KTH ROYAL INSTITUTE
OF TECHNOLOGY

Degree Project in the Field of Technology Information Technology and the Main Field of Study Computer Science and Engineering

Second cycle, 30 credits

A comparative study of the Data Warehouse and Data Lakehouse architecture

PHILIP SALQVIST

Stockholm, Sweden, 2023

A comparative study of the Data Warehouse and Data Lakehouse architecture

PHILIP SALQVIST

Degree Programme in Information and Communication Technology
Date: December 10, 2023

Supervisor: Bo Peng
Examiner: Stefano Markidis
School of Electrical Engineering and Computer Science
Swedish title: En komparativ studie av Data Warehouse- och Data Lakehouse-arkitektur

Abstract

This thesis aimed to assess a given Data Warehouse against a well-suited Data Lakehouse in terms of read performance and scalability. Using the TPC-DS benchmark, these systems were tested with synthetic datasets reflecting the specific needs of a Decision Support (DSS) system. Moreover, this research aimed to determine whether certain categories of queries resulted in notably large discrepancies between the systems. This might help pinpoint the architectural differences that cause these discrepancies. Initial research identified BigQuery and Delta Lake as top candidates due to their exceptional read performance and scalability, prompting further investigation into both.

The most significant latency difference was noted in the initial benchmark using a dataset scale of 2 GB, with BigQuery outperforming Delta Lake. As the dataset size grew, BigQuery's latency increased by 336%, while Delta Lake's went up by just 40%. However, BigQuery still maintained a significant overall lower latency across all scales. Detailed query analysis showed BigQuery excelling especially with complex queries, those involving extensive aggregation and multiple join operations, which have a high potential for generating large intermediate data during the shuffle stage. It was hypothesized that some of the read performance discrepancies could be attributed to BigQuery's in-memory shuffling capability, whereas Delta Lake might spill intermediate data to the disk. Delta Lake's hardware utilization metrics further supported this theory, displaying a trend where peaks in memory usage and disk write rate coincided with queries showing high discrepancies. Meanwhile, CPU utilization remained low. This pattern suggests an I/O-bound system rather than a CPU-bound one, possibly explaining the observed performance differences. Future studies are encouraged to explicitly monitor shuffle operations, aiming for a more rigorous correlation between high-discrepancy queries and data spillage during the shuffle phase. Further research should also include larger dataset sizes; this thesis was constrained to a maximum dataset size of 64 GB due to limited resources.

Keywords

Data-Intensive Computing, Data Lakehouse, BigQuery, Delta Lake, Data storage system, Data Lakehouse architecture

Sammanfattning

Denna uppsats undersökte ett givet Data Warehouse i jämförelse med ett lämpligt Data Lakehouse med fokus på läsprestanda och skalbarhet. Med hjälp av TPC-DS benchmark testades dessa system med syntetiska dataset som speglade kundens specifika behov. Vidare syftade forskningen till att avgöra om vissa kategorier av queries resulterade i märkbart stora skillnader mellan systemen. Detta för att identifiera de teknologiska aspekter hos systemen som orsakar dessa skillnader. Den inledande litteraturstudien identifierade BigQuery och Delta Lake som toppkandidater på grund av deras läsprestanda och skalbarhet, vilket ledde till ytterligare undersökning av båda.

Den mest påtagliga skillnaden i latens noterades i den initiala jämförelsen med ett dataset av storleken 2 GB, där BigQuery presterade bättre än Delta Lake. När datamängden skalades upp, ökade BigQuery's latens med 336%, medan Delta Lakes ökade med endast 40%. Dock bibehöll BigQuery en avsevärt lägre total latens för samtliga datamängder. Detaljerad analys visade att BigQuery presterade särskilt bra under komplexa queries som involverade omfattande aggregering och flera join-operationer, vilka har en hög potential för att generera stora datamängder under shuffle-fasen. Det antogs att skillnaderna i latens delvis kunde tillskrivas BigQuery's in-memory shuffle-kapacitet, medan Delta Lake riskerade att spilla data till disk. Delta Lakes hårdvaruanvändning stödde denna teori ytterligare, där toppar i minnesanvändning och skrivhastighet till disk sammanföll med queries som visade höga skillnader, samtidigt som CPU-användningen förblev låg. Detta mönster tyder på ett I/O-bundet system snarare än ett CPU-bundet, vilket möjligen förklarar de observerade prestandaskillnaderna. Framtida studier uppmuntras att explicit övervaka shuffle-operationer, med målet att mer noggrant koppla queries som uppvisar stora skillnader med dataspill under shuffle-fasen. Ytterligare forskning bör också inkludera större datamängdstorlekar; denna avhandling var begränsad till en maximal datamängdstorlek på 64 GB på grund av begränsade resurser.

Nyckelord

Data-intensiv databehandling, Data Lakehouse, Data Lakehouse-arkitektur, BigQuery, Delta Lake, Datalagringssystem

Acknowledgments

First and foremost, I would like to thank my supervisor at KTH, Bo Peng, for contributing with her scientific experience and guidance throughout the thesis. And finally, I would like to thank my examiner Stefano Markidis for overseeing the thesis work at large.

Stockholm, December 2023
Philip Salqvist

Contents

List of Figures

List of Tables

List of acronyms and abbreviations

ACID	Atomicity, Consistency, Isolation, Durability
AWS	Amazon Web Services
BI	Business Intelligence
BSC	Barcelona Supercomputing Center
CAP	Consistency, Availability, and Partition tolerance
CoW	Copy on Write
CTE	Common Table Expression
DBMS	Database Management System
DL	Data Lake
DLH	Data Lakehouse
DSS	Decision Support
DW	Data Warehouse
EC2	Amazon Elastic Compute Cloud
ELT	Extract Load Transform
ETL	Extract Transform Load
ETLed	Extracted, Transformed and Loaded
GCC	Google Cloud Compute
GCP	Google Cloud Platform
GFS	Google File System
ML	Machine Learning
MoR	Merge on Read
S3	Amazon Simple Storage Service
TPC	Transaction Processing Performance Council
TPC-DS	Transaction Processing Performance Council Decision Support
TTDA	Two-tier Data Architecture
VW	Virtual Warehouse

Chapter 1

Introduction

1.1 Background

1.1.1 Research

Database Management Systems (DBMSes) have gone through several revolutions in recent decades. One such revolution was the introduction of Data Warehouses (DWs), which was introduced in the late 1980s by IBM researchers Berry Devlin and Paul Murphy [1]. The story of data warehousing began with helping businesses to gather analytical insights and Decision Support (DSS). DWs were advantageous in several ways, one being their promise to offer Atomicity, Consistency, Isolation, Durability (ACID) transactions. However, the wide adoption of modern technologies has led to a dramatic increase in data collection. This data revolution creates new demands on DBMSes, especially when dealing with issues such as data volume, velocity, and variety [1].

To address these issues, Data Lakes (DLs) were introduced, which offered scalable data storage for low cost and unstructured data support. Unfortunately, this kind of technology was limited in providing ACID transactions as well as performance. To offer these necessary DSS capabilities, data has to be Extracted, Transformed and Loaded (ETLed) to a DW. While this type of Two-tier Data Architecture (TTDA) has been dominant for big data storage during the 2010s, new issues concerning reliability and data staleness were introduced. [2] The Data Lakehouse (DLH) architecture [2] on the other hand, uses a meta layer on top of a DL to offer (i) reliable data management, (ii) support for machine learning and data science, and (iii) high-performance ACID transactions.

1.2 Research Question

How does the latency of the selected DW and DLH vary under different dataset scales, which category of queries contribute to the highest discrepancy in read performance between the systems, and why? The following subquestions should be answered for the research question to be addressed:

RQ1 How does the latency of the DLH and DW change as the data set size increases?

RQ2 What are the patterns of discrepancy in latency between the DLH and DW when performing complex query operations?

RQ3 Are there specific categories of queries where the discrepancies in performance between the DLH and DW are particularly pronounced? If so, what could be the reasons behind these discrepancies?

1.3 Purpose

The objective of this thesis is to compare a given DLH with a chosen DW. The research investigates read performance under different request workloads and scales, focusing on measuring latency for queries relevant to DSS use cases. The results are relevant to the distributed systems field and can be useful for producers of similar systems being investigated in thesis. This is also an opportunity for a master's student in Computer Science, specializing in Software Engineering, to deepen their knowledge in this research area.

1.4 Delimitations

The research of this thesis is focused on investigating DLHes and DWs specifically, and not DBMS systems in general. The research is also only considering the read performance of these technologies, since DSS operations mainly revolves around informational retrieval. Furthermore, to get a more complete view of read performance, throughput should be investigated as well while this thesis only considers latency. The scaling of data set sizes are limited by the costs associated with it. The goal is to present a pattern when scaling the systems which one could simply use to extrapolate for even larger data set sizes.

Chapter 2

Background

2.1 Data Architectures

2.1.1 On-premise Data Warehouse

The on-premise DW is a large, centralized repository of structured and organized data that is used for reporting, analytics, and Business Intelligence (BI). It is designed to support decision-making by providing a comprehensive view of an organization's historical and current data. DWs typically store data from multiple sources, such as transactional databases, and transform the data into a common format to enable analysis and reporting. The data written to these DWs is written using schema-on-write. Schema-on-write is a data processing approach that involves defining a data schema upfront and then enforcing that schema when data is written to a database or data storage system. With schema-on-write, data is structured and validated before it is stored, according to the Extract Transform Load (ETL) approach, ensuring that it conforms to a predetermined structure. This is in part what ensures that the data is optimized for BI analytics. [2] [1] Furthermore, DWs are especially fit to perform DSS operations because of its ability to ensure ACID properties as well as providing high SQL performance. Data warehouses employ various methods to achieve optimal performance, including storing frequently accessed data on high-speed devices such as SSDs, maintaining statistics to improve query optimization, creating efficient access methods such as indexes, and optimizing the data format and compute engine to work in tandem. These techniques enable data warehouses to deliver faster query response times, handle larger data volumes, and support complex analytical queries. However, this generation of data warehouses originally coupled storage and compute,

which makes storage less scalable since the computational resources have to scale together with it. [2]

2.1.2 Data Lake

A DL is a large storage repository that enables organizations to store massive amounts of raw data in a flexible and scalable manner. Unlike traditional DWs, which typically require data to be structured and processed before it can be stored, DLs allow organizations to store data in its native format, without the need for prior processing or modeling. This makes it easier to store and manage large volumes of unstructured or semi-structured data, such as text, images, videos, and sensor data. [2] [3] Data lakes often use a schema-on-read storage format, where data is structured and validated after it is stored, according to the Extract Load Transform (ELT) approach.

However, data lakes also have some limitations. Because data lakes are designed to store raw data in an open schema-on-read format, they do not provide the same level of data consistency and control as traditional data warehouses. This can make it more challenging to ensure data quality and accuracy, and to maintain compliance with data governance and regulatory requirements. Additionally, data lakes typically do not provide support for ACID transactions or high-performance SQL queries, which are important features of many DSS systems.

2.1.3 Two-tier Data Architecture

Traditional on-premise DWs and DLs comes with it's respective inherent drawbacks. The traditional DW can't handle the large scale of modern unstructured big data generation, while DLs on the other hand can't provide ACID properties and high performance SQL queries that are necessary parts of a DSS system.

The solution to these trade-offs has in part been to separate storage and compute in a TTDA. In this manner, data can be stored in a highly scalable and cheap DL, while the computing resources can be scaled up and down depending on demand, independent of the storage size. Meanwhile, data can be ETLed into a separate DW, where appropriate ACID properties and high performance SQL queries are enabled.

However, while the TTDA solve some of the problems that arose due to high scale unstructured big data generation, new problems with this architecture have emerged and been identified in previous research [2]:

1. `Total cost of ownership.` While DL storage is cheap, some studies indicate that the total cost of ownership might be relatively high due to the doubled storage cost for data being copied into the DW, as well as the continuous ETL processing that is needed.

2. `Reliability.` Maintaining consistency between the DL and DW is a challenging and expensive task. It requires ongoing engineering efforts to ETL data between the two systems and ensure that it can support high-performance DSS. Moreover, every step of this process carries the risk of errors or issues that can lead to data quality problems, such as variations in the underlying engines of the DL and DW.

3. `Data staleness.` Compared to the DL, the information stored in the DW is often outdated, and it can take days to load new data. This represents a regression from the on-premise generation of analytics systems, which allowed for immediate access to new operational data. According to a study conducted by Dimensional Research and Fivetran [4], a large percentage of analysts (86%) rely on data that is no longer up-to-date, and a significant portion (62%) frequently experience delays waiting for engineering resources to become available.

4. `Limited support for advanced analytics.` Despite significant research on the integration of Machine Learning (ML) and data management, leading ML systems like TensorFlow, and PyTorch do not perform well on warehouse data. Unlike BI queries that extract small amounts of data, ML systems require processing of large and complex non-SQL programming. Warehouse vendors suggest exporting the data to files, but this approach adds complexity and further delays the data's freshness. Alternatively, users can run ML systems on DL data in open formats, but this results in the loss of key warehouse management features like ACID transactions, data versioning, and indexing.

To summarize, traditional on-premise DWs and DLs have inherent drawbacks that limit their ability to handle modern unstructured big data. A TTDA has emerged as a solution to these trade-offs by separating storage and compute, but this approach introduces new challenges such as high total cost of ownership, reliability issues, data staleness, and limited support for advanced analytics.

2.1.4 Data Lakehouse

Since the DLH is a fairly recent concept, there are still differing opinions in how one should conceptualize its architecture. E.g., Tomislav et al. [5] defines the DLH architecture as a combination of the Data Lake and Data Warehouse to provide the desirable features of both. However, in this thesis the term TTDA is used to draw a distinction between this architecture and the DLH. Rather, the DLH is defined as by Armbrust et al. [2]. At a high level by this definition, the DLH architecture includes three main components: a DL, a **metadata layer**, and a **compute layer**.

The DL is a centralized repository that stores large volumes of structured, unstructured and semi-structured data, usually in a standard open file format such as Apache Parquet or ORC [6] [7]. It is typically implemented using a low-cost cloud object store like Amazon S3, Azure Data Lake Storage, or Google Cloud Storage [8] [9] [10]. Data is stored in its native format, without any transformations, and can be accessed by different data processing engines. This type of storage solution enables flexibility and high scalability at low cost.

The **metadata layer** in the DLH architecture is a critical component that helps to provide a unified view of the data stored in the DL. This layer enables the implementation of ACID transactions and other management features, such as data quality enforcement and governance. Metadata layers like Delta Lake, Apache Iceberg, and Apache Hudi [11] [12] [13] are designed to provide a unified view of data and enable organizations to manage metadata and ensure data governance. They are easy to adopt, and organizations are rapidly adopting them as they provide similar or better performance to raw Parquet/ORC DLs while adding useful management features like transactions, zero-copy cloning, and time travel to past versions of a table. The metadata layer is also a natural place to implement data quality enforcement and governance features such as access control and audit logging. [2]

While a metadata layer can provide management capabilities, it alone is insufficient to achieve optimal SQL performance. To achieve top-of-the-line performance, data warehouses utilize various techniques such as storing frequently accessed data on fast storage devices like SSDs, keeping up-to-date statistics, constructing effective access methods such as indexes, and co-optimizing the data format and compute engine. To achieve similar capabilities, the DLH typically caches hot data on faster devices, while optimizing queries made on cold data using a combination of data layout optimizations and auxiliary data structures such as zone maps. [2]

The **compute layer** is another critical component of the DLH architecture,

which provides a unified processing platform that enables users to perform different types of workloads on the data, such as batch processing, ACID SQL queries, stream processing, and machine learning. One of the key benefits of the compute layer is that it allows users to perform processing on the data in its original place, without the need to ETL data to a separate DW. This reduces the time and complexity of data processing and enables users to derive insights and make decisions in near real-time.

These features of the DLH architecture has the potential to solve some of the issues related to the TTDA, including total cost of ownership, reliability, data staleness, and the limited support for advanced analytics.

2.2 Data Processing

In the coming section, the topic of distributed data processing is presented briefly. Since this is a largely broad topic, the focus is limited to the Consistency, Availability, and Partition tolerance (CAP) theorem, as well as map, shuffle, and reduce, since these are the most relevant subjects for this thesis.

2.2.1 CAP theorem

The CAP theorem, often referred to as Brewer's theorem after its originator Eric Brewer, presents a fundamental principle governing the behavior and design of distributed data systems. The theorem postulates that there are three primary attributes that these systems can provide: Consistency, Availability, and Partition Tolerance. However, the central tenet of the CAP theorem asserts that a distributed system can only guarantee two out of these three attributes at any given time. [14]

- Consistency: All nodes (or data storage components) in the system see the same data at the same time. This ensures that there is a single up-to-date copy of the data, regardless of which node is accessed. [14]
- Availability: The system is operational and responsive to requests at all times. This means that every request made to the system receives a response, without the guarantee that it contains the most recent data version. [14]
- Partition Tolerance: The system continues to function even when network partitions (communication breakdowns between nodes) occur.

Essentially, even if a part of the network goes down, the system as a whole remains operational. [14]

2.2.2 Map, Shuffle, Reduce

The Map, Shuffle, and Reduce paradigm underpins the core logic of many distributed processing frameworks, with the most famous being the MapReduce framework introduced by Google. This paradigm facilitates the processing and generation of vast data sets using a distributed cluster. [15]

- Map: At this phase, the input data is processed and transformed into a set of intermediate key-value pairs. Individual nodes in the cluster execute the map function, working on different subsets of the input data. As a result, data becomes organized based on some attributes or properties, which are used as the keys. [15]
- Shuffle: Post the map phase, the shuffle process redistributes the intermediate key-value pairs. The goal is to organize these pairs in such a way that all values for a given key are grouped together. This often involves transferring data between nodes in the cluster, ensuring that all values for specific keys end up on the same node. [15]
- Reduce: Once the data is shuffled and sorted by keys, the reduce phase commences. Here, the intermediate key-value pairs are processed to generate a smaller set of values. Typically, the reduce function aggregates, filters, or combines the data in some manner. The result is a consolidated data set that summarizes or represents the larger input. [15]

2.3 Data Warehouse Selection

In the following section, all DWs being presented adhere to a modern TTDA. However, these will simply be referred to as DWs.

2.3.1 Redshift

The Amazon Web Services (AWS) Redshift architecture consists of two main components: the control plane and the data plane. The control plane provides workflows to monitor and manage the database. The data plane includes the database engine, which provides data storage and SQL execution. The

Redshift data plane uses a cluster architecture to provide scalable storage and compute resources for processing large data workloads. The cluster consists of two types of nodes: leader nodes and compute nodes. [16]

The cluster leader node is the entry point for all client connections to the Redshift cluster. It acts as a coordinator for all the compute nodes in the cluster, managing the distribution of queries, data, and workload across the compute nodes. [16]

The compute nodes store and process data in parallel across multiple nodes to provide high performance and scalability. Each compute node contains one or more slices, and each slice is a portion of the total data stored in the cluster. The number of compute nodes and slices can be increased or decreased depending on the workload demands. [16]

Finally, it should be noted that each slice stores data in a column-oriented manner. This approach provides significant performance benefits for data warehousing workloads because queries typically involve aggregations and analysis of specific columns rather than entire rows. Each column within each slice is encoded in a chain of one or more fixed size data blocks. Furthermore, data blocks are replicated both within the database instance and within Amazon Simple Storage Service (S3) as backup to provide durability. [16]

2.3.2 BigQuery

BigQuery is a distributed DW using columnar storage and a tree architecture to perform queries with high efficiency, throughput, and compression ratio. It is a serverless technology, meaning that the underlying infrastructural management and resource allocation is abstracted away from the user. Rather, such infrastructural and resource management is handled automatically based on demand. It should be noted that the computing resources allocated to a given BigQuery instance are measured as a unit called slots, rather than CPUs. BigQuery is the publicly available implementation of Dremel [17], which have been used internally at Google since 2006 [18]. Since BigQuery is implemented on the same underlying architecture as Dremel [18], the background of BigQuery will be performed by investigating the architecture of Dremel in the coming subsection.

2.3.2.1 Dremel

Dremel is a distributed query engine designed to efficiently process large-scale datasets. It was developed by Google to meet the growing demand for a fast

and scalable solution to analyze massive datasets. The architecture of Dremel is based on several core components, including a distributed file system, a query engine (queries are executed using a tree architecture), and a columnar storage system.

Dremel is designed to work with Google's distributed file system, known as Google File System (GFS). GFS is a highly scalable and fault-tolerant file system that allows Dremel to read and write data in parallel across multiple nodes in the cluster. This distributed file system ensures that data is always available, even in the event of node failures. [17]

The query engine of Dremel is responsible for processing SQL-like queries against the data stored in the distributed file system. The query engine is highly parallelized and distributed, allowing it to process queries in a highly scalable manner. Dremel's query execution engine uses a tree architecture with a root node that distributes the query to multiple intermediate nodes. Each intermediate node performs a specific operation on the data, and the process continues until the leaf nodes are reached, which gathers the final result of the query by reaching the storage layer or access the data on local disk. [17]

Dremel stores data in a columnar format, rather than the traditional row-based format. This allows for highly efficient data access, as queries can skip over irrelevant columns, and data can be read in a highly compressed format. The columnar storage system also enables highly efficient processing of aggregate functions, which are commonly used in DSS queries. [17]

Finally, Dremel capitalizes on in-memory data structures, specifically storing intermediate data in memory instead of writing it to disk. This in-memory shuffle strategy accelerates the process of reorganizing data between stages of execution. When vast datasets are queried, such speed-ups can drastically reduce latency. [19]

2.4 Data Lakehouse Selection

2.4.1 Delta Lake

Delta Lake is an open-source data management system that is built on top of Apache Spark [20], offering a unified data management system that can handle both batch and streaming data. The Delta Lake architecture consists of three core components, the delta storage layer, the delta table, and the delta engine. [21]

In the delta storage layer, data is stored in a DL, enabling storage at low cost [21]. Moreover, Delta Lake is structured around the concept of a write-

ahead transaction log, which comprises a sequence of immutable, append-only data files that record all changes made to a Delta Table. The transaction log provides various benefits, including ACID transactions, version control, and time travel. Furthermore, Delta Lake is designed so that all metadata is kept in the underlying DL, and transactions are achieved using optimistic concurrency protocols against the DL, enabling storage and compute to be scaled separately. Additionally, Delta Lake utilizes this architecture to offer advanced features like upsert operations, caching, audit logs, schema enforcement and schema modification. [22]

The table content is stored in an open column-oriented file format using Apache Parquet [6]. Using an established storage format enables Delta Lake to integrate with existing processing engines such as Spark SQL and Structured Streaming using Apache Spark's data source API [23] [24] [20]. Moreover, data is updated using a Copy on Write (CoW) strategy [25].

