Evolution of Spark SQL and an Exploration into its Performance Evaluation

Project:- Big Data Management – Analytics and Data Warehouse – Spark SQL (Team 12)
Course:- Database Theory CS6051 (Professor: Seokki Lee)

Motivation

Nowadays, everything is data-driven, therefore making the most of Big Data is essential. The aim of this project is to investigate Spark SQL's ability to handle both organized and unstructured data. In particular, it explores the function of the catalyst optimizer in running complex queries using joins and aggregations. Our investigation is inspired by the goal of improving the depth of data-driven analyses through the study, which focuses on optimizing Spark SQL's scalability and efficiency within Apache Spark.

Summary of paper

The study article presents Spark SQL, an essential part of Apache Spark that uses SQL queries to handle structured and semi-structured data. Spark SQL makes data processing easy by addressing the problem of connecting big data frameworks with relational databases. SQL queries optimised for distributed execution within Spark clusters enable users to perform sophisticated operations like JOIN and GROUP BY. Through the use of sophisticated optimisations, the Catalyst optimizer improves speed, and its compatibility with Apache Hive enables the seamless conversion of Hive workloads. The Catalyst Optimizer, a complex query optimization architecture essential for improving speed, is at the core of Spark SQL. Predicate pushdown and constant folding are two of the optimizations used by Catalyst Optimizer, which cleverly rearranges SQL query operations to reduce disc I/O and data shifting. It greatly increases the performance of Spark SQL queries by dynamically optimising the query execution strategy, allowing for smooth processing over large distributed datasets. Across addition to guaranteeing quick query execution, this optimisation process enables Spark SQL to adjust to changing workloads and data patterns, preserving excellent performance across a range of analytical scenarios. In this work, will demonstrate query processing in spark SQL and its optimization.

Limitations

This study focuses only on exploring Spark SQL and its catalyst optimizer techniques. Additionally, we are analysing Spark SQL's query execution time in comparison with other SQL tools, to be presented in our analysis. The referenced document has limitations, lacking depth in real-world deployment challenges, detailed exploration of system behaviour under varied workloads, and comprehensive comparisons with similar systems, hindering a fair assessment of Spark SQL's advantages.

System requirements and installation

To setup spark SQL there are certain prerequisites that needs to be meet. Java with minimum version of 8 is required to run spark sql. Scala support is necessary as spark is written in this particular functional language. Hadoop distributed file system (HDFS) should be installed because it was intended to store and handle massive amounts of data over a network of devices. It enables fault tolerance and high availability, allowing Spark to analyse massive volumes of data that may not fit on the storage of a single system. In our case, the system is successfully installed, and we are currently using spark 3.2.4 version, with Scala 2.12.15 using hdfs as a default storage system. In general, there are various ways to setup spark environment, while working on large scale data processing, analysts prefer to create clusters or instances in the cloud environment. We setup the environment in the local mode as it makes easy to access and flexible for testing purposes.

The environment which was installed in the local mode has been working successfully. The system was tested in various ways such as checking the environment variables, executing the commands, creating the sql tables and retrieving the data. Scala is a popular programming language for creating Spark applications. Spark offers high-level APIs for a variety of programming languages, including Java and Python. Scala provides access to SQL capability in Spark. Spark SQL is a module that allows you to run SQL queries and interact with structured data straight from Spark applications. In view of catalyst optimizer, an attempt has been performed to check whether we are able to access the optimized logical plan consequently, the test run was completely successful and spark web UI is also accessible. It is web based dashboard which helps us to understand about the behaviour and performance of spark jobs.

Dataset Description

In the age of big data, the ecommerce industry relies on data analysis for valuable insights. This study utilizes a Kaggle-sourced ecommerce dataset to perform Spark SQL query processing. The dataset includes customer orders from various Indian states, order details (quantity, category, amount), and monthly sales targets. After data cleaning, attributes were appropriately redefined from strings to their respective data types.

