Shuffle Overhead Analysis for the Layered Data Abstractions

by

Pratik Barhate

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2021 by the
Graduate Supervisory Committee:

Jia Zou, Chair
Ming Zhao
Mohamed Sarwat Abdelghany Aly Elsayed

ARIZONA STATE UNIVERSITY

May 2021

ABSTRACT

Apache Spark is one of the most widely adopted open-source Big Data processing engines. High performance and ease of use for a wide class of users are some of the primary reasons for the wide adoption. Although data partitioning increases the performance of the analytics workload, its application to Apache Spark is very limited due to layered data abstractions. Once data is written to a stable storage system like Hadoop Distributed File System (HDFS), the data locality information is lost, and while reading the data back into Spark's in-memory layer, the reading process is random which incurs shuffle overhead.

This report investigates the use of metadata information that is stored along with the data itself for reducing shuffle overload in the join-based workloads. It explores the Hyperspace library to mitigate the shuffle overhead for Spark SQL applications. The article also introduces the Lachesis system to solve the shuffle overhead problem. The benchmark results show that the persistent partition and co-location techniques can be beneficial for matrix multiplication using SQL (Structured Query Language) operator along with the TPC-H analytical queries benchmark. The study concludes with a discussion about the trade-offs of using integrated stable storage to layered storage abstractions. It also discusses the feasibility of integration of the Machine Learning (ML) inference phase with the SQL operators along with cross-engine compatibility for employing data locality information.

ACKNOWLEDGMENTS

I am grateful to my thesis advisor Dr. Jia Zou, Assistant Professor in the School of Computing, Informatics, and Decision Systems Engineering. Prof. Zou provided me with careful guidance and supported me to complete my thesis work. I learned a lot about doing quality research work under the advisor's supervision while we worked together to get the results for the research paper for the Lachesis system which is now accepted for publication at VLDB 2021. I am also grateful to Dr. Ming Zhao and Dr. Mohamed Sarwat for their time and serving as the committee members for my Master's thesis.

I would like to thank the members of the Cactus Data-Intensive Systems Lab at ASU for helping me to understand various research work. Thanks to inputs from one of my colleagues at the Cactus Lab, Amitabh Das, I was able to understand the Lachesis code base steadily.

I am thankful to my parents who supported me with all the work I have done and are proud of the work I am doing at ASU. I am delighted to have amazing roommates and friends who helped me throughout my semesters at ASU.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

INTRODUCTION

Over the past decade, distributed data processing using frameworks like Hadoop ("Apache Hadoop" 2020), Spark ("Apache Spark" 2021), and Flink ("Apache Flink" 2020) have attracted a lot of attention and are widely adopted for various applications. All of these frameworks are based on the MapReduce paradigm (Dean and Ghemawat 2008) for processing large volumes of data on commodity hardware. Shuffle operation is the major bottleneck for the execution efficiency of such systems. Various expensive tasks like network I/O, disk I/O and serialization-deserialization are required to complete the shuffle operation. There are various conditions and tasks which trigger the shuffle operation in the distributed frameworks. Join is one of the most expensive operation which triggers shuffle operation if the data locality is not as required. To avoid such expensive tasks the datasets involved in the join operation can be co-located.

Conditions and processes to co-locate the datasets are different in each of the frameworks. Hadoop has two components; Hadoop Distributed File System (HDFS) (Shvachko et al. 2010) and MapReduce (Dean and Ghemawat 2008) with the purpose of storage and computation respectively. Such systems can take advantage of the data locality on stable storage. While Spark and Flink have in-memory data abstractions that are used during the executions with no correlation with the stable storage like HDFS or Object Stores. The movement of data between such a system with decoupled storage architecture is inevitable. Although the reading process can be streamlined to mitigate the overhead. Hyperspace library built by the team at Microsoft helps the Spark SQL users to create and manage indexes stored on the same storage system as

that of the data. This report studies the problem of persistent partitions for Apache Spark and explores the Hyperspace library to mitigate the issue for structured data processing.

Lachesis (Zou et al. 2020) is the distributed system that enables users to create automatic persistent partitions over the datasets by learning from the historical workload. The processing and storage data abstractions in Lachesis are integrated, making it possible to manage the underlying data more efficiently. This report will highlight the issues of extending the Lachesis work to Spark ("Apache Spark" 2021) and also show that similar work is required for multi-tenant environments.

Contributions of this thesis report can be summarized as follows:-

1. Survey various ways to reduce the shuffle overhead in Apache Spark. Explore the working of the Hyperspace library to accelerate Spark SQL.
2. Benchmark the Hyperspace library for the performance improvements observed over TPC-H queries and matrix multiplication.
3. Discuss the feasibility and advantages of extending the matrix multiplication over the SQL operators.

