23] Jan Schneider et al. “Assessing the Lakehouse: Analysis, Requirements and Definition.” In: *ICEIS (1)*. 2023, pp. 44–56.

[Sin+22] Ismaila Idris Sinan et al. “A Comparison of Data-Driven and Data-Centric Architectures using E-Learning Solutions”. In: *2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS)*. IEEE. 2022, pp. 1–6.

[Son] Dremio Sonar. In: (). URL: `https://docs.dremio.com/current/sonar/`.

[Sto] Azure Data Lake Storage. In: (). URL: `https://azure.microsoft.com/en-us/products/storage/data-lake-storage`.

[Vas09] Panos Vassiliadis. “A survey of extract–transform–load technology”. In: *International Journal of Data Warehousing and Mining (IJDWM)* 5.3 (2009), pp. 1–27.

[Voh16] Deepak Vohra. In: *Apache Parquet. Apress, Berkeley, CA, 325–335. https://doi.org/10.1007/978-1-4842-2199-0*$_8$ (2016).

[Wik] Apache Spark - Wikipedia. In: (). URL: `https://en.wikipedia.org/wiki/Apache_Spark`.

A Code samples

```
version: "3.9"

services:
  minioserver:
    image: minio/minio
    ports:
      - 9000:9000
      - 9001:9001
    environment:
      MINIO_ROOT_USER: minioadmin
      MINIO_ROOT_PASSWORD: minioadmin
    container_name: minio
    command: server /data --console-address ":9001"

  spark_notebook:
    image: alexmerced/spark33-notebook
    container_name: notebook
    volumes:
      - ./warehouse:/home/docker/warehouse
      - ./notebooks:/home/docker/notebooks
      - ./datasets:/home/docker/datasets
    env_file: .env
    ports:
      - 8888:8888

  nessie:
    image: projectnessie/nessie
    container_name: nessie
    ports:
      - "19120:19120"

networks:
  default:
    name: iceberg_env
    driver: bridge
```

```
# AWS_REGION is used by Spark
AWS_REGION=us-east-1
# This must match if using minio
MINIO_REGION=us-east-1
# Used by pyIceberg
AWS_DEFAULT_REGION=us-east-1
# AWS Credentials (this can use minio credential, to be filled in later)
AWS_ACCESS_KEY_ID=WhjLMMR7QrlhFJplGGoe
```

```
AWS_SECRET_ACCESS_KEY=kxyA2czmeQPWFT1OoySNzLBwpfGn9nKnq2wYtJ9F
# If using Minio, this should be the API address of Minio Server
AWS_S3_ENDPOINT=http://minioserver:9000
# Location where files will be written when creating new tables
WAREHOUSE=s3a://warehouse/
# URI of Nessie Catalog
NESSIE_URI=http://nessie:19120/api/v1
```

```python
import pyspark
from pyspark.sql import SparkSession
import os

## DEFINE SENSITIVE VARIABLES
NESSIE_URI = os.environ.get("NESSIE_URI") ## Nessie Server URI
WAREHOUSE = os.environ.get("WAREHOUSE") ## BUCKET TO WRITE DATA TOO
AWS_ACCESS_KEY = os.environ.get("AWS_ACCESS_KEY") ## AWS CREDENTIALS
AWS_SECRET_KEY = os.environ.get("AWS_SECRET_KEY") ## AWS CREDENTIALS
AWS_S3_ENDPOINT= os.environ.get("AWS_S3_ENDPOINT") ## MINIO ENDPOINT

conf = (
    pyspark.SparkConf()
    .setAppName('app_name')
    .set('spark.jars.packages',
    'org.apache.iceberg:iceberg-spark-runtime-3.3_2.12:1.3.1,'
    'org.projectnessie.nessie-integrations:nessie-spark-extensions-3.3_2.12:0.67.0,'
    'software.amazon.awssdk:bundle:2.17.178,'
    'software.amazon.awssdk:url-connection-client:2.17.178')
    .set('spark.sql.extensions',
    'org.apache.iceberg.spark.extensions.IcebergSparkSessionExtensions,'
    'org.projectnessie.spark.extensions.NessieSparkSessionExtensions')
    .set('spark.sql.catalog.nessie', 'org.apache.iceberg.spark.SparkCatalog')
    .set('spark.sql.catalog.nessie.uri', NESSIE_URI)
    .set('spark.sql.catalog.nessie.ref', 'main')
    .set('spark.sql.catalog.nessie.authentication.type', 'NONE')
    .set('spark.sql.catalog.nessie.catalog-impl',
    'org.apache.iceberg.nessie.NessieCatalog')
    .set('spark.sql.catalog.nessie.s3.endpoint', AWS_S3_ENDPOINT)
    .set('spark.sql.catalog.nessie.warehouse', WAREHOUSE)
    .set('spark.sql.catalog.nessie.io-impl', 'org.apache.iceberg.aws.s3.S3FileIO')
    .set('spark.hadoop.fs.s3a.access.key', AWS_ACCESS_KEY)
    .set('spark.hadoop.fs.s3a.secret.key', AWS_SECRET_KEY)
)

## Starting Spark Session
spark = SparkSession.builder.config(conf=conf).getOrCreate()
print("Spark Running")

