University of Piraeus

Department of Digital Systems

MSc Information Systems & Services

Big Data and Analytics

Comparative Analysis of SQL Queries Performance on Vehicle Sensor Data in RDBMS and Apache Spark

Koutsimpogiorgos Grigorios

ME2022

Supervisor
Prof. Doulkeridis Christos

Piraeus, September 2023

Abstract

In today's digital era, the exponential growth in data volume, variety, and velocity has necessitated the exploration of advanced techniques for storing and analyzing big data. The continuous improvement of hardware has led to the development of new technologies for data storage and processing by expanding the traditional data storages technologies and analysis frameworks. Many organizations are turning to distributed computing frameworks to process and analyze large datasets.

One of the most popular technologies in this field is Apache Spark, a fast and general-purpose cluster computing system. However, traditional relational databases, such as Oracle, are still widely used for data storage and retrieval. In this thesis we are comparing the performance of a specific set of queries, on a Vehicle Sensor Dataset, executed on both a traditional RDBMS system as well as on Apache spark. Our goal is to determine whether modern technologies can perform as well or even better than relational databases when it comes to processing and analyzing large data sets like in our case. Additionally, the thesis explores the optimization techniques that can be used to improve the performance of Spark and Oracle.

Table of Contents

Περίληψη

Στη σημερινή ψηφιακή εποχή, η αύξηση στον όγκο, την ποικιλομορφία και την ταχύτητα των δεδομένων έχει καταστήσει αναγκαία την αναζήτηση προηγμένων τεχνικών αποθήκευσης και ανάλυσης μεγάλων δεδομένων. Η συνεχής βελτίωση των υπολογιστών και η ανάπτυξη νέων τεχνολογιών για την αποθήκευση και επεξεργασία δεδομένων επεκτείνουν τις παραδοσιακές τεχνολογίες αποθήκευσης δεδομένων και πλαισίων ανάλυσης. Πολλοί οργανισμοί στρέφονται σε κατανεμημένα υπολογιστικά συστήματα για την επεξεργασία και ανάλυση μεγάλων συνόλων δεδομένων.

Μία από τις πιο δημοφιλείς τεχνολογίες σε αυτόν τον τομέα είναι το Apache Spark, ένα γρήγορο και γενικής χρήσης σύστημα που λειτουργεί πάνω σε υπολογιστικούς κόμβους. Ωστόσο, τα παραδοσιακά σχεσιακά συστήματα βάσεων δεδομένων, όπως η Oracle, χρησιμοποιούνται ακόμα ευρέως για την αποθήκευση και ανάκτηση δεδομένων. Σε αυτήν την διπλωματική εργασία συγκρίνουμε την απόδοση ενός συγκεκριμένου συνόλου ερωτημάτων, πάνω σε ένα σύνολο δεδομένων συλλεγμένα από αισθητήρες οχημάτων, που εκτελέστηκαν τόσο σε ένα παραδοσιακό σύστημα σχεσιακής βάσης δεδομένων όσο και στο Apache Spark . Στόχος μας είναι να προσδιορίσουμε εάν οι σύγχρονες τεχνολογίες μπορούν να επιτύχουν ίση ή ακόμα καλύτερη απόδοση από τις σχεσιακές βάσεις δεδομένων όταν πρόκειται για την επεξεργασία και ανάλυση μεγάλων συνόλων δεδομένων, όπως στην περίπτωσή μας. Επιπλέον, η εργασία εξερευνά τις τεχνικές βελτιστοποίησης που μπορούν να χρησιμοποιηθούν για τη βελτίωση της απόδοσης του Spark καθώς και της Oracle.

1. Introduction

In recent years, big data has become an increasingly important topic and an integral part of many organizations' operations, with an increasing amount of data being generated and collected per day. Many organizations have begun to recognize the value of large sets of data as they can be used to support decision making and gain insights that can drive business growth. However, as the volume of data grows, so does the complexity of managing and processing it. Two popular technologies that are commonly used to manage and process (big) data are relational database management systems (RDBMS) and Apache Spark.

Relational databases, such as Oracle, MySQL, and SQL Server, have been widely used for decades to store, manage and query structured data. They are based on the relational model and use SQL (Structured Query Language) as the main means of data manipulation. RDBMSs are well-established, stable, and provide a powerful and flexible way to model and query data, making them a popular choice for managing structured data.

Apache Spark, is a more recent technology that has emerged as a powerful big data processing framework. It is built on top of the Hadoop ecosystem and provides a fast, in-memory data processing engine that can handle both structured and unstructured data. It provides a distributed computing model that allows it to scale out to large clusters of machines. Spark also has an interactive shell called Spark SQL, which allows for the use of SQL for querying data.

This thesis aims to compare the performance of the specific set of SQL queries in both RDBMS and Apache Spark for the Vehicle Sensor Dataset. The research will focus on identifying the strengths and weaknesses of each technology in terms of query performance and ease of use as well as investigating various performance optimization techniques for both Oracle RDBMS and Spark. The goal of this study is to provide insights that can help the organization to make informed decisions when choosing a technology for managing and processing big data.

2. Databases

2.1 Relational Database Management System

The Relational Database Management system (RDBMS) is a database management system that is based on the relational model introduced by Edgar F.Codd, an IBM Researcher [S1]. According to Codd's relational model, an RDBMS gives the opportunity to users to construct, update, manage and interact with a relational database.

In relational databases the data is structured into rows and columns, which collectively form a table. These tables can be interconnected to numerous other tables based on common data to each other, which are referred as keys. A key that can uniquely identify of a particular row, it is called primary key. When that primary key is incorporated into a record in another table, it is referred as foreign key. That connection gives the opportunity to users to create new datasets from data existing in various related tables with a single query.

The basic data structure used by relational databases are tables and views, with the primary components being columns and rows. The data must adhere to a strict schema, which allows a database management system to heavily co-optimize the data storage and processing. The most common RDBMSs today are using the Structured Query Language (SQL) and are mostly used to perform CRUD (create, read, update and delete) operations.

The two main SQL workloads for relational databases can be:

- **Online transaction processing (OLTP) workloads**

  OLTP workloads are typically high-concurrency, low-latency, simple queries that read or update a few records at a time. Bank account transactions are a typical example of an OLTP workload.

- **Online analytical processing (OLAP)**

OLAP workloads, like reporting, are typically complex queries (involving aggregates and joins) that require high-throughput scans over many records. Modern Datawarehouse solutions are OLAP systems.

2.1.1 Benefits of Relation Databases

The RDBMS offers a method for storing and fetching data, ensuring convenient access to individual data points. Organizations employ relational databases when they require the handling of structured data. Some of their main advantages of using RDBMS to manage and store data are:

- **Flexibility:**
  Adding, updating and deleting tables and relationships, adjusting data as necessary can take place without changing the database structure or impacting existing applications.

- **ACID compliance:**
  RDBMS supports ACID (Atomicity, Consistency, Isolation, Durability) operations.

- **Ease of use:**
  Complex queries can be performed using SQL which enables even non-technical users to interact with them.

- **Database Normalization:**
  The technique of normalization can be used reducing data redundancy and improving data integrity.

- **Data Integrity**:
  Referential integrity refers to the accuracy and consistency of data and it is achieved by using the primary and foreign keys.

These benefits collectively make relational databases a powerful and reliable choice for managing structured data. By providing flexibility, ACID compliance, ease of use through SQL, database normalization, and data integrity, RDBMS systems empower organizations to effectively store, manage, and analyze their data, driving informed decision-making and enabling efficient data operations.

2.1.2 Limitations of Relation Databases

One of the main limitations of RDBMS systems is scalability. The increased volume of data, leads to difficulty of storing and retrieving data quickly and efficiently. This can be particularly challenging when working with very large data sets.

Another limitation of RDBMSs regards the handling of unstructured data. RDBMSs are designed to work with structured data, where each record has a predefined set of fields and data types. However, many modern data sources, such as social media, IoT devices, and log files, generate unstructured data, which can be difficult to store and query using traditional RDBMSs. Additionally, RDBMSs are not designed for handling data with high velocity, which can be a problem for real-time or near real-time analytics applications.

2.1.3 Structured Query Language (SQL)

Structured Query Language (SQL) is a programming language for storing and processing information in relational database management systems (RDBMS). It serves as the standard language for interacting with databases, including popular RDBMSs like Oracle and Microsoft SQL Server. SQL is used to create and modify database structures, insert and update data, and retrieve data using queries. SQL queries vary from simple commands that retrieve a small amount of data to complex queries that process larger amounts of data and perform advanced calculations. The performance of SQL queries can be critical for the overall performance of a database, and optimizing the performance of queries can significantly improve the efficiency of a database.

2.1.3.1 SQL Query Optimization in Oracle

SQL queries are an essential component of any database management system, and their performance can have a significant impact on the overall performance of the database. SQL query performance optimization involves identifying and addressing bottlenecks in the queries to improve their performance. There are several techniques for optimizing SQL queries in a RDBMS system and an Oracle database.

One technique for optimizing SQL queries in Oracle is the use of bind variables. Bind variables allow a SQL statement to be executed multiple times with different values, without the need to parse and optimize the statement each time it is executed. This can significantly improve the performance of queries that are executed frequently with different values, such as those used in web applications.

Another technique for optimizing SQL queries in Oracle is the use of indexes. Indexes allow the database to quickly locate and retrieve data from specific columns or rows, improving the performance of queries that filter or sort data. There are several types of indexes available in Oracle, including B-tree indexes, bitmap indexes, and function-based indexes. Choosing the right indexing strategy can significantly improve the performance of the queries.

Materialized views can also be used to improve the performance of complex queries. A materialized view is a separate table that stores the results of a query, allowing the database to quickly retrieve the results without having to re-execute the query each time. They can be useful for queries that are executed frequently or take a long time to run, but they come with a cost as they require additional storage and maintenance overhead.

The hardware and the infrastructure that the database runs on can also have an impact on the query performance. Considering factors like CPU, memory, storage, and network bandwidth, can help ensure that the database has the resources it needs to execute queries efficiently.

Finally, the use of the EXPLAIN PLAN command can provide a better understanding on how the database executes the query and find ways to optimize it.

Optimizing SQL queries in Oracle is an important task that can significantly improve the performance of the database. By using techniques like bind variables, indexing, materialized views, and choosing the right hardware and infrastructure, we can help ensure that the database is running efficiently. It is important to regularly monitor and optimize the performance of the queries.

By using these techniques, we can help ensure that the SQL queries are optimized for maximum performance. It is important to regularly monitor and optimize the performance of the queries to ensure that the database is running efficiently.

2.2 Non-Relational Databases

The non-relational database, NoSQL, is another mechanism for storing and retrieving data but, unlike the relational database, they are not using tables, rows, primary keys or foreign keys. Instead, they utilize a storage model tailored to the unique needs of the data type they are storing.

Rather than using Structured Query Language (SQL) as in relational databases, NoSQL databases utilize Object-Relational Mapping (ORM). ORM allows for crafting queries in one's chosen programming language. Notable ORMs include Java, Javascript, .NET, and PHP. Some of the more popular NoSQL databases are MongoDB (Document based), Apache Cassandra (Columnar), Redis (Key Value Store), Neo4j (Graph Database) and Apache HBase (Tabular) with each one using a different type.

2.2.1 NoSQL Databases Types

NoSQL databases can be classified into different types based on their data models.

Here are four commonly recognized types of NoSQL databases:

- Document data stores:

  Data are stored in units known as “document, typically represented in JSON format and can be encoded in multiple manners. These documents don't adhere to a fixed schema or structure. An application can query and filter the fields within a document using field values. An advantage of the document stores is that is not required the same data structure for all the documents which provide great flexibility.

- Columnar data stores:

  In columnar databases the data is stored into columns which is similar to the relational approach but the advantage is in the denormalization logic to structing sparse data.

- Key Value Store:

  The key value store databases store the data in a simply collection of key-value pairs in which the key works as a unique identifier. This kind of databases are can be portioned and can achieve high horizontal scaling, greater than other types of databases can achieve.