The rest of the report is composed as follows: Chapter 2 is the literature survey, Chapter 3 explains the background and details of Apache Spark, Chapter 4 describes the problem statement, in short, Chapter 5 describes the Hyperspace library and how it helps to accelerate the Spark SQL applications, Chapter 6 introduces the Lachesis system, Chapter 7 presents the TPC-H results over two index combinations, along with distributed matrix multiplication results over Spark SQL, Chapter 8 concludes this document by summarizing the results.

Chapter 2

LITERATURE SURVEY

2.1 Co-locating Partitions for Big Data Applications

Arranging the parts of data across the cluster to optimal use for processing is a crucial task for accelerating the Big Data workloads. Hadoop (Shvachko et al. 2010) is one of the widely used systems to store and process large volumes of data. The concept of MapReduce (Dean and Ghemawat 2008) is easy for the users to understand and build complicated applications on top of the underlying data. Although, it is lacking in the indexing and partitioning techniques for efficiently storing the data in such a way that the applications can take advantage of the data placement across the cluster. There have been multiple efforts to optimize the data placement for the Hadoop system. The following paragraphs will discuss some of these approaches to solve the co-location issue.

The data placement strategy for Hadoop was limited to using the load balancer (Olston et al. 2008) to evenly distribute the data among all the worker nodes. Applications processing just the single file can be optimized by simple balancing of the data load. More complex applications sourcing multiple datasets need to perform join operation which needs more sophisticated techniques to be optimized. Such applications can benefit from co-locating the related datasets in Hadoop, which is a difficult task (Jiang et al. 2010) as the applications cannot control the location where the data is to be stored across the HDFS. CoHadoop (Eltabakh et al. 2011) extends the existing Hadoop system by introducing the new property to the files stored over

the HDFS. This new property for a file named `locator` is used to modify the data placement policy for the given file. Each file on the system is assigned to a single `locator` whereas each `locator` can be associated with multiple files. The files sharing the `locator` will have their blocks of data being co-located across the nodes as shown in the figure 1.

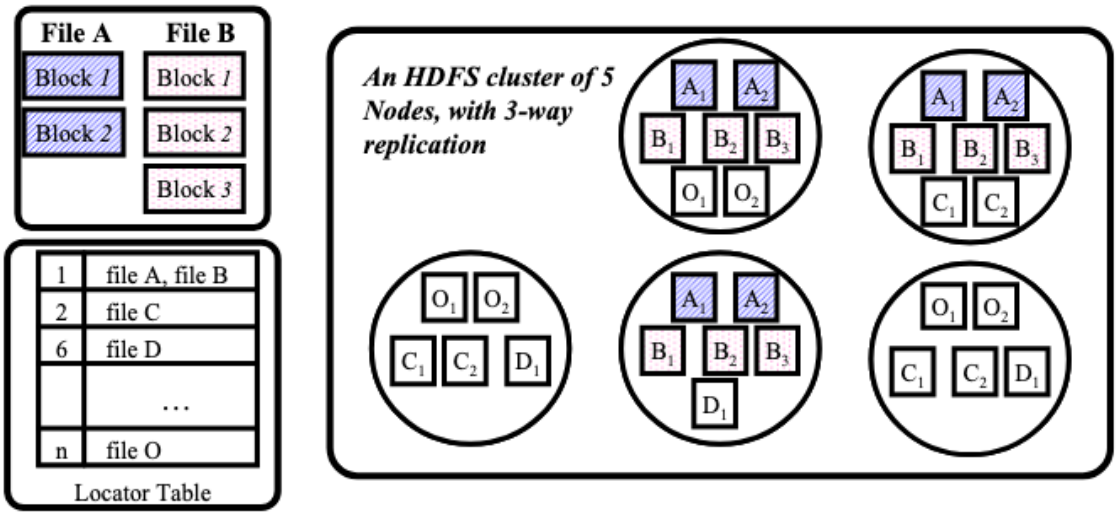

Figure 1: Co-located Datasets A and B in CoHadoop

The table as seen in the figure 1 is the additional data structure to keep track of all the locator information and files on the namenode. The `locator table` is injected into the namenode structure modifying the original implementation. The table is maintained in the main memory when the system is in the running state. The application users can hint the system to colocate datasets by assigning the same locator to both the datasets. The application needs to be developed using the MapReduce interface of the Hadoop system. Other external applications developed using modern frameworks like Apache Spark, Apache Flink, etc. cannot take advantage of such a modified Hadoop system.

