Parallel Data Retrieval in Distributed Data Warehouse

Ashirwad Pradhan 2019H1120043P, Shashank S 2019H1120054P

Abstract

The growing need for storage of huge amounts of data has certainly shown that scaling vertically is not an optimal solution to the problem. The huge amounts of data in a data warehouse can be queried more efficiently if the data is stored in a distributed fashion. Thus, to scale the system horizontally we need an efficient architecture which will enable us to query data parallelly from the underlying distributed data storage. Our report proposes a solution which provides an architecture and an appropriate read query optimization method based on SQL query to efficiently query the data. The report also discusses the running time cost based on read speed metrics that we have obtained from the test results based on comparing synchronous execution vs. asynchronous execution. This report also discusses optimal cases where the architecture is expected to give better results.

Keywords

Distributed Data Warehouse, Asynchronous Execution, Parallel SQL Execution, Read Optimized

Introduction

Hardware Efficiency

Recent advancements in storage technology have enabled us to gain more read and write speeds as compared to a few years back and has made I/O operations faster than ever. A few years back the advent of solid-state drives (popularly known as **SSD**) based on flash storage technology has enabled I/O operations to be at par with the growth rate of computing power. In the past, every application and software architecture has mostly been limited by the speed of I/O operations which made overall time efficiency for solving a problem worse. Still, it is the same story as most of the problems are purely limited by the data speeds. To make the I/O process more efficient we can look up to distributed storage architecture as a solution. The storage disks can be read parallelly as we query the data. Some examples include distributed data stores in which the data is stored on a number of nodes which can either be used to provide parallelism or redundancy.

Distributed File System

To efficiently manage the above hardware solution we also need an effective software layer which will manage the low level read and write operations for the developers or users. Several solutions have been proposed for that which includes **Apache's Hadoop(HDFS)**, **Google's GFS**

and **Borg** to name a few. They provide us an abstraction to read and write data from a distributed file system and not worry about the implementation details. They also help us with additional utility functions such as managing the cluster, enable journaling and also helps in achieving redundancy and fault tolerance of the stored data.

Multithreading

As we know that data retrieval is highly I/O intensive, we can spawn multiple threads to handle this efficiently. The number of threads that are spawned can be very very larger than the number of threads that can actually run on the multiprocessor machine. The idea is, as the I/O operations are very slow the CPU will have to wait most of the time for data to arrive. Hence even if the number of threads that can be run on a multithreaded processor is 4 we can spawn for example about 50 or more threads simultaneously so that our CPU runs busy at all times. But still, the number 50 is problem-dependent. If the program is making too many network calls and disk requests then this number is fine. The optimum number of threads to spawn in an I/O intensive task is completely task dependent but surely it should be much larger than it is required for a computation heavy task.

Asynchronous Execution

Asynchronous execution refers to the concurrent execution of the program. In the concurrent execution, the thread utilizes a single thread of execution but it doesn't wait for blocking calls (blocking I/O calls). Instead, it utilizes that waiting time for executing other tasks (computation, making another I/O call, etc.) therefore providing an illusion that the program is running parallelly. The blocking calls are generally those I/O calls where the CPU just waits for the I/O to arrive thereby losing effective time as the I/O calls are 100's of magnitude slower than computation times. Many programming models have come up in recent times (**Javascript's Promises, Java's CompletableFuture, Python's asyncio**) which allows us to use this pattern to take advantage of concurrency.

Problem Statement

Before discussing the proposed solution for improving I/O time by parallelly retrieving data from the data warehouse there are few things we need to understand about the nature of the problem.
Our problem mainly focuses on technologically distributed data warehouses in which there are multiple local data warehouses. This can be due to the fact that the business is geographically distributed and each region is big enough (data-wise) to construct a warehouse to perform BI queries.

The data warehouses are physically separated, although they have one logical view of all the data warehouses taken together. If we need to perform a query we need to collate all the data into a central repository and then use the traditional centralized warehouse techniques to query the data.

If we look more closely at the arrangement of data, the data is already in a distributed fashion across the geographical locations. We can utilize this pattern to query data efficiently without the overhead of collating the data to a central repository and then querying from it.
Formally we can represent the problem as follows:

Given Input:

Let $\mathbf{DW = [\ DW_1 , DW_2 , DW_3 , \dots , DW_n\ ]}$ represents the set of data warehouses we have, which are distributed geographically.

Each data warehouse contains multiple tables $\mathbf{T = [\ T_1 , T_2 , T_3 , \dots , T_n\ ]}$ which contains rows of data.

Each table $\mathbf{T_i}$ contains rows of data $\mathbf{d_i = [\ d_{i1} , d_{12} , d_{13} , \dots , d_{im}\ ]}$ which are sequentially arranged.