The delta engine increases the efficiency of parallel processing through Apache Spark [20] by employing various methods such as optimizing data layout with Z-Order, a clustering technique that works in multiple dimensions. Additionally, the delta engine offers the ability to compress smaller files while still adhering to ACID compliance. The optimization process involves a bin-packing or Z-ordering algorithm. [21] [22]

When operations like re-partitioning, aggregating, or joining data are performed in Spark, data shuffling across partitions and nodes becomes necessary. Given that this operation can be network and I/O intensive, Spark uses a combination of in-memory processing and disk storage to manage shuffled data efficiently. Thereby, Delta Lake uses in memory shuffle by default, but it can spill data to disk if there is not enough RAM. [26]

2.4.2 Apache Iceberg

Apache Iceberg is an open-source system that supports the DLH architecture by providing a columnar table format on top of object storage systems. It ensures transactional and consistent data by collecting metadata in manifest lists and manifest files related to the tables' snapshots. The columnar storage format together with the system's ability to divide the data into partitions, allows for faster queries and more efficient processing of large data sets. [21] [13] Moreover, both CoW and Merge on Read (MoR) are provided as data updating strategies [25].

The design of Apache Iceberg's metadata management is based on the table's snapshots, which are created with each commit to the table. The system

uses a tree structure to track data files, building a link between a snapshot, its manifest list, the manifest files, and the data files. The manifest list contains the list of the manifest files and information about each of these files, such as the range of values covered by the files related to the manifest file. The manifest file contains metadata and statistics related to its files. [21] [13]

In summary, this design pattern enables Apache Iceberg to implement the transactional layer on top of the object storage, providing various capabilities such as ACID support, schema evolution, hidden partitions, partition evolution, time travel, and rollback. [21] [13]

2.4.3 Apache Hudi

Apache Hudi is a data processing framework designed to optimize upserts, deletes, and incremental processing over data file systems. Similar to Delta Lake and Apache Iceberg, Apache Hudi is implemented as a layer on top of the DL. The framework prioritizes stream data ingestion and capturing data changes to facilitate quick ingestion and analysis of streamed data. This is useful in scenarios where only recently ingested data is required for analysis. Incremental processing ensures that only new data is processed, increasing query performance while avoiding reprocessing old data. [21] [12]

Apache Hudi stores tables in directories, with each directory representing the table's partitions. These partitions contain file groups divided into slices, where each slice contains data in parquet format. Two types of update strategies are provided by Apache Hudi, with the trade-off depending on the workload's nature: CoW and MoR [25]. [21] [12]

To support the LakeHouse architecture design, Apache Hudi offers several features, including fast upserts and deletes enabled by its index, ACID compliance, and optimistic concurrency control to handle concurrent writes. Data layout optimization is achieved through file-level statistics, and support for rollbacks is also provided. Additionally, Apache Hudi enables lineage tracking. [21] [12]

2.4.4 Snowflake

Snowflake was originally considered a data warehouse [27]. However, the platform has recently been started to be considered as the first DLH, only created before the term DLH was coined [28] [5]. The architecture of Snowflake can be divided into three layers: Data Storage layer, Virtual Warehouse layer, and Cloud Services layer.

The data storage layer is designed to be highly scalable and cost-effective, while also supporting a variety of data formats. Snowflake stores data in a columnar format, which is optimized for analytical processing and compression. Data is stored in cloud-based object storage, such as S3, and is automatically partitioned and scaled as needed. Snowflake also includes built-in data replication and disaster recovery capabilities, ensuring high availability and reliability. [29]

The Virtual Warehouse (VW) is the computing resource in the Snowflake architecture. It works as an abstraction layer that presents clusters of Amazon Elastic Compute Cloud (EC2) instances as a single VW to the user. The worker nodes of each VW are individual EC2 instances that can be created, resized or destroyed at any point in time. Each query runs on exactly one VW, ensuring strong isolation for queries. Each user can have multiple VWs running concurrently, with each VW running multiple queries. Local caching and file stealing techniques are used to improve the hit rate and avoid redundant caching of individual table files across worker nodes of a VW. The corresponding SQL execution engine reaches high performance due to columnar storage, vectorized execution (by avoiding materialization of intermediate results in the execution pipeline), and push-based execution which enhances cache efficiency. [29]

Finally, the cloud services layer provides various services such as security, metadata management, query optimization, and workload management. This layer is responsible for managing user access, ensuring data security, and optimizing query performance. [2]

2.5 Previous Benchmarks

2.5.1 TPC-DS

The Transaction Processing Performance Council Decision Support (TPC-DS) benchmark is a standardized benchmark for measuring the performance of decision support systems, which are used for complex queries and data analysis. The TPC-DS benchmark is designed to simulate a real-world decision support workload by generating a set of SQL-based queries that represent typical DSS operations. [30]

Furthermore, it is designed to be both scalable and comprehensive, with a wide range of query types and complexity levels. The benchmark includes 99 queries that cover a variety of DSS operations, such as complex reporting, ad-hoc analysis, and data mining, as well as 12 data maintenance queries. The

queries are designed to simulate different types of decision support operations, such as customer analysis, inventory management, and supply chain analysis. The schema of the relational database is in third normal form, and mimics a rather standard schema of a retail product supplier, holding information such as customer, product, and order data. [30]

Moreover, the benchmark also includes a data generator, which can generate large volumes of data to simulate a realistic decision support workload. The data generator allows users to specify the size of the dataset, as well as other parameters such as the number of customers, products, and transactions. The data set can scale from a size of 1 GB up to 100 TB in total. [30]

In terms of performance metrics, the benchmark measures query throughput, query latency, and data loading time. The benchmark also includes a power metric, which measures the overall performance and energy efficiency of the system. [30]

2.5.2 Delta Lake, Apache Hudi, and Apache Iceberg

The benchmarking study performed by Jain et al. [25] compares the performance of three different lakehouse storage systems: Apache Hudi, Delta Lake, and Apache Iceberg [12] [11] [13]. The study focuses on three key areas of comparison: end-to-end performance, the performance of different data ingestion strategies, and the performance of different metadata access strategies during query planning.

The first area of comparison evaluates the effects of different DLH storage formats on load times and query performance using the TPC-DS benchmark suite. The study loads 3 TB of TPC-DS data into each of the lakehouse systems, then runs all TPC-DS queries three times and reports the median runtime. The study found that Delta Lake outperformed both Hudi and Iceberg in terms of query performance, with TPC-DS running 1.4x faster on Delta Lake than on Hudi and 1.7x faster on Delta Lake than on Iceberg. The study also found that Hudi was almost ten times slower than Delta Lake and Iceberg in terms of loading time, likely due to its optimization for keyed upserts rather than bulk data ingestion. [25]

The second area of comparison evaluated the performance of MoR and CoW update strategies using both an end-to-end benchmark based on the TPC-DS data refresh maintenance benchmark and a synthetic microbenchmark with varying merge source sizes. CoW was tested for all DLHes, while MoR only was tested for Iceberg and Hudi, since Delta Lake only uses CoW. [25]

In the former benchmark, query performance is measured before and after maintenance operations, simulated by the TPC-DS benchmark, is performed. Results showed that maintenance in Hudi MoR were 1.3× faster than in Hudi CoW, but at the cost of 3.2× slower queries post-maintenance. Both Hudi CoW and MoR had poor write performance during the initial load due to additional pre-processing to distribute the data by key and rebalance write file sizes. Delta's performance on both merges and reads was competitive, despite using only CoW, due to generating fewer files, faster scans, and a more optimized MERGE command during maintenance operations. Maintenance in Iceberg with MoR were 1.4× faster than CoW, and post-maintenance query performance remained similar. [25]

The third area of comparison evaluated the performance of different metadata access strategies during query planning, including from cold starts. The study found that Hudi performed best for very small queries because it caches query plans. The study also found that Iceberg uses the Spark Data Source v2 API instead of the Data Source v1 API used by Delta Lake and Hudi, which sometimes results in less performant query plans over Iceberg due to the less mature nature of the DSv2 API. [25]

2.5.3 Delta Lake

Databricks Delta Lake achieved a new world record on the TPC-DS benchmark, with a scale factor of 100TB. The benchmark consisted of 99 queries and measured the performance of the system in terms of response time and throughput. Databricks Delta Lake was able to complete the benchmark in 3.2 hours with a throughput of 526,100 queries per hour, which is the highest throughput ever achieved on this benchmark. [31] [32]

The benchmark was conducted on the Databricks Unified Analytics Platform, which is a cloud-based platform that integrates Delta Lake with Apache Spark. The platform was deployed on AWS and used 1,440 EC2 instances with a total of 57,600 vCPUs and 2.6 PB of storage. [31] [32]

Moreover, the result was further supported in research done by Barcelona Supercomputing Center (BSC), which ran a different benchmark derived from TPC-DS. The research concluded that Delta Lake performed 2.7x faster than the similarly sized Snowflake setup. [31]

2.5.4 BigQuery, and Redshift

The comparison by [3] et al. between BigQuery and Redshift was done based on three main criteria: speedup with increasing processing power, scaleup with increasing dataset size, and cost-effectiveness. In this thesis, the focus is on performance alone, so cost-effectiveness is not discussed going forward.

For Redshift, the evaluation was done on two sets of experiments: speed-up and scale-up. In the speed-up test, the data size was kept constant while increasing the number of nodes, and the time each query takes was measured. The run time for queries 1 and 6 decreases almost linearly as the number of nodes increases, while the speed-up for the other two queries, especially query 2, are worse due to low degree of parallelism. In the scale-up test, both the data size and the number of nodes were increased at the same rate. The runtime for all the queries stays relatively constant across all scales. There are some fluctuations, but there is no general increasing trend in any of the queries. This means that Redshift has good scale-up for different kinds of queries. [3]

For BigQuery, since it provides the user with a blackbox query executor, the user has no control over how many server nodes are actually used for processing. Hence, the evaluation was done by measuring the runtime of all TPC-H queries with increasing dataset sizes. Queries 1 and 6 exhibit very good scaling, and BigQuery takes almost the same time to execute the query irrespective of dataset size. This is because queries 1 and 6 are simple scans over the dataset to compute aggregate values and are massively parallelizable. [3]

The study moves on to evaluate the performance of BigQuery and RedShift in terms of query runtime. A baseline performance is established by running queries on a commodity desktop machine with PostgreSQL 9.1 database management software. Both BigQuery and RedShift are faster than PostgreSQL, and the difference increases with dataset size. For TPC-H query 1, BigQuery outperforms RedShift, even in hot cache performance. However, for TPC-H query 2, RedShift outperforms BigQuery as the dataset size increases. The authors concludes that Redshift scales better when performing more complex queries, since BigQuery is not designed to handle multiple joins, it rather expects data to be nested in itself. [3]

2.6 Choice of Candidates

Based on all knowledge gathered in Chapter 2, BigQuery was selected as the DW, while Delta Lake was selected as the DLH for further evaluation. Even

though all candidates fulfill the basic needs required as a DSS system, these were selected based on their suggested query performance and scalabiliity.

The comparison between BigQuery and Redshift showed some differences in read performance, although not any significant differences. The study investigated suggests that Redshift performs slightly better on complex queries that involves multiple joins. However, BigQuery seemed to outperform Redshift on other queries, which can be explained by BigQuery's ability to scale compute proportional to demand and data size. Even though the differences between the two DWs are not that significant, BigQuery was chosen based on the scalability that its serverless implementation provides.

The selection was more obvious when comparing the lakehouse candidates. Based on the benchmarking study by Jain et al. [25], Delta Lake was chosed for further analysis since it outperforms Apache Hudi and Apache Iceberg in terms of query performance and loading time for bulk data ingestion. Delta Lake also has competitive performance on merges and reads, despite using only CoW, due to generating fewer files, faster scans, and a more optimized MERGE command during maintenance operations. In addition, Delta Lake uses CoW only, which makes maintenance simpler and faster compared to Apache Hudi's MoR and CoW update strategies. All of these metrics are important to consider when building a DSS system, while the most important metric for this thesis is read performance. Delta Lake's impressive read performance obviously makes it the more promising candidate for this particular use case.

Snowflake seemed like a good choice because of its maturity and scalability. There is a lack of academic literature comparing Snowflake to Delta Lake, although some gray literature was found to compare the two. The BSC have compared Delta Lake against Snowflake as a third party, and concluded that Delta Lake outperforms Snowflake by 2.7x in terms of speed. Furthermore, Transaction Processing Performance Council (TPC) have conducted research on Delta Lake and observed the highest throughput ever achieved on the TPC-DS 100 TB benchmark [32]. On that basis, Delta Lake was chosen over Snowflake for further analysis.

Chapter 3

Method

In this chapter the experimental method is described. First, the data used within the experiment is presented and which scale factors are used. Thereafter, the experimental design is described by presenting workloads, test environment, measurements made during experiments, and finally a section describing the analysis.

3.1 Data

For the experiments, a standardized DSS benchmark, TPC-DS, was employed. This choice was influenced by several factors. The benchmark offered a diverse range of queries designed to simulate the operations of a standard DSS system. Furthermore, the TPC-DS data generator included functionality to scale the dataset, which was utilized to evaluate the scalability of the systems under evaluation. The data scales included in the study were 2, 4, 8, 16, 32, and 64 GB. It's important to note that the dataset exclusively comprises structured data. While the DW only accommodates structured data, DLHes can handle both unstructured and semi-structured data. However, this research does not investigate the DLHes' capability to query these types of data.

3.2 Experimental design

3.2.1 Workload

To assess the performance of each system, read operations were performed on both the Delta Lake (DLH) and BigQuery (DW) systems under varying request workloads and data set sizes. Synthetic DSS queries provided by the TPC-DS

benchmark were used to simulate varying request workloads, ranging from simple to complex queries. The queries were consistent across scales, and were executed on the synthetic datasets generated by TPC-DS.

3.2.2 Environment

All benchmarks were run in Google Cloud Compute (GCC) in the us-east1 region. Although the benchmarking of Delta Lake was conducted within a Databricks environment, the underlying compute capabilities were provided by Google Cloud Platform (GCP). A e2-highmem-16 instance was used as a client to run the benchmarks towards BigQuery. It's crucial to note that BigQuery is a serverless technology, allowing its resources to scale with the dataset. Therefore, we observed the maximum number of slots allocated to BigQuery under the heaviest workload conditions and noted the associated costs. We then allocated resources to Delta Lake at an equivalent cost to account for BigQuery's serverless capabilities. The maximum number of slots allocated to BigQuery was found to be 200. Consequently, Delta Lake was configured with a cluster of e2-standard-32 instances, scaling from a minimum of 1 node to a maximum of 3 nodes. This configuration resulted in costs comparable to the 200 slots allocated to BigQuery. Default conditions of BigQuery were used when performing benchmarks, and thus there is no need to further specify the BigQuery configuration below in Table 3.1. Additionally, it's important to note that BigQuery Client is primarily responsible for formulating, submitting, and collecting the results of queries. It doesn't handle the actual data processing workload, which is managed by BigQuery's underlying infrastructure. Given this, the memory performance of BigQuery Client is unlikely to impact the latency results. Thus, our focus has been on comparing the computing power of BigQuery against the Databricks cluster, as these are the primary drivers of query performance. More detailed information regarding the configuration of the two clusters can be found in Appendix A.

Cluster Type	GCP Instance Type	Virtual CPU's	Memory	Local Disk Size	Local Disk Type	Nodes
DeltaLake Cluster	e2-standard-32	32	128GB	3TB	SSD	3
BigQuery Client	e2-highmem-16	16	128GB	0GB	None	1

Table 3.1: GCP hardware resources

3.2.3 Measurements

3.2.3.1 Latency

During the benchmarking process, the primary metric recorded was the latency of the queries executed on the TPC-DS dataset. This latency represents the duration from the initiation of a query to its completion, offering a direct indication of each system's responsiveness.

3.2.3.2 Profiling

Grafana dashboards were used to measure the hardware utilization of the two systems during experimentation. For BigQuery, due to its serverless nature, many of the deeper system metrics were not accessible to end-users. Therefore, the profiling primarily focused on slot utilization and the number of bytes scanned. On the Databricks side, given it was running on GCC, a broader range of metrics was available for study. CPU utilization was tracked to see how the cluster's resources were used across the benchmarking tasks. Network-related metrics, such as bytes sent and received, were recorded to understand data transfer patterns. Memory usage together with read and write on disk metrics were recorded to offer insights into activities like data spillage and shuffling, as well as general storage access patterns.

3.2.4 Analysis

To begin with, average latency was calculated for each scale to get an overview of performance differences. Additionally, a discrepancy analysis was performed to identify the specific queries that resulted in the highest discrepancies between the systems. These queries could then be selected for a more detailed analysis, aiming to find distinct performance features of the two platforms. This was done by categorizing the queries based on complexity in terms of the amount of aggregation, joins and the potential for shuffling.

3.2.4.1 Average Latency

For every executed query, specific latency values were recorded for both Delta Lake and BigQuery. Given the array of queries and scales involved, an average latency was computed for each platform across all scales. This was determined by taking the mean of the latencies of all individual queries for a specific scale:

$$\text{Average Latency}_{\text{Scale}_i} = \frac{1}{n} \sum_{j=1}^{n} \text{Latency}_{\text{Query}_j, \text{Scale}_i} \tag{3.1}$$

Where n represents the total number of queries. This calculation provides an average perspective on performance, offering insights into the overall responsiveness of each system across varying scales.

3.2.4.2 Discrepancy Analysis

Firstly, the absolute difference in latency between the platforms for each query was computed, while the platform which performed faster was identified:

$$\text{Discrepancy}_{\text{Query}_i, \text{Scale}_j} = |\text{Latency}_{\text{Query}_i, \text{Scale}_j, \text{DL}} - \text{Latency}_{\text{Query}_i, \text{Scale}_j, \text{BQ}}| \tag{3.2}$$

Next, the average absolute discrepancy was calculated for each query over all dataset scales:

$$\text{Average Discrepancy}_{\text{Query}_i} = \frac{1}{m} \sum_{j=1}^{m} \text{Discrepancy}_{\text{Query}_i, \text{Scale}_j} \tag{3.3}$$

Where m is the count of dataset scales. To offer a relative perspective, the overall average discrepancy, which helps to place individual query discrepancies in context, was found as:

$$\text{Overall Average Discrepancy} = \frac{1}{n} \sum_{i=1}^{n} \text{Average Discrepancy}_{\text{Query}_i} \tag{3.4}$$

Following this, discrepancies were normalized:

$$\text{Normalized Discrepancy}_{\text{Query}_i} = \frac{\text{Average Discrepancy}_{\text{Query}_i}}{\text{Overall Average Discrepancy}} \tag{3.5}$$

Finally, discrepancies were sorted in descending order to find the queries to move forward with for further analysis.

3.2.4.3 Query Categorization

In the analysis, each query was categorized based on it's complexity in terms of the amount of aggregation, joins, and the potential for shuffling. Aggregation was either categorized as **Intensive**, **Medium**, or **None**. The amount of joins was categorized as either **Multi-Join**, **Single-Join**, or **None**. Finally, the potential for shuffling was categorized based on the amount of aggregation and join operations.

Aggregation	Join	Potential for shuffling
Intensive	Multi-Join	High
Intensive	Single-Join	High
Medium	Multi-Join	High
Intensive	None	Medium
None	Multi-Join	Medium
Medium	Single-Join	Medium
Medium	None	Low
None	Single-Join	Low
None	None	Low

Table 3.2: Describes how the potential for shuffling is inferred based on the amount of aggregation and join operations

This analysis aimed to discern any patterns among queries exhibiting significant discrepancies between the two systems. Additionally, the hardware metrics collected during the most resource demanding benchmark were analyzed in order to further support any patterns found during the query categorization.

Chapter 4

Results and Discussion

In the following chapter the most relevant collected results are presented. First, aggregated latency results for all scales are presented and discussed, together with measured peak CPU utilization of both BigQuery and Delta Lake. Additionally, a more in depth discrepancy analysis follows that identifies which queries have the highest average discrepancy between the two systems. These queries are categorized based on complexity. A discussion revolving hardware utilization follows that hypothesizes why the high discrepancies occur for these particular categories. Due to the high amount of data collected, results are included in their entirety in Appendix B and Appendix C for brevity.

4.1 Aggregated results

This section describes the results in an aggregated format, presenting how overall latency changes across all scales for both BigQuery and Delta Lake. Furthermore, the peak CPU utilization observed during the benchmarks are presented. The purpose of this section is to present an overview of the results.

4.1.1 Average latency

We can see clear patterns in the average latency of BigQuery and Delta Lake across the range of scales, as shown in Figure 4.1 and Table 4.1.

From 2 GB to 64 GB, Delta Lake consistently had significantly higher average latency compared to BigQuery. The difference was most evident at the 2 GB scale, where Delta Lake's latency was 9.1s longer than BigQuery's, thereby 650% larger. However, it's notable that Delta Lake exhibited a decrease in latency when moving to the 4 GB scale giving a difference of 7,7

s, giving a more moderate but still significant increase of 513%. This means that the 2 GB scale could be interpreted as somewhat of an outlier. Continuing however, we can see a gradual increase in latency with scale progression for Delta Lake.

Furthermore, while the latency for BigQuery rising methodically from 1.4s at 2 GB to 6.1s at 64 GB — which represents a 336% increase — Delta Lake's latency trajectory was more moderate, increasing from 10.5s at the 2 GB scale to 14.7s at the 64 GB scale, marking only a 40% increase.

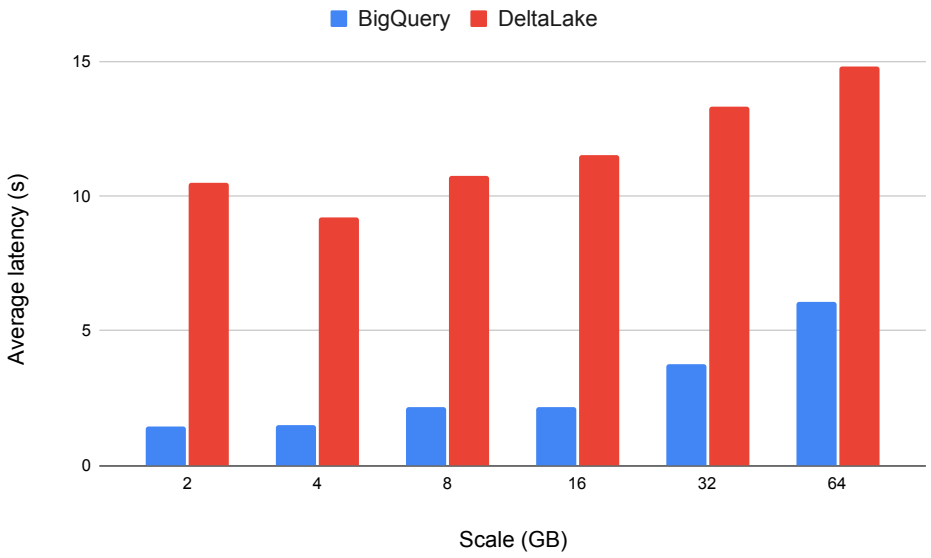


Figure 4.1: Average latency across scales

Scale (GB)	BigQuery Average Latency (s)	DeltaLake Average Latency (s)
2	1.4	10.5
4	1.5	9.2
8	2.2	10.8
16	2.1	11.5
32	3.8	13.3
64	6.1	14.7

Table 4.1: Average latency across scales

4.1.2 Peak CPU Utilization

Peak CPU Utilization was monitored while running the benchmarks. The results are presented across all scales in Figure 4.2 to provide an overview of resource allocation.

With the increase in data scale, CPU usage for both BigQuery and Delta Lake rose. However, BigQuery showed higher CPU usage than Delta Lake at the majority of the scales. A standout observation is at the 32 GB scale, where BigQuery's CPU consumption reached 91.0%, which was notably higher than Delta Lake's 43.3%.