Scenario 1

Spark SQL is a library that operates on the foundation of Spark and it is accessible via the command-line terminal. While Spark SQL does not provide a stand-alone data warehouse, it can communicate with a variety of data sources and storage systems. Spark SQL does not have its own storage system; instead, on it using its built-in metadata storage as its data storage backend. The warehouse directory is a location on HDFS , were Spark stores table data and managed tables metadata. Spark SQL quickly saves and manages tables by exploiting metadata management features, promising smooth integration and robust data handling. Utilizing the dataset, five (2 simple and 3 complex queries) SQL queries are formed. The support of procedural language is necessary in spark to create a table in the spark SQL. In this work, we utilized Scala programming language as a support to the SQL for creation of tables. Firstly, data which was in CSV format is converted into a data frame, then a temporary table or view has been created. As procedure languages provide support to the relational processing, a table was created and using the view, the data was transferred into the respective table.

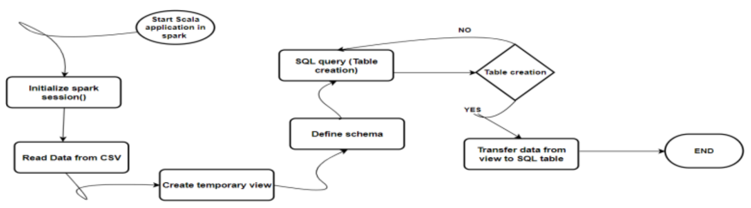

Start Scala application in spark
Initialize spark session()
Read Data from CSV
Create temporary view
Define schema
SQL query (Table creation)
Table creation
NO
YES
Transfer data from view to SQL table
END

Figure 1:- Table creation in spark SQL

In spark SQL, when an analyst wants to retrieve data with specific conditions, the query goes through various phases. Let's say analyst wants to count number of orders received from each state. When the SQL query to count orders by state is conducted in Spark SQL, it is parsed, resulting in a logical plan outlining the required operations. optimizer then applies transformations and rule-based refinements to this logical plan. Following that, a physical execution plan is created, which details the particular processes for distributed processing throughout the cluster. Optionally, code generation takes place, compiling specific query sections into bytecode for faster execution. The optimized plan reads, processes, and aggregates data across cluster nodes. The results are pooled, and the total number of orders per state is calculated. The pictorial representation of the system flow is represented in figure 2. In simple words, at presentation we will be demonstrating the process of table creation in spark SQL using functional programming language and will showcase the query processing at the backend of spark SQL.

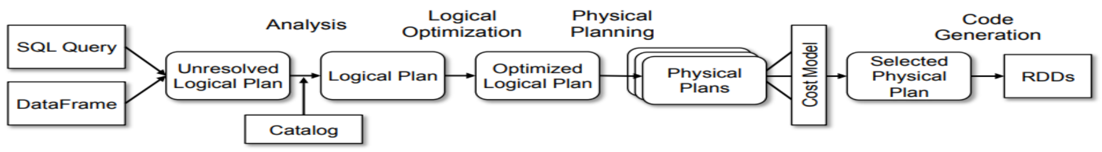
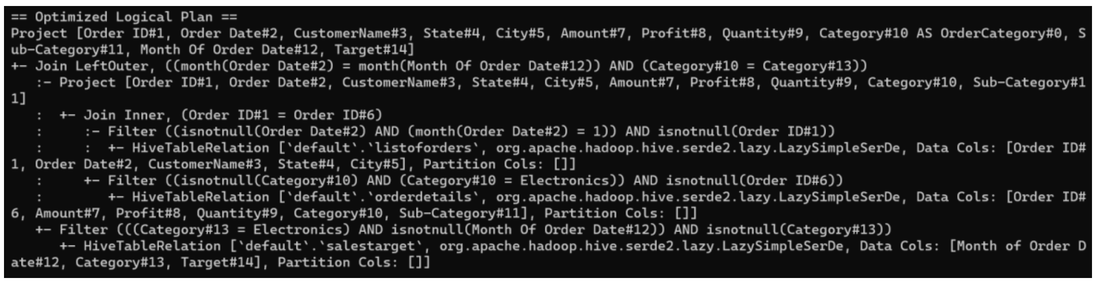