- Graph database:

  Graph databases are designed to store relations efficiently between entities when data are interconnected and it is one of the most complex type of databases.

Overall, these NoSQL database types offer diverse approaches to data storage and retrieval, catering to various use cases and data requirements

3. Big Data Frameworks

3.1 Big Data at Google

In the rise of the 21st century, as the volume of data continued to grow exponentially, major companies faced the challenge of managing and storing massive amounts of information. Google was at the forefront of tackling this issue.

Traditional storage systems like relational databases (RDBMS) and imperative programming approaches were inadequate for the scale of data that Google needed to handle, especially for indexing and searching the vast number of documents on the web. To address this need for a scalable infrastructure capable of storing and processing large datasets, Google developed a set of cluster-based technologies[S2].

Among these technologies are the Google File System (GFS) [S3], which provided a distributed file storage system designed for reliability and scalability. MapReduce [S4], another key technology, enabled parallel processing of data across clusters, allowing for efficient data processing. Additionally, Google developed Bigtable [S5], a distributed, highly scalable NoSQL database, which served as the underlying storage system for various Google services.

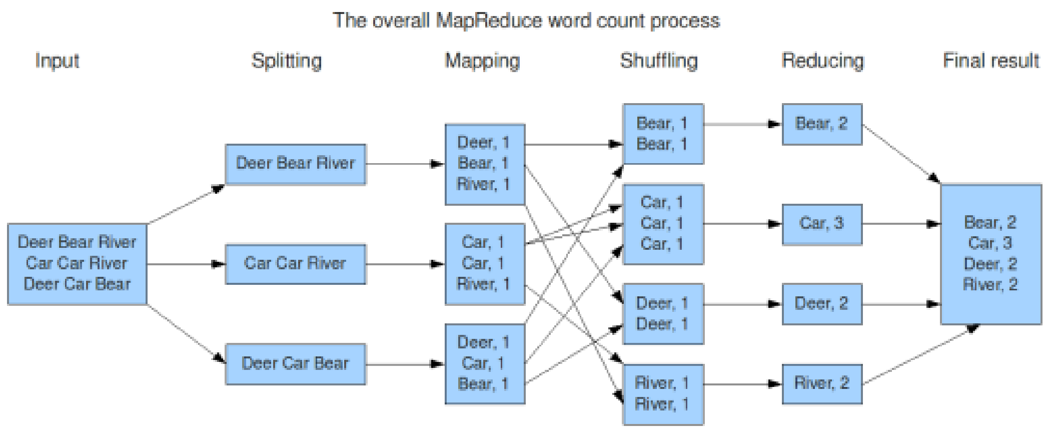


Figure 1 MapReduce example diagram
http://blog.jteam.nl/wp- content/uploads/2009/08/MapReduceWordCountOverview1.png

The GFS was able to provide a fault-tolerant and distributed filesystem across a cluster of computers to store unstructured files and Bigtable was able to offer scalable storage of structure data across the GFS.

The MapReduce proposed a parallel programming model for processing large scale data sets distributed over GFS and Bigtable.

The work that Google did was proprietary but they were also many more companies dealing with same data challenges. The three aforementioned papers about GFS, MR and Bigtable provided solutions and inspired other researchers who they were trying to tackle the same issues.

3.2 Apache Hadoop

Google's research team's computational challenges and solutions served as the inspiration for the development of the Hadoop Distributed File System (HDFS) [S6] and the Apache Hadoop framework. HDFS, in particular, adopted the concept of splitting files into large blocks and distributing them across a cluster of nodes.

The Apache Hadoop framework is an open-source software suite designed to leverage the power of distributed computing to solve problems involving massive amounts of data. Hadoop achieves this by dividing the data into manageable chunks and distributing them across multiple nodes in a cluster. This distributed storage approach allows Hadoop to take advantage of data locality, where each node processes only the data it holds, resulting in faster and more efficient data processing. [S7]

Furthermore, Hadoop employs the MapReduce programming model as a framework for distributed computing. The data processing tasks are packaged as code and distributed to the nodes, allowing for parallel execution across the cluster. This parallelism and data locality enable Hadoop to handle large-scale data processing tasks in a scalable and efficient manner.

Map Function: $M(key, value) \rightarrow list(key', value')$

Reduce Function: $R(key', list(value')) \rightarrow list(key', value'')$

MapReduce Paradigm

By adopting the principles and technologies developed by Google, Hadoop has become a widely-used solution for big data processing, providing organizations with the ability to leverage distributed computing and handle massive amounts of data effectively.

The three main components of Apache Hadoop are:

1. **HDFS**:
   Hadoop Distributed File System is a specialized file system designed for storing vast amounts of data across a cluster of computres. It enables data to be stored across numerous nodes in the cluster which ensures data security and fault tolerance.

2. **MapReduce**:
   Hadoop Distributed File System is a dedicated file system to store big data with a cluster of commodity hardware or cheaper hardware with streaming access pattern. It enables data to be stored at multiple nodes in the cluster which ensures data security and fault tolerance.

3. **YARN**:
   stands for Yet Another Resource Negotiator. It is a dedicated operating system for Hadoop which is responsible for managing computing resources in clusters and using them for scheduling users' applications in Hadoop. The various types of scheduling are First Come First Serve, Fair Share Scheduler and Capacity Scheduler etc. [S8].

3.2.1 Hadoop Ecosystem

The Apache Hadoop ecosystem encompasses a wide range of components within the Apache Hadoop software library. In addition to the core components like HDFS, MapReduce, and YARN, the ecosystem includes various open-source projects and complementary tools. The diagram below provides an overview of some commonly used tools within the Hadoop ecosystem.

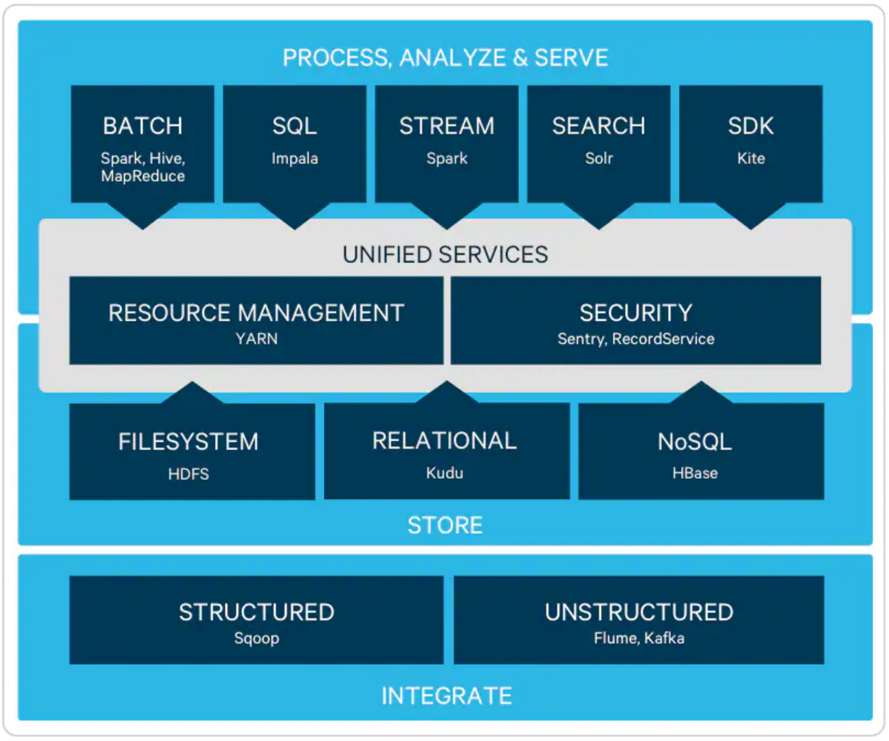


Figure 2 Hadoop Ecosystem

While Hadoop has been a popular choice for developers working with big data, it does have certain limitations. One of the drawbacks is the need to write intermediate results to the local disk for each pair of MapReduce tasks. This frequent disk I/O operation can significantly impact performance and lead to longer execution times for large MapReduce jobs.

Furthermore, as additional tools were introduced to support different workloads such as machine learning, streaming, and interactive SQL-like queries, the operational complexity of the Hadoop framework increased. This complexity posed challenges for users in terms of learning and managing the various components effectively.

4. Apache Spark

As the data landscape continued to evolve, new challenges arose which exposed limitations of Hadoop's MapReduce paradigm. The major challenge was the need for easier, faster and more efficient processing of data than Hadoop and MapReduce. Apache Spark emerged as a response to those challenges. Apache Spark is an open-source data processing engine that has gained significant popularity in the field of big data processing. It is a multi-language engine for executing data engineering, data science, and machine learning on single-node machines or clusters. [S9]

It was developed at UC Berkeley in 2009 and has been open sourced since the year 2010 under BSD license. It finally became a project of Apache Software Foundation in the year 2013 and now is the biggest project of the Apache foundation.

One of the standout features of Apache Spark that separates it from the previous big data analytics software is its in-memory cluster computing capability, which increases the processing speed of an application. Compared to Hadoop, Spark's performance can be up to 100 times faster in memory and about 10 times faster on disk [S10]

Spark provides a programming interface that allows developers to work with entire clusters, leveraging implicit data parallelism and ensuring fault tolerance. Its design aims to accommodate a wide range of workloads, making it a flexible choice for diverse data processing needs.

Apache Spark has become a vital tool in the big data analytics landscape, offering enhanced performance and versatility for processing large-scale datasets. Its robustness, scalability, and extensive ecosystem of libraries and tools have contributed to its widespread adoption in the industry.

4.1 Spark Core and Components

Spark core or the computing engine of Spark contains all the basic functionality including the four distinct components as libraries for diverse workloads. These components are Spark SQL, Spark MLlib, Spark Structured Streaming, and GraphX. Spark core also host the API which defines the Resilient Distributed Datasets (RDD) [S11] which is an immutable distributed collection of objects, providing fault tolerance and distributed computing capabilities in Apache Spark.

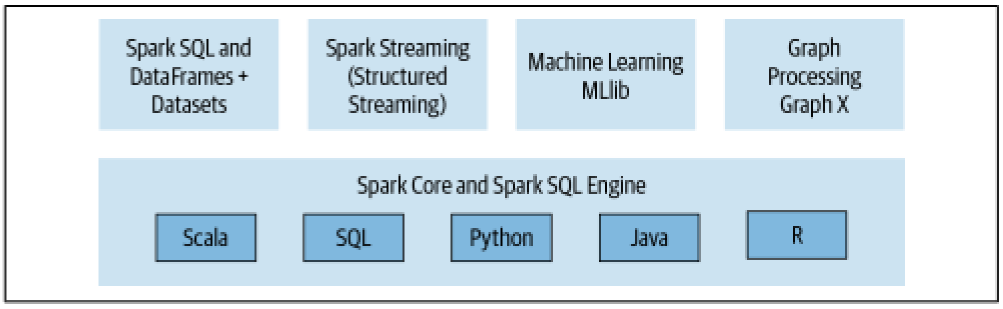

Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scala
SQL
Python
Java
R

Figure 3 Apache Spark components

When we write Spark application, Spark converts it to a DAG(Directed Acyclic Graph) and executes it in the core engine. That means that the Spark code, regardless of the API used (Java, Scala, Python, SQL), decompose to highly compacted bytecode and it is executed in the workers' JVM across the cluster.

4.1.1 SPARK SQL

Spark SQL is an Apache Spark's module that integrate relational processing with Spark's functional programming API[S12]. It gives the opportunity to manipulate Dataframes, a data abstraction introduces by Spark SQL, in Scala, Java or Python. It supports many data sources as it can read data stored in RDBMS tables or from file formats like CSV, txt, JSON, Avro, Parquet etc.

Also, it gives the opportunity to query data using SQL-like queries combined with Spark's Structure APIs in Java, Scala, Python or R. Spark SQL is ANSI SQL:2003[S12] compliant and functions as a SQL-engine.