One interesting point relates to the 8 GB to 16 GB transition for BigQuery. Even though there was an uptick in CPU usage from 46.5% to 62.5%, BigQuery's latency reduced. This suggests that BigQuery could be allocating CPU resources efficiently, allowing it to process larger volumes of data without a corresponding rise in latency.

It's possible that Delta Lake's operations are more I/O bound than CPU bound. If Delta Lake spends more time waiting for data to be read from or written to storage, it could be using the CPU less intensively during these periods. This can contrast with BigQuery which might be more optimized for parallel processing and performs more operations in-memory, thus utilizing the CPU more heavily.

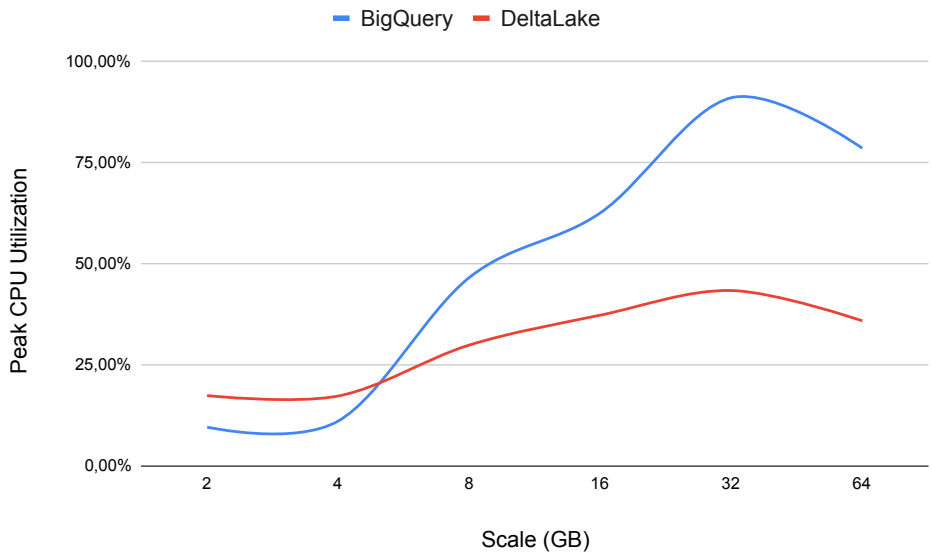


Figure 4.2: Peak CPU utilization across scales

4.2 Discrepancy Analysis

This section presents the results from a more granular perspective. To begin with, the queries with the largest difference in average latency over all scales are presented. Furthermore, a normalized version of the numbers is presented to compare the specific query performance to the overall average latency. Thereafter, these queries are selected for further analysis where they are categorized based on their complexity. Finally, hardware resource metrics are presented.

4.2.1 Query Discrepancy

To understand the variability in performance between BigQuery and Delta Lake for specific queries, the average latency discrepancies were analyzed. Figure 4.3 and Table 4.2 showcases the five queries that recorded the most significant differences in their execution times across all scales.

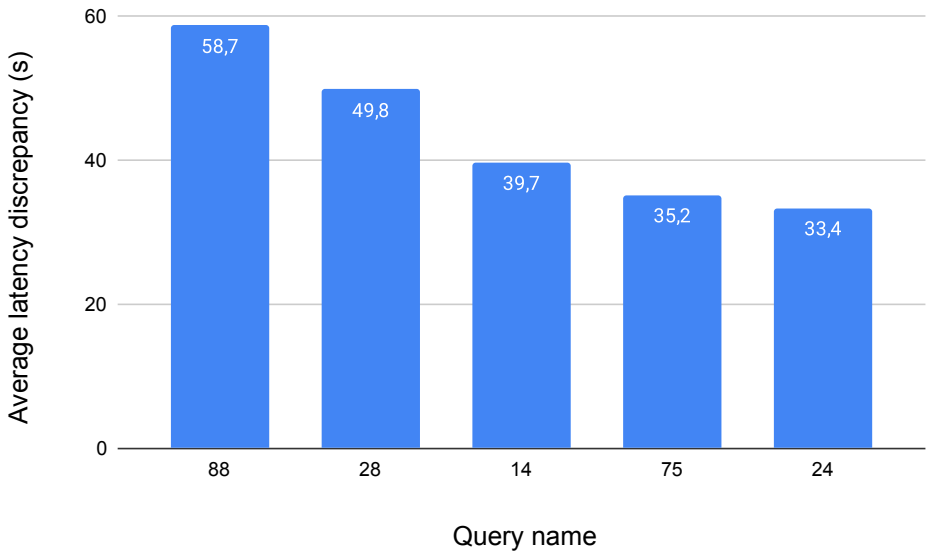


Figure 4.3: Queries with the largest average discrepancy

Examining normalized discrepancies, we can observe that Query 88 exhibits the highest value at 6.5, emphasizing its pronounced variation in performance between the two platforms. This is followed by Queries 28, 14, 75, and 24 in descending order of their normalized discrepancies. Notably, for

all these queries, BigQuery was the predominant faster platform, consistently outperforming Delta Lake when handling these specific query types.

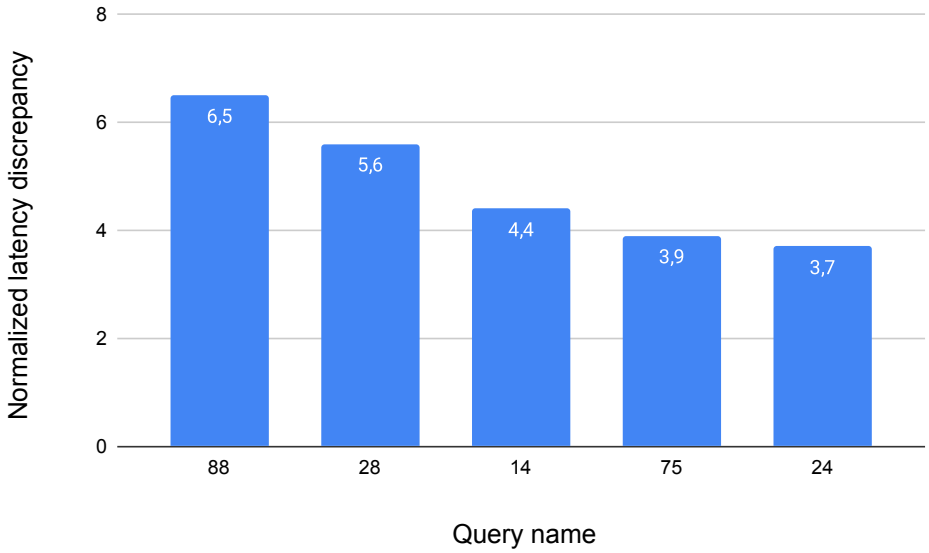


Figure 4.4: Normalized values of queries with the largest average discrepancy

Rank	Query name	Average discrepancy	Normalized discrepancy	Majority faster platform
1	88	58.7	6.5	BigQuery
2	28	49.8	5.6	BigQuery
3	14	39.7	4.4	BigQuery
4	75	35.2	3.9	BigQuery
5	24	33.4	3.7	BigQuery

Table 4.2: Queries with the largest average latency discrepancy

4.2.2 Query Categorization

To better understand the performance discrepancies observed between BigQuery and Delta Lake, we delve into the inherent characteristics of our test queries. By examining the nature and operations within each query, especially focusing on aggregation and data shuffling, we aim to discern patterns that might hint at why one system might consistently outperform the other. Following this, we present a detailed breakdown of five representative

queries with the highest observed discrepancies, to point to the potential factors influencing their execution times on each platform. The queries are summarized in a list format rather than to be inserted in their entirety into the main body of the thesis. The full queries can be found in Appendix D. The following table summarizes the categorization of the top five queries with the highest discrepancy:

Rank	Query name	Start time	End time	Aggregation	Join	Potential for shuffling
1	Query 88	10:35:17	10:36:30	Intensive	Multi-Join	High
2	Query 28	10:22:00	10:23:02	Intensive	None	Medium
3	Query 14	10:18:31	10:19:19	Intensive	Multi-Join	High
4	Query 75	10:31:46	10:32:20	Intensive	Multi-Join	High
5	Query 24	10:20:47	10:21:40	Intensive	Multi-Join	High

Table 4.3: Queries with the largest average latency discrepancy, categorized based on complexity

4.2.2.1 Query 88

Query 88 involves retrieving sales data based on specific time intervals and demographics. The structure of the query is outlined as follows:

- **Tables Involved**:
  - `store_sales`
  - `household_demographics`
  - `time_dim`
  - `store`
- **Join Operations**: The query involves multiple join operations across the mentioned tables.
- **Filtering Criteria**:
  - Filtering based on specific hours and minutes to delineate time intervals.
  - Filtering based on household demographics, specifically the dependency count and vehicle count.
  - Specific store name criterion.

- **Aggregation**: The core of this query involves counting sales entries based on the set criteria.
- **Complexity**:
  - Multiple sub-queries representing different time intervals, all of which are then joined together.
  - Intensive filtering and join operations, which can lead to significant data shuffling.

Based on its structure and operations, Query 88 can be categorized as a complex, multi-join, and aggregation-intensive query. Given the intensive shuffling expected from such a query, systems with optimized in-memory operations and efficient handling of intermediate data storage may have a performance advantage.

4.2.2.2 Query 28

Query 28 aims to retrieve aggregated sales metrics for different brackets of sold quantities. These brackets define ranges of quantities sold and are associated with certain price, coupon, and cost criteria. The structure of the query can be broken down as:

- **Tables Involved**:
  - `store_sales`
- **Join Operations**: No explicit join operations with other tables are conducted. However, the query involves multiple sub-queries which are then combined to form the result.
- **Filtering Criteria**:
  - Filtering based on different ranges of `ss_quantity`.
  - Each sub-query has conditions related to either `ss_list_price`, `ss_coupon_amt`, or `ss_wholesale_cost`.
- **Aggregation**: Each sub-query performs aggregation operations to calculate:
  - Average of `ss_list_price`
  - Count of `ss_list_price`

  - Count of distinct `ss_list_price`

- **Complexity**:
  - Contains six similar sub-queries, each filtering and aggregating data based on different criteria.
  - Extensive use of `OR` clauses, which can lead to broad scans and increased computational requirements.
  - Due to the absence of joins, shuffling might not be as significant as in join-intensive queries. However, the repeated and varied aggregation in each sub-query can still make the query computationally heavy.

Based on its structure and operations, Query 28 can be categorized as a complex, aggregation-intensive query with multiple sub-queries.

4.2.2.3 Query 14

Query 14 is designed to compute sales metrics for items across various sales channels (store, catalog, web) for a specified time range. It highlights items whose sales have surpassed an average sales value. The structure of the query is as follows:

- **Tables Involved**:
  - `store_sales`
  - `catalog_sales`
  - `web_sales`
  - `item`
  - `date_dim`
- **Join Operations**: Several joins are performed:
  - `store_sales` with `item` and `date_dim`
  - `catalog_sales` with `item` and `date_dim`
  - `web_sales` with `item` and `date_dim`
- **Filtering Criteria**:
  - Time range criteria based on `d_year` and `d_moy`.

  - Item-specific filters determined by cross-referencing sales across channels.

- **Aggregation**:
  - Computing the average sales.
  - Summing sales and counting the number of sales for specific item attributes across channels.

- **Complexity**:
  - The use of multiple Common Table Expressions (CTEs) to define item sets and average sales computations.
  - Several sub-queries, each with their filtering, joining, and aggregation operations.
  - Use of UNION ALL operations to combine results from different sales channels.

Query 14 can be categorized as a highly complex multi-channel sales analysis query that involves heavy join operations which could imply significant data shuffling.

4.2.2.4 Query 75

Query 75 is structured to aggregate and compare sales figures for items across different sales channels. It then contrasts these figures between two successive years, highlighting significant discrepancies. The structure of the query is as follows:

- **Tables Involved**:
  - `catalog_sales`
  - `store_sales`
  - `web_sales`
  - `item`
  - `date_dim`
  - `catalog_returns`
  - `store_returns`
  - `web_returns`

- **Join Operations**: Several joins are executed:
  - `catalog_sales` with `item` and `date_dim`
  - `store_sales` with `item` and `date_dim`
  - `web_sales` with `item` and `date_dim`
- **Filtering Criteria**:
  - The criterion based on the `i_category` 'Men'.
  - Specific year comparison between `curr_yr.d_year` and `prev_yr.d_year`.
  - Relative sales count difference between two years.
- **Aggregation**:
  - Summation of sales count and sales amount across various dimensions.
  - Computation of the difference in sales count and sales amount between two years.
- **Complexity**:
  - Implementation of a CTE all_sales that performs aggregation after consolidating sales data across channels.
  - Incorporation of UNION ALL operations to combine results from distinct sales channels.
  - A self-join operation performed on the CTE all_sales to compare year-on-year figures.
  - Left joins with respective returns tables to adjust sales data with returned item figures.

Query 75 can be categorized as a multi-channel sales comparative analysis query. The heavy join operations combined with aggregations could lead to extensive data shuffling and computational needs.

4.2.2.5 Query 24

Query 24 is developed to aggregate net profits from store sales. This analysis involves filtering data based on specific conditions and then presenting results with a constraint on the aggregated net profit. The structure of the query is as follows:

- **Tables Involved**:
  - `store_sales`
  - `store_returns`
  - `store`
  - `item`
  - `customer`
  - `customer_address`
- **Join Operations**: Several joins are performed:
  - `store_sales` with `store_returns` on ticket number and item.
  - `store_sales` with `item`, `customer`, `store`, and `customer_address` based on appropriate keys.
- **Filtering Criteria**:
  - Matching between the customer birth country and the country in the customer's address.
  - The store's market ID.
  - The color attribute of the item.
  - The total net profit surpassing a threshold based on the average net profit.
- **Aggregation**:
  - Summation of the net profit based on customer name and store name.
  - Computation of average net profit.
- **Complexity**:
  - Utilization of a CTE 'ssales' that combines results after joining multiple tables and performing initial aggregation.
  - Nested subquery used in the HAVING clause to determine a threshold for the aggregated net profit.
  - Multiple attributes are used in the GROUP BY clause.

Given the multi-table joins and multiple aggregations, Query 24 could entail considerable data processing and shuffling.

4.2.3 Hardware Utilization

During the benchmark test of our dataset at scale 64 GB, the memory consumption across three instances provided valuable insights into the performance dynamics of the systems. Observing Figure 4.5, we can see that Instance 1 exhibited a consistent memory usage profile, hovering between 46.7 GiB to 46.9 GiB without any marked deviations.

On the other hand, Instance 2 presented a more varied pattern. Starting with a baseline of 18.9 GiB at 10:17:00, it experienced a considerable peak in memory usage at 10:21:00, registering 48.8 GiB. This dynamic behavior continued, reaching its maximum at 10:35:00 with a usage of 78.7 GiB, shortly after. Subsequent to these peaks, the memory consumption reverted to a range between 19.9 GiB and 27.0 GiB, though it peaked again to 54.3 GiB by 10:48:00.

Instance 3 initiated with a memory usage of 23.2 GiB. Over time, it displayed periodic memory fluctuations. Although there were some increases and decreases, the most notable peak was observed at 10:49:00, reaching 44.6 GiB.

When correlating the query execution times provided in Figure 4.3 with Figure 4.5, it's plausible that the specific queries, especially those involving multi-join operations and bearing high shuffling potential, can significantly influence the memory consumption patterns. This is especially true for Instance 2, where Query 14 and Query 24 are highly suspect in inducing the notable spike at 10:21:00. Further, the extreme spike in Instance 2 at 10:35:00 strongly correlates with the execution of Query 88, suggesting that this particular query could be very memory-intensive or inefficiently optimized for Delta Lake.

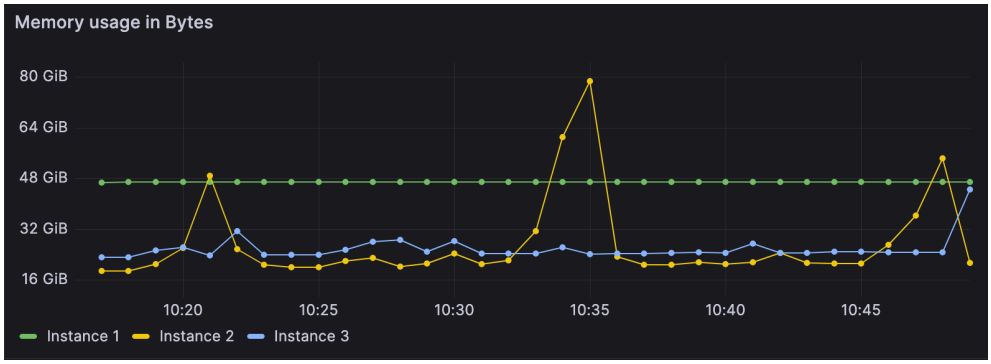


Figure 4.5: Memory usage while benchmarking Delta Lake on scale 64

An analysis of 4.6, which represents the disk write rates across our three instances, reveals notable behavior that aligns with our earlier observations related to memory consumption. For Instance 1, the disk writes appear relatively stable, with no massive spikes. The rate hovers between 2.09 MB/s and 3.86 MB/s.

Instance 2 and 3 however, see evident spikes in disk write rates around 10:19:00. Instance 2 jumps to a rate of 31.3 MB/s, while Instance 3 peaks at 22.2 MB/s. These peaks for both instances coincides with the spike in memory usage observed around the same time for Instance 1.

Given that this timing aligns with the execution of Query 14, it's plausible that it is suggestive that this query induces operations that require intensive in-memory processing such as shuffling. When the available memory is exhausted, the system will start writing the intermediate data onto the disk, leading to increased disk activity. This behavior, termed as spilling, can degrade performance as disk operations are notably slower than in-memory operations.

Furthermore, we can observe a significant spike in disk write rate at around 10:35 up to 15.3 MB/s. The rise peaks at around 10:37 recording a disk write rate of 43.6 MB/s. Building on our previous observation, it is suggestive that this sudden rise in disk activity, when combined with the simultaneous peak in memory usage at the same time, strongly indicates that the system was struggling to keep up with the demands of Query 88.

This behavior is consistent with what we've seen for other queries like Query 14 and Query 24, suggesting that Query 88, like its predecessors, involves resource-intensive operations such as shuffling data, which again partly seem to spill over to disk.

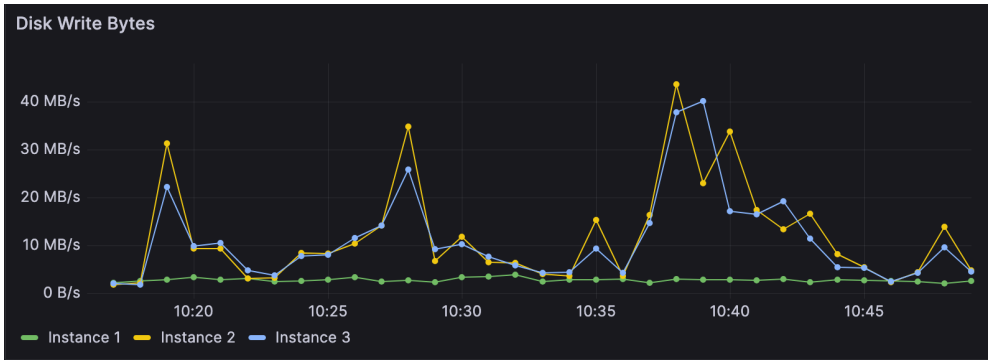


Figure 4.6: Bytes written to disk while benchmarking Delta Lake on scale 64

None of the instances seem to reach, or even approach, the maximum CPU utilization. The peak observed is around 0.358 for Instance 3, which translates to only 35.8% of the CPU's full capacity. Even during instances of increased workloads or complex queries, the CPU does not reach its maximum utilization.

Comparing this with our prior observations from Figures 4.5 and 4.6, we notice a significant I/O activity during the same time frames. The above patterns suggest an I/O bound system. An I/O bound system is characterized by spending more time in waiting for I/O operations to complete than performing computational tasks. This means the system's performance is largely determined by the rate at which it can read and write to external data sources rather than the rate it can process data.

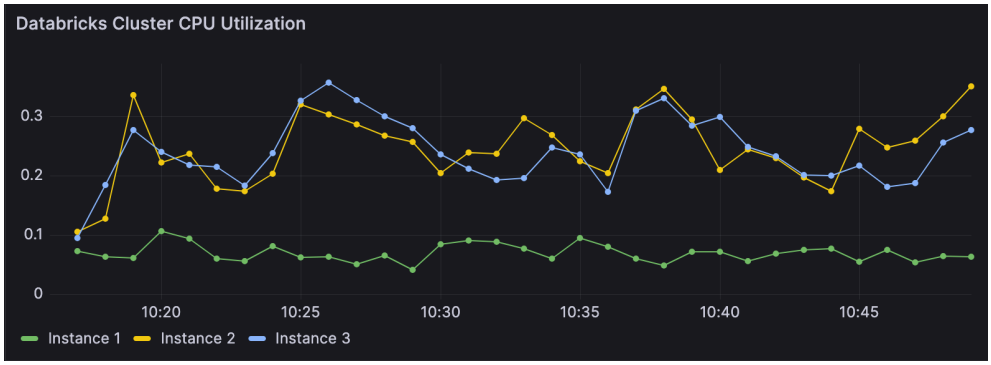


Figure 4.7: CPU utilization while benchmarking Delta Lake on scale 64

Chapter 5

Conclusions and Future work

5.1 Conclusions

The aim of this study was to address several key questions pertaining to the latency of the selected DW and DLH under varying dataset scales and to discern which categories of queries most influenced the performance differences between the systems. Specifically, we sought to answer:

RQ1 How does the latency of the DLH and DW change as the data set size increases?

RQ2 What are the patterns of discrepancy in latency between the DLH and DW when performing complex query operations?

RQ3 Are there specific categories of queries where the discrepancies in performance between the DLH and DW are particularly pronounced? If so, what could be the reasons behind these discrepancies?

A number of data storage systems were first investigated as potential DW and DLH candidates. A literature study was conducted to find the differences and similarities among these systems and to investigate prior benchmarks made in order to gain further insights. These storage systems were then further examined based on their basic properties and performance. Upon investigation, specifically BigQuery and Delta Lake were selected for further analysis based on their suggested read performance and scalability.

To evaluate the performance differences between Delta Lake (DLH) and BigQuery (glsDW), the TPC-DS benchmark was employed. The benchmark enabled the generation of datasets of sizes ranging from 2 to 64 GB,

offering a spectrum to assess scalability. The primary metric of interest was query latency, complemented by profiling metrics used for detailed hardware utilization analysis. After experimentation, analysis began with computing average latency for both systems across all dataset scales. A deeper discrepancy analysis then pinpointed specific queries exhibiting the most pronounced performance variations between the platforms. These queries underwent further analysis, being categorized based on complexity attributes such as aggregation, join operations, and potential for shuffling, aiming to reveal patterns that might account for the observed performance distinctions.