Figure 2:- Phases involved in query processing on spark SQL

Scenario 2

An essential Spark SQL feature called Catalyst Optimizer enters the scene in this situation. The abstract syntax tree's (AST) structure is accessible to it for manipulation. Catalyst is in charge of improving query performance and operates at the logical plan level. The logical plan is carefully examined, and a variety of optimization approaches are used. These methods include constant folding, which eliminates the need to recompute constant expressions within the query during execution by computing them during the optimization step. Another optimization called "predicate pushdown" reduces the quantity of data that has to be processed by pushing filters as near as feasible to the data source. Additionally, Catalyst manages more intricate optimizations such as those for join techniques, aggregations, and projections, all of which are intended to improve the effectiveness of query execution. In order to identify the most effective physical execution approach, Catalyst develops several optimized logical plans and investigates alternative cost models. For, example let say that an analyst wants to get aware of optimization process that catalyst is using to optimize the query and enhancing the performance for retrieving information about orders, order details, and sales targets for a specific category and month execution (Query 2).



```
== Optimized Logical Plan ==
Project [Order ID#1, Order Date#2, CustomerName#3, State#4, City#5, Amount#7, Profit#8, Quantity#9, Category#10 AS OrderCategory#0, S
ub-Category#11, Month Of Order Date#12, Target#14]
+- Join LeftOuter, ((month(Order Date#2) = month(Month Of Order Date#12)) AND (Category#10 = Category#13))
   :- Project [Order ID#1, Order Date#2, CustomerName#3, State#4, City#5, Amount#7, Profit#8, Quantity#9, Category#10, Sub-Category#1
1]
   :  +- Join Inner, (Order ID#1 = Order ID#6)
   :     :- Filter ((isnotnull(Order Date#2) AND (month(Order Date#2) = 1)) AND isnotnull(Order ID#1))
   :     :  +- HiveTableRelation [`default`.`listoforders`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, Data Cols: [Order ID#
1, Order Date#2, CustomerName#3, State#4, City#5], Partition Cols: []]
   :     +- Filter ((isnotnull(Category#10) AND (Category#10 = Electronics)) AND isnotnull(Order ID#6))
   :        +- HiveTableRelation [`default`.`orderdetails`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, Data Cols: [Order ID#
6, Amount#7, Profit#8, Quantity#9, Category#10, Sub-Category#11], Partition Cols: []]
   +- Filter (((Category#13 = Electronics) AND isnotnull(Month Of Order Date#12)) AND isnotnull(Category#13))
      +- HiveTableRelation [`default`.`salestarget`, org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe, Data Cols: [Month of Order D
ate#12, Category#13, Target#14], Partition Cols: []]
```

Figure 3:- Optimized logical plan for query 2

The Spark SQL Catalyst optimizer strategically optimizes query performance by pushing down filters, reordering joins, and converting left outer joins to inner joins based on filter criteria and non-null checks. These optimizations, illustrated in Query 2's execution plan, significantly enhance response times, especially for large datasets. The optimizer starts by filtering relevant entries from 'listoforders,' broadcasting data using "BroadcastExchange." A similar broadcast occurs for 'orderdetails' data filtered by the 'Electronics' category. To minimize data shuffling, "BroadcastHashJoin" connects matching records, and 'Project' operations select specific output columns. Multiple 'Filter' operations further refine datasets. The 'AdaptiveSparkPlan' optimizes the plan based on runtime statistics, enhancing efficiency. This study (scenario 2) demonstrates the impactful optimization process of the Spark SQL Catalyst optimizer, showcasing its vital role in maximizing query plan efficiency, making it ideal for large-scale data processing.

Contributions

In our collaborative project, it's important to note that all team members actively participated in every aspect, reflecting our shared commitment to a comprehensive approach. While we are specifically highlighting individual contributions in sub-parts, it is crucial to recognize that our collective efforts were involved in all aspects of the project. All the team members worked together in researching and selecting pertinent research papers, system installation, and understanding the working process of catalyst optimizer. Uday majorly focussed on Creation of schemas in spark SQL, SQL query formulation, and Catalyst optimization in logical plan of query (Scenario 2), while Naveen thoroughly worked on Spark SQL architecture, Creation of schemas in spark SQL, and SQL query formulation (Scenario 1). Harshini majorly worked on ensuring that data was cleaned, SQL query formulation and Catalyst optimization methods(Scenario 2) further enriched our project. Simultaneously, Siva handled reference papers, dataset curation, and relational architecture in Spark SQL (Scenario 1). Additionally, our collaboration extended to collective efforts in report formatting, ensuring a well-rounded and cohesive project outcome.