4.1.2 Spark MLib

Spark comes with an open-source distributed library called Mlib which contains all the common Machine Learning (ML) algorithms like classification, regression and clustering. Mllib provides efficient functionality for a wide range of learning settings and includes several underlying statistical, optimization, and linear algebra primitives[S14] . All of these methods are designed to scale out across a cluster.

4.1.3 Spark Streaming

Stream processing is defined as the continuous processing of endless streams of data. Spark Streaming extension of the core Spark API and it responsible for processing live streams of data. Apache Spark Streaming is the previous generation of Apache Spark's streaming engine. After Apache Spark 2.0 a new higher-level API, Structured Streaming has been implemented[S15]. Apache Spark Structured Streaming was built on top of the Spark SQL Engine and DataFrame-based API. Under the hood of the Spark's Structured Streaming model, all the fault tolerance and late-data semantics aspects are handling by the Spark SQL engine. In later versions, the range of streaming data sources has been extended to include Apache Kafka, Kinesis and HDFS-based or cloud storage.

4.1.4 GraphX

GraphX is an embedded graph processing framework built on top of Apache Spark[S16]. It is a library for manipulating graphs performing graph-parallel computations. Like other Apache Spark's components, GraphX extends the Spark RDD API. GraphX can bring low-cost fault tolerance by using the distributed dataflow frameworks. Finally, it provides various operators for manipulating graphs and offers the standard graph algorithms for analysis, connections and traversals.

4.2 Spark Architecture

Spark is a cluster computing framework, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce. It is a distributed data processing engine with all its components working collaboratively on a cluster of machines.

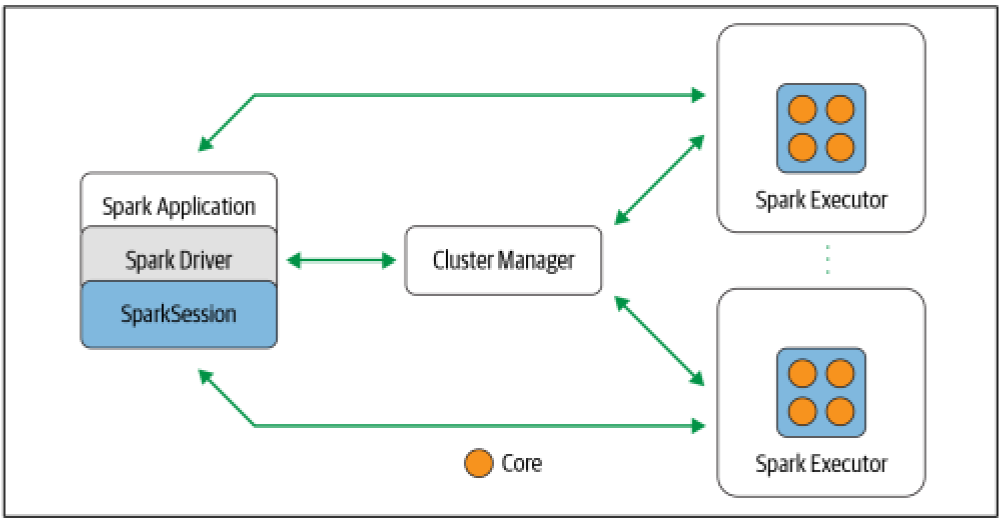


Figure 4 Apache Spark components & architecture

At a high level, any spark application consists of a driver program that is the manager of the Spark program and responsible for orchestrating parallel operations on the cluster. It interfaces with the Cluster Manager to get the resources and lunch the executors. Spark can run across different cluster managers. A Master/Worker node architecture is implemented with one primary Master node (sometimes and a second one as secondary Master node) and many workers (executors).

4.2.1 Spark driver

The driver is the main process of a Spark Application and it is responsible for instantiating a SparkSession. It is the main controller of the execution of a Spark Application and maintains all of the states of the Spark cluster. It must interface with the cluster manager in order to request physical resources for Spark's executors. Then it transforms all the Spark operations into DAG computations, it schedules them and finally distribute their execution as tasks across the executors. When the resources are allocated, it communicates with the executors directly. The driver can run either in the cluster or on the client machine that is running the Spark application.

At the end of the day, this is just a process on a physical machine that is responsible for maintaining the state of the application running on the cluster.

4.2.2 SparkSession

From Spark 2.0 and onwards, the SparkSession became a unifies entry point of all underlying Spark functionality. It subsumes previous entry points to Spark[S17] and made working with Spark simple and easier. So, instead of having a SparkContext, SQLContext, HiveContext, SparkConf, and StreamingContext, now all of it is encapsulated in a Spark session. In order to maintain backward compatibility, the individual contexts and their respective methods remained available.

In a standalone Spark application, the user can create a SparkSession using one of the high-level APIs in any of the available programming languages while, in the Spark shell the SparkSession is created automatically and can be accessed via a global variable called spark or sc.

4.2.3 Spark Executors

Spark executors run on the worker nodes in the cluster and they perform tasks from the Spark driver. Their main job is to take the tasks, that the Spark driver has assign to them, execute them and then report back to the Spark driver their state and the results. The executors communicate with the driver program and it is they’re responsibility for executing the tasks on the worker nodes. In most cases there is only one executor that operates per node. Usually, the executors run for the entirely lifetime of a Spark application. If though an executor fails, Spark can continue to run the program by recalculating only the lost data.

4.2.4 Cluster Manager

The cluster manager is responsible for managing and allocating resources for the cluster on which the Spark application runs. Spark support four cluster managers: the built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes and either of them can be launched on-premise or in the cloud for a spark application to run.

Mode	Spark Driver	Spark Executor	Cluster Manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's Node Manager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on Node Managers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

Table 1 Apache Spark deployment modes

4.2.4.1 Cluster Manager Types

- **Standalone**

  It is a simple cluster manager already included with Spark and make it easy to set up a cluster and execute applications on it. It contains one master and several workers, each having a configures memory size and CPU cores.

- **Hadoop YARN**

  It was introduced in Hadoop 2.0. It supports utilizing varied data processing frameworks on a distributed resource pool. It is placed on the same nodes as Hadoop's Distributed File System (HDFS) which give an extra advantage as it allows Spark to access HDFS data swiftly, on the same nodes where the data is kept.
  YARN can be used easily by setting an environment variable that points to the user's Hadoop configuration directory and then submitting jobs to a special master URL using spark-submit.

- **Apache Mesos**

It is a general common-purpose cluster manager that can run also Hadoop MapReduce and service applications. The restriction for using Apache Mesos though is that applications can run only on cluster mode, something which we will talk more in depth later.

- **Kubernetes**

  It is an open-source system for automating deployment, scaling and management of containerized applications. It is a relative new addition as it was only introduced with the launch of Spark 2.3 but it has been declared as generally available with the release of Spark 3.1.

4.2.5 Execution Mode

An execution mode determines whether the aforementioned resources are physically located when a Spark application runs.

- **Cluster mode**

  It is the most common way to run a Spark Application. In cluster mode, the user submits a JAR file, Python or R script to the cluster manager. The cluster manager launches the driver program on one of the worker nodes inside the cluster, in addition to the executor process. Cluster mode is most often used for running production level jobs.

- **Client mode**

  In this mode the driver will be launched on the machine where the spark-submit command was executed. This means that the client machine is responsible for maintaining the Spark driver process and the cluster manager maintains the executor processes. These machines are usually referred as gateway machines or edge nodes.

4.3 Spark Application

The core of every Spark application is the Spark driver program, which creates the SparkSession object. When the SparkSession is initialized, spark operations can be performed using any provided API.

4.3.1 Spark Jobs

During the execution of a Spark application, the driver converts the application to one or more Spark jobs. Then each job is transformed into a DAG. This is the Spark's execution plan, where each node withing a DAG could be one or more Spark stages.

4.3.2 Spark Stages

Stages are part of the DAG nodes and are created based on the operations that can be performed either serially or in parallel. Some Spark operation may not take place in a single stage and they could be divided into multiple stages.

4.3.3 Spark Tasks

Each stage includes Spark tasks (unit of execution) which are federated across each Spark executor. Each task maps to a single core and works on a single partition of data.

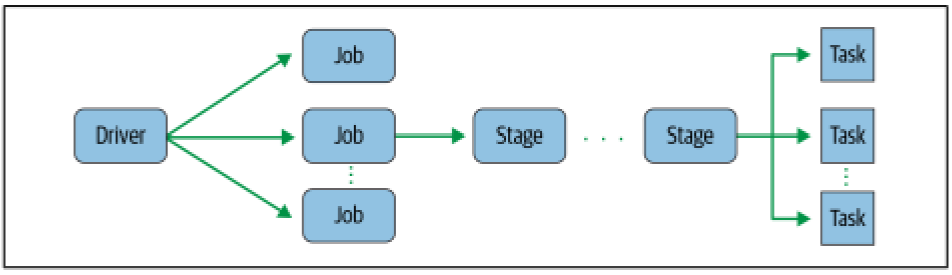


Figure 5 Apache SparkJ Jobs, Stages and Tasks

4.4 Transformations, Actions and Lazy Evaluation

The two types of Spark operations are Transformations and Actions.

With transformations, Spark transform a DataFrame into a new DataFrame without altering the original one, making each DataFrame immutable.

All types of transformations are evaluated lazily. That means that their results are not calculated immediately, but they are recorded or remembered. A recorded lineage allows Spark to create a more optimal execution plan by rearranging certain transformations, coalesce them, or optimizing them into stages and provide a more efficient execution.

Lazy evaluation is Spark's strategy for delaying execution until an action take place. An action triggers the lazy evaluation of all the recorded transformations until that point. The lazy evaluation allows Spark to optimize the queries by peeking into the chained transformations, lineage and data immutability provide fault tolerance.

In the below table there is an example of some basics Transformations and Actions.

Transformations	Actions
orderBy()	show()
groupBy()	take()
filter()	count()
select()	collect()
join()	save()

Table 2 Apache Spark Transformation & Actions

4.4.1 Narrow and Wide Transformations

Transformations can be further classified as Narrow and Wide regarding to their dependencies. Transformations where a single output partition can be calculated from a single input partition is called Narrow transformation. An example of a narrow transformation is filter() because it operates on a single partition and produce the result without any data exchange. An example of a wide Transformation is groupBy(), where a shuffle of data from each of the executor's partitions across the cluster will be needed.

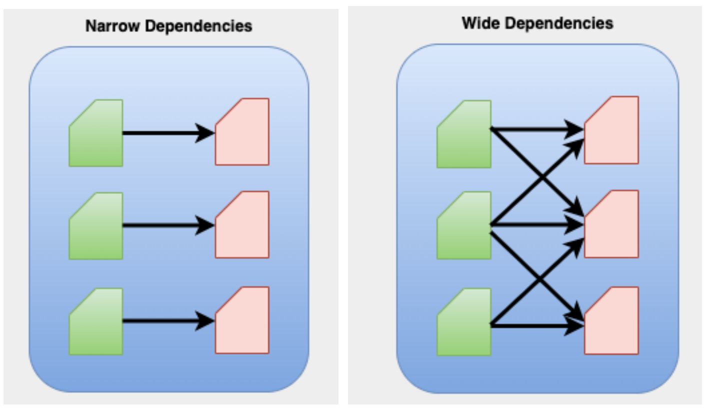


Figure 6 Narrow and Wide Transformations

4.5 Spark Optimization

Like any distributed system, Spark can benefit from optimization and performance tuning to ensure that it is running more efficiently and effectively. Below are described some of the key areas that can be considered when optimizing and tuning the performance of Spark

Cluster configuration

Configuration of the Spark cluster can have a significant impact on the performance. Should be ensured that the cluster has sufficient resources, including CPU, memory, and storage, to meet the needs of the workloads. Also, there is the need of considering the number and size of the worker nodes and the network infrastructure connecting them.