From the onset, it was apparent that BigQuery consistently held a latency advantage over Delta Lake. The disparity in latency was most pronounced at the 2 GB scale. However, as the scale increased, BigQuery exhibited a steeper latency progression, marking a 336% increase, compared to Delta Lake's steadier 40% increase. Despite this trend, in absolute terms, BigQuery still maintained lower latency values across all tested scales.

Furthermore, resource utilization metrics revealed that BigQuery's efficient use of CPU, particularly at the 32 GB scale, could be a determinant in its latency advantage. This observation alludes to BigQuery's capability to efficiently allocate computational resources, potentially leveraging parallel processing and in-memory operations more effectively than its counterpart. We hypothesized that Delta Lake, in contrast, appeared to be more I/O-bound, implying its performance could be primarily dictated by its interactions with disk rather than computational ability.

Upon examining discrepancies across various queries, it was observed that BigQuery frequently outperformed Delta Lake in cases of higher complexity and intensiveness. For example, queries that were aggregation-intensive, or those with multiple sub-queries and extensive join operations, leading to a high shuffle potential, presented the most significant performance differences. These findings suggest that BigQuery may possess optimizations for specific operations, enabling it to manage certain complex queries more efficiently than Delta Lake.

Looking at hardware metrics for Delta Lake during the highest dataset scale of 64 GB, specifically memory consumption patterns provided key insights. While Instance 1 remained steady in its memory usage, Instance 2 revealed pronounced fluctuations, particularly during the execution of Queries 14, 24, and 88. The correlation of these spikes in memory with increased disk write rates indicated instances of data spillage, where the system resorted to writing intermediate data onto the disk when memory was exhausted. This phenomenon, compounded by the fact that the CPU utilization remained

suboptimal even during high-demand periods, solidified the hypothesis of an I/O-bound system.

In addressing RQ1, we can conclude that while both systems have their inherent benefits and drawbacks, it is apparent that BigQuery outperforms Delta Lake in terms of latency across all scales examined. However, BigQuery's latency increased sharply as dataset sizes expanded, whereas Delta Lake displayed a more gradual increase. Despite this difference in progression, BigQuery maintained lower latency values across all scales.

Addressing RQ2, BigQuery consistently outperformed Delta Lake when handling queries categorized as aggregation intensive, with multiple join operations, leading to a high potential for large intermediate data during shuffling stages.

Turning to RQ3, we hypothesize that the pattern identified in RQ2 emerge due to the fact that BigQuery handles the shuffle stage in memory. In contrast, Delta Lake risks spilling data to disk when the memory is exhausted, which to some extent is supported by examining the hardware utilization metrics.

5.2 Future work

The current investigation has gathered insights regarding the performance and resource utilization characteristics of BigQuery and Delta Lake. Building on these findings, there are several avenues that future research can explore to further enhance our understanding of these platforms.

Our study limited its scope to datasets up to 64 GB. Future investigations should extend this limit, testing the behavior of both platforms on data scales extending beyond 64 GB, potentially venturing into terabyte or even petabyte scales. Such research would further clarify the scalability and performance traits of both systems.

One hypothesis formulated in this study was the impact of shuffle operations on performance, particularly in relation to disk spillage. Dedicated research focusing on the intricacies of shuffle operations, capturing both their number and intensity, would be beneficial. This could provide a clearer correlation between complex queries, shuffling intensity, and the consequent disk spillage, thereby revealing deeper insights into the performance bottlenecks. Furthermore, the hypothesis could be tested by switching from the current local SSD based file system to a temporary file system backed by DRAM, and compare the latency metrics to the ones in this study.

Future work can also delve deeper into the underlying architectures of BigQuery and Delta Lake. By understanding the native optimizations and architectural nuances of each platform, we could better comprehend their performance characteristics and potentially identify areas for improvement or further optimization.

5.3 Sustainability and Ethics

Databases, especially when operating at large scales, contribute significantly to the power consumption of data centers. By choosing a more efficient database platform, organizations can reduce their energy consumption, achieving both economical and environmental sustainability.

References

[1] A. Nambiar and D. Mundra, "An overview of data warehouse and data lake in modern enterprise data management," *Big Data and Cognitive Computing*, vol. 6, no. 4, 2022. doi: 10.3390/bdcc6040132. [Online]. Available: https://www.mdpi.com/2504-2289/6/4/132 [Pages 1 and 3.]

[2] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics," in *Proceedings of CIDR*, 2021. [Pages 1, 3, 4, 6, and 13.]

[3] R. Hai, C. Quix, and M. Jarke, "Data lake concept and systems: a survey," *arXiv preprint arXiv:2106.09592*, 2021. [Pages 4 and 16.]

[4] "The data analyst survey." [Online]. Available: https://www.fivetran.com/blog/analyst-survey [Page 5.]

[5] D. Oreščanin and T. Hlupić, "Data lakehouse - a novel step in analytics architecture," in *2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO)*, 2021. doi: 10.23919/MIPRO52101.2021.9597091 pp. 1242–1246. [Pages 6 and 12.]

[6] "Apache parquet." [Online]. Available: https://parquet.apache.org/ [Pages 6 and 11.]

[7] "Apache orc." [Online]. Available: https://orc.apache.org/ [Page 6.]

[8] "Google cloud storage." [Online]. Available: https://cloud.google.com/storage/ [Page 6.]

[9] "Azure data lake storage." [Online]. Available: https://azure.microsoft.com/en-us/products/storage/data-lake-storage [Page 6.]

[10] "Amazon s3." [Online]. Available: https://aws.amazon.com/s3/ [Page 6.]

[11] "Delta lake." [Online]. Available: https://delta.io/ [Pages 6 and 14.]

[12] "Apache hudi." [Online]. Available: https://hudi.apache.org/ [Pages 6, 12, and 14.]

[13] "Apache iceberg." [Online]. Available: https://iceberg.apache.org/ [Pages 6, 11, 12, and 14.]

[14] E. Brewer, "Towards robust distributed systems," 07 2000. doi: 10.1145/343477.343502 p. 7. [Pages 7 and 8.]

[15] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. [Page 8.]

[16] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan, "Amazon redshift and the case for simpler data warehouses," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015. doi: 10.1145/2723372.2742795. ISBN 9781450327589 p. 1917–1923. [Online]. Available: https://doi.org/10.1145/2723372.2742795 [Page 9.]

[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010. [Pages 9 and 10.]

[18] "An inside look at google bigquery | bigquery | white paper - google cloud platform." [Online]. Available: https://www.yumpu.com/en/document/view/11328597/an-inside-look-at-google-bigquery-pdf-google-cloud-platform [Page 9.]

[19] A. Gubarev, D. Delorey, G. M. Romer, H. Ahmadi, J. Shute, J. J. Long, M. Tolton, M. Pasumansky, N. Shivakumar, S. Melnik, S. Min, and T. Vassilakis, "Dremel: A decade of interactive sql analysis at web scale," 2020, pp. 3461–3472. [Online]. Available: http://www.vldb.org/pvldb/vol13/p3461-melnik.pdf [Page 10.]

[20] "Apache spark." [Online]. Available: https://spark.apache.org/ [Pages 10 and 11.]

[21] S. A. Errami, H. Hajji, K. A. El Kadi, and H. Badir, "Spatial big data architecture: From data warehouses and data lakes to the lakehouse," *Journal of Parallel and Distributed Computing*, 2023. [Pages 10, 11, and 12.]

[22] M. Armbrust, T. Das, L. Sun, B. Yavuz, S. Zhu, M. Murthy, J. Torres, H. van Hovell, A. Ionescu, A. Łuszczak *et al.*, "Delta lake: high-performance acid table storage over cloud object stores," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 3411–3424, 2020. [Page 11.]

[23] "Spark sql." [Online]. Available: https://spark.apache.org/sql/ [Page 11.]

[24] "Structured streaming." [Online]. Available: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html [Page 11.]

[25] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, and M. Zaharia, "Analyzing and comparing lakehouse storage systems." [Pages 11, 12, 14, 15, and 17.]

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A {Fault-Tolerant} abstraction for {In-Memory} cluster computing," in *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, 2012, pp. 15–28. [Page 11.]

[27] "Snowflake for data warehouse." [Online]. Available: https://www.snowflake.com/en/data-cloud/workloads/data-warehouse/ [Page 12.]

[28] "Selling the data lakehouse." [Online]. Available: https://medium.com/snowflake/selling-the-data-lakehouse-a9f25f67c906 [Page 12.]

[29] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, ser. SIGMOD '16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2882903.2903741. ISBN 9781450335317 p. 215–226. [Online]. Available: https://doi.org/10.1145/2882903.2903741 [Page 13.]

[30] R. O. Nambiar and M. Poess, "The making of tpc-ds." in *VLDB*, vol. 6, 2006, pp. 1049–1058. [Pages 13 and 14.]

[31] “Databricks sets official data warehousing performance record.” [Online]. Available: https://www.databricks.com/blog/2021/11/02/databricks-sets-official-data-warehousing-performance-record.html [Page 15.]

[32] “Tpc-ds result highlights.” [Online]. Available: https://www.tpc.org/tpcds/results/tpcds_result_detail5.asp?id=121103001 [Pages 15 and 17.]

Appendix A

Configurations

A.1 Databricks Cluster Configuration

```
{
    "autoscale": {
        "min_workers": 1,
        "max_workers": 3
    },
    "cluster_name": "Philip Salqvist's Cluster",
    "spark_version": "12.2.x-scala2.12",
    "spark_conf": {
        "spark.sql.crossJoin.enable": "true"
    },
    "gcp_attributes": {
        "use_preemptible_executors": false,
        "availability": "ON_DEMAND_GCP",
        "zone_id": "HA"
    },
    "node_type_id": "e2-standard-32",
    "driver_node_type_id": "e2-standard-32",
    "ssh_public_keys": [],
    "custom_tags": {},
    "spark_env_vars": {
        "PYSPARK_PYTHON": "/databricks/python3/bin/
            python3"
    },
    "autotermination_minutes": 120,
```

```
    "enable_elastic_disk": false,
    "cluster_source": "UI",
    "init_scripts": [],
    "single_user_name": "philipsalqvist02@gmail.com
       ",
    "enable_local_disk_encryption": false,
    "data_security_mode": "
       LEGACY_SINGLE_USER_STANDARD",
    "runtime_engine": "STANDARD",
    "cluster_id": "0810-131317-gtka26jh"
}
```


A.2 BigQuery Client Configuration


```
    {
  "canIpForward": false,
  "confidentialInstanceConfig": {
    "enableConfidentialCompute": false
  },
  "cpuPlatform": "Unknown CPU Platform",
  "creationTimestamp": "2023-08-07T07
     :01:44.204-07:00",
  "deletionProtection": false,
  "description": "",
  "disks": [
    {
      "architecture": "X86_64",
      "autoDelete": true,
      "boot": true,
      "deviceName": "bigquery-client",
      "diskSizeGb": "60",
      "guestOsFeatures": [
        {
          "type": "UEFI_COMPATIBLE"
        },
        {
          "type": "VIRTIO_SCSI_MULTIQUEUE"
        },
```

```
        {
          "type": "GVNIC"
        },
        {
          "type": "SEV_CAPABLE"
        }
      ],
      "index": 0,
      "interface": "SCSI",
      "kind": "compute#attachedDisk",
      "licenses": [
        "https://www.googleapis.com/compute/v1/
          projects/debian-cloud/global/licenses/
          debian-11-bullseye"
      ],
      "mode": "READ_WRITE",
      "source": "https://www.googleapis.com/compute
        /v1/projects/tpc-ds-bigquery/zones/us-
        east1-b/disks/bigquery-client-1",
      "type": "PERSISTENT"
    }
  ],
  "displayDevice": {
    "enableDisplay": true
  },
  "fingerprint": "FbGZ2J9zyDA=",
  "id": "5106888646909052840",
  "keyRevocationActionType": "NONE",
  "kind": "compute#instance",
  "labelFingerprint": "42WmSpB8rSM=",
  "lastStartTimestamp": "2023-08-10T00
    :31:29.955-07:00",
  "lastStopTimestamp": "2023-08-10T11
    :42:32.717-07:00",
  "machineType": "https://www.googleapis.com/
    compute/v1/projects/tpc-ds-bigquery/zones/us-
    east1-b/machineTypes/e2-highmem-16",
  "metadata": {
    "fingerprint": "Wh5M4tbAowQ=",
```

```
    "kind": "compute#metadata"
  },
  "name": "bigquery-client-1",
  "networkInterfaces": [
    {
      "accessConfigs": [
        {
          "kind": "compute#accessConfig",
          "name": "External NAT",
          "networkTier": "PREMIUM",
          "type": "ONE_TO_ONE_NAT"
        }
      ],
      "fingerprint": "up4PXyXJQlA=",
      "kind": "compute#networkInterface",
      "name": "nic0",
      "network": "https://www.googleapis.com/
          compute/v1/projects/tpc-ds-bigquery/global
          /networks/default",
      "networkIP": "10.142.0.3",
      "stackType": "IPV4_ONLY",
      "subnetwork": "https://www.googleapis.com/
          compute/v1/projects/tpc-ds-bigquery/
          regions/us-east1/subnetworks/default"
    }
  ],
  "reservationAffinity": {
    "consumeReservationType": "ANY_RESERVATION"
  },
  "scheduling": {
    "automaticRestart": false,
    "instanceTerminationAction": "STOP",
    "onHostMaintenance": "TERMINATE",
    "preemptible": true,
  },
  "selfLink": "https://www.googleapis.com/compute/
     v1/projects/tpc-ds-bigquery/zones/us-east1-b/
     instances/bigquery-client-1",
  "serviceAccounts": [
```

```
    {
      "email": "442449641627-compute@developer.
          gserviceaccount.com",
      "scopes": [
        "https://www.googleapis.com/auth/cloud-
            platform"
      ]
    }
  ],
  "shieldedInstanceConfig": {
    "enableIntegrityMonitoring": true,
    "enableSecureBoot": false,
    "enableVtpm": true
  },
  "shieldedInstanceIntegrityPolicy": {
    "updateAutoLearnPolicy": true
  },
  "startRestricted": false,
  "status": "TERMINATED",
  "tags": {
    "fingerprint": "nNZ0SA7CJyk=",
    "items": [
      "https-server"
    ]
  },
  "zone": "https://www.googleapis.com/compute/v1/
      projects/tpc-ds-bigquery/zones/us-east1-b"
}
```

Appendix B

Benchmark Tables

B.1 BigQuery

B.1.1 Scale 2

Name	Runtime
1	1.254
2	1.212
3	0.797
4	2.504
5	1.965
6	1.298
7	0.835
8	0.731
9	1.893
10	2.539
11	3.898
12	0.663
13	1.088
14	2.07
15	0.973
16	1.5
17	3.898
18	2.312
19	1.013
20	0.649
21	0.7
22	0.92
23	3.841
24	1.511
25	2.883
26	0.581
27	0.621
28	1.217
29	3.86
30	0.86
31	1.129
32	0.716
33	0.742

BigQuery results scale 2 queries 1-33

Name	Runtime
34	0.807
35	2.568
36	0.76
37	0.568
38	1.901
39	1.102
40	1.159
41	0.389
42	0.433
43	0.51
44	1.296
45	0.979
46	1.182
47	2.2
48	1.344
49	1.859
50	1.414
51	2.829
52	0.535
53	0.666
54	1.068
55	0.482
56	0.906
57	1.488
58	1.04
59	1.018
60	1.005
61	0.983
62	0.687
63	0.7
64	7.454
65	1.977
66	0.892

BigQuery results scale 2 queries 34-66

Name	Runtime
67	2.597
68	0.949
69	1.642
70	1.188
71	0.722
72	1.63
73	0.797
74	3.608
75	1.875
76	0.651
77	1.334
78	4.657
79	0.89
80	2.057
81	1.035
82	0.918
83	1.151
84	1.031
85	1.539
86	0.698
87	1.846
88	1.33
89	0.712
90	0.576
91	1.019
92	0.53
93	1.083
94	1.572
95	1.912
96	0.541
97	2.065
98	1.384
99	0.612

BigQuery results scale 2 queries 67-99

B.1.2 Scale 4

Name	Runtime
1	1.717
2	1.081
3	0.787
4	2.394
5	3.342
6	1.278
7	1.01
8	0.757
9	1.366
10	1.898
11	3.004
12	0.701
13	1.197
14	1.613
15	1.196
16	1.202
17	3.601
18	2.011
19	0.878
20	0.735
21	0.716
22	0.854
23	3.298
24	3.037
25	3.633
26	0.71
27	0.619
28	1.223
29	3.467
30	0.967
31	1.178
32	0.669
33	0.791

BigQuery results scale 4 queries 1-33

Name	Runtime
34	0.762
35	3.378
36	0.752
37	0.755
38	1.805
39	0.943
40	1.359
41	0.435
42	0.512
43	0.582
44	1.359
45	1.092
46	1.285
47	1.933
48	0.916
49	1.966
50	2.414
51	1.933
52	0.472
53	0.71
54	1.504
55	0.506
56	0.994
57	2.012
58	1.177
59	0.897
60	0.801
61	1.044
62	0.706
63	0.716
64	7.933
65	1.429
66	1.008

BigQuery results scale 4 queries 34-66

Name	Runtime
67	1.925
68	0.983
69	1.478
70	1.276
71	0.758
72	2.118
73	0.682
74	1.96
75	3.078
76	0.73
77	1.401
78	4.35
79	0.829
80	3.333
81	1.134
82	0.7
83	1.497
84	1.416
85	1.762
86	0.664
87	1.805
88	1.277
89	0.781
90	0.537
91	0.914
92	0.601
93	2.508
94	1.908
95	3.7
96	0.431
97	1.344
98	1.553
99	0.634

BigQuery results scale 4 queries 67-99

B.1.3 Scale 8

Name	Runtime
1	2.198
2	1.073
3	0.606
4	4.023
5	2.096
6	1.818
7	0.94
8	0.995
9	2.288
10	2.453
11	3.683
12	1.102
13	1.418
14	2.181
15	1.378
16	1.725
17	6.365
18	2.881
19	1.229
20	0.88
21	1.071
22	1.477
23	4.465
24	4.777
25	6.989
26	1.084
27	0.8
28	1.701
29	7.497
30	1.731
31	1.527
32	0.843
33	1.066

BigQuery results scale 8 queries 1-33

Name	Runtime
34	1.399
35	5.756
36	1.084
37	1.119
38	3.801
39	1.584
40	3.124
41	0.475
42	0.634
43	0.669
44	1.818
45	1.946
46	1.497
47	2.363
48	1.121
49	2.667
50	4.481
51	2.553
52	0.733
53	0.928
54	1.84
55	0.581
56	1.144
57	2.596
58	2.214
59	1.13
60	1.089
61	0.995
62	0.774
63	0.878
64	10.182
65	1.697
66	1.099

BigQuery results scale 8 queries 34-66

Name	Runtime
67	2.67
68	1.606
69	1.871
70	1.318
71	0.923
72	3.607
73	0.997
74	4.039
75	6.869
76	0.759
77	1.249
78	6.399
79	0.859
80	5.489
81	2.1
82	0.765
83	1.432
84	1.775
85	2.208
86	0.84
87	2.926
88	1.707
89	0.86
90	0.677
91	1.321
92	0.777
93	4.21
94	1.281
95	5.376
96	0.574
97	1.206
98	1.894
99	0.681

BigQuery results scale 8 queries 67-99

B.1.4 Scale 16

Name	Runtime
1	3.119
2	1.231
3	1.111
4	3.016
5	2.279
6	1.619
7	1.049
8	0.981
9	2.856
10	2.692
11	3.645
12	0.851
13	1.644
14	2.188
15	1.338
16	3.022
17	1.839
18	3.466
19	1.066
20	0.851
21	9.648
22	0.901
23	9.839
24	7.332
25	1.598
26	0.965
27	0.845
28	1.552
29	1.804
30	1.762
31	1.766
32	1.265
33	1.017

BigQuery results scale 16 queries 1-33

Name	Runtime
34	1.237
35	5.007
36	0.899
37	0.84
38	3.005
39	1.192
40	4.076
41	0.439
42	0.549
43	0.793
44	1.36
45	1.553
46	1.499
47	2.237
48	1.368
49	2.172
50	1.084
51	2.175
52	0.654
53	0.784
54	1.479
55	0.578
56	1.057
57	1.745
58	1.335
59	1.241
60	2.009
61	0.8
62	0.705
63	0.801
64	10.965
65	1.35
66	1.088

BigQuery results scale 16 queries 34-66

Name	Runtime
67	2.142
68	1.46
69	1.722
70	1.441
71	1.535
72	4.596
73	0.95
74	2.811
75	5.778
76	0.821
77	1.247
78	7.861
79	1.127
80	5.345
81	2.29
82	0.807
83	1.344
84	2.277
85	3.238
86	1.047
87	3.341
88	1.584
89	0.885
90	0.85
91	1.05
92	0.788
93	1.141
94	1.899
95	8.06
96	0.587
97	1.625
98	1.527
99	0.673

BigQuery results scale 16 queries 67-99

B.1.5 Scale 32

Name	Runtime
1	3.073
2	2.23
3	0.754
4	14.338
5	2.517
6	2.892
7	1.436
8	1.329
9	2.105
10	3.527
11	10.627
12	1.107
13	1.99
14	6.354
15	3.471
16	6.221
17	4.997
18	7.274
19	1.915
20	1.308
21	0.721
22	1.0
23	14.022
24	6.854
25	5.235
26	1.255
27	1.336
28	1.859
29	4.91
30	2.326
31	4.188
32	2.89
33	2.132

BigQuery results scale 32 queries 1-33

Name	Runtime
34	1.104
35	4.866
36	1.659
37	0.964
38	6.464
39	1.417
40	2.332
41	0.559
42	1.241
43	1.136
44	1.823
45	2.662
46	2.514
47	4.927
48	1.862
49	4.333
50	3.23
51	4.468
52	0.983
53	1.783
54	4.595
55	1.355
56	2.346
57	2.861
58	4.091
59	2.537
60	2.416
61	1.06
62	0.818
63	1.171
64	24.506
65	3.955
66	1.987

BigQuery results scale 32 queries 34-66

Name	Runtime
67	6.73
68	2.528
69	2.194
70	1.929
71	2.934
72	5.946
73	1.225
74	8.646
75	13.392
76	1.48
77	2.144
78	29.47
79	1.488
80	11.033
81	4.389
82	1.362
83	1.524
84	0.957
85	2.444
86	1.087
87	6.391
88	3.123
89	1.639
90	0.817
91	1.258
92	1.331
93	2.631
94	3.498
95	8.355
96	0.791
97	5.048
98	2.234
99	1.03

BigQuery results scale 32 queries 67-99

B.1.6 Scale 64

Name	Runtime
1	2.941
2	2.49
3	1.675
4	24.926
5	4.498
6	5.357
7	2.215
8	1.779
9	4.1
10	5.252
11	22.759
12	1.423
13	3.082
14	10.292
15	4.22
16	6.371
17	9.723
18	4.535
19	4.61
20	1.577
21	1.182
22	1.74
23	21.625
24	18.349
25	10.214
26	1.418
27	2.1
28	2.473
29	9.776
30	4.711
31	9.039
32	1.73
33	3.137