Hadoop++ (Dittrich et al. 2010) takes a slightly different approach to improve the performance of the analytics workloads. Instead of extending and altering the existing system, it introduces an indexing concept that can help to reduce the shuffle overhead during the join operation. To create indexes the implementation required the knowledge of the schema for the underlying data. By analyzing the schema and the anticipated workload Hadoop++ creates an additional meta-data file about the key distribution which can be used by the applications during the load time. Schema information of the underlying data is also leveraged by HadoopDB (Abouzeid et al. 2009) to accelerate the analytics workload. Although, it makes use of the Database Management System (DBMS) as the stable storage and alters the Hadoop interfaces. Hadoop++ can have similar performance gains with no change to the existing Hadoop system. The MapReduce jobs which are not aware of the indexes and join conditions can be analyzed (Cafarella and Ré 2010) to be useful through the Hadoop++ module.

2.2 Automated Partitioning

Analyzing numerous workloads in a multi-tenant environment to partition the underlying data is a tedious task. There have been multiple efforts to automate the process of designing the physical layout of the database. This section will describe the various approaches taken for the relational databases in brief.

In a shared-nothing environment for parallel databases, the partition advisor (Rao et al. 2002) in the IBM DB2 system recommends the ideal schemes for the physical layout of the database by analyzing the query optimizer for various workloads. Agrawal, Narasayya, and Yang 2004 take an integrated approach for vertical as well

as horizontal partitioning. It increases the performance along with the manageability of the system. Agrawal, Narasayya, and Yang 2004, says that indexing, partitioning, and materialized views all must be taken into consideration together to reduce the efforts to maintain the databases along with efficient executions. Nehme and Bruno 2011, introduced a proposition for considering the data movement across Massively Parallel Processors (MPP). It decides which data need to be replicated over the ones which need to be distributed across nodes based on some column. AdaptDB (Lu et al. 2017) moves small chunks of data at the time of the execution in such a way that the next execution over the data will need less shuffle, for doing so, it introduces an `hyper-join` algorithm which will only shuffle the data which is not already co-located. Eadon et al. 2008 approach co-locates the two tables identified by the foreign key relationships between the tables. Although, deciding the optimal relation in the case of multiple relational dependencies can be a difficult task. LegoBase (Shaikhha, Klonatos, and Koch 2018) can switch between various intermediate representations and chooses the one most suitable for the workload. It can identify multiple replicas as the candidate distribution for the current execution.

Other than the cost-based models there have been efforts to use modern Machine Learning (ML) techniques. Hilprecht, Binnig, and Roehm 2019 showed that using Deep Reinforcement Learning (DRL) for finding the optimal scheme to maximize the performance of most of the queries in the cloud-based deployments for the Online Analytical Processing (OLAP) workloads. Hilprecht, Binnig, and Röhm 2020 says that their approach can find the ideal candidate even for the various deployment environments provided by the cloud providers. The DRL based techniques have been used to optimize the numerous parameters of the DBMS, for example, QTune (Li et al. 2019) and CBDTune (Zhang et al. 2019). Some of the automatic partitionings

work for the Online Transaction Processing (OLTP) system are Horticulture (Pavlo, Curino, and Zdonik 2012), Schism (Curino et al. 2010), and Sword (Kumar et al. 2014).

2.3 Indexing for Big Data

Indexing is a common technique used across all database technologies to retrieve the required information in a cost-efficient way. There are many available techniques to create indexes over relational data, although to create an index over the complex nested objects we need to maintain the hierarchy and the depth of a predicate. Maier and Stein 1986 introduced an early approach to creating indexes over the complex nested objects. It presents the idea of storing the indexes as the sequence of classpaths. One more challenge in creating and maintaining indexes is the growing need for a dynamic multi-tenant environment. It is difficult to find an optimal layout for the data warehouses for running ad-hoc queries. Sattler, Schallehn, and Geist 2005 propose the use of dynamic configuration during the run time. It suggests that the indexes and partitions created using automated advisory systems may not be very useful for all the queries, especially the ones to get a quick report or short results dynamically. The solution (Sattler, Schallehn, and Geist 2005) is the use of an additional automated system to decide whether to use certain indexes for a query at the runtime. It can also modify some of the index configurations if it is helpful.