Data representation and partitioning

The way the data are represented and partitioned can have an impact on Spark performance. An optimized data representation should be chosen according to the type of processing that takes place and the data should be portioned in a way that allows Spark to effectively parallelize the workloads. Repartitioning is one the techniques in Spark to optimize the execution of queries by reshuffling the data across partitions and It can have a significant impact on the performance of operations.

At its essence, repartitioning in Spark involves redistributing the data across a specified number of partitions. By default, Spark might not always partition data optimally for the specific operations you're going to perform. Repartitioning allows for manual intervention, ensuring that the data is distributed in a manner that's more conducive to efficient querying.

By reportioning, a more balancing load can be achieved as uneven distribution of data across nodes (data skew) can lead to performance issues. Additionally, join operations can become for efficient and data shuffling can be reduced.

Caching and persistence

Spark provide the possibility to cache data in memory or persist it to disk, which can improve the performance of iterative or interactive workloads. Caching is a mechanism to speed up operations that access the same dataset multiple times.

The `cache()` method on a DataFrame, use the default storage level, which is StorageLevel.MEMORY_ONLY. This means that the DataFrame is stored in memory as deserialized JVM objects. If the DataFrame does not fit into memory, some partitions will not be cached and they will have to be recomputed every time that they are needed.

With the `persist()` method, StorageLevel that determines how the dataset is stored can be specified. With StorageLevel.MEMORY_AND_DISK, the DataFrame is stored in memory as deserialized JVM objects, but if it doesn't fit in memory, the excess partitions spill to disk. This ensures that even if the DataFrame is larger than the available memory, all its partitions remain available for quick access without needing any re computation.

The trade-offs between memory and disk usage should be considered when deciding which data to cache or persist.

Executor and task configuration

The executors and tasks that Spark uses to execute the workloads can also be optimized for performance. Factors like the number of cores and memory per executor and the number of tasks per executor should be taken under consideration when configuring a Spark application.

The optimization and tuning of these factors, can help ensure that Spark applications are running efficiently and effectively. The specific optimization and tuning strategies that will be most effective will always depend on the characteristics of the workloads and the resources available to the cluster.

5 Problem statement

In our use-case we are having a database with tables filled with records collected from sensors mounted on moving vehicles. In the current setting the data that are been collected are stored in an Oracle database. This configuration, while robust and reliable, presents significant challenges as these records frequently undergo complex queries. The queries could be of various types like aggregated queries, queries with filter conditions or queries that require full table scan which often having multiple joins and nested queries. Another challenge is the increasing data volumes as this tables could be populated each day with additional data.

The primary objective of this thesis, hence, is to explore the benefits and drawbacks of transitioning from the current Oracle database setup - a baseline solution and the prevailing production setup in this use-case - to a more distributed, highly scalable environment such as Apache Spark. The main focus is on optimizing the execution time of the queries using Spark's inherent capabilities of handling large datasets in a distributed manner.

The goal is to assess the capabilities and efficiency of these two technologies in handling and processing large datasets for various types of SQL Queries. By conducting an experimental comparison, we can gain insights into the strengths and weaknesses of each approach and determine which technology is better suited for the specific data processing scenarios we are facing. This study will not only compare their general strengths and weaknesses but will be tailored to the specific challenges we are facing on our use-case for the vehicular sensor data. Additionally, we will explore optimization techniques to improve the performance of both Oracle and Spark in executing the beforementioned type of SQL queries.

The outcome of this research would be to determine whether executing the set of queries in a big data framework like Spark could offer advantages on execution time over the current baseline solution of Oracle. This study will provide empirical evidence as to whether the shift to a distributed environment like Spark could present a significant improvement, for those types of queries, over a traditional database system like an Oracle Database.

Even though our results could have a broader implication, in our case we will focus on the specific use-case and for the specific set of queries.

6 Database schema & Environment

6.1 Database Schema and Design

The Dataset used in our use-case for our experiments consists entries that have been collected from sensors on moving objects and their related metrics. The initial data and the DDLs of the tables were provided by Company X and it has a real-world application scenario. The database design adheres to the third normal form (3NF) to avoid data redundancy and maintain data integrity. The schema of the database is depicted below:

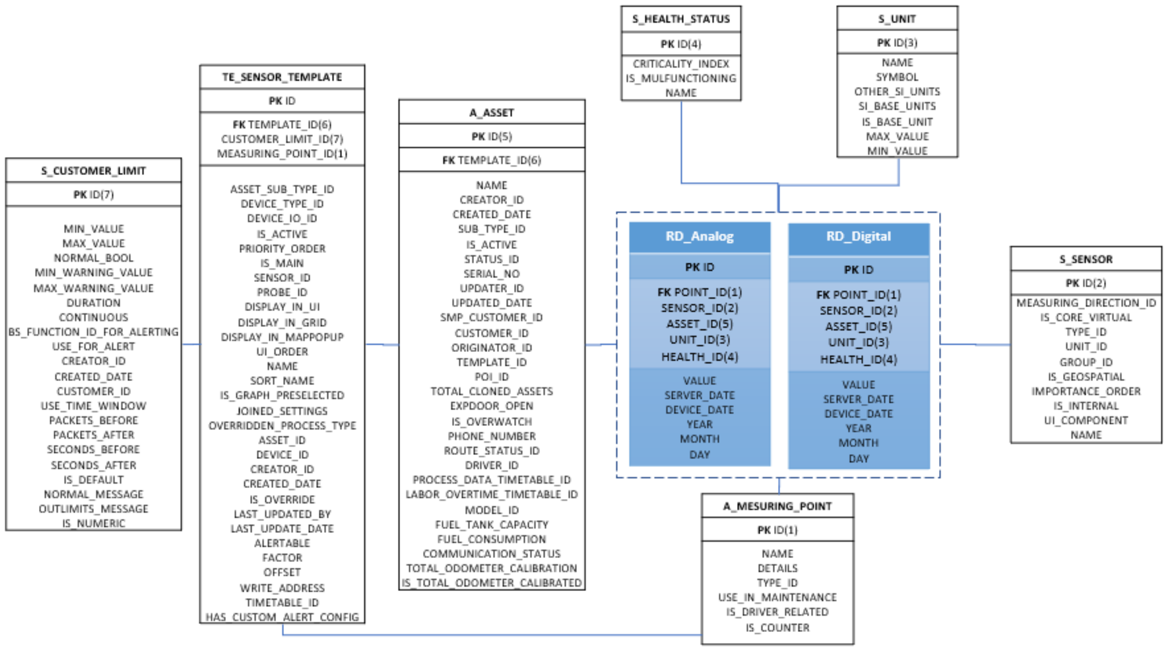


Figure 7 Database schema representation

6.1.1 Tables description

As we can see, we have two main tables, rd_sensor_analog_history and rd_sensor_digital_history. These tables store the historical data from analog and digital sensors respectively. The key attributes include asset_id, point_id, device_date, server_date, sensor_id, health_id and value with the ID as Primary Key. The key attributes are also Foreign Keys (FK) for the rest of the configuration tables with

- ASSET_ID to be FK of A_ASSET table

- POINT_ID to be FK of A_MEASURING_POINT table
- SENSOR_ID to be FK of S_SENSOR table
- HEALTH_ID to be FK of S_HEALTH_STATUS table
- UNIT_ID to be FK of S_UNIT table

The union of those two tables is our main entity and the core of our database. These tables have the information sent by the sensors while the rest of the tables are used as configuration tables.

Then we have TE_SENSOR_TEMPLATE which stores metadata related to different sensor templates, their types, their measurements and other configuration settings. Constraints are added to enforce non-null values for most of the fields. The ID is the primary key while SENSOR_ID and CUSTOMER_LIMIT_ID are Foreign Keys (FK) referencing to the S_SENSOR table and S_CUSTOMER_LIMIT respectively. It is also connected with A_ASSET table with the TEMPLATE_ID column.

The table S_CUSTOMER_LIMIT is used for setting and monitoring various limits for customers with ID as Primary Key.

The A_ASSET table is at the heart of our database, capturing comprehensive information about all assets, which in our case our vehicles. The ID is the Primary Key which is a unique identifier for all assets.

The S_SENSOR table represents the various sensors available in the system where the ID is the primary key having a unique identifier for each sensor.

The S_UNIT table manages the units of measurement.

The S_HEALTH_STATUS table stores information related to the health status of the sensors. The column CRITICALITY_INDEX is an indicator of how much severe or not the health of the sensor is. When 0 the sensor is normal, when 1 it means that it has accuracy issues.

The A_MEASURING_POINT table captures information related to specific measuring points where sensors are fitted.

The final dataset was based on the initial data provided by Company X which were augmented with additional data in order to increase the size of our test dataset. These were the datasets that we executed our experiments on.

Our purpose is to test whether it would be an advantage of executing those queries on a big data framework, instead of a relational database.

6.2 Data Flow

In our business case every day we get 10 million new records which are appended to rd_sensor_analog_history and rd_sensor_digital_history. The way that the two main tables are populated daily is the following:

We have one thousand (1000) unique assets where:

- each asset sends every day one thousand (1.000) batches and
- each batch consists of 10 rows.

The total number of rows per day can be calculate with the following equation:

$$Total\ Rows = Number\ of\ Assets\ x\ Number\ of\ Batches\ x\ Number\ of\ Rows\ per\ Batch$$

So, for one day we have 10 million rows, while for the time span of a full week we will have 70 million records.

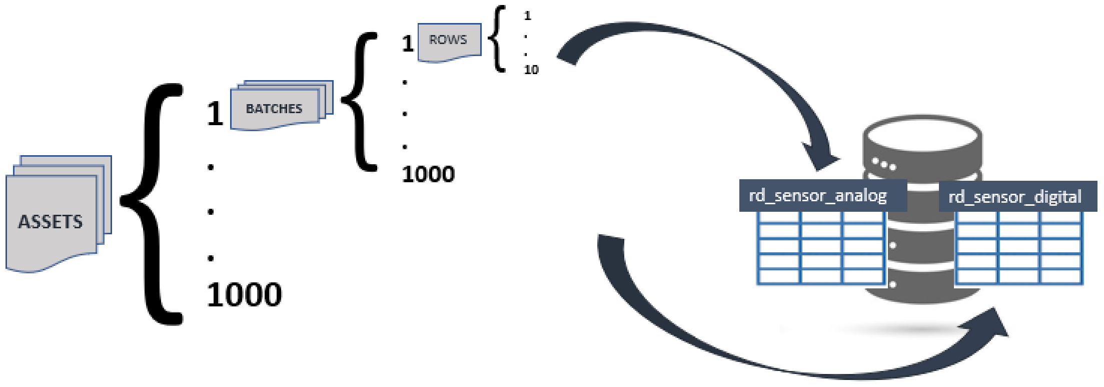


Figure 8 Daily Data Flow

6.3 Working Environment

The working environment consists of two main components. The relation database and the Apache Spark.

The Oracle database version that was used was the Oracle Database 23c and it was set up on a local working environment with 2 CPU cores and 8GB of RAM running on Windows 10 machine.

For Spark we employed two distinct computational environments. Initially, we utilized the same local working environment that was used for Oracle Database as we are aiming to facilitate a direct comparison between the two systems. Thus, utilizing a mutual environment ensured that external variables were minimized and that any observed differences in performance were related to the systems themselves and not influenced by differing operational conditions. In this local setting, the Spark version that was deployed was 3.0.2, with Scala version 2.12. Additionally, we integrated Spark with the Jupyter Notebook environment using the Spylon-kernel.

Jupyter Notebook is an open-source interactive computing environment that enables researchers and data scientists to create, share, and document live code, visualizations, and explanatory text, fostering reproducible research and collaborative analysis[S17]. Spylon-kernel [S18] is an integration tool designed to bridge the gap between the Jupyter Notebook environment and Apache Spark, enabling users to execute Spark computations directly within notebooks using the Scala programming language.

This integration made sure we could harness Spark's powerful processing right within the interface of Jupyter. This meant that Spark tasks could be executed in the Notebook while still using Scala as programming language, which is the language in which Spark is written.