BigQuery results scale 64 queries 1-33

Name	Runtime
34	1.359
35	6.291
36	2.496
37	1.368
38	10.596
39	2.675
40	3.514
41	1.011
42	1.761
43	1.735
44	1.976
45	6.267
46	4.235
47	6.475
48	3.288
49	6.225
50	6.122
51	6.706
52	1.377
53	1.962
54	7.689
55	1.722
56	3.23
57	3.568
58	5.803
59	3.947
60	3.39
61	1.2
62	1.107
63	2.021
64	68.292
65	5.208
66	2.64

BigQuery results scale 64 queries 34-66

Name	Runtime
67	14.528
68	4.105
69	2.764
70	3.09
71	2.896
72	6.831
73	1.679
74	18.818
75	7.051
76	2.249
77	2.919
78	39.002
79	2.236
80	18.085
81	3.805
82	1.752
83	1.734
84	1.288
85	3.989
86	1.269
87	12.167
88	4.443
89	2.388
90	1.022
91	1.331
92	2.116
93	7.56
94	6.243
95	16.108
96	0.882
97	5.662
98	3.586
99	1.538

BigQuery results scale 64 queries 67-99

B.2 Databricks

B.2.1 Scale 2

Name	Runtime
1	30.659
2	28.015
3	2.178
4	23.188
5	16.187
6	2.703
7	4.453
8	2.86
9	12.685
10	4.422
11	17.227
12	1.791
13	3.797
14	55.901
15	1.592
16	42.179
17	6.01
18	4.666
19	1.884
20	1.395
21	1.643
22	4.08
23	33.548
24	39.475
25	4.645
26	4.019
27	5.174
28	85.34
29	7.842
30	4.954
31	7.006
32	2.532
33	3.665

Databricks results scale 2 queries 1-33

Name	Runtime
34	2.573
35	6.135
36	3.609
37	8.606
38	6.588
39	3.255
40	11.494
41	1.178
42	1.313
43	2.572
44	16.003
45	1.66
46	2.702
47	6.977
48	2.616
49	22.848
50	8.548
51	7.382
52	1.302
53	2.936
54	3.057
55	1.186
56	2.672
57	4.983
58	2.635
59	8.754
60	2.907
61	1.768
62	8.92
63	2.658
64	36.486
65	5.228
66	5.1

Databricks results scale 2 queries 34-66

Name	Runtime
67	5.34
68	1.619
69	3.141
70	4.921
71	2.436
72	13.945
73	1.999
74	10.019
75	36.464
76	19.72
77	4.947
78	29.829
79	2.446
80	23.578
81	4.544
82	8.202
83	2.881
84	8.457
85	13.438
86	2.8
87	7.658
88	49.434
89	3.356
90	14.64
91	2.328
92	2.547
93	14.62
94	24.221
95	25.429
96	8.223
97	7.154
98	4.206
99	8.631

Databricks results scale 2 queries 67-99

B.2.2 Scale 4

Name	Runtime
1	15.91
2	17.854
3	2.195
4	19.47
5	11.404
6	2.498
7	3.434
8	3.674
9	8.647
10	3.792
11	11.419
12	1.784
13	2.996
14	34.186
15	2.031
16	25.279
17	4.992
18	3.58
19	1.844
20	1.637
21	1.629
22	4.519
23	19.141
24	25.024
25	3.62
26	3.142
27	3.265
28	41.226
29	7.426
30	4.87
31	6.849
32	2.499
33	3.252

Databricks results scale 4 queries 1-33

Name	Runtime
34	2.769
35	6.233
36	3.274
37	8.417
38	6.862
39	3.976
40	12.796
41	1.222
42	1.363
43	2.779
44	15.544
45	2.116
46	3.187
47	7.678
48	2.742
49	23.353
50	8.129
51	8.137
52	1.578
53	2.957
54	3.365
55	1.342
56	2.866
57	6.354
58	2.936
59	8.66
60	3.165
61	1.735
62	8.371
63	2.917
64	37.308
65	5.744
66	5.191

Databricks results scale 4 queries 34-66

Name	Runtime
67	7.384
68	2.043
69	3.346
70	5.224
71	2.664
72	17.42
73	1.908
74	10.573
75	36.18
76	18.81
77	5.189
78	31.077
79	2.605
80	24.702
81	4.847
82	7.673
83	2.995
84	8.498
85	11.68
86	2.958
87	7.375
88	53.237
89	3.564
90	14.078
91	2.643
92	2.709
93	14.805
94	23.582
95	28.802
96	9.342
97	9.178
98	5.091
99	9.192

Databricks results scale 4 queries 67-99

B.2.3 Scale 8

Name	Runtime
1	7.159
2	18.409
3	2.269
4	25.36
5	14.37
6	3.43
7	4.765
8	4.52
9	11.052
10	5.177
11	24.053
12	2.122
13	3.81
14	43.396
15	2.33
16	29.02
17	5.398
18	4.096
19	2.217
20	1.385
21	1.854
22	4.915
23	21.443
24	33.258
25	5.514
26	4.196
27	4.675
28	51.411
29	11.439
30	4.941
31	10.602
32	3.224
33	4.339

Databricks results scale 8 queries 1-33

Name	Runtime
34	3.208
35	8.215
36	5.038
37	9.238
38	9.2
39	4.251
40	13.213
41	1.34
42	1.357
43	2.464
44	18.573
45	2.319
46	3.271
47	10.056
48	2.878
49	26.018
50	9.75
51	10.821
52	1.536
53	3.266
54	4.493
55	1.627
56	6.607
57	8.245
58	3.355
59	10.121
60	3.45
61	1.956
62	8.81
63	2.891
64	42.693
65	7.537
66	5.08

Databricks results scale 8 queries 34-66

Name	Runtime
67	11.101
68	2.244
69	3.82
70	6.209
71	2.985
72	19.464
73	2.383
74	11.368
75	38.321
76	20.332
77	5.699
78	33.891
79	3.062
80	25.994
81	4.973
82	10.732
83	3.657
84	8.917
85	13.595
86	3.148
87	9.134
88	60.795
89	6.939
90	14.999
91	3.059
92	4.026
93	16.774
94	27.905
95	28.121
96	10.01
97	8.632
98	3.831
99	10.109

Databricks results scale 8 queries 67-99

B.2.4 Scale 16

Name	Runtime
1	6.951
2	19.47
3	2.506
4	27.821
5	12.749
6	2.944
7	3.871
8	3.634
9	10.461
10	5.039
11	19.624
12	1.593
13	3.61
14	39.443
15	2.645
16	30.125
17	6.84
18	4.71
19	2.498
20	1.958
21	1.879
22	5.644
23	28.879
24	36.632
25	6.096
26	4.166
27	4.817
28	54.483
29	9.751
30	5.661
31	9.422
32	2.566
33	4.216

Databricks results scale 16 queries 1-33

Name	Runtime
34	3.087
35	8.446
36	3.798
37	11.536
38	9.133
39	5.535
40	12.83
41	1.196
42	1.564
43	2.61
44	21.066
45	3.089
46	3.599
47	9.713
48	2.99
49	25.967
50	11.243
51	12.521
52	1.634
53	3.49
54	5.07
55	1.882
56	3.243
57	8.611
58	3.239
59	12.089
60	3.871
61	2.398
62	8.962
63	3.152
64	46.858
65	11.625
66	6.429

Databricks results scale 16 queries 34-66

Name	Runtime
67	22.772
68	2.89
69	4.754
70	7.728
71	4.346
72	27.308
73	3.163
74	22.242
75	41.758
76	23.47
77	6.111
78	36.06
79	3.4
80	28.042
81	6.739
82	11.112
83	3.486
84	8.958
85	12.619
86	3.492
87	10.918
88	61.876
89	4.331
90	15.232
91	2.515
92	2.478
93	18.78
94	27.12
95	30.84
96	10.466
97	6.697
98	2.8
99	11.222

Databricks results scale 16 queries 67-99

B.2.5 Scale 32

Name	Runtime
1	9.481
2	28.8
3	3.145
4	35.408
5	14.835
6	3.819
7	3.907
8	3.674
9	10.242
10	5.307
11	21.229
12	2.016
13	3.634
14	40.686
15	3.581
16	32.48
17	9.421
18	6.691
19	4.218
20	2.969
21	2.767
22	8.839
23	46.323
24	50.947
25	6.96
26	3.888
27	4.355
28	56.281
29	11.051
30	5.785
31	11.607
32	2.905
33	4.18

Databricks results scale 32 queries 1-33

Name	Runtime
34	3.603
35	9.253
36	3.99
37	10.659
38	12.09
39	4.965
40	13.067
41	1.21
42	1.556
43	2.92
44	21.036
45	3.362
46	4.28
47	9.786
48	3.246
49	25.26
50	11.9
51	14.263
52	1.814
53	3.389
54	4.798
55	1.707
56	3.774
57	9.843
58	3.364
59	22.655
60	4.054
61	3.016
62	10.99
63	3.789
64	46.101
65	9.097
66	6.104

Databricks results scale 32 queries 34-66

Name	Runtime
67	31.465
68	3.956
69	4.892
70	6.783
71	3.58
72	31.344
73	2.976
74	25.942
75	51.486
76	31.482
77	9.001
78	51.786
79	5.811
80	32.961
81	8.508
82	10.606
83	3.771
84	11.233
85	15.283
86	3.811
87	13.429
88	69.071
89	4.094
90	16.738
91	2.396
92	2.758
93	21.149
94	27.812
95	31.356
96	9.71
97	8.454
98	3.141
99	10.67

Databricks results scale 32 queries 67-99

B.2.6 Scale 64

Name	Runtime
1	11.463
2	44.211
3	3.384
4	41.969
5	18.124
6	4.71
7	6.987
8	4.559
9	14.016
10	6.999
11	26.164
12	2.729
13	4.716
14	48.206
15	4.199
16	36.464
17	8.702
18	7.313
19	3.274
20	2.413
21	2.035
22	7.89
23	42.577
24	53.923
25	7.633
26	4.647
27	4.034
28	62.385
29	14.001
30	6.544
31	10.433
32	2.848
33	4.105

Databricks results scale 64 queries 1-33

Name	Runtime
34	4.091
35	11.489
36	4.839
37	13.148
38	18.48
39	9.356
40	17.124
41	1.598
42	2.829
43	3.47
44	26.449
45	6.211
46	6.101
47	13.395
48	4.827
49	31.221
50	12.401
51	16.069
52	1.95
53	3.69
54	5.104
55	1.867
56	3.827
57	9.984
58	3.547
59	22.761
60	4.36
61	2.791
62	10.831
63	3.527
64	48.722
65	9.997
66	6.199

Databricks results scale 64 queries 34-66

Name	Runtime
67	49.422
68	4.09
69	5.153
70	6.953
71	4.113
72	31.815
73	2.877
74	22.011
75	46.433
76	30.877
77	6.069
78	42.528
79	4.584
80	30.688
81	6.496
82	10.647
83	3.939
84	12.43
85	14.885
86	3.866
87	17.244
88	73.433
89	7.598
90	18.083
91	2.398
92	4.841
93	22.442
94	31.665
95	33.619
96	9.934
97	9.407
98	3.667
99	11.815