The most popular open-source implementation of MapReduce (Dean and Ghemawat 2008), Hadoop (Shvachko et al. 2010), became the standard tool for aggregating large volumes of data over the past decade. There have been numerous efforts to improve the performance of the Hadoop framework for analyzing the growing demand for complex analytical queries. The default implementation is suitable for scanning through a

large amount of data to generated distilled reports. To execute highly selective queries scanning through all of the data will have the worst performance. HAIL (Hadoop Aggressive Indexing Library) (Romero et al. 2015) proposes an idea to create an index over the multiple replicas stored across the HDFS cluster. Each of the replicas can be associated with an indexing strategy optimizing the indexes for various use cases by using simple structures multiple times. Although, the creation of static indexes is not enough for the modern demands for ad-hoc queries. HAIL also takes advantage of the idea of using `adaptive indexing` (Idreos, Kersten, Manegold, et al. 2007) to create indexes incrementally. Such a dual pipeline of indexing policy can help to optimize the selective queries over the high volume data lakes.

A specific analytics use case with certain basic criteria for the applications can be designed to be used for a particular domain or a small team within an organization. The multidimensional model is used to plan the layout of the cube of data into various n-dimensional tables. These multidimensional models are very effective for planning efficient applications for a small analytics use case. Richter et al. 2014 introduces a suggestion to create secondary indexes over the large datasets across the Hadoop cluster to support the active small analytics applications. It also suggests the use of secondary indexes for HBase (George 2011), which is a NoSQL database implemented on top of Hadoop HDFS.

The demand for in-memory distributed computing frameworks like Spark is increasing at an accelerated pace. Although, managing the underlying data across the huge volumes of data lakes is a huge problem. One of the main issues while handling the data lakes in a multi-tenant environment is the data integrity issues. The data used by various teams across the cluster might not be consistent as there are not any cross key checks for maintaining the consistency. Delta Lake (Armbrust et al. 2020)

proposed a solution to the issues by trying to bring the ACID (atomicity, consistency, isolation, durability) properties to the Big Data analytics space. It brings in the properties along with the flexibility of the computation layer. Delta Lake system maintains the information about which objects are part of a Delta table in an ACID manner, using a write-ahead log that is itself stored on the stable storage along with the data itself. The log also contains metadata such as min/max statistics for each data file.

2.4 Storage Elasticity

As the demand for the growth in the overall volume of the datasets might be varying according to the requirements there is a need to address the elasticity for the storage requirements. Taking advantage of the cloud services like Amazon S3, Google Buckets, Azure Object Store, etc. is a cost-effective way to provide the elasticity for the underlying storage requirements. Snowflake (Vuppalapati et al. 2020) introduces two layers of storage. The first one is the ephemeral layer which uses high-speed storage technology for delivering the low latency performance for the queries. The second layer is the disaggregated storage which can be easily extended as per the requirements taking advantage of the cloud-based services. Snowflake parencitevuppalapati2020building also provides the user with the consistency guarantees for which it uses additional services which are consistent like RDBMS to maintain the underlying data over the disaggregated storage. This reduces the efficiency of accessing the data by any other external frameworks Spark, Flink, etc. Delta Lake (Armbrust et al. 2020) takes a different approach to store the log file on the same disaggregated storage along with data. Although there is a disadvantage to the Delta Lake approach too, as for

transactions that perform writes, clients need a way to ensure that only a single writer can create the next log record. The responsibility of maintaining the single writer is pushed to the client writing the data.

Chapter 3

BACKGROUND

Spark is one of the early frameworks to take advantage of the distributed memory. As the data is represented by an in-memory abstraction the iterative algorithms were able to take advantage of such abstractions and significantly reduce the execution time. Although the shuffle operation required to perform various tasks like reduce and join is still similar to the original MapReduce framework and involves local disk I/O making it one of the expensive operations.

3.1 Resilient Distributed Dataset

Resilient Distributed Dataset (RDD) (Zaharia et al. 2012) is the in-memory data abstraction provided by Apache Spark which can be used to apply high-level transformations over the shared dataset across the cluster. The main idea behind RDD is that the user wants to apply coarse transformations to all the sets of data over the shared memory cluster. It abstracted the shared memory states of the data from the user and thus made it easy for the users to implement more complex algorithms. The core development of the framework is built using Scala programming language and provides interfaces for Scala, Java, and Python. Many of the new developments are done in the area of reducing the friction between the Python interface and the Java Virtual Machine (JVM).

One of the main features of the RDD is that it is fault-tolerant. If a node goes down the part of the RDD can be recomputed using the lineage graph represented by

a Directed Acyclic Graph (DAG) data structure. The RDD can be generated deterministically from stable storage like HDFS or from another RDD. The materialization process is only triggered when an `action` is called. An `action` represents a task where a user wants to get the results after all the computations like collect, write or count. All the tasks which fall under the category of `transformations` are used to build the lineage of the RDD mapping the steps to reach the result.