While this setup is suitable for developmental purposes and testing, it does not exploit Spark's primary advantage: distributed data processing across multiple nodes in a cluster. Since Spark is a distributed computing framework designed to process vast amounts of data across many nodes, by its nature was design for clusters. It has been built on the principal of distributing data and computations across multiple nodes where data can be partitioned and processed in parallel across nodes. Thus, a distributed environment was needed for further exploration.

For the distributed environment of the Spark cluster, after comparing and evaluating various options like a Hadoop cluster, a spark stand-alone cluster and cloud-based environments, the Azure Databricks decided to be the optimal approach.

Azure Databricks[S20] is a cloud-based collaborative environment for big data analytics. It provides a managed Apache Spark platform that seamlessly integrates with other Azure services, offering scalability, reliability, and simplified cluster management.

The Spark cluster in Azure Databricks consisted of one Master node and three workers. Each node is equipped with 14GB of main memory and 4 cores. Additionally, the Azure Databricks Spark Cluster was utilizing Databricks runtime version 12.2 LTS using Spark 3.0.2 and Scala 2.12.

Since Azure Databricks was decided to be used for the implementation of the Spark cluster, another component of the azure ecosystem, the Azure Blob Storage [S21], was decided as the main Data storage option for the Big Data tasks.

7 Experimental Study and Evaluation

As part of our research, we conducted performance tests on two datasets: one for a single day and another for a full week. These datasets were used to evaluate the performance of four queries in both Oracle and Spark. The configuration tables remain the same in both datasets while the two main tables, rd_sensor_analog_history and rd_sensor_digital_history are appended with data. The cumulative size of the two main tables for the daily dataset is 1 GB and a total of 10 million rows. The corresponding size of the dataset for the whole week is 7 GB and 70 million rows.

We conduct our experiments on the aforementioned datasets for four different queries. Two queries, Q1a and Q2, were provided by the Company X and are commonly used to derive valuable business insights. The third one, Q3, was an aggregated query that was specifically designed by us, following the same business logic, to assess the performance of the two systems when handling aggregated results. Additionally, one more query was created (Q1b), based on the first one provided (Q1a), to examine the behavior of the system when we are using a where clause.

7.1 Queries

7.1.1 Query 1 (Q1)

a) Without filter

This query extracts a comprehensive view for all the sensors by reading both digital and analog sensor data, along with their associated attributes.

The core of the query is a UNION ALL operation on the digital and analog sensor history tables, rd_sensor_digital_history and rd_sensor_analog_history. By joining with the configuration tables, we are able to retrieve enriched information for each sensor.

An extra column is created with a nested CASE statement and it is called message. When the unit id is 1, it compares the sensor value (rd.val) with a Boolean flag (li.normal_bool) in the customer limit table. If they are equal, the normal_message from the customer limit table is returned; otherwise, the outlimits_message is returned.

This query, thus, presents an enriched view of the sensor history, incorporating readings, sensor details, associated units, and health status, along with customer-specific information and any potential messages based on the data readings. For this query we do not apply any filter condition and a full scan of the table will be implemented.

```
SELECT mp.id mespiid
      ,mp.name mspname
      ,te.name templname
      ,se.id
      ,se.name
      ,u.name unitname
      ,u.symbol
      ,rd.val
      ,rd.device_date
      ,hs.name healthanme
      ,li.min_value
      ,li.max_value
      ,CASE
            WHEN u.id = 1
                  THEN CASE
                              WHEN li.normal_bool = rd.val
                                    THEN li.normal_message
                              ELSE li.outlimits_message
                        END
            END message
FROM (
      SELECT asset_id
            ,point_id
            ,device_date
            ,sensor_id
            ,health_id
            ,VALUE val
      FROM rd_sensor_digital_history

      UNION ALL

      SELECT asset_id
            ,point_id
            ,device_date
            ,sensor_id
            ,health_id
            ,VALUE val
      FROM rd_sensor_analog_history
      ) rd
LEFT JOIN a_measuring_point mp ON rd.point_id = mp.id
LEFT JOIN s_sensor se ON rd.sensor_id = se.id
LEFT JOIN a_asset a ON rd.asset_id = a.id
LEFT JOIN s_unit u ON se.unit_id = u.id
LEFT JOIN s_health_status hs ON rd.health_id = hs.id
LEFT JOIN te_sensor_template te ON mp.id = te.measuring_point_id
      AND a.template_id = te.template_id
LEFT JOIN s_customer_limit li ON te.customer_limits_id = li.id
```

b) With filter

This query uses the same tables and the joins like the previous one but in this case a where clause is added to filter the search criteria only for one ASSET (WHERE rd.asset_id = 1000)

7.1.2 Query 2 (Q2)

The second query refers to the alerts section and is used to display related alerts to the customers, regarding their assets. The SQL query is designed to retrieve detailed information regarding alerts triggered by various sensor readings. It is structured to provide comprehensive data, including sensor identification, time of reading, the sensor value, as well as detailed sensor, unit, and health status information.

This information is gathered from several tables including 'rd_sensor_digital_history' and 'rd_sensor_analog_history', which store sensor reading histories and then are joined with several other tables to enrich it with additional details. The output of this query can be used to generate alerts or to populate UI components that provide sensor data details and statuses.

```sql
SELECT se.id sensorid
      ,rd.device_date latestutcdate
      ,rd.val sensorvalue
      ,se.name sensorname
      ,u.name unitname
      ,u.symbol unitsymbol
      ,hs.name healthstatus
      ,te.display_in_ui displayinui
      ,te.display_in_grid displayingrid
      ,te.display_in_mappopup displayinmappopup
      ,te.ui_order uiorder
      ,a.id assetid
      ,te.sort_name sortname
      ,CASE
            WHEN u.id = 1
                  THEN CASE
                              WHEN li.normal_bool = rd.val
                                    THEN li.normal_message
                              ELSE li.outlimits_message
                              END
            END message
      ,te.measuring_point_id measuringpointid
      ,amp.name measuringpointname
      ,te.name customermeasuringpointname
      ,te.is_graph_preselected isgraphpreselected
```

```
FROM (
      SELECT asset_id
            ,point_id
            ,sensor_id
            ,device_date
            ,server_date
            ,TO_CHAR(value) AS val
            ,health_id
      FROM rd_sensor_digital_history

      UNION ALL

      SELECT asset_id
            ,point_id
            ,sensor_id
            ,device_date
            ,server_date
            ,TO_CHAR(value) AS val
            ,health_id
      FROM rd_sensor_analog_history
      ) rd


LEFT JOIN a_asset a ON rd.asset_id = a.id
LEFT JOIN a_measuring_point amp ON rd.point_id = amp.id
LEFT JOIN s_health_status hs ON rd.health_id = hs.id
      AND hs.id != 1
LEFT JOIN s_sensor se ON rd.sensor_id = se.id
LEFT JOIN s_unit u ON se.unit_id = u.id
LEFT JOIN a_measuring_point mp ON rd.point_id = mp.id
LEFT JOIN te_sensor_template te ON mp.id = te.measuring_point_id
      AND a.template_id = te.template_id
LEFT JOIN s_customer_limit li ON te.customer_limits_id = li.id
```

7.1.3 Query 3 (Q3)

The third query is an aggregated one and counts the health condition of the assets. This query is used to give an overview of the health status of all sensor readings across both digital and analog sensors, by providing a count of sensor readings per health status. This data can be instrumental in providing insights into the overall health and performance of the sensors in the vehicles.

```
SELECT count(*)
      ,rd.health_id
      ,hs.name
FROM (
      SELECT asset_id
            ,point_id
            ,sensor_id
            ,device_date
            ,server_date
            ,TO_CHAR(value) AS val
            ,health_id
      FROM rd_sensor_digital_history

      UNION ALL

      SELECT asset_id
            ,point_id
            ,sensor_id
            ,device_date
            ,server_date
            ,TO_CHAR(value) AS val
            ,health_id
      FROM rd_sensor_analog_history
      ) rd
JOIN s_health_status hs ON rd.health_id = hs.id
      AND hs.id != 1
GROUP BY rd.health_id,hs.name
```

7.2 Evaluation method & Query benchmarking approach

Our goal is to compare the execution time of each query in both Oracle and Spark. In the case of Oracle, this evaluation was more straightforward as we could measure the execution time using an integrated development environment (IDE) such as SQL Developer. To ensure accurate results, we implemented a PL/SQL script to iterate over the whole results dataset in order to validate that the query was executed over the entire result set and not just over a sample of it. An example of the PL/SQL script described above for the first query (Q1a) is presented below.

```
DECLARE
  t1 timestamp;
  t2 timestamp;
  l_name varchar2(30);
BEGIN
  t1 := systimestamp;
  FOR c1 IN (SELECT mp.id mespiid
                        ,mp.name mspname
                        ,te.name templname
                        ,se.id
                        ,se.name
                        ,u.name unitname
                        ,u.symbol
                        ,rd.val
                        ,rd.device_date
                        ,hs.name healthanme
                        ,li.min_value
                        ,li.max_value
                        ,CASE
                              WHEN u.id = 1
                                    THEN CASE
                                          WHEN li.normal_bool = rd.val
                                                THEN li.normal_message
                                          ELSE li.outlimits_message
                                          END
                              END message
                  FROM (
                        SELECT asset_id
                              ,point_id
                              ,device_date
                              ,sensor_id
                              ,health_id
                              ,value val
                        FROM rd_sensor_digital_history

                        UNION ALL

                        SELECT asset_id
                              ,point_id
                              ,device_date
                              ,sensor_id
                              ,health_id
                              ,value val
                        FROM rd_sensor_analog_history
                        ) rd
                  LEFT JOIN a_measuring_point mp ON rd.point_id =
mp.id
                  LEFT JOIN s_sensor se ON rd.sensor_id = se.id
                  LEFT JOIN a_asset a ON rd.asset_id = a.id
                  LEFT JOIN s_unit u ON se.unit_id = u.id
                  LEFT JOIN s_health_status hs ON rd.health_id = hs.id
                  LEFT JOIN te_sensor_template te ON mp.id =
te.measuring_point_id
                        AND a.template_id = te.template_id
                  LEFT JOIN s_customer_limit li ON
te.customer_limits_id = li.id) LOOP
     l_name := c1.id;
     t2 := systimestamp;
  END LOOP;
  dbms_output.put_line('Start: '||t1);
  dbms_output.put_line('  End: '||t2);
  dbms_output.put_line('Elapsed Seconds: '||TO_CHAR(t2-t1,
'SSSS.FF'));
END;
```

However, evaluating Spark queries was more complex due to its lazy evaluation mechanism that has been already mentioned. Spark delays the execution of transformations until a Spark action is performed. To assess the performance of Spark queries, we leveraged the 'noop' (no operation) option. By specifying 'noop' as the format for writing a DataFrame, Spark would perform the same computations used in writing a file but without actually saving any records. This allowed us to evaluate the query performance without the overhead of data storage.