Databricks results scale 64 queries 67-99

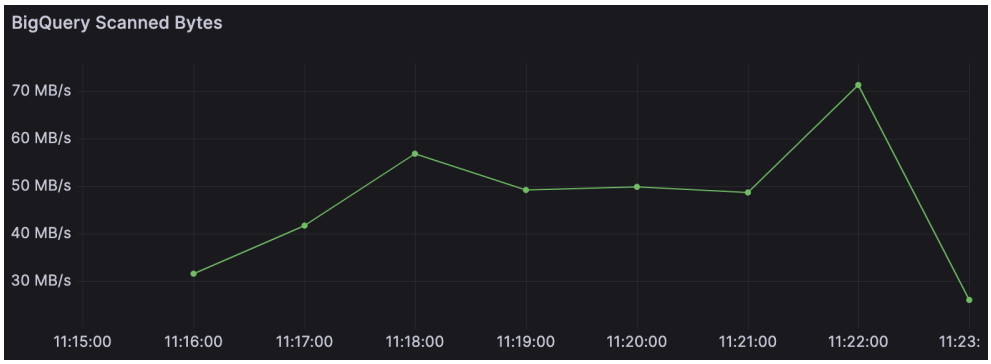
Appendix C

Profiling

C.1 BigQuery

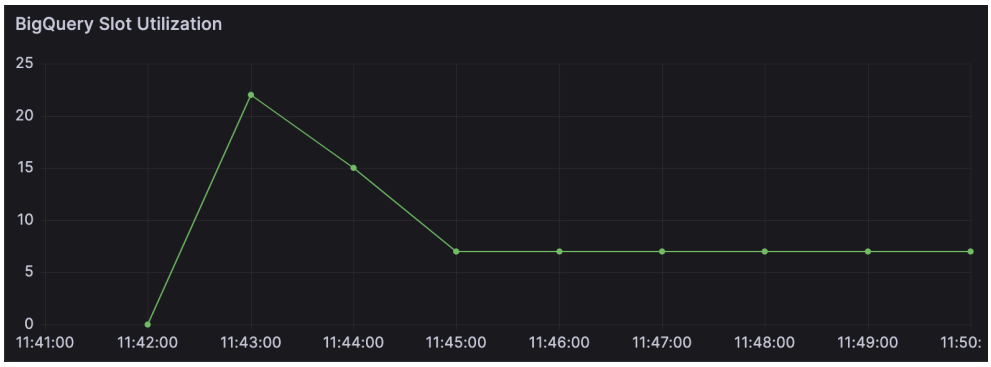
C.1.1 Scale 2

BigQuery slot utilization



BigQuery scanned bytes

C.1.2 Scale 4

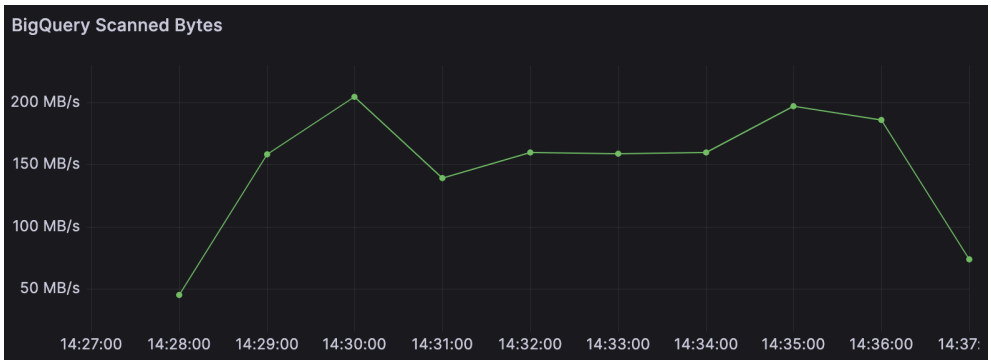


BigQuery slot utilization

BigQuery scanned bytes

C.1.3 Scale 8

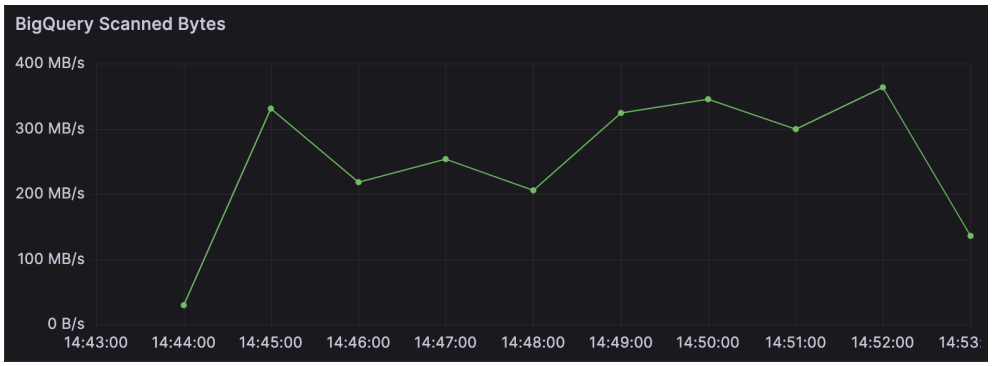
BigQuery slot utilization



BigQuery scanned bytes

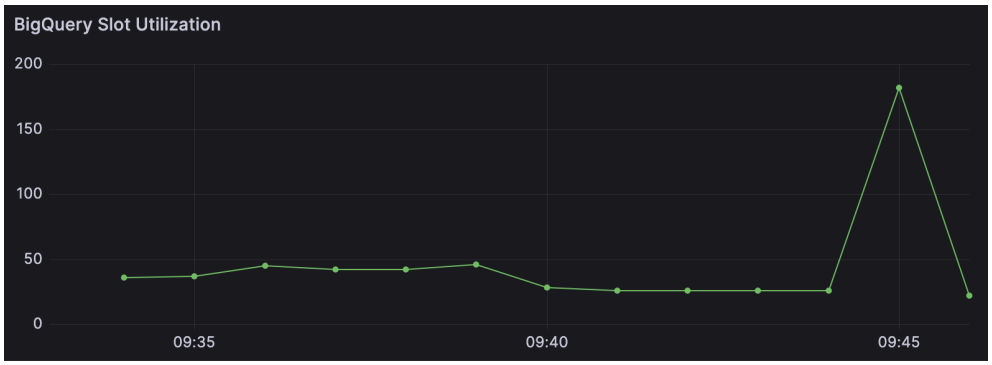
C.1.4 Scale 16

BigQuery slot utilization

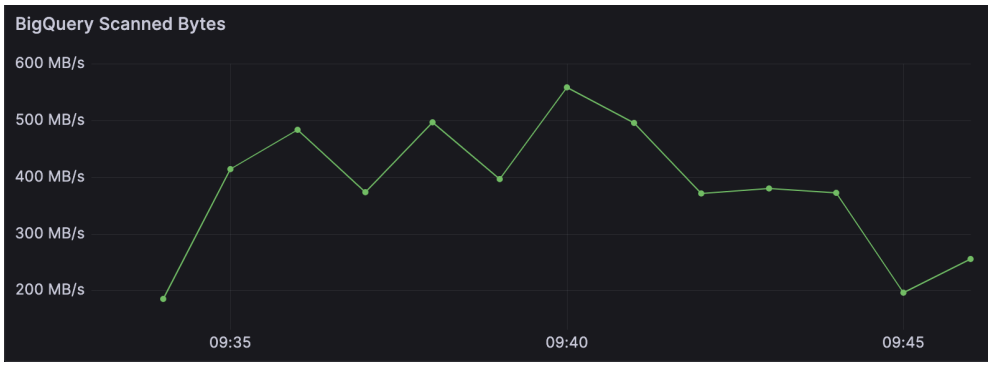


BigQuery scanned bytes

C.1.5 Scale 32

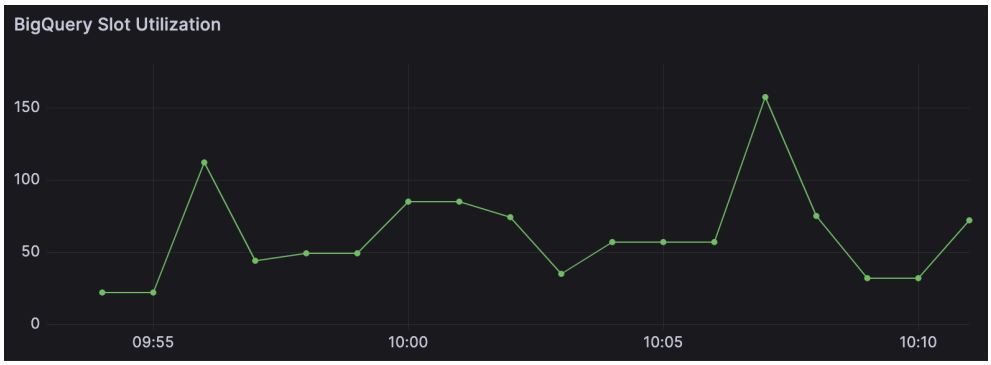


BigQuery slot utilization



BigQuery scanned bytes

C.1.6 Scale 64

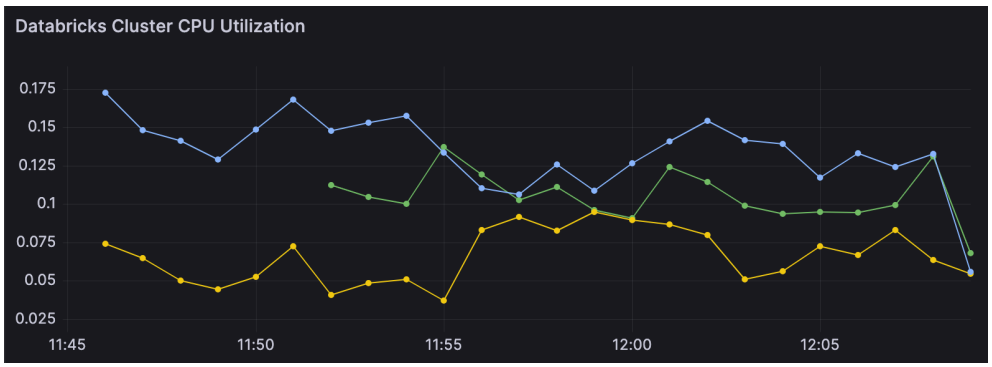


BigQuery slot utilization

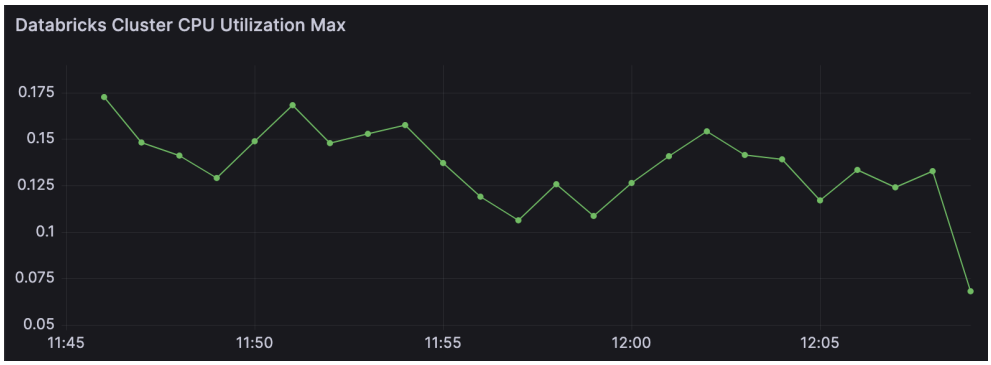
BigQuery scanned bytes

C.2 Databricks

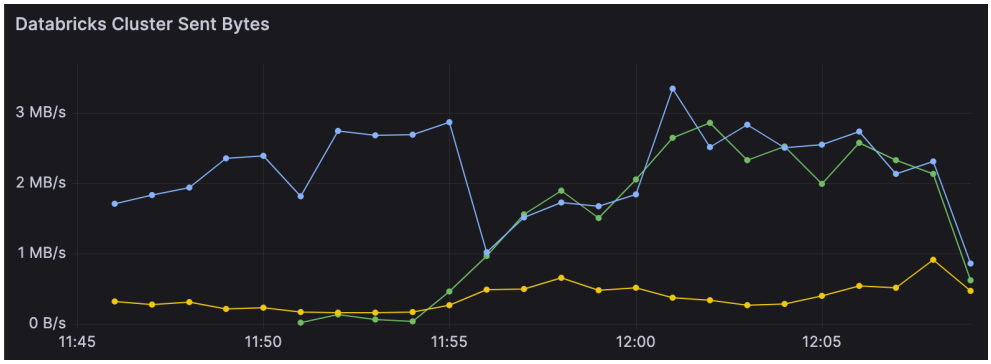
C.2.1 Scale 2



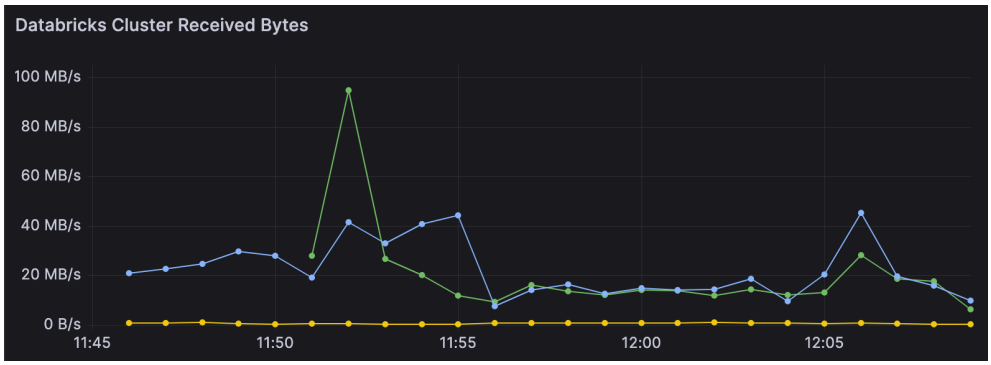
Databricks cluster cpu utilization



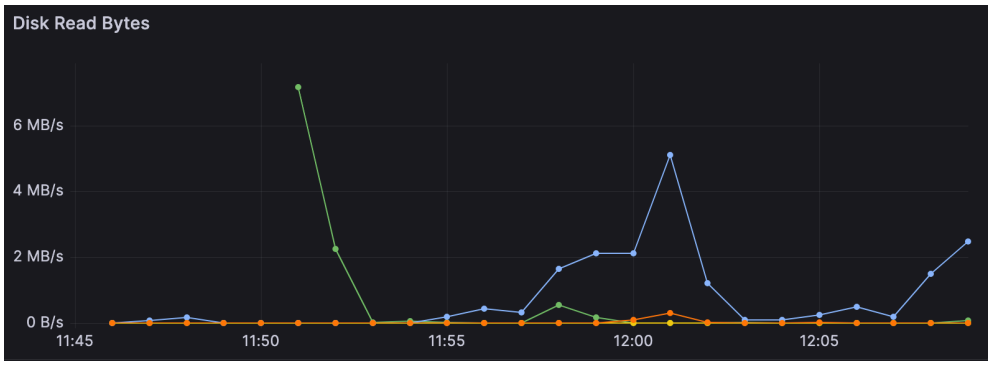
Databricks cluster cpu utilization max



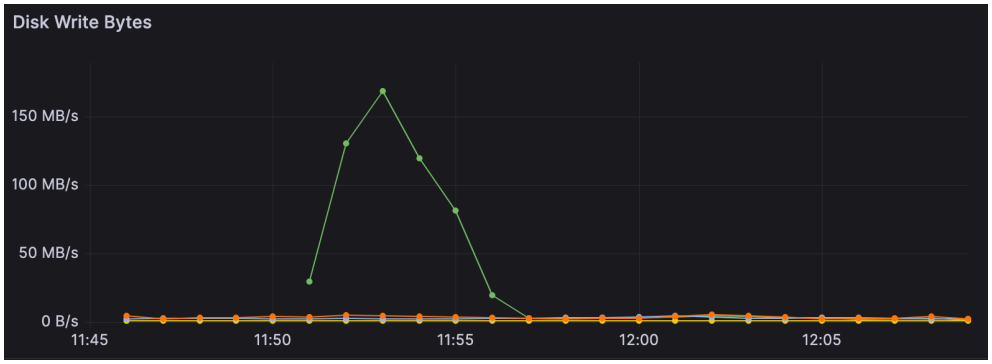
Databricks cluster sent bytes



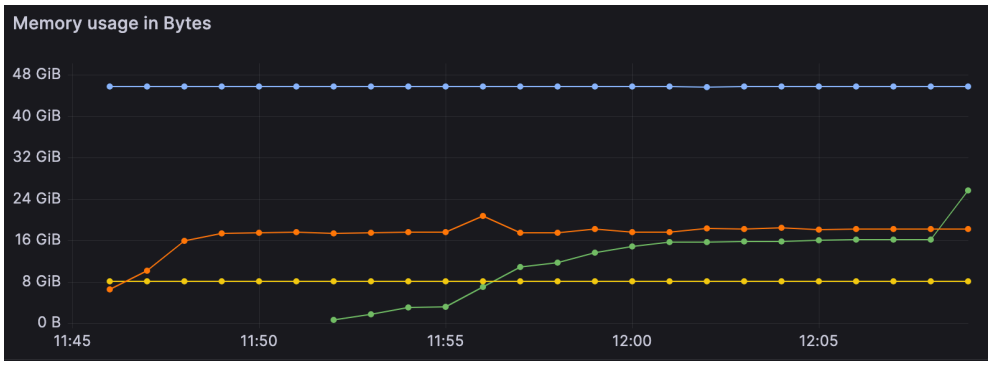
Databricks cluster received bytes



Databricks cluster disk read bytes

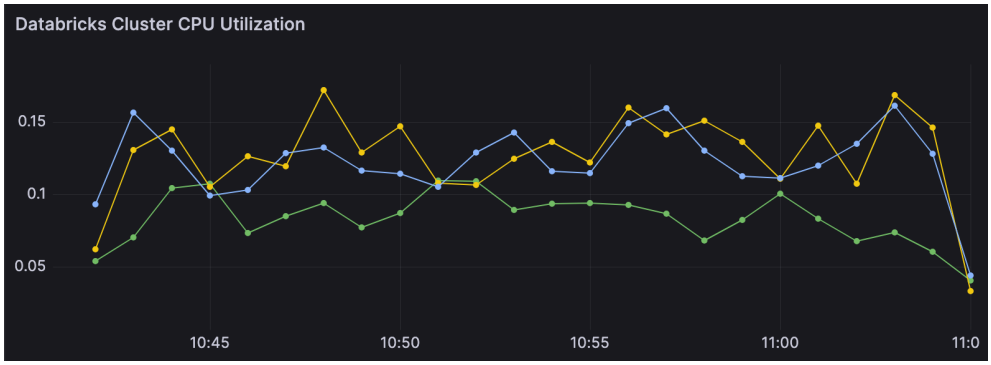


Databricks cluster disk write bytes

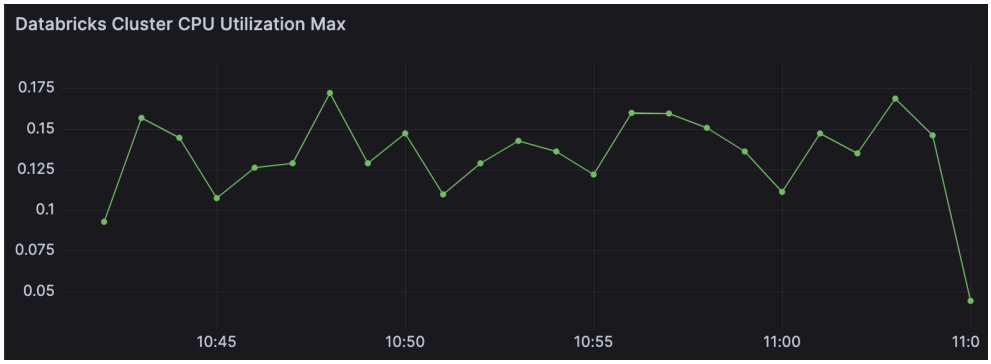


Databricks cluster memory usage

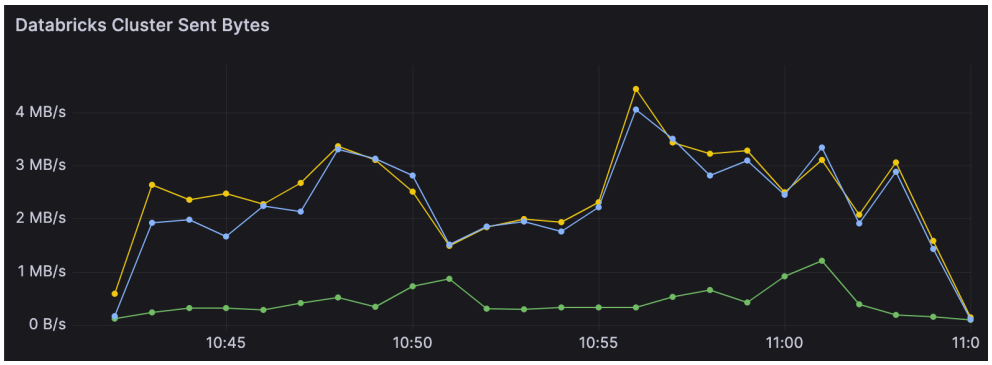
C.2.2 Scale 4



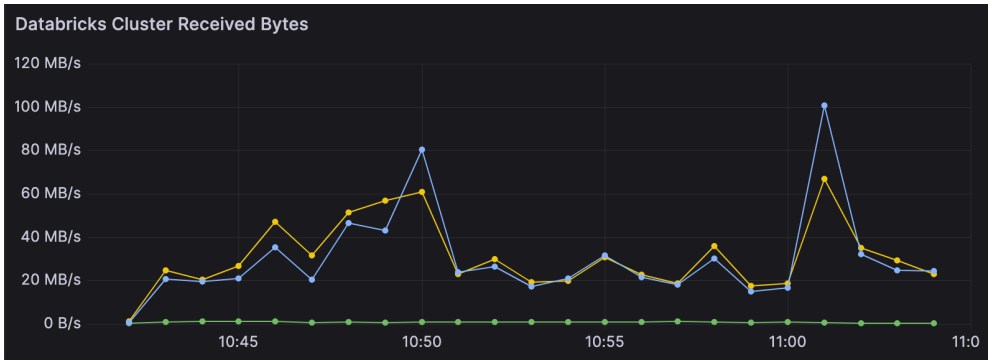
Databricks cluster cpu utilization



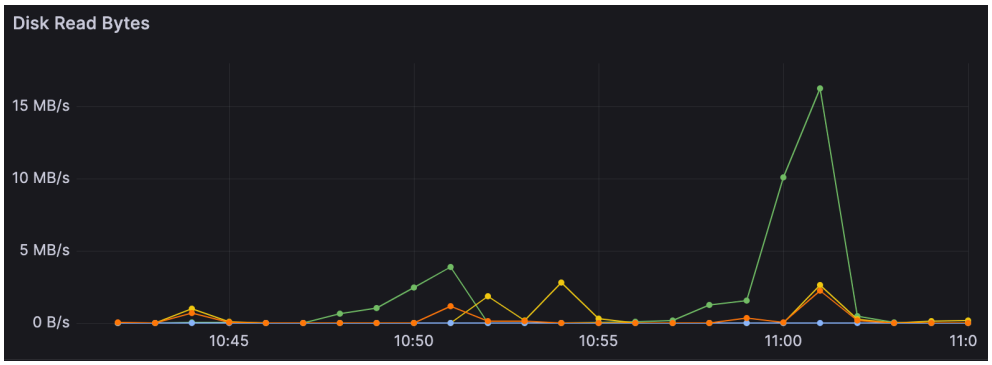
Databricks cluster cpu utilization max



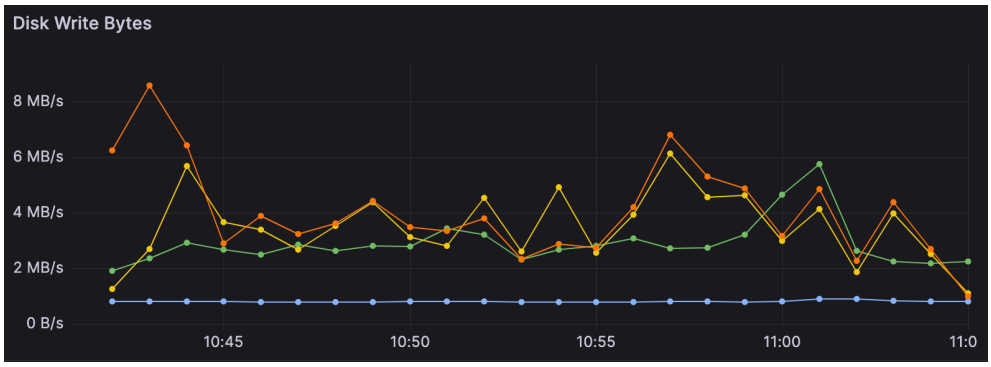
Databricks cluster sent bytes



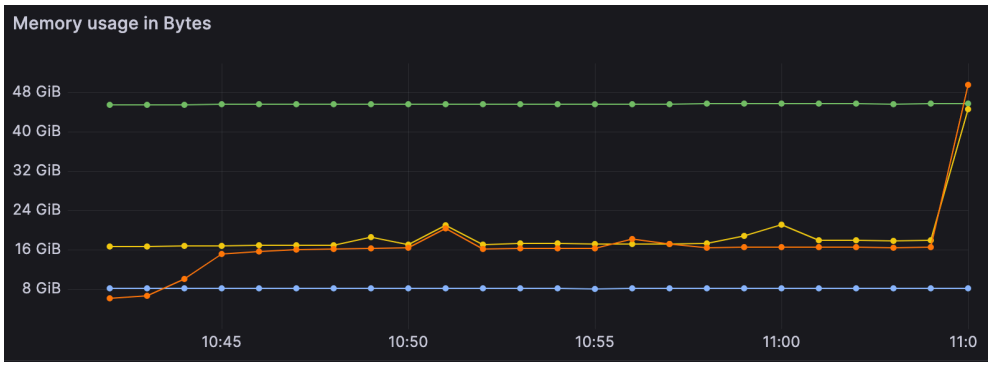
Databricks cluster received bytes



Databricks cluster disk read bytes

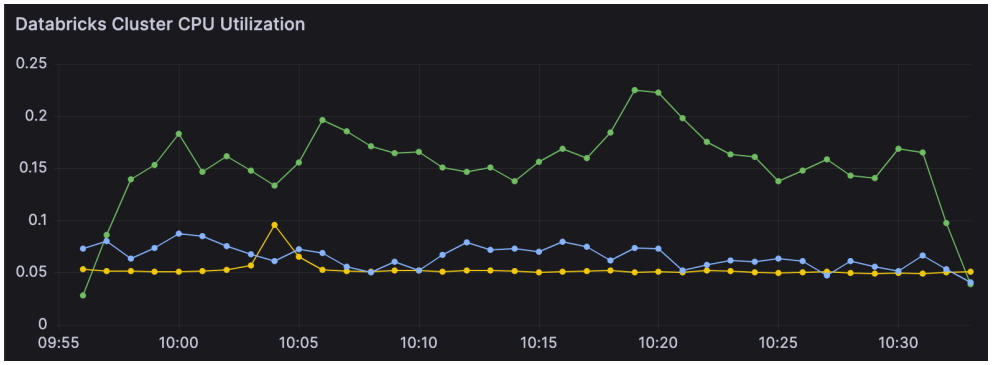


Databricks cluster disk write bytes

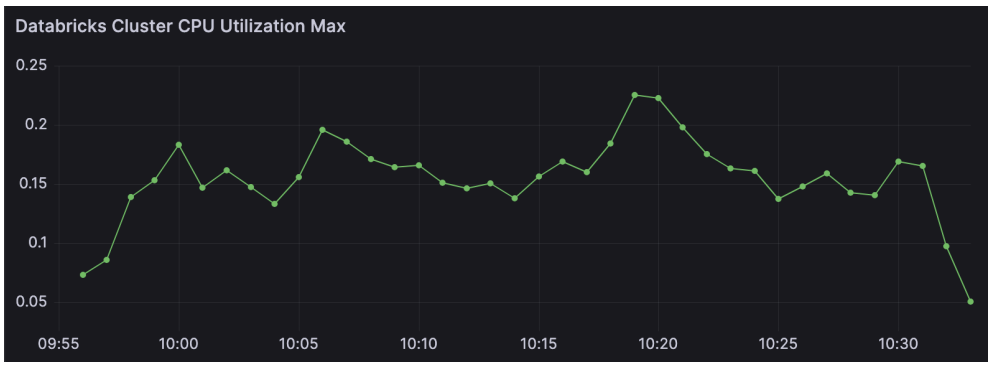


Databricks cluster memory usage

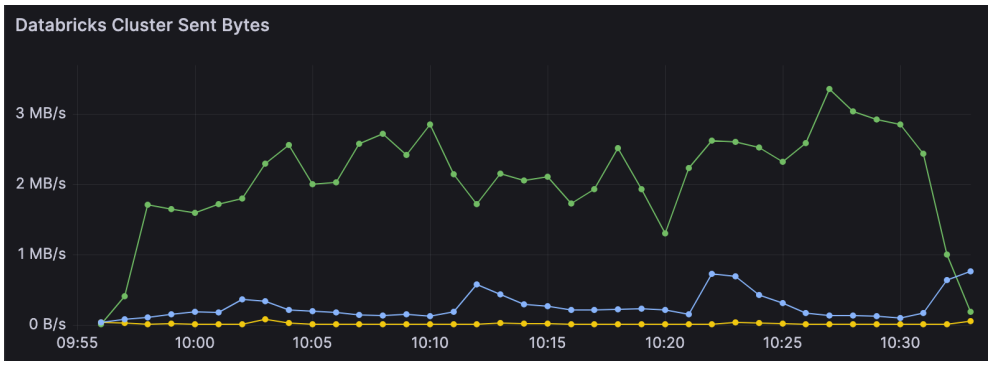
C.2.3 Scale 8



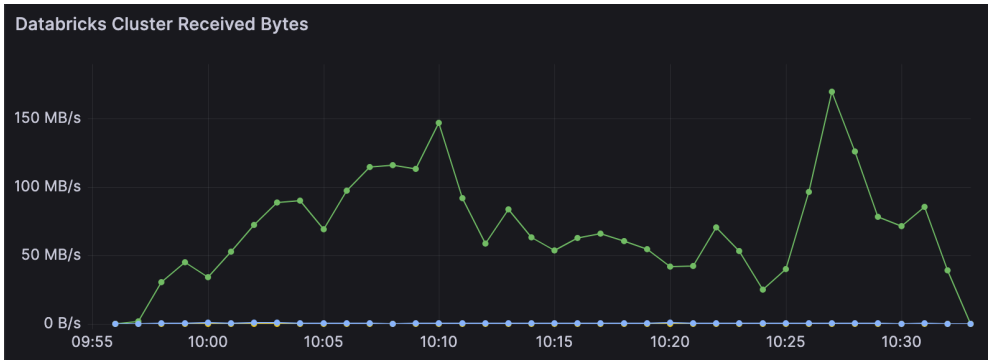
Databricks cluster cpu utilization



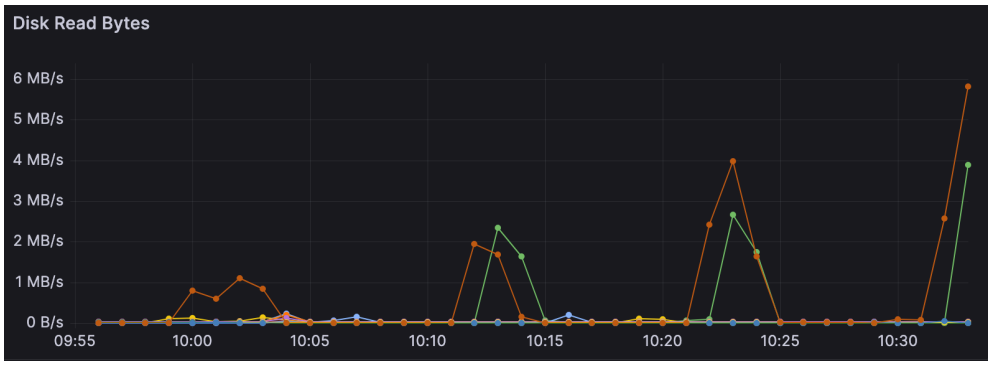
Databricks cluster cpu utilization max



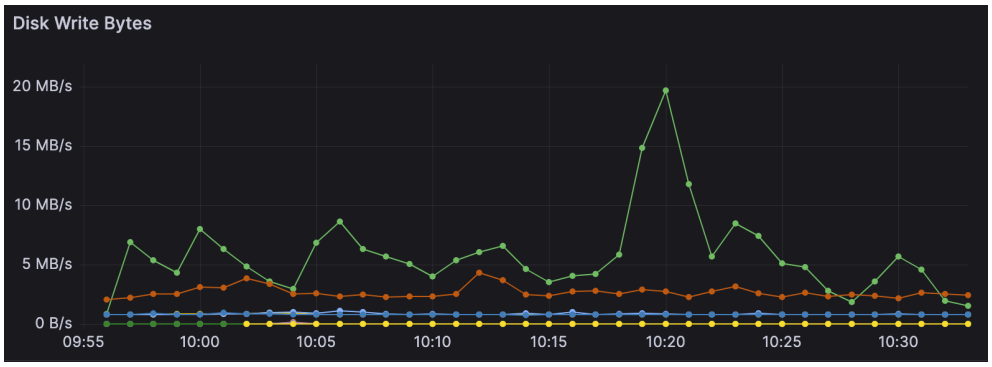
Databricks cluster sent bytes



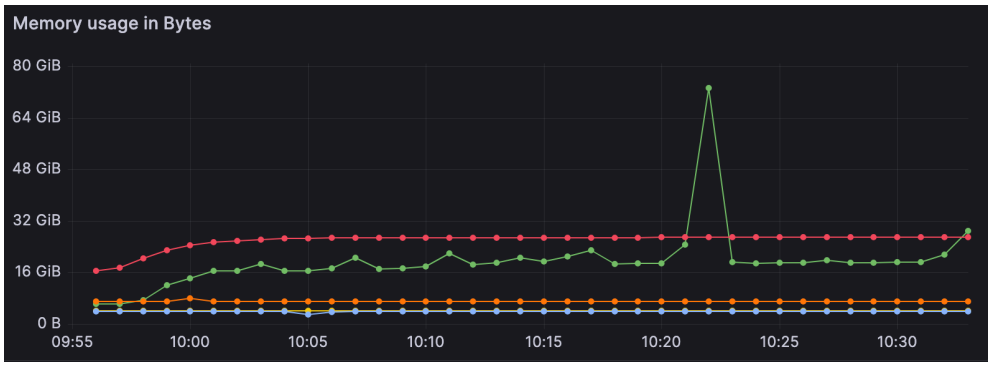
Databricks cluster received bytes



Databricks cluster disk read bytes

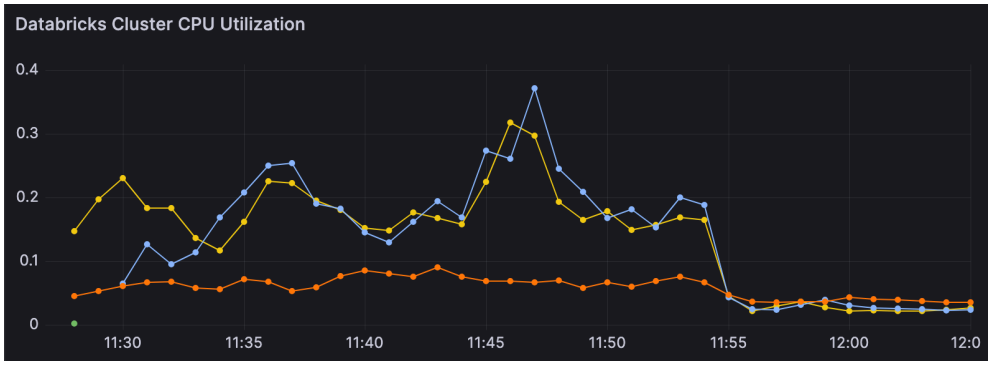


Databricks cluster disk write bytes

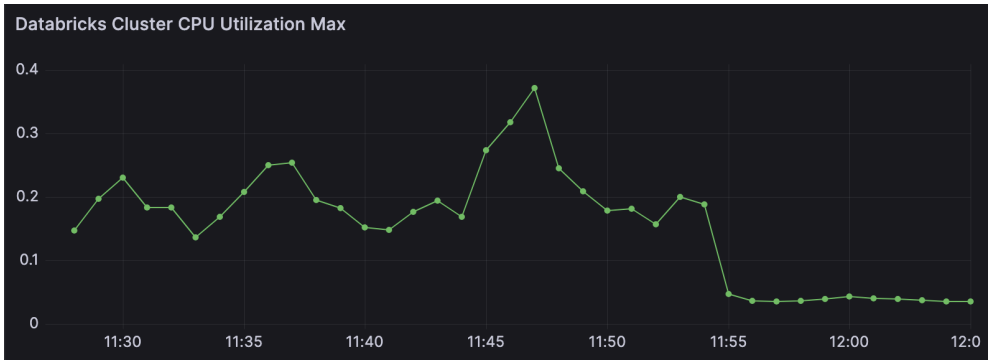


Databricks cluster memory usage

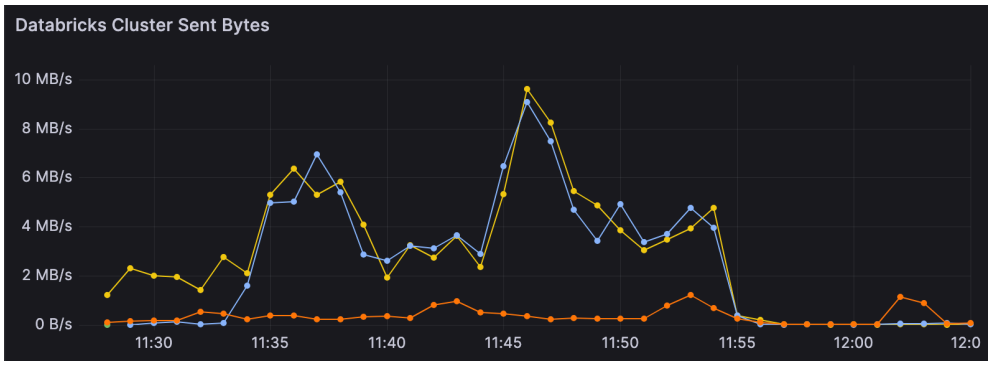
C.2.4 Scale 16



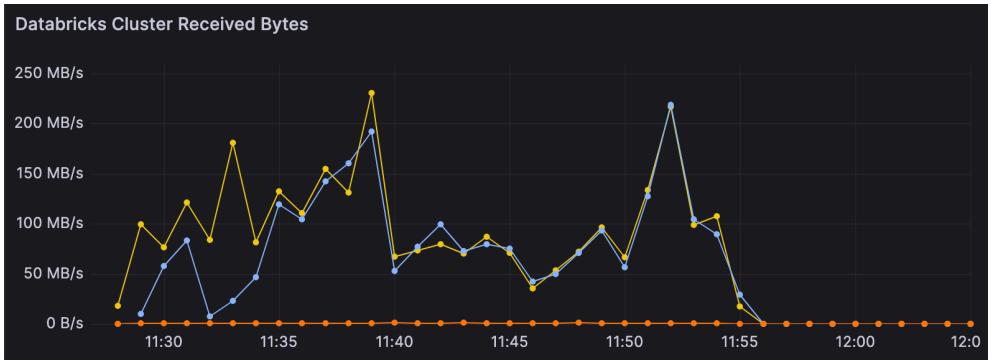
Databricks cluster cpu utilization



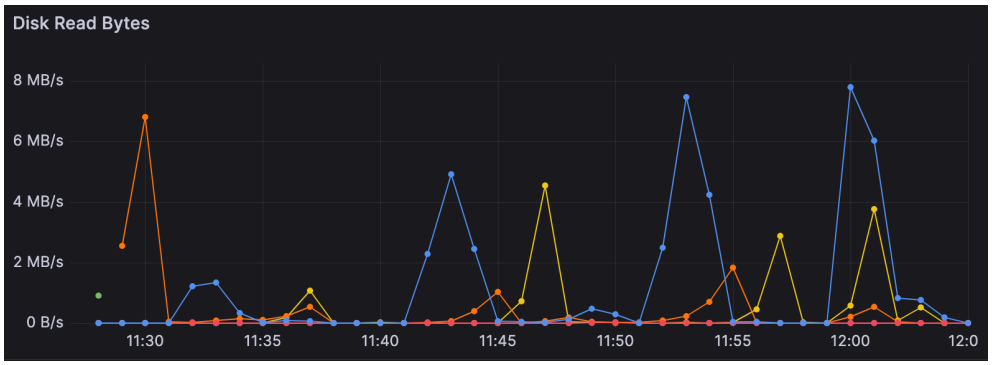
Databricks cluster cpu utilization max



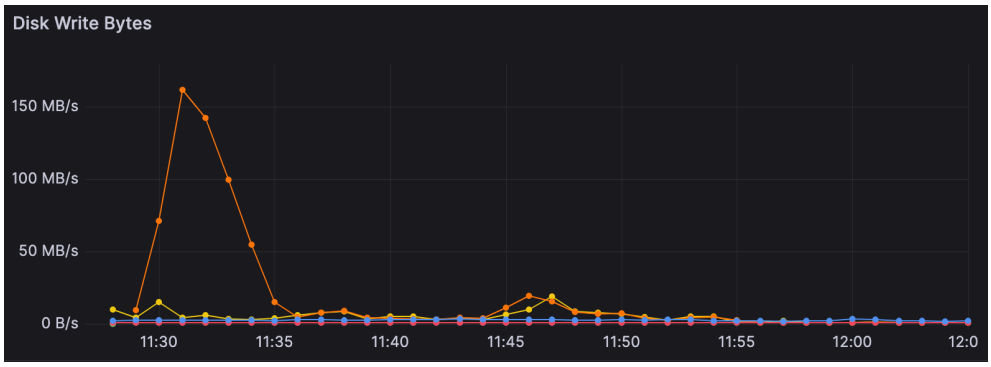
Databricks cluster sent bytes



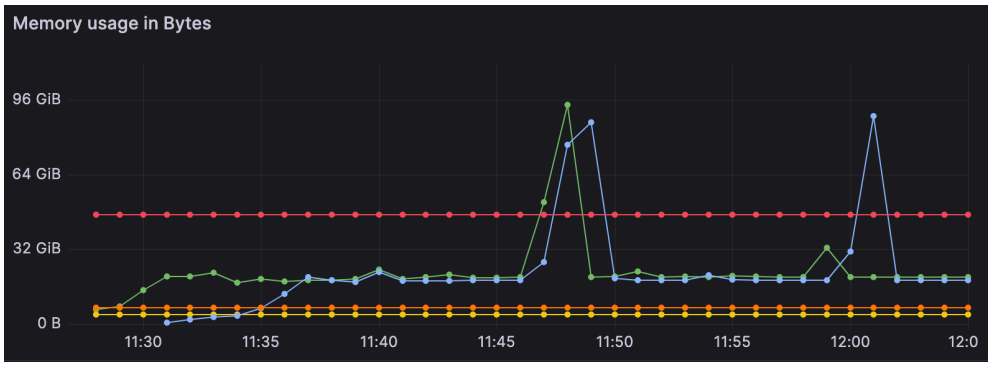
Databricks cluster received bytes



Databricks cluster disk read bytes

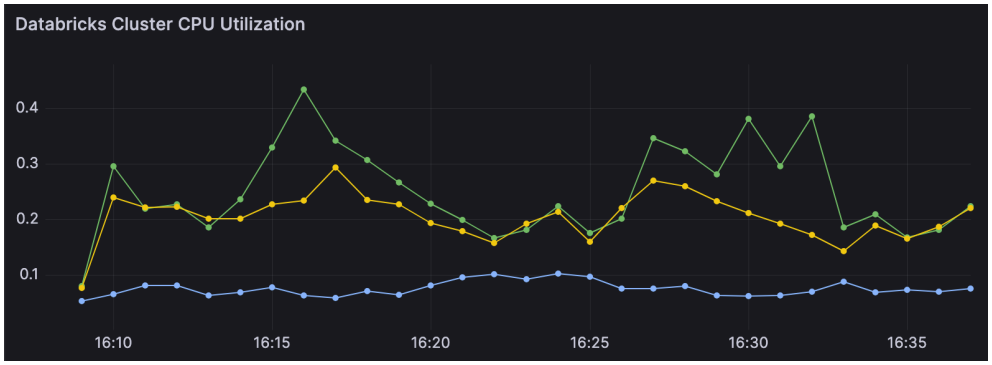


Databricks cluster disk write bytes

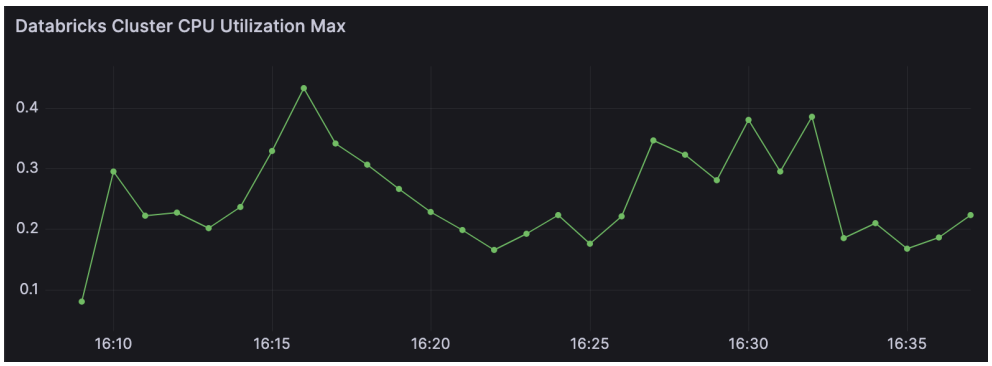


Databricks cluster memory usage

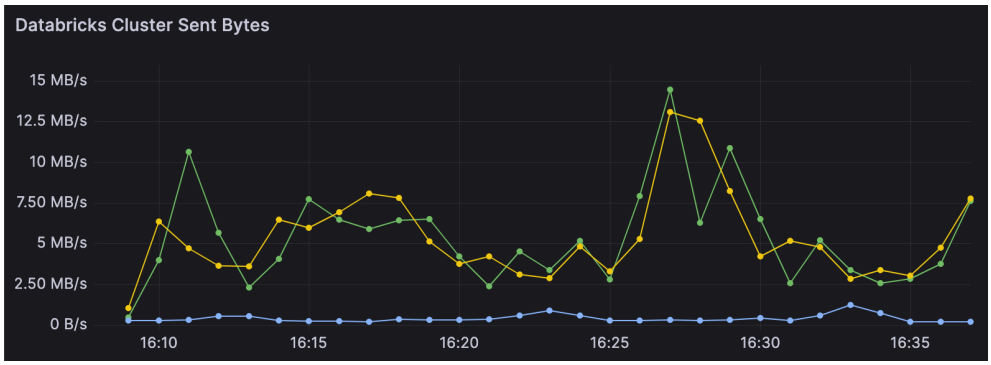
C.2.5 Scale 32



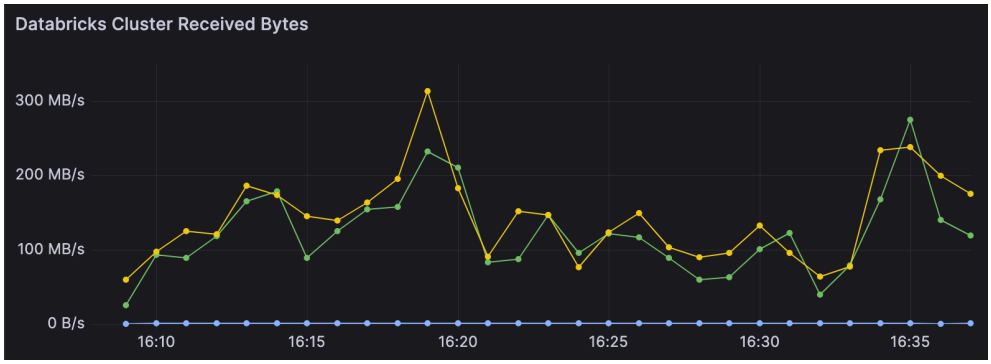
Databricks cluster cpu utilization



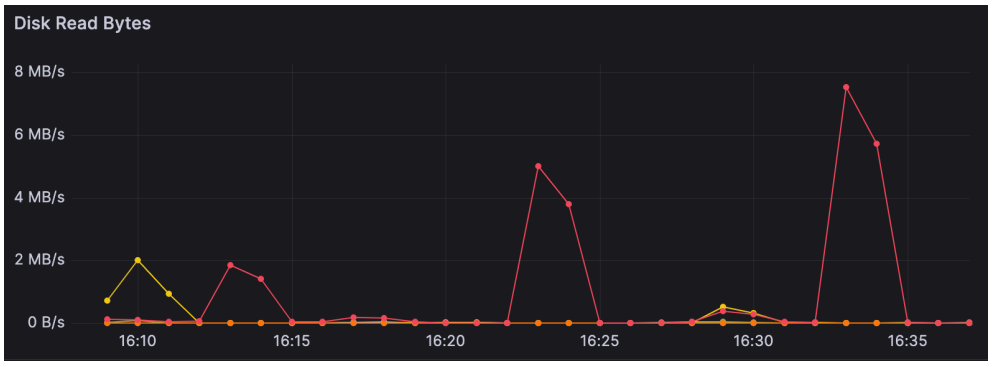
Databricks cluster cpu utilization max



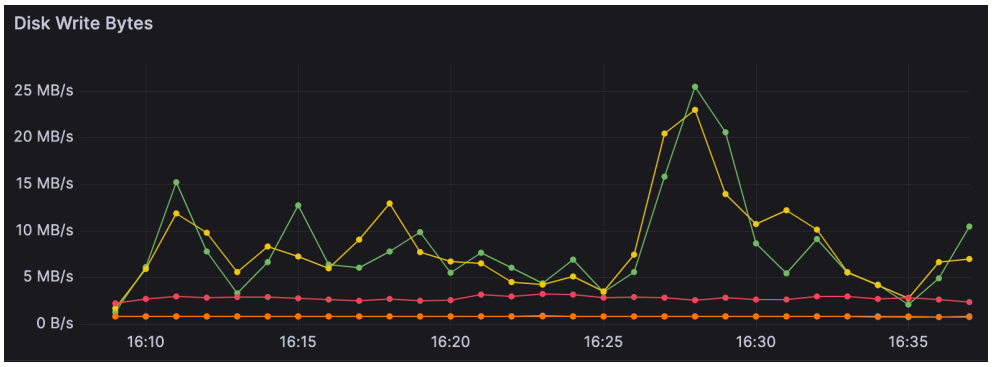
Databricks cluster sent bytes



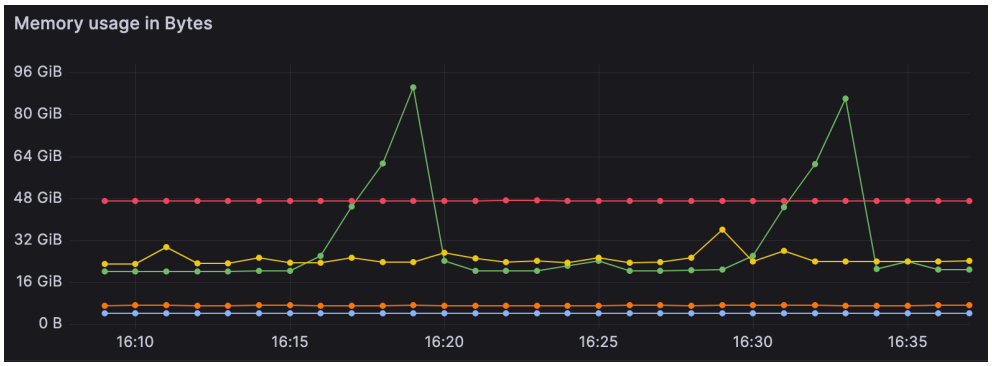
Databricks cluster received bytes



Databricks cluster disk read bytes

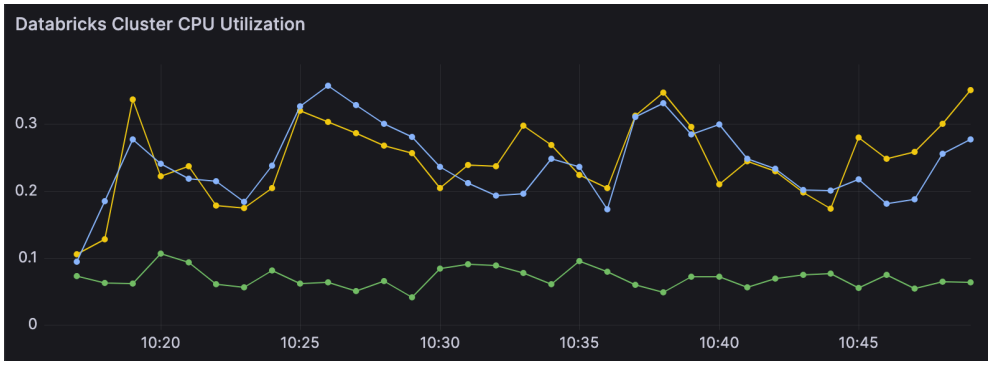


Databricks cluster disk write bytes

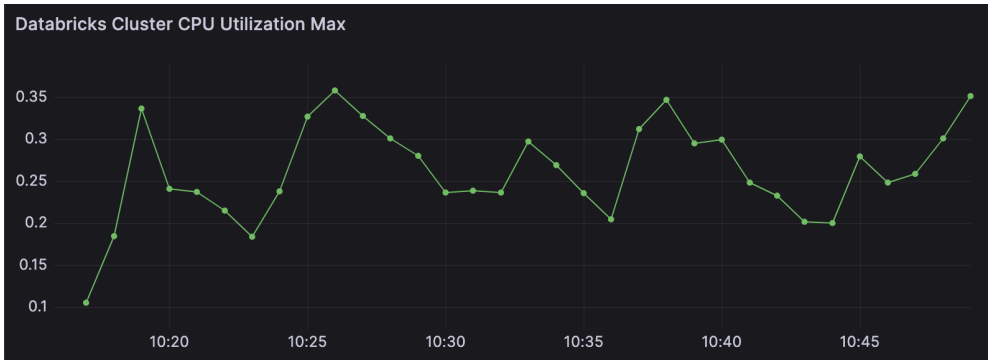


Databricks cluster memory usage

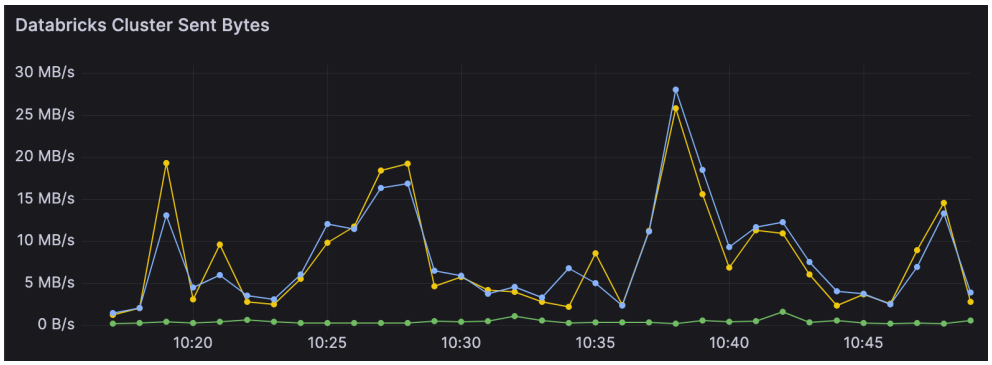
C.2.6 Scale 64



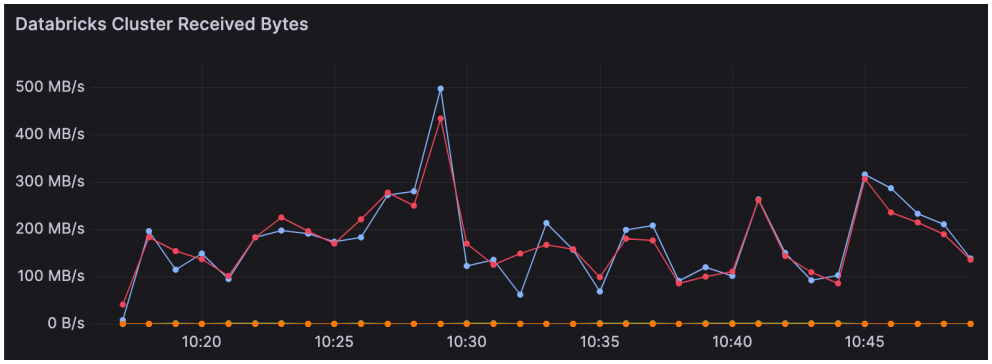
Databricks cluster cpu utilization



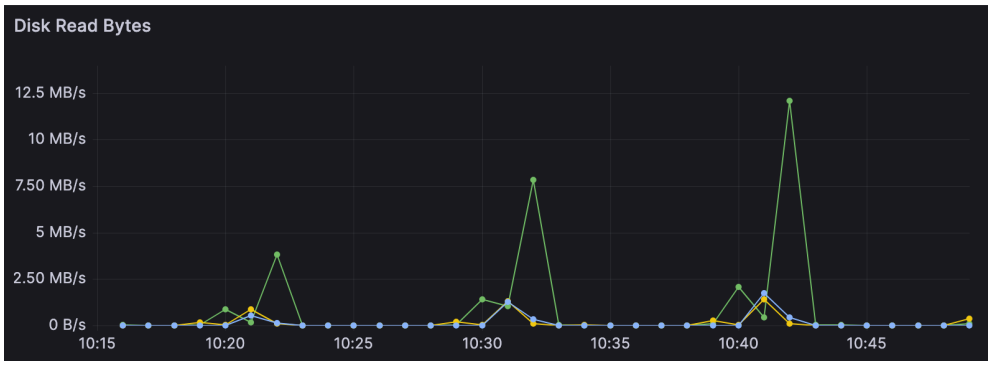
Databricks cluster cpu utilization max



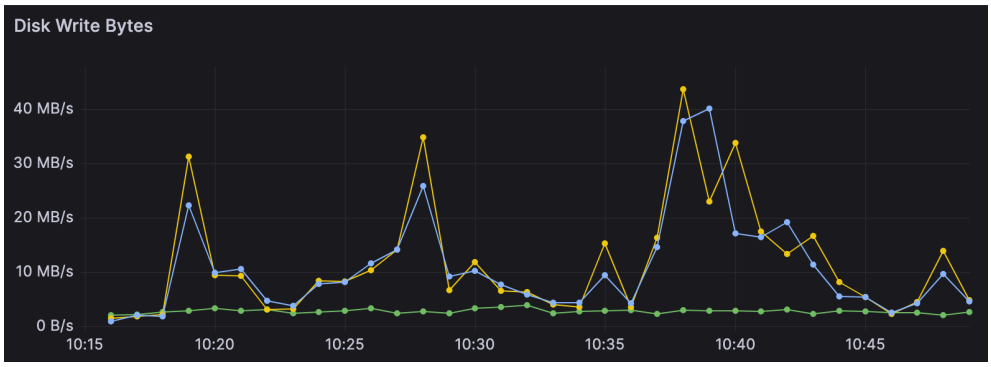
Databricks cluster sent bytes



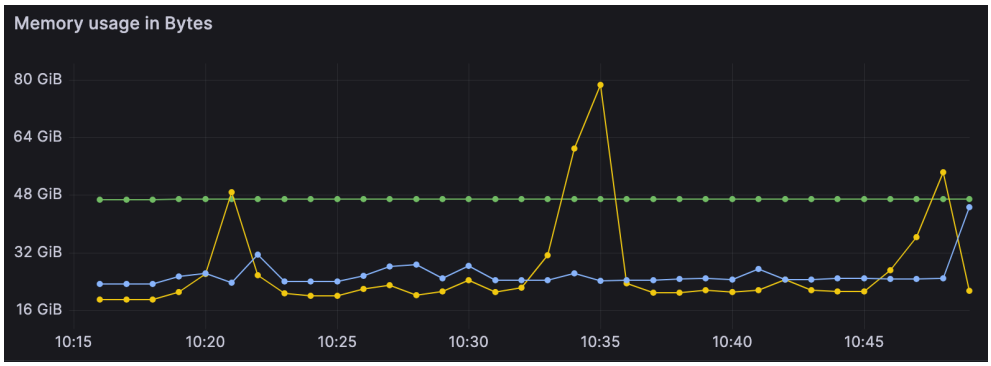
Databricks cluster received bytes



Databricks cluster disk read bytes



Databricks cluster disk write bytes



Databricks cluster memory usage

Appendix D

TPC-DS Queries

D.1 Query 88

```
select  *
from
 (select count(*) h8_30_to_9
 from store_sales, household_demographics , time_dim,
    store
 where ss_sold_time_sk = time_dim.t_time_sk
     and ss_hdemo_sk = household_demographics.hd_demo_sk
     and ss_store_sk = s_store_sk
     and time_dim.t_hour = 8
     and time_dim.t_minute >= 30
     and ((household_demographics.hd_dep_count = -1 and
        household_demographics.hd_vehicle_count<=-1+2)
        or
          (household_demographics.hd_dep_count = 2 and
             household_demographics.hd_vehicle_count
             <=2+2) or
          (household_demographics.hd_dep_count = 3 and
             household_demographics.hd_vehicle_count
             <=3+2))
     and store.s_store_name = 'ese') s1,
 (select count(*) h9_to_9_30
 from store_sales, household_demographics , time_dim,
    store
 where ss_sold_time_sk = time_dim.t_time_sk
     and ss_hdemo_sk = household_demographics.hd_demo_sk
```

```
18      and ss_store_sk = s_store_sk
19      and time_dim.t_hour = 9
20      and time_dim.t_minute < 30
21      and ((household_demographics.hd_dep_count = -1 and
           household_demographics.hd_vehicle_count<=-1+2)
           or
22           (household_demographics.hd_dep_count = 2 and
              household_demographics.hd_vehicle_count
              <=2+2) or
23           (household_demographics.hd_dep_count = 3 and
              household_demographics.hd_vehicle_count
              <=3+2))
24      and store.s_store_name = 'ese') s2,
25  (select count(*) h9_30_to_10
26  from store_sales, household_demographics , time_dim,
       store
27  where ss_sold_time_sk = time_dim.t_time_sk
28      and ss_hdemo_sk = household_demographics.hd_demo_sk
29      and ss_store_sk = s_store_sk
30      and time_dim.t_hour = 9
31      and time_dim.t_minute >= 30
32      and ((household_demographics.hd_dep_count = -1 and
           household_demographics.hd_vehicle_count<=-1+2)
           or
33           (household_demographics.hd_dep_count = 2 and
              household_demographics.hd_vehicle_count
              <=2+2) or
34           (household_demographics.hd_dep_count = 3 and
              household_demographics.hd_vehicle_count
              <=3+2))
35      and store.s_store_name = 'ese') s3,
36  (select count(*) h10_to_10_30
37  from store_sales, household_demographics , time_dim,
       store
38  where ss_sold_time_sk = time_dim.t_time_sk
39      and ss_hdemo_sk = household_demographics.hd_demo_sk
40      and ss_store_sk = s_store_sk
41      and time_dim.t_hour = 10
42      and time_dim.t_minute < 30
43      and ((household_demographics.hd_dep_count = -1 and
           household_demographics.hd_vehicle_count<=-1+2)
           or
```

```
44            (household_demographics.hd_dep_count = 2 and
                 household_demographics.hd_vehicle_count
                 <=2+2) or
45            (household_demographics.hd_dep_count = 3 and
                 household_demographics.hd_vehicle_count
                 <=3+2))
46      and store.s_store_name = 'ese') s4,
47  (select count(*) h10_30_to_11
48  from store_sales, household_demographics , time_dim,
       store
49  where ss_sold_time_sk = time_dim.t_time_sk
50      and ss_hdemo_sk = household_demographics.hd_demo_sk
51      and ss_store_sk = s_store_sk
52      and time_dim.t_hour = 10
53      and time_dim.t_minute >= 30
54      and ((household_demographics.hd_dep_count = -1 and
           household_demographics.hd_vehicle_count<=-1+2)
           or
55            (household_demographics.hd_dep_count = 2 and
                 household_demographics.hd_vehicle_count
                 <=2+2) or
56            (household_demographics.hd_dep_count = 3 and
                 household_demographics.hd_vehicle_count
                 <=3+2))
57      and store.s_store_name = 'ese') s5,
58  (select count(*) h11_to_11_30
59  from store_sales, household_demographics , time_dim,
       store
60  where ss_sold_time_sk = time_dim.t_time_sk
61      and ss_hdemo_sk = household_demographics.hd_demo_sk
62      and ss_store_sk = s_store_sk
63      and time_dim.t_hour = 11
64      and time_dim.t_minute < 30
65      and ((household_demographics.hd_dep_count = -1 and
           household_demographics.hd_vehicle_count<=-1+2)
           or
66            (household_demographics.hd_dep_count = 2 and
                 household_demographics.hd_vehicle_count
                 <=2+2) or
67            (household_demographics.hd_dep_count = 3 and