Function:

There is a query function **Q** which takes in a data warehouse and required tables in the warehouse and gives us a subset of all the rows **S** in the table as a result of the query based on some matching condition. $\mathbf{S = Q(\ DW_i , T_w)}$ **, where** $\mathbf{T_w \subseteq T.}$

Goal and constraints:

If we take a sequential approach in which we collate all the tables of the data warehouse across all locations and then perform the query execution we will incur an additional overhead to collate the tables. The overhead can be represented as $\mathbf{O = n(DW)\ x\ n(T_w)}$, where **n( )** represents the cardinality of the set. The main goal of our solution is to **min(O)** effectively making it a minimization problem. Here **min(O)** represents minimize objective function **O**.

By minimizing the overhead function we can achieve better performance given both the storage constraints and the query runtime constraints. In the next section, we propose our solution in which we will discuss a solution of asynchronous execution for parallel data retrieval which will minimize the overhead function **O**.

Proposed Approach

1. Architecture

Our proposed architecture is composed of the following components.

➔ *Client*

The client provides us an interface in which we input the query. This is a command-line program that acts as a logical entry point to the system. This takes in a SQL query which is then sent to the QueryManager. The client also provides status regarding the execution of the query. The status includes logs from various stages of the query execution. A query can either be in one of the states **[submitted, running, completed/failed]**.

➔ *Query Manager*

The query manager is the main entry point of the backend. It manages all the other processes and tasks which are required for the execution of the query. This also keeps track of the status of the

query and coordinates the query to be moved from one process to another. Query Manager manages the **parser**, **task manager** and the **aggregator** components of the architecture. These components are discussed below.

➔ *Parser*

The user initially provides the query to the client. The client now passes this query to the query manager in the raw form. The query manager now feeds the raw query into the parser. The parser parses the raw query into **two parts of the optimized query**. The idea behind this SQL query optimization is explained below in the optimization section. The main idea is to minimize the extra cost of the data that needs to be moved from the remote local data warehouse to a central warehouse for the processing.

➔ *Task Manager*

For every remote local data warehouse, the query manager spawns up a task manager. Each task manager controls the part query execution related to a single remote local data warehouse. Each task manager takes in the first part of the optimized SQL query provided by the parser and sends it to the SQL processing system in the remote local data warehouse. Every task manager when successfully completed contains the part result set of the part optimized query.

Every instance of the task manager is **asynchronous** in operation. The task managers are spawned parallelly making the connection to the SQL processing system in the remote local data warehouse. The overall running time of the task managers is governed by the slowest performing task manager.

Each task manager during the execution of the query has one of the three states [**submitted, running, completed/failed]**.

➔ *Aggregator*

The aggregator takes in the part result set of all the task managers and again performs the part SQL processing which is identified by the second part of the optimized query from the parser. The aggregator when successfully completes the SQL processing contains the final result of the query which can be stored in the central repository which is much less than the data itself.

➔ *SQL Processing System*

The SQL processing system in our architecture is based on Apache Spark. The data (in the data warehouse) is stored on HDFS for distributed storage and concurrent reading and writing of data blocks.

2. Flow of Execution

We show how the query evaluation process takes place.

- The user enters the query in the client. Let's say: **q = "select b.*,a.CategoryName from Categories a inner join Products b on a.CategoryID = b.CategoryID where b.Discontinued='N' order by b.ProductName "**
- The query manager now takes in the above query and passes it into the parser. The parser generates two parts of the optimized query set.

The parser produces output as: **qs = ["select * from Categories a inner join Products b on a.CategoryID = b.CategoryID where b.Discontinued=1 " , "select b.*,a.CategoryName from**

Categories order by b.ProductName"]. The query set **qs** represents the optimized query set. The idea behind this optimization is discussed in the optimization section.

- The task manager now takes **qs[1]** of the query set and executes it on the SQL processing system and generates a part result. This happens for all the remote local data warehouses and it is a parallel process. The task manager produces a **part result set** corresponding to the data into the warehouse. **[prs_1 prs_2 prs_3 ... prs_m]** where m represents the number of local data warehouses.