Below are the Spark scripts that were executed for the first query (Q1a) in order to calculate the executed time. The two scripts correspond to the two different ways to execute a Spark query, with DataFrame API on Scala and with SparkSQL using the 'noop' option.

```
//Q1a with DataFrame API

// Read congig csv files into csv
val df_mespoint = spark.read.option("header",true).csv("")
val df_sensor = spark.read.option("header",true).csv("")
val df_asset = spark.read.option("header",true).csv("")
val df_unit = spark.read.option("header",true).csv("")
val df_health = spark.read.option("header",true).csv("")
val df_senstempl = spark.read.option("header",true).csv("")
val df_custlimit = spark.read.option("header",true).csv("")

// Read RD csv files for 1DAY into df
val df_rdA = spark.read.option("header",true).csv("")
val df_rdD = spark.read.option("header",true).csv("")

val joindf1a = df_rdA.unionAll(df_rdD).as("rd")
                .join(df_mespoint.as("mp"), col("rd.point_id") ===
col("mp.id"), "left")
                .join(df_sensor.as("se"),col("rd.sensor_id") ===
col("se.id"),"left" )
                .join(df_asset.as("a"), col("rd.asset_id") ===
col("a.id"),"left" )
                .join(df_unit.as("u"), col("se.unit_id") ===
col("u.id"),"left" )
                .join(df_health.as("hs"), col("rd.health_id") ===
col("hs.id"),"left" )
                .join(df_senstempl.as("te"), col("mp.id") ===
col("te.measuring_point_id") &&

col("a.template_id") === col("te.template_id"),"left" )
                .join(df_custlimit.as("li"),
col("te.customer_limits_id") === col("li.id"),"left" )
```

```
val maindf1a = joindf1a
.select(
    $"mp.id".alias("mespiid"),
    $"mp.name".alias("mspname"),
    $"te.name".alias("templname"),
    $"se.id".alias("sensorid"),
    $"se.name".alias("sensorname"),
    $"u.name".alias("unitname"),
    $"u.symbol".alias("unitsymbol"),
    $"rd.value".alias("sensorvalue"),
    $"rd.device_date".alias("latestutcdate"),
    $"hs.name".alias("healthanme"),
    $"li.min_value",
    $"li.max_value",
    when($"u.id" === 1, when($"li.normal_bool" === $"rd.value",
$"li.normal_message").otherwise($"li.outlimits_message")).alias("mes
sage")
)
.write.format("noop").mode("overwrite").save()
```

```
//Q1a with SparkSQL

// Read congig csv files into csv
val df_mespoint = spark.read.option("header",true).csv("")
val df_sensor = spark.read.option("header",true).csv("")
val df_asset = spark.read.option("header",true).csv("")
val df_unit = spark.read.option("header",true).csv("")
val df_health = spark.read.option("header",true).csv("")
val df_senstempl = spark.read.option("header",true).csv("")
val df_custlimit = spark.read.option("header",true).csv("")
// Read RD csv files for 1DAY into df
val df_rdA = spark.read.option("header",true).csv("")
val df_rdD = spark.read.option("header",true).csv("")


// Register DataFrames as TempViews
df_rdA.createOrReplaceTempView("rd_sensor_digital_history1")
df_rdD.createOrReplaceTempView("rd_sensor_analog_history1")
df_mespoint.createOrReplaceTempView("a_measuring_point1")
df_sensor.createOrReplaceTempView("s_sensor1")
df_asset.createOrReplaceTempView("a_asset1")
df_unit.createOrReplaceTempView("s_unit1")
df_health.createOrReplaceTempView("s_health_status1")
df_senstempl.createOrReplaceTempView("te_sensor_template1")
df_custlimit.createOrReplaceTempView("s_customer_limit1")
```

```
val sqlQ1a = spark.sql("""
  SELECT mp.id mespiid,
      mp.name mspname,
      te.name templname,
      se.id,
      se.name,
      u.name unitname,
      u.symbol,
      rd.val,
      rd.device_date,
      hs.name healthanme,
      li.min_value,
      li.max_value,
      CASE
            WHEN u.id = 1
                  THEN CASE
                              WHEN li.normal_bool = rd.val
                                    THEN li.normal_message
                              ELSE li.outlimits_message
                              END
            END message
      FROM (
            SELECT asset_id, point_id, device_date, sensor_id,
health_id, VALUE val FROM rd_sensor_digital_history1
            UNION ALL
            SELECT asset_id, point_id, device_date, sensor_id,
health_id, VALUE val FROM rd_sensor_analog_history1
      ) rd
      LEFT JOIN a_measuring_point1 mp ON rd.point_id = mp.id
      LEFT JOIN s_sensor1 se ON rd.sensor_id = se.id
      LEFT JOIN a_asset1 a ON rd.asset_id = a.id
      LEFT JOIN s_unit1 u ON se.unit_id = u.id
      LEFT JOIN s_health_status1 hs ON rd.health_id = hs.id
      LEFT JOIN te_sensor_template1 te ON mp.id = te.measuring_point_id
AND a.template_id = te.template_id
      LEFT JOIN s_customer_limit1 li ON te.customer_limits_id =
li.id""")
      .write.format("noop").mode("overwrite").save()
```

The 'noop' approach was used for the first three queries (Q1a, Q1b & Q2), while for the aggregated query (Q3), counting the result and calling with the `.show()` method was sufficient to evaluate its performance. Here, it worths mentioning that even though `.count()` in spark is an action when it is used with aggregated data as part of an aggregation transformation (like `groupBy().count()`) it returns a *RelationalGroupedDataset* object. That means that it works as a transformation and it returns a new Dataframe. For this reason, it is important to use the `.count()` method inside the aggregation and then triggering it with an action like `.show()`.

7.3 Experiments & Results

In the preceding chapters, we laid the foundation for our study, detailing the methodology, tools, and frameworks in use. This chapter aims to bring these elements together and present a cohesive analysis of our experimental findings.

In the experimental evaluation, we aimed to compare the performance of the selected four queries on both Oracle and Spark platforms. Initially, the queries were executed without the incorporation of optimization techniques. Subsequent to this preliminary run, the same queries were re-executed with optimization strategies applied. Given the multifaceted nature of Spark, which offers various modes for query execution, we chose to implement and assess two prominent approaches: using SparkSQL and the DataFrame API in Scala. The experiments took place on the local environment to be comparable between the platforms. Moreover, the experiments and Spark evaluations were extended to a distributed computing environment, Azure Databricks. This inclusion not only enhanced the depth of our study but also mirrored the real-world scenarios.

One-Day Dataset

For the dataset of one day, as already mentioned, an extensive analysis was concluded by running all four queries on both platforms (Oracle and Spark) on the local environment and on Azure Databricks. Each query was executed multiple times and the average of the execution time is presented.

Execution Time Without Optimizations (seconds):

Query	Oracle	Spark DataFrame API	Spark SQL	Databricks DataFrame API	Databricks SparkSQL
Q1a	50.0	30.0	25.0	8.0	7.0
Q1b	7.0	16.0	12.0	6.5	5.0
Q2	55.0	25.0	20.0	8.0	7.0
Q3	7.0	15.0	13.0	5.0	5.0

Table 3 Execution Times in Seconds without Optimization for 1 Day dataset

For Oracle the queries Q1b, where a filter selection was applied, and Q3, the aggregated one, were those that performed better in the current set up. While Q1a and Q2 are general retrieval queries that require a full table scan, Q1b introduces specific conditions which in systems, like Oracle, can be highly efficient. Q3 is essentially an aggregation query, and Oracle is highly optimized for such operations. Moreover, the exclusion of hs.id != 1 can also facilitate faster data processing. Spark, being a distributed system, introduces overhead for task partitioning, shuffling, and data movement, which for such aggregation tasks, can introduce a delay for machines with limited computing power.

The Q1a and Q2 perform better on Spark as they are comprehensive queries without too many filters. Spark can efficiently distribute the load and process the data concurrently.

Queries in Spark were executed with two methods, Spark DataFrame API and SparkSQL. While both of them provide ways to process data, Spark SQL tends to be more expressive and human-readable. However, the DataFrame API, with its programmatic nature, often allows for more fine-tuned optimizations. Given the similarity of the operations in both of the queries, as they have the same business logic and will produce the same result set, the logical plans will be very similar. Catalyst will convert the logical plan into the same physical plan, because the operations' intent is the same and that is the reason why the differences in practice are negligible in terms of performance.

Running Spark on Azure Databricks reduced the execution times noticeably. Even with a relatively smaller dataset, Databricks showcased superior performance, emphasizing the optimized and distributed nature of Spark, which allows it to harness the power of multiple nodes.

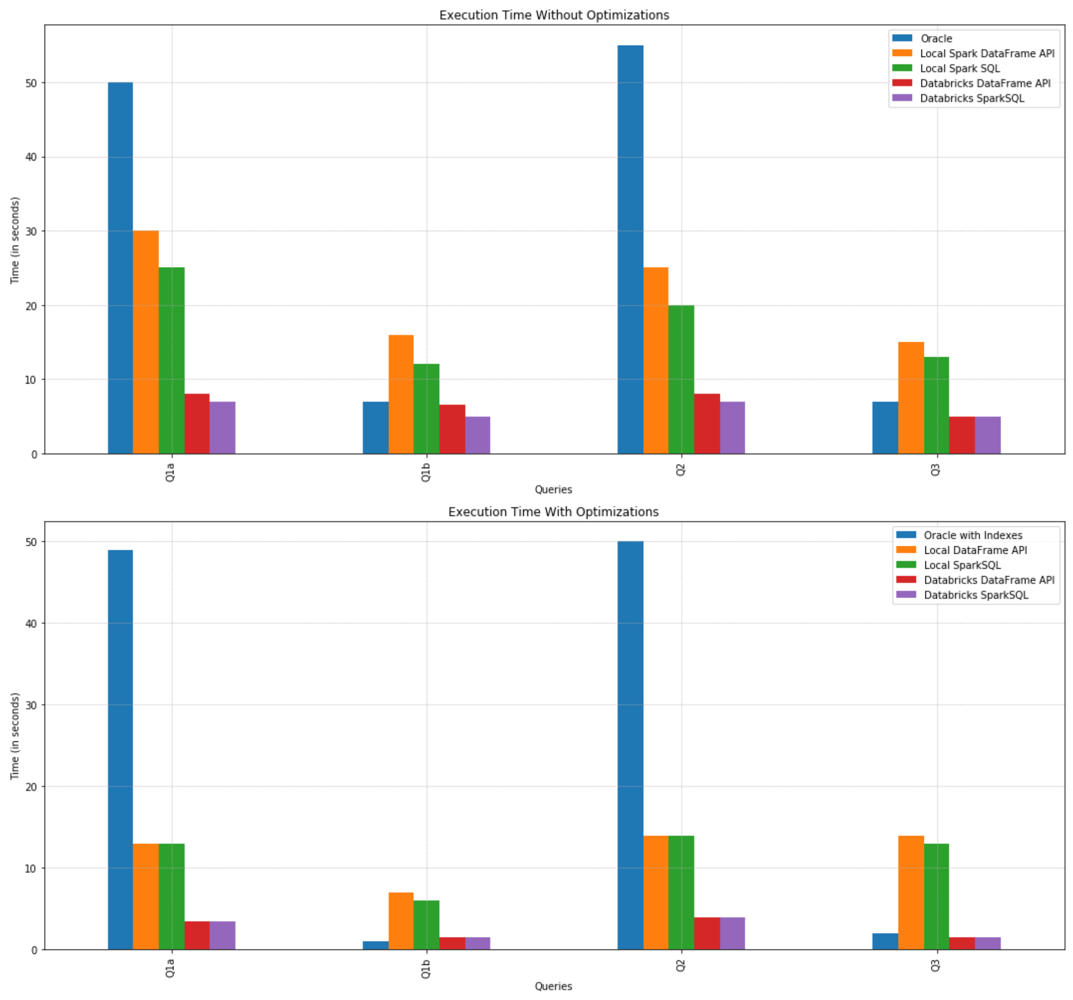


Figure 9 One Day Execution times

After the initial tests, optimization techniques were applied in both environments to understand their impact on performance. For the Oracle database the optimization technique that was applied, was the creation of Indexes. Indexes are data structures that allow databases to locate rows faster. Without indexes, the database system would need to scan the entire table to find a specific row. By using indexes, a roadmap is for the database to quickly find the data without having to search every row. In this case, the indexes were created on the primary keys of all tables, along with one composite index for each of the main tables on the foreign keys used for the joining operations.