                 household_demographics.hd_vehicle_count
                 <=3+2))
```

```
68      and store.s_store_name = 'ese') s6,
69   (select count(*) h11_30_to_12
70   from store_sales, household_demographics , time_dim,
        store
71   where ss_sold_time_sk = time_dim.t_time_sk
72       and ss_hdemo_sk = household_demographics.hd_demo_sk
73       and ss_store_sk = s_store_sk
74       and time_dim.t_hour = 11
75       and time_dim.t_minute >= 30
76       and ((household_demographics.hd_dep_count = -1 and
             household_demographics.hd_vehicle_count<=-1+2)
             or
77             (household_demographics.hd_dep_count = 2 and
                household_demographics.hd_vehicle_count
                <=2+2) or
78             (household_demographics.hd_dep_count = 3 and
                household_demographics.hd_vehicle_count
                <=3+2))
79       and store.s_store_name = 'ese') s7,
80   (select count(*) h12_to_12_30
81   from store_sales, household_demographics , time_dim,
        store
82   where ss_sold_time_sk = time_dim.t_time_sk
83       and ss_hdemo_sk = household_demographics.hd_demo_sk
84       and ss_store_sk = s_store_sk
85       and time_dim.t_hour = 12
86       and time_dim.t_minute < 30
87       and ((household_demographics.hd_dep_count = -1 and
             household_demographics.hd_vehicle_count<=-1+2)
             or
88             (household_demographics.hd_dep_count = 2 and
                household_demographics.hd_vehicle_count
                <=2+2) or
89             (household_demographics.hd_dep_count = 3 and
                household_demographics.hd_vehicle_count
                <=3+2))
90       and store.s_store_name = 'ese') s8
91   ;
```

D.2 Query 28

```
SELECT *
FROM   (SELECT Avg(ss_list_price)            B1_LP,
               Count(ss_list_price)          B1_CNT,
               Count(DISTINCT ss_list_price) B1_CNTD
        FROM   store_sales
        WHERE  ss_quantity BETWEEN 0 AND 5
               AND ( ss_list_price BETWEEN 18 AND 18 +
                  10
                    OR ss_coupon_amt BETWEEN 1939 AND
                       1939 + 1000
                    OR ss_wholesale_cost BETWEEN 34
                       AND 34 + 20 )) B1,
       (SELECT Avg(ss_list_price)            B2_LP,
               Count(ss_list_price)          B2_CNT,
               Count(DISTINCT ss_list_price) B2_CNTD
        FROM   store_sales
        WHERE  ss_quantity BETWEEN 6 AND 10
               AND ( ss_list_price BETWEEN 1 AND 1 + 10
                    OR ss_coupon_amt BETWEEN 35 AND 35
                        + 1000
                    OR ss_wholesale_cost BETWEEN 50
                       AND 50 + 20 )) B2,
       (SELECT Avg(ss_list_price)            B3_LP,
               Count(ss_list_price)          B3_CNT,
               Count(DISTINCT ss_list_price) B3_CNTD
        FROM   store_sales
        WHERE  ss_quantity BETWEEN 11 AND 15
               AND ( ss_list_price BETWEEN 91 AND 91 +
                  10
                    OR ss_coupon_amt BETWEEN 1412 AND
                       1412 + 1000
                    OR ss_wholesale_cost BETWEEN 17
                       AND 17 + 20 )) B3,
       (SELECT Avg(ss_list_price)            B4_LP,
               Count(ss_list_price)          B4_CNT,
               Count(DISTINCT ss_list_price) B4_CNTD
        FROM   store_sales
        WHERE  ss_quantity BETWEEN 16 AND 20
               AND ( ss_list_price BETWEEN 9 AND 9 + 10
```

```
                      OR ss_coupon_amt BETWEEN 5270 AND
                         5270 + 1000
                      OR ss_wholesale_cost BETWEEN 29
                         AND 29 + 20 )) B4,
       (SELECT Avg(ss_list_price)            B5_LP,
               Count(ss_list_price)          B5_CNT,
               Count(DISTINCT ss_list_price) B5_CNTD
        FROM   store_sales
        WHERE  ss_quantity BETWEEN 21 AND 25
               AND ( ss_list_price BETWEEN 45 AND 45 +
                  10
                      OR ss_coupon_amt BETWEEN 826 AND
                         826 + 1000
                      OR ss_wholesale_cost BETWEEN 5 AND
                          5 + 20 )) B5,
       (SELECT Avg(ss_list_price)            B6_LP,
               Count(ss_list_price)          B6_CNT,
               Count(DISTINCT ss_list_price) B6_CNTD
        FROM   store_sales
        WHERE  ss_quantity BETWEEN 26 AND 30
               AND ( ss_list_price BETWEEN 174 AND 174 +
                   10
                      OR ss_coupon_amt BETWEEN 5548 AND
                         5548 + 1000
                      OR ss_wholesale_cost BETWEEN 42
                         AND 42 + 20 )) B6
LIMIT 100;
```

D.3 Query 14

```
WITH item_ss AS (
    SELECT DISTINCT
        iss.i_brand_id,
        iss.i_class_id,
        iss.i_category_id
    FROM   store_sales,
        item iss,
        date_dim d1
    WHERE  ss_item_sk = iss.i_item_sk
        AND ss_sold_date_sk = d1.d_date_sk
```

```
11          AND d1.d_year BETWEEN 1999 AND 1999 + 2
12 ), item_cs AS (
13     SELECT DISTINCT
14         ics.i_brand_id,
15         ics.i_class_id,
16         ics.i_category_id
17     FROM   catalog_sales,
18         item ics,
19         date_dim d2
20     WHERE  cs_item_sk = ics.i_item_sk
21         AND cs_sold_date_sk = d2.d_date_sk
22         AND d2.d_year BETWEEN 1999 AND 1999 + 2
23 ), item_ws AS (
24     SELECT DISTINCT
25         iws.i_brand_id,
26         iws.i_class_id,
27         iws.i_category_id
28     FROM   web_sales,
29         item iws,
30         date_dim d3
31     WHERE  ws_item_sk = iws.i_item_sk
32         AND ws_sold_date_sk = d3.d_date_sk
33         AND d3.d_year BETWEEN 1999 AND 1999 + 2
34 ), item_intersect AS (
35     SELECT
36         item_ss.i_brand_id    brand_id,
37         item_ss.i_class_id    class_id,
38         item_ss.i_category_id category_id
39     FROM item_ss
40     JOIN item_ws ON item_ss.i_brand_id = item_ws.
          i_brand_id
41         AND item_ss.i_class_id = item_ws.i_class_id
42         AND item_ss.i_category_id = item_ws.
              i_category_id
43     JOIN item_cs ON item_ss.i_brand_id = item_cs.
          i_brand_id
44         AND item_ss.i_class_id = item_cs.i_class_id
45         AND item_ss.i_category_id = item_cs.
              i_category_id
46 ), cross_items AS (
47          SELECT i_item_sk ss_item_sk
48          FROM   item,
```

```
                item_intersect
          WHERE  i_brand_id = brand_id
                 AND i_class_id = class_id
                 AND i_category_id = category_id),
     avg_sales
     AS (SELECT Avg(quantity * list_price) average_sales
         FROM   (SELECT ss_quantity   quantity,
                        ss_list_price list_price
                 FROM   store_sales,
                        date_dim
                 WHERE  ss_sold_date_sk = d_date_sk
                        AND d_year BETWEEN 1999 AND 1999
                            + 2
                 UNION ALL
                 SELECT cs_quantity   quantity,
                        cs_list_price list_price
                 FROM   catalog_sales,
                        date_dim
                 WHERE  cs_sold_date_sk = d_date_sk
                        AND d_year BETWEEN 1999 AND 1999
                            + 2
                 UNION ALL
                 SELECT ws_quantity   quantity,
                        ws_list_price list_price
                 FROM   web_sales,
                        date_dim
                 WHERE  ws_sold_date_sk = d_date_sk
                        AND d_year BETWEEN 1999 AND 1999
                            + 2) x)
SELECT channel,
               i_brand_id,
               i_class_id,
               i_category_id,
               Sum(sales),
               Sum(number_sales)
FROM  (SELECT 'store'                                channel,
              i_brand_id,
              i_class_id,
              i_category_id,
              Sum(ss_quantity * ss_list_price) sales,
              Count(*)
                 number_sales
```

```
87          FROM   store_sales,
88                 item,
89                 date_dim
90          WHERE  ss_item_sk IN (SELECT ss_item_sk
91                                FROM   cross_items)
92                 AND ss_item_sk = i_item_sk
93                 AND ss_sold_date_sk = d_date_sk
94                 AND d_year = 1999 + 2
95                 AND d_moy = 11
96          GROUP  BY i_brand_id,
97                    i_class_id,
98                    i_category_id
99          HAVING Sum(ss_quantity * ss_list_price) > (SELECT
                average_sales
100                                                   FROM
                                                        avg_sales
                                                        )