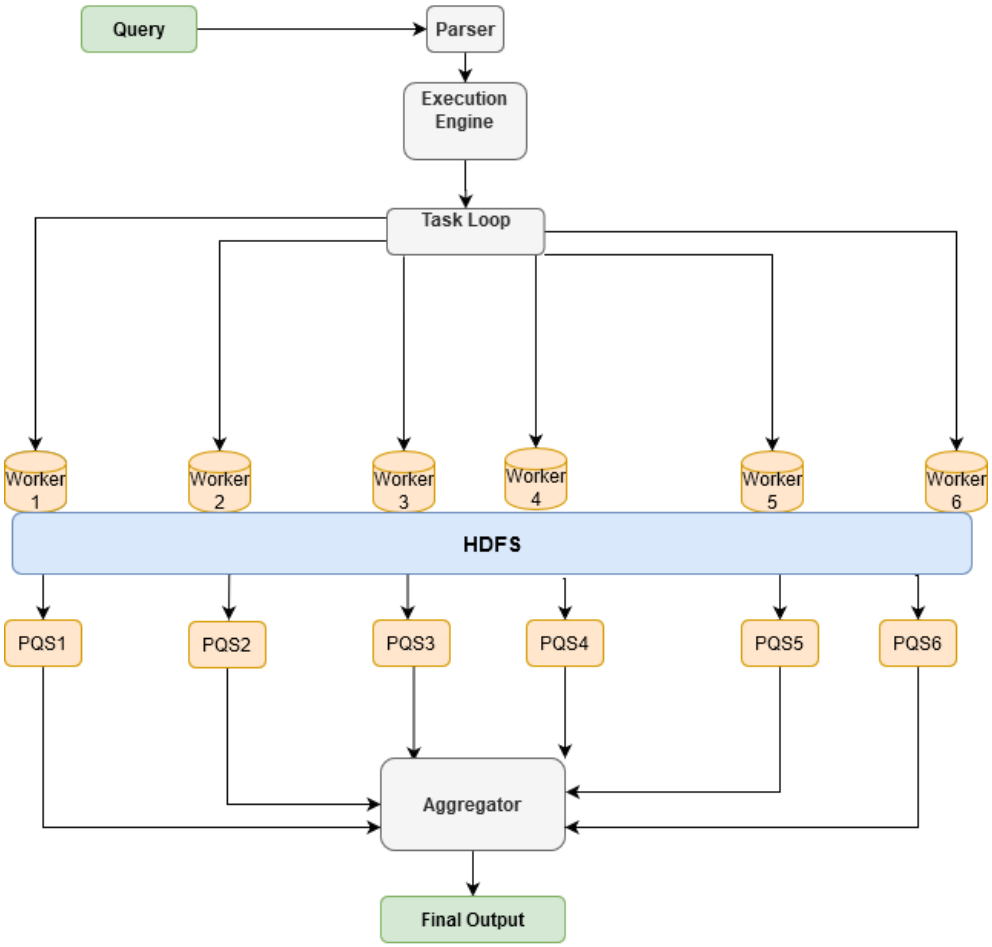

Fig: Architecture Diagram

- The aggregator now combines all the elements in part result set from the above step into a single structure. **prs = combine(prs_1 prs_2 prs_3 ... prs_m)**. The aggregator now takes the

combined result set and processes it with **qs[2]** in its SQL processing system to produce the final result. **fs = agg(prs, qs[2])** where agg is the function which operates **qs[2]** with **prs**.

The entire process can be summarized as follows:

1. Read q from the user.
2. qs = parser**(** q **)**
3. For every DW_i in DW.
    a. $\_prs_i$ = task_manager(DW_i , qs[1])
    :*The above step is parallel with respect to other* DW_i
    b. prs_i = Q(DW_i , qs[1] , T_w)
    Note: Step 3 is asynchronous with respect to the query manager.
4. prs = combine(prs_1 prs_2 prs_3 ... prs_m)
5. fs = agg(prs, qs[2])

3. Optimizations

The following optimizations were done in order to minimize the objective function **O** as described in the previous section.

SQL Query optimization

This optimization function is defined as $\mathbf{OP_1}$.

The parser takes in the raw query and optimizes it to produce two parts. The key idea is to split the raw query efficiently based on the order of execution of the SQL clauses.

The first part of the optimized query set consists of "**select ***", "**join**" and "**where**" clauses. These clauses need the whole dataset to compute the correct result.

The second part of the query set consists of "**select <col>**", "**group by**", "**order by**" and "**having**" clauses. These clauses take the output of where clause and then operate on the dataset to compute the final result.

Therefore we use this idea to create the optimized query set and using the first part we essentially minimize the number of rows that needs to be transferred thereby reducing the overhead. The task manager which performs this step ensures that only the rows which are actually required to compute the final result need to be moved around. Moreover, this operation can be done parallelly as these operations produce disjoint sets.

The second part of the query needs to be performed sequentially as the whole second part relies on all the part result sets from all the local data warehouses.