Execution Time With Optimizations (seconds):

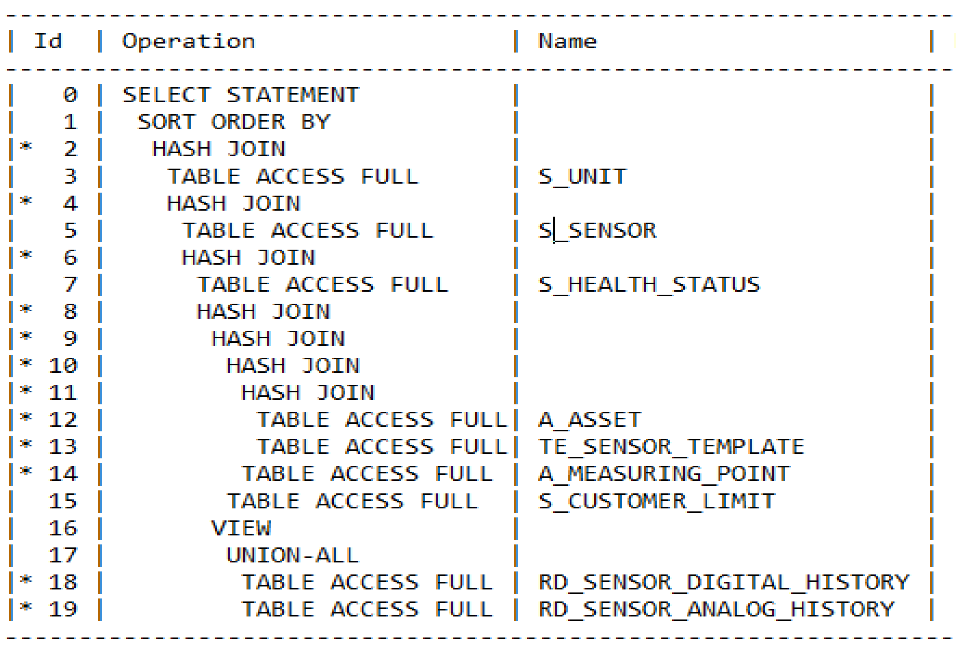
Query	Oracle	Spark DataFrame API	Spark SQL	Databricks DataFrame API	Databricks SparkSQL
Q1a	49.0	13.0	13.0	3.5	3.5
Q1b	1.0	7.0	6.0	1.5	1.5
Q2	50.0	14.0	14.0	4.0	4.0
Q3	2.0	14.0	13.0	1.5	1.5

Table 4 Execution Times in Seconds with Optimization for 1 Day dataset

The introduction of indexes in Oracle made a marked difference, especially for the aggregated query Q3 and queries that involved filtering like the Q1b.

An examination of the execution plan of Oracle for Q1b before and after the indexes showcases the impact of the index on the specific query.

Q1b Execution Plan in Oracle Before Index



```
----------------------------------------------------------------
| Id  | Operation                    | Name                     |
----------------------------------------------------------------
|   0 | SELECT STATEMENT             |                          |
|   1 |  SORT ORDER BY               |                          |
|*  2 |   HASH JOIN                  |                          |
|   3 |    TABLE ACCESS FULL         | S_UNIT                   |
|*  4 |    HASH JOIN                 |                          |
|   5 |     TABLE ACCESS FULL        | S_SENSOR                 |
|*  6 |     HASH JOIN                |                          |
|   7 |      TABLE ACCESS FULL       | S_HEALTH_STATUS          |
|*  8 |      HASH JOIN               |                          |
|*  9 |       HASH JOIN              |                          |
|* 10 |        HASH JOIN             |                          |
|* 11 |         HASH JOIN            |                          |
|* 12 |          TABLE ACCESS FULL| A_ASSET                     |
|* 13 |          TABLE ACCESS FULL| TE_SENSOR_TEMPLATE          |
|* 14 |         TABLE ACCESS FULL | A_MEASURING_POINT           |
|  15 |        TABLE ACCESS FULL  | S_CUSTOMER_LIMIT            |
|  16 |       VIEW                   |                          |
|  17 |        UNION-ALL             |                          |
|* 18 |         TABLE ACCESS FULL | RD_SENSOR_DIGITAL_HISTORY   |
|* 19 |         TABLE ACCESS FULL | RD_SENSOR_ANALOG_HISTORY    |
----------------------------------------------------------------
```

Q1b Execution Plan in Oracle After Index

```
--------------------------------------------------------------------------------
| Id  | Operation                                 | Name                        |
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                          |                             |
|*  1 |  HASH JOIN                                |                             |
|   2 |   TABLE ACCESS FULL                       | S_UNIT                      |
|*  3 |   HASH JOIN                               |                             |
|   4 |    TABLE ACCESS FULL                      | S_SENSOR                    |
|*  5 |    HASH JOIN                              |                             |
|   6 |     TABLE ACCESS FULL                     | S_HEALTH_STATUS             |
|*  7 |     HASH JOIN                             |                             |
|*  8 |      HASH JOIN                            |                             |
|*  9 |       HASH JOIN                           |                             |
|* 10 |        HASH JOIN                          |                             |
|  11 |         TABLE ACCESS BY INDEX ROWID BATCHED| A_ASSET                    |
|* 12 |          INDEX RANGE SCAN                 | IDX_A_ASSET                 |
|  13 |         TABLE ACCESS FULL                 | TE_SENSOR_TEMPLATE          |
|  14 |        TABLE ACCESS FULL                  | A_MEASURING_POINT           |
|  15 |       TABLE ACCESS FULL                   | S_CUSTOMER_LIMIT            |
|  16 |      VIEW                                 |                             |
|  17 |       UNION-ALL                           |                             |
|  18 |        TABLE ACCESS BY INDEX ROWID BATCHED | RD_SENSOR_DIGITAL_HISTORY  |
|* 19 |         INDEX RANGE SCAN                  | IDX_RD_DIGITAL_COMP         |
|  20 |        TABLE ACCESS BY INDEX ROWID BATCHED | RD_SENSOR_ANALOG_HISTORY   |
|* 21 |         INDEX RANGE SCAN                  | IDX_RD_ANALOG_COMP          |
--------------------------------------------------------------------------------
```

However, for the rest of the Queries, no big difference was spotted. This can be attributed to the absence of filtering conditions, leading to a full table scan by the optimizer.

Spark on the other hand, applies some automatic optimizations using its Catalyst optimizer, which analyzes the user's program and determines the most efficient execution plan for data operations. These optimizations are performed behind the scenes, helping to improve the performance and efficiency of Spark applications.

One notable optimization is the broadcast join, where Spark broadcasts smaller datasets to the executors. When a dataset is broadcasted, it is sent to all worker nodes once and cached. This can be represented as an optimization to reduce data transfer:

Without broadcast: $Data \times Nodes$

With broadcast: $Data + Nodes$

Spark employs this technique to mitigate the need for shuffling data during the join process.

The cost of shuffling can be represented as:

$$C_{shuffle} = N \times S \times P$$

Equation 1 Shuffling equation

Where:

- N is the number of nodes
- S is the average size of data sent to a node
- P is the overhead of opening a connection to a node

By broadcasting smaller datasets to the executors, Spark avoids the costly data shuffling operation, resulting in improved performance.

However, by our experience we know that Spark may sometimes try to broadcast larger datasets as well, resulting in decreased performance. This option can be manually disabled by setting the spark.sql.autoBroadcastJoinThreshold parameter to '-1', emphasizing the importance of examining the execution plan to understand Spark's execution strategy.

These Spark optimizations were applied by default in the first place, and that is another reason why Spark on some queries performed better than Oracle before applying any custom optimization.

Further to the Spark default optimizations, two optimization techniques, repartitioning and caching, were applied.

The main Dataframe was repartitioned based on the columns used in the join conditions and then cached. For our specific use and case and with the current set up we could utilize `.cache()` as the Dataframe fits in memory. That way all operations will be held entirely on memory which could offer faster access times.

Since, in our case repartition was intended to optimize the join operations we first applied the repartition and after we cached the DataFrame. Any operation post-caching (like joins) can take advantage of the co-located data. This can speed up these operations as, data that needs to be joined might already be in the same partition, reducing the need for further shuffling.

Upon applying the techniques of repartitioning and caching, there was a significant drop in the execution times in all the queries executed on the local Spark environment except the aggregated one, Q3, where it almost remained the same. These methods optimize Spark's in-memory processing capabilities and its distributed data partitioning. The improvements were more pronounced in some queries than others, indicating the varying influence of optimization techniques based on the nature of the query.

The combination of Spark optimizations and Azure Databricks led to further reduced execution times. The contrast between local Spark and Databricks was more evident with optimizations, emphasizing the advantages of a distributed environment. Even with a relatively smaller dataset, Databricks showcased superior performance, emphasizing its optimized nature.

Whole-week Dataset

It was important for our experiments to be implemented on a larger dataset so the same procedure like before was implemented for the whole-week dataset as well. The bigger dataset increases the complexity of the execution making very important for our performance analysis of the two platforms as it offers a broader perspective on their performance.

<u>Execution Time Without Optimizations (seconds):</u>

Query	Oracle	Spark DataFrame API	Spark SQL	Databricks DataFrame API	Databricks SparkSQL
Q1a	450.0	125.0	135.0	37.0	36.0
Q1b	3.0	85.0	58.0	23.0	22.0
Q2	500.0	127.0	130.0	39.0	38.0
Q3	53.0	95.0	86.0	22.0	22.0

Table 5 Execution Times in Seconds without Optimization for whole-week dataset

We observed similar trends when evaluating the larger dataset of the full week.

Again, the initial execution of the queries happened before any custom optimization was applied. Oracle's performance was again poorer, when a full table scan was required was again poorer compare to Spark, for both local and Databricks implementations indicating Oracle's relative inefficiency for handling large-scale datasets on those scenarios. Those were the queries Q1a and Q2 where the difference between the two platforms was noticeable. On contrast for the aggregated query Oracle could perform better than Spark running on the same environment. The biggest advantage of Oracle was for the query, Q1b, where we apply a filter. In this case Oracle achieved the best performance even when compared to the execution times of the Databricks cluster.

For those queries that required the full table scan the performance of Spark was significant better compared to Oracle running on the same environment. Even compared to the one-day dataset the difference of the two frameworks was increased, highlighting once more that Spark's true potential shines through in scenarios with bigger data volumes.

Additionally, the gap between local Spark instances and Azure Databricks widened as well. Databricks, with its multi-node capabilities, significantly outperformed its local counterpart on all queries. It also outperformed Oracle on the aggregated query Q3 but not for the one with the filter condition, Q1b, where still Oracle hold the best performance.