The lazy computation is a double-edged sword, as it can trigger a high number of re-computations in an unreliable environment or the jobs with a very large lineage graph. To avoid these re-computations Spark provides a checkpointing mechanism to store the intermediate RDD onto a local disk. Users have to skillfully use this feature to optimize their use cases.

3.2 Data I/O from Stable Storage

The data represented by the RDD abstraction can be persisted to one of the stable storage using Java serialization. To improve the serialization performance, Kryo serialization library is used by default. The partitioner associated with the RDD is not serialized along with the data and hence the information is lost; once the data moves out of Spark's RDD abstraction. This layered data abstraction is defined by the system the data is currently stored.

As the partitioner information is lost after writing the data to stable storage while reading the data back into the RDD abstraction the Spark's file scan process reads the data into random nodes. In figure 2 above, let's consider there are two datasets with three unique values for a key:- purple, yellow, and green. As the sets are loaded at

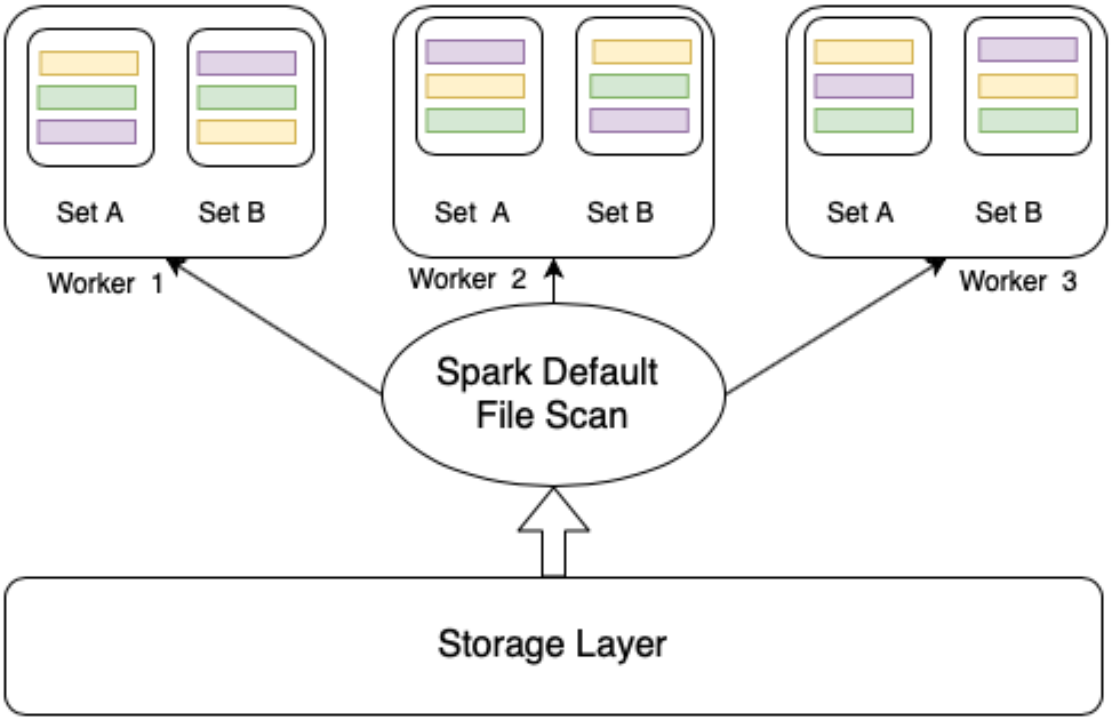

Figure 2: Spark Default File Scan

the worker (node) it will trigger shuffle operation if a join (or similar) transformation is applied over the sets.

3.3 Co-location Condition in Apache Spark

The RDDs can be co-located in Spark's in-memory abstraction when both the RDDs were shuffled as part of a computation graph (DAG) to produce a result and share the partitioner object (Karau and Warren 2017). To meet this condition we have to process the RDDs once to obtain a result that will co-locate the RDDs. The transformation after the first result can take advantage of such co-located sets.

This is perfectly fine for the iterative algorithms as it takes time to process the first iteration and then from the second iteration onwards, the set can be co-located

to avoid the shuffle operation. This is suitable for the algorithms like PageRank (Brin and Page 1998), K-Means, and all the other iterative algorithms.

3.4 Sort-Merge-Join operator

To perform Join operations over the high volume datasets Spark uses the Sort-Merge-Join technique as it scales more efficiently than the Hash-Join. As the volume of the data increases the size of the HashTable to maintain the keys increase significantly adversely affecting the task. High memory requirement might also lead to an Out-Of-Memory (OOM) error. Hence, the report further focuses on the Sort-Merge-Join operator.