101         UNION ALL
102         SELECT 'catalog'                          channel,
103                i_brand_id,
104                i_class_id,
105                i_category_id,
106                Sum(cs_quantity * cs_list_price) sales,
107                Count(*)
                      number_sales
108         FROM   catalog_sales,
109                item,
110                date_dim
111         WHERE  cs_item_sk IN (SELECT ss_item_sk
112                               FROM   cross_items)
113                AND cs_item_sk = i_item_sk
114                AND cs_sold_date_sk = d_date_sk
115                AND d_year = 1999 + 2
116                AND d_moy = 11
117         GROUP  BY i_brand_id,
118                   i_class_id,
119                   i_category_id
120         HAVING Sum(cs_quantity * cs_list_price) > (SELECT
                average_sales
121                                                   FROM
```

```
                                                              avg_sales
                                                              )
        UNION ALL
        SELECT 'web'                                  channel,
               i_brand_id,
               i_class_id,
               i_category_id,
               Sum(ws_quantity * ws_list_price) sales,
               Count(*)
                  number_sales
        FROM   web_sales,
               item,
               date_dim
        WHERE  ws_item_sk IN (SELECT ss_item_sk
                              FROM   cross_items)
               AND ws_item_sk = i_item_sk
               AND ws_sold_date_sk = d_date_sk
               AND d_year = 1999 + 2
               AND d_moy = 11
        GROUP  BY i_brand_id,
                  i_class_id,
                  i_category_id
        HAVING Sum(ws_quantity * ws_list_price) > (SELECT
            average_sales
                                                     FROM

                                                        avg_sales
                                                        ))
                                                        y
GROUP  BY channel, i_brand_id, i_class_id, i_category_id
ORDER  BY channel,
          i_brand_id,
          i_class_id,
          i_category_id
LIMIT 100;
```

D.4 Query 75

```
WITH all_sales
     AS (SELECT d_year,
```

```
3                 i_brand_id,
4                 i_class_id,
5                 i_category_id,
6                 i_manufact_id,
7                 Sum(sales_cnt) AS sales_cnt,
8                 Sum(sales_amt) AS sales_amt
9           FROM  (SELECT d_year,
10                        i_brand_id,
11                        i_class_id,
12                        i_category_id,
13                        i_manufact_id,
14                        cs_quantity - COALESCE(
                             cr_return_quantity, 0)
                                    AS
15                        sales_cnt,
16                        cs_ext_sales_price - COALESCE(
                             cr_return_amount, 0.0) AS
17                        sales_amt
18                 FROM   catalog_sales
19                        JOIN item
20                          ON i_item_sk = cs_item_sk
21                        JOIN date_dim
22                          ON d_date_sk = cs_sold_date_sk
23                        LEFT JOIN catalog_returns
24                                ON ( cs_order_number =
                                     cr_order_number
25                                  AND cs_item_sk =
                                        cr_item_sk )
26                 WHERE  i_category = 'Men'
27                 UNION ALL
28                 SELECT d_year,
29                        i_brand_id,
30                        i_class_id,
31                        i_category_id,
32                        i_manufact_id,
33                        ss_quantity - COALESCE(
                             sr_return_quantity, 0)     AS
34                        sales_cnt,
35                        ss_ext_sales_price - COALESCE(
                             sr_return_amt, 0.0) AS
36                        sales_amt
37                 FROM   store_sales
```

```
38                          JOIN item
39                            ON i_item_sk = ss_item_sk
40                          JOIN date_dim
41                            ON d_date_sk = ss_sold_date_sk
42                          LEFT JOIN store_returns
43                                  ON ( ss_ticket_number =
                                       sr_ticket_number
44                                    AND ss_item_sk =
                                          sr_item_sk )
45                  WHERE  i_category = 'Men'
46                  UNION ALL
47                  SELECT d_year,
48                         i_brand_id,
49                         i_class_id,
50                         i_category_id,
51                         i_manufact_id,
52                         ws_quantity - COALESCE(
                              wr_return_quantity, 0)     AS
53                         sales_cnt,
54                         ws_ext_sales_price - COALESCE(
                              wr_return_amt, 0.0) AS
55                         sales_amt
56                  FROM   web_sales
57                         JOIN item
58                           ON i_item_sk = ws_item_sk
59                         JOIN date_dim
60                           ON d_date_sk = ws_sold_date_sk
61                         LEFT JOIN web_returns
62                                ON ( ws_order_number =
                                     wr_order_number
63                                  AND ws_item_sk =
                                        wr_item_sk )
64                  WHERE  i_category = 'Men') sales_detail
65          GROUP  BY d_year,
66                    i_brand_id,
67                    i_class_id,
68                    i_category_id,
69                    i_manufact_id)
70  SELECT prev_yr.d_year                          AS
       prev_year,
71                 curr_yr.d_year                          AS
                      year1,
```

```
72                curr_yr.i_brand_id,
73                curr_yr.i_class_id,
74                curr_yr.i_category_id,
75                curr_yr.i_manufact_id,
76                prev_yr.sales_cnt                       AS
                     prev_yr_cnt,
77                curr_yr.sales_cnt                       AS
                     curr_yr_cnt,
78                curr_yr.sales_cnt - prev_yr.sales_cnt AS
                     sales_cnt_diff,
79                curr_yr.sales_amt - prev_yr.sales_amt AS
                     sales_amt_diff
80  FROM   all_sales curr_yr,
81         all_sales prev_yr
82  WHERE  curr_yr.i_brand_id = prev_yr.i_brand_id
83         AND curr_yr.i_class_id = prev_yr.i_class_id
84         AND curr_yr.i_category_id = prev_yr.i_category_id
85         AND curr_yr.i_manufact_id = prev_yr.i_manufact_id
86         AND curr_yr.d_year = 2002
87         AND prev_yr.d_year = 2002 - 1
88         AND curr_yr.sales_cnt / prev_yr.sales_cnt
89             < 0.9
90  ORDER  BY sales_cnt_diff
91  LIMIT 100;
```

D.5 Query 24

```
1   WITH ssales
2        AS (SELECT c_last_name,
3                   c_first_name,
4                   s_store_name,
5                   ca_state,
6                   s_state,
7                   i_color,
8                   i_current_price,
9                   i_manager_id,
10                  i_units,
11                  i_size,
12                  Sum(ss_net_profit) netpaid
13            FROM   store_sales,
```

```
                store_returns,
                store,
                item,
                customer,
                customer_address
        WHERE   ss_ticket_number = sr_ticket_number
                AND ss_item_sk = sr_item_sk
                AND ss_customer_sk = c_customer_sk
                AND ss_item_sk = i_item_sk
                AND ss_store_sk = s_store_sk
                AND c_birth_country = Upper(ca_country)
                AND s_zip = ca_zip
                AND s_market_id = 6
        GROUP   BY c_last_name,
                   c_first_name,
                   s_store_name,
                   ca_state,
                   s_state,
                   i_color,
                   i_current_price,
                   i_manager_id,
                   i_units,
                   i_size)
SELECT c_last_name,
       c_first_name,
       s_store_name,
       Sum(netpaid) paid
FROM   ssales
WHERE  i_color = 'papaya'
GROUP  BY c_last_name,
          c_first_name,
          s_store_name
HAVING Sum(netpaid) > (SELECT 0.05 * Avg(netpaid)
                       FROM   ssales);
```

TRITA-EECS-EX-2024:38