Asynchronous execution and Parallel processing

The query manager needs to call **M** instances of the task manager in order to execute the first part of the optimized query set where **M** is the number of local data warehouses. While executing one task manager the query manager needs to wait for the result to arrive thereby wasting CPU cycles. So we can execute all the **M** instances asynchronously and effectively

utilize the CPU time. The total time to call these **M** instances sequentially would have been $\mathbf{T_s} = \mathbf{t_1} + \mathbf{t_2} + \mathbf{t_3} + \ldots + \mathbf{t_m}$ where $\mathbf{t_i}$ is time taken by the i-th instance.
But by executing asynchronously we need time $\mathbf{T_a} = \mathbf{max(\ t_1, t_2, t_3, \ldots, t_m\ )}$. We can easily verify that $\mathbf{T_a} <= \mathbf{T_s}$.
Therefore the above two optimization steps add an important factor in optimizing the objective function **O**.

Results and Analysis

The proposed architecture which is built upon asynchronous paradigm is compared with the synchronous execution for the same set of queries. We picked up five sample queries for measuring the performance metrics. These queries consisted of join clauses, querying large tables, with group by and order by clauses.
Here we measured the time required for each query to get executed with optimizations both on and off. We can clearly see that the asynchronous execution gives us better times than the synchronous execution. We have kept the SQL query optimization the same in both cases.

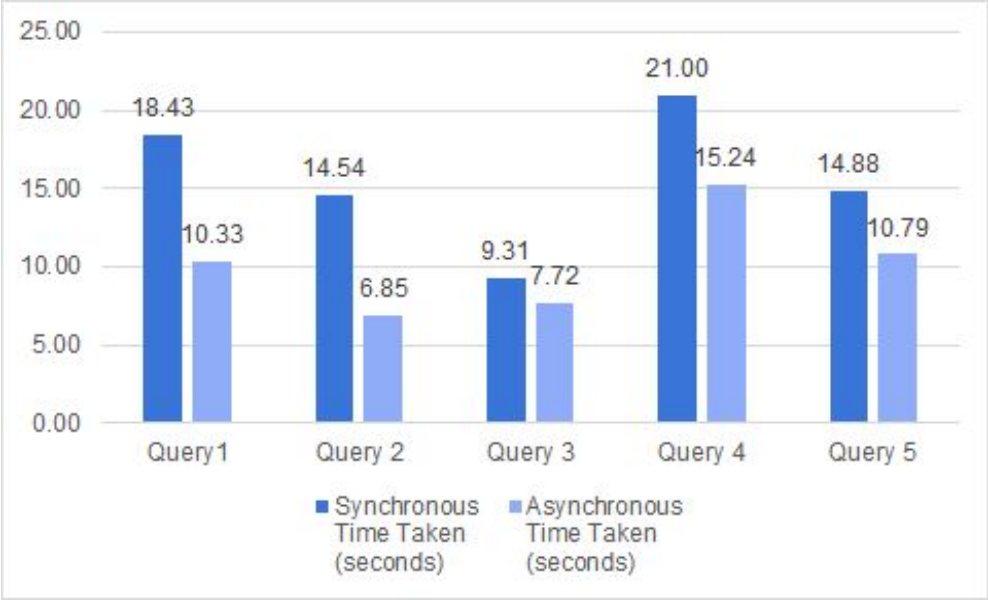

Fig: Comparing Execution Time

We can see a decrease in execution time between **17% - 43%**.
We have also compared the I/O speeds when optimizations are on or off. Here we also see that asynchronous execution clearly outperforms synchronous execution. In the below graph we can see a performance gain between **17% - 27%** in terms of I/O speed.

Fig: Comparison of I/O speed

Conclusion and Future Work

This architecture provides us an efficient platform to execute an SQL query parallelly by leveraging the distributedness of technologically distributed data warehouses. The above approach shows us that an asynchronous approach to parallely execute SQL queries results in much faster query execution than the sequential approach taken in a centralized data warehouse.
This architecture can be further extended to support subqueries and other DDL functions like create, insert, update for improved SQL support.

Project Link

The project can be found at tpsql @Github. All the steps to test the project is mentioned in the project page.

References

[1] Y. Song, X. Chen, C. Li, K. Xuan, J. Wang, G. Liuy "Design and construction of the data warehouse based on Hadoop ecosystem", In 12th Int. Workshop on *Emerging Technologies and Scientific Facilities Controls PCaPAC2018*, Hsinchu, Taiwan, 2018
[2] W.H Inmon, *Building the Data Warehouse*, 3rd edition, Wiley.
[3] Raghu Ramkrishnan, and Johannes Gherke, *Database Management System*, 2nd edition.
[4] Apache Software Foundation, Apache Tajo™: A big data warehouse system on Hadoop (https://tajo.apache.org/)
[5] Amin Yousef Noaman, "Distributed data warehouse architecture and design", A thesis submitted to the Faculty of graduate studies in partial fulfilment of of requirements for the degree of Doctor of Philosophy in Computer science, university of Mantioba, 2000