```

```
## Test Run a Query
spark.sql("CREATE TABLE IF NOT EXISTS nessie.test1 (name string)
USING iceberg;").show()
spark.sql("INSERT INTO nessie.test1 VALUES ('test');").show()
spark.sql("SELECT * FROM nessie.test1;").show()
```

```
## LOAD A CSV INTO AN SQL VIEW
csv_df = spark.read.format("csv").option("header", "true").load("../datasets/
sales_data_sample.csv")
csv_df.createOrReplaceTempView("sales_data")
## CREATE AN ICEBERG TABLE FROM THE SQL VIEW
spark.sql("CREATE TABLE IF NOT EXISTS nessie.sale
s USING iceberg AS SELECT * FROM sales_data;").show()
## QUERY THE ICEBERG TABLE
spark.sql("SELECT * FROM nessie.sales limit 10;").show()
# Demonstration of zero copy experimentation using nessie
## QUERY THE COUNT OF ENTRIES
spark.sql("SELECT Count(*) as Total FROM nessie.sales").show()
## CREATE A BRANCH WITH NESSIE
spark.sql("CREATE BRANCH IF NOT EXISTS demo IN nessie")
## SWTICH TO THE NEW BRANCH
spark.sql("USE REFERENCE demo IN nessie")
## DELETE ALL RECORDS WHERE countryOfOriginCode = 'FR'
spark.sql("DELETE FROM nessie.sales WHERE COUNTRY = 'France'")
## QUERY THE COUNT OF ENTRIES
spark.sql("SELECT Count(*) as Total FROM nessie.sales").show()
## SWITCH BACK TO MAIN BRANCH
spark.sql("USE REFERENCE main IN nessie")
## QUERY THE COUNT OF ENTRIES
spark.sql("SELECT Count(*) as Total FROM nessie.sales").show()
## MERGE THE CHANGES
spark.sql("MERGE BRANCH demo INTO main IN nessie")
## QUERY THE COUNT OF ENTRIES
spark.sql("SELECT Count(*) as Total FROM nessie.sales").show()
# Performing update
spark.sql("UPDATE nessie.sales SET PRODUCTLINE = 'Books' where COUNTRY = 'Norway'")
## QUERY THE ICEBERG TABLE
spark.sql("SELECT * FROM nessie.sales where COUNTRY ='Norway' limit 10;").show()
## Table History
spark.sql("SELECT * FROM nessie.sales.history").show()
## Listing manifest files
spark.sql("SELECT * FROM nessie.sales.files").show()
## Listing snapshots
spark.sql("SELECT * FROM nessie.sales.snapshots").show()
## Listing records at a particular timestamp (time travel)
spark.sql("SELECT * FROM nessie.sales TIMESTAMP AS OF '2024-02-05 06:10' ").show()
```

```python
# Switching to ml_branch
spark.sql("USE REFERENCE ml_branch IN nessie")
ml_df = spark.read.format("iceberg").load("nessie.sales")
ml_df_pandas = ml_df.toPandas()

import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = ml_df_pandas

# Drop non-numeric columns for simplicity
numeric_data = data.drop(columns=['ORDERDATE','STATUS', 'PRODUCTLINE', 'PRODUCTCODE',
'CITY', 'STATE', 'POSTALCODE', 'COUNTRY' 'DEALSIZE'])

# Preprocess the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(numeric_data)

# Perform K-means clustering
kmeans = KMeans(n_clusters=3, random_state=42)
clusters = kmeans.fit_predict(scaled_data)

# Add cluster labels to the original dataset
data['Cluster'] = clusters
# Apply PCA for visualization
pca = PCA(n_components=2)
pca_data = pca.fit_transform(scaled_data)

# Create a DataFrame for the PCA-transformed data
pca_df = pd.DataFrame(data=pca_data, columns=['PC1', 'PC2'])

# Add cluster labels to the PCA DataFrame
pca_df['Cluster'] = clusters

# Visualize the clusters
plt.figure(figsize=(10, 6))
for cluster in pca_df['Cluster'].unique():
    plt.scatter(pca_df.loc[pca_df['Cluster'] == cluster, 'PC1'],
                pca_df.loc[pca_df['Cluster'] == cluster, 'PC2'],
                label=f'Cluster {cluster}', alpha=0.7)

plt.title('Clusters Visualized with PCA')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.grid(True)
plt.show()
```

```
# Switching to bi_branch
spark.sql("USE REFERENCE bi_branch IN nessie")
iceberg_table_name = "nessie.sales"

# SQL query to create a new Iceberg table containing only entries from USA
query = f"""
CREATE TABLE nessie.sales_filtered
USING iceberg
AS
SELECT *
FROM nessie.sales
WHERE COUNTRY = 'USA'
"""

spark.sql("SELECT Count(*) as Total FROM nessie.sales_filtered").show()

# Execute the SQL query
spark.sql(query)

bi_df = spark.read.format("iceberg").load("nessie.sales_filtered")
bi_stat = bi_df.toPandas()

import pandas as pd
import matplotlib.pyplot as plt


# Count the number of orders per state
orders_per_state = bi_stat['STATE'].value_counts()

# Plot the graph
plt.figure(figsize=(10, 6))
orders_per_state.plot(kind='bar')
plt.xlabel('State')
plt.ylabel('Number of Orders')
plt.title('Number of Orders per State')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
# Switching to ml_branch
spark.sql("USE REFERENCE ml_branch IN nessie")

# Analyze the clusters
cluster_centers = scaler.inverse_transform(kmeans.cluster_centers_)
cluster_centers_df = pd.DataFrame(cluster_centers, columns=numeric_data.columns)
```

```python
print(cluster_centers_df)

# Get cluster labels for each data point
data['Cluster'] = clusters

# Print the centroids of each cluster
print("Cluster Centroids:")
print(cluster_centers_df)

# Analyze sample data points from each cluster
for cluster_label in range(3):
    print(f"\nData points in Cluster {cluster_label}:")
    cluster_data = data[data['Cluster'] == cluster_label].head(5)  # Displaying 5 dat
    print(cluster_data)
```