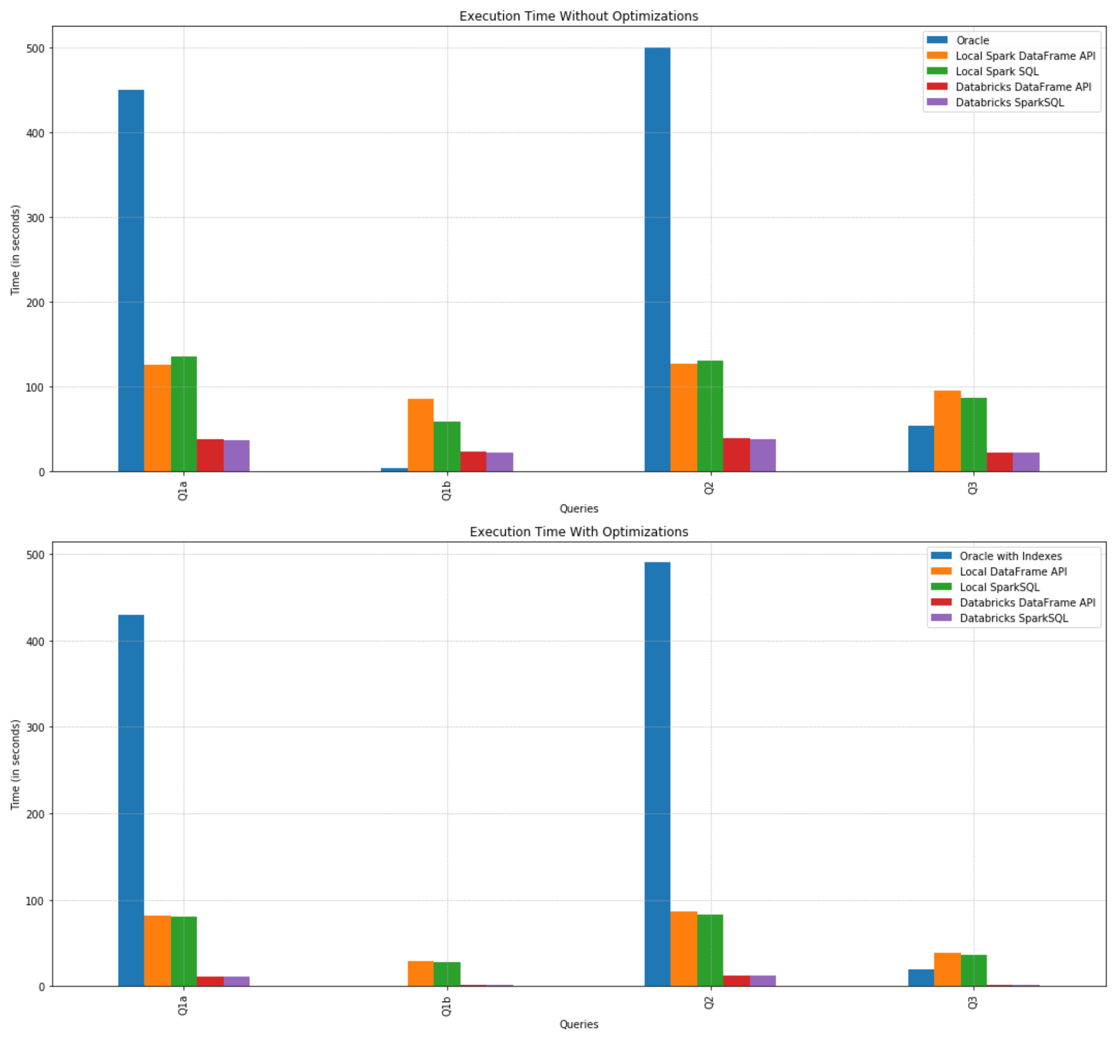


Figure 10 Whole-week execution times

After weaving in again the same optimization techniques as described above the performance of the same queries reevaluated. The impact of the indexes on Oracle for Queries Q1b and Q3 was again noticeable, reducing the execution time drastically but, again with out any adding any significant improvement to the rest of the queries.

<u>Execution Time With Optimizations (seconds):</u>

Query	Oracle	Spark DataFrame API	Spark SQL	Databricks DataFrame API	Databricks SparkSQL
Q1a	430.0	82.0	81.0	11.0	11.0
Q1b	0.5	29.0	28.0	2.0	3.0
Q2	490.0	87.0	83.0	13.0	14.0
Q3	20.0	39.0	36.0	1.5	2.0

Table 6 Execution Times in Seconds with Optimization for 1 Week dataset

The impact on Spark optimizations was again more noticeable on both environments, as with the one-day Dataset. But, the synergy between Spark distributed environment on Databricks and the optimizations was what resulted to the best performance in almost all of the queries. The advantages of distributed processing, as exhibited by Spark in environments like Azure Databricks, made a difference in the performance of the queries, especially when dealing with extensive datasets where the benefits of parallel processing and distribution become exceedingly clear. Even though, for the filtering query, Q1b, Oracle's performance was still better, the difference was marginal.

Datasets comparison

Finally, we proceeded with a comparison in the percentage increase of the execution time between the two datasets, 1-day and whole-week. The graph (figure 11) illustrates the scalability of each platform in handling larger datasets As the dataset expands from a 1-day span to a full week, there's a corresponding rise in execution times across every platform and for all queries. This is expected due to the larger volume of data being processed.

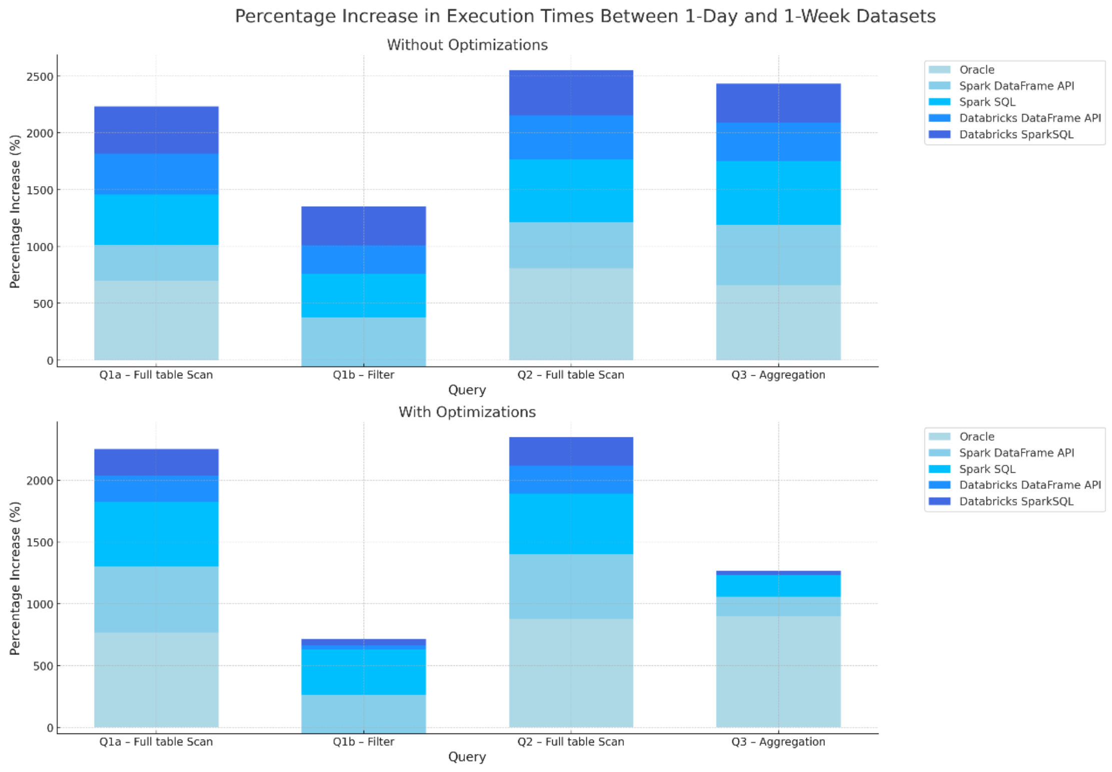


Figure 11 Percentage increase in execution times

Oracle consistently demonstrated significant percentage increases in execution times when transitioning from the smaller to the larger dataset. Especially for the types of queries that required a full table scan, Oracle's execution time soared, being up to 8 times longer on the whole-week dataset. This suggests potential scalability concerns for Oracle when dealing with extensive data. In contrast, Spark showcased better adaptability to the increased dataset size, with relatively smaller percentage increases in execution times. Additionally, the effectiveness of optimizations was evident in the reduced percentage increases for some queries, highlighting the importance of tailored optimization techniques.

It is worth mentioning that the increase on the Databricks platform was marginal, emphasizing its optimized distributed processing capabilities in distributed environments.

8 Conclusion & Future Work

In conclusion, the evaluation of the four queries on both Spark and Oracle SQL led us to some interesting insights regarding their performance. Initially, without implementing any optimization techniques, Spark demonstrated improved execution times for most of the queries compared to Oracle, particularly for the larger dataset and especially when was executed on a distributed environment. However, the aggregated query and the query with the filter condition performed better in Oracle, in some cases. That indicated than not all queries can show improved performance when moving to Spark. The performance variations among different queries express the need of handling each query separate and there is no one unique solution for all problems.

The effect of optimizations, both in Oracle and Spark, highlights the importance of fine-tuning but again the systems should be tailoring according to the data and queries. We noticed that is not only the queries that were not created equal but also the optimizations are not all equal. While some queries significantly benefit from specific optimization techniques, others might remain relatively unaffected. This emphasizes once more the importance of understanding the data and query characteristics.

Additionally, and especial for Spark, the platform on which the queries will be executed plays a pivotal role on the performance. As shown, Azure Databricks significantly enhanced Spark's performance in comparison to a local configuration on both scenarios with and without optimizations.

On the other hand, the difference between SparkSQL and DataFrame API in performance terms is negligible. Both have their advantages and disadvantages and it is on the user's hand to decide which fits best his purpose.

Overall, this study highlights the importance of considering optimization techniques and understanding the strengths and characteristics of each system and each query. Both Spark and Oracle have their own advantages and use cases and it is important to effectively utilize the appropriate optimizations and tailor them to the specific workload and dataset size, in order to achieve efficient and high-performance data processing and analysis.

For queries with complex joins, filtering conditions and aggregations executed on moderate size Datasets Oracle remains a reliable platform. However, for larger datasets, particularly when the nature of the query necessitates reading the entirety of the dataset, Spark emerges as a viable alternative that can enhance query performance. Furthermore, Spark offers the possibility of a distributed computing environment, bringing about significant performance boosts which could have an impact on the vast majority of workload.

Nonetheless, while considering this approach, it is essential to weigh these benefits against potential challenges like increased costs and the complexity of setting up and managing a distributed system.

8.1 Future work and improvements

Moving forward, there are several areas that can be explored for further research and improvements.

Firstly, expanding the cluster size by adding more nodes can potentially enhance the performance and scalability of the executed queries. The horizontal scale up is one of the main advantages of the distributed computing.

Secondly, exploring the integration of Spark with SQL engines through JDBC drivers could be beneficial, especially in corporate environments where relational databases are already established. This integration would enable direct connectivity to SQL engines and would eliminate the need for data transfers, even though that in our case it would not impact the query performance.

Furthermore, there are various other optimization techniques and configurations in Spark that can be investigated and could potentially lead to further improvements in query performance and overall system efficiency.

Lastly, it is important to continue monitoring and evaluating the advancements and updates in both Oracle and Spark technologies. As new versions and optimizations are released, it is essential to assess their impact on performance and explore opportunities for incorporating them into the existing system.

By continuing to explore these avenues and staying abreast of the latest developments, it is possible to further enhance the performance and capabilities of both Oracle and Spark in processing and analyzing large datasets.

9 References

[S1] Codd, E. F. (1970), 'A Relational Model of Data for Large Shared Data Banks', Commun. ACM 13 (6), 377-387.

[S2] Passing, Johannes. (2008). The Google File System and its application in MapReduce.

[S3] S. Ghemawat, H. Gobioff, S.-T. Leung, The Google File System, ACM SOSP

[S4] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters

[S5] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. “Bigtable: A Distributed Storage System for Structured Data”, 2006.

[S6] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System," 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010

[S7] Wang, Yandong; Goldstone, Robin; Yu, Weikuan; Wang, Teng (October 2014). "Characterization and Optimization of Memory-Resident MapReduce on HPC Systems". 2014

[S8] Murthy, Arun (15 August 2012). "Apache Hadoop YARN – Concepts and Applications". hortonworks.com. Hortonworks

[S9] https://spark.apache.org/

[S10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 10–10

[S11] Zaharia, M. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of the Ninth USENIX NSDI Symposium on Networked Systems Design and Implementation (San Jose, CA, Apr. 2527, 2012).

[S12] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational data processing in Spark. In Proceedings of the ACM SIGMOD International Conference on Management of Data. ACM, 1383–1394

[S13] https://en.wikipedia.org/wiki/SQL:2003

[S14].Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine learning in apache spark. Journal of Machine Learning Research 17, 34 (2016), 1—7

[S15] https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

[S16]. J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 599--613, 2014.

[S17] https://www.databricks.com/blog/2016/08/15/how-to-use-sparksession-in-apache-spark-2-0.html

[S18] Kluyver, T. et al., 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In F. Loizides & B. Schmidt, eds. Positioning and Power in Academic Publishing: Players, Agents and Agendas. pp. 87–90.

[S19] https://pypi.org/project/spylon-kernel/

[S20] https://azure.microsoft.com/en-us/products/databricks

[S21] https://azure.microsoft.com/en-us/products/storage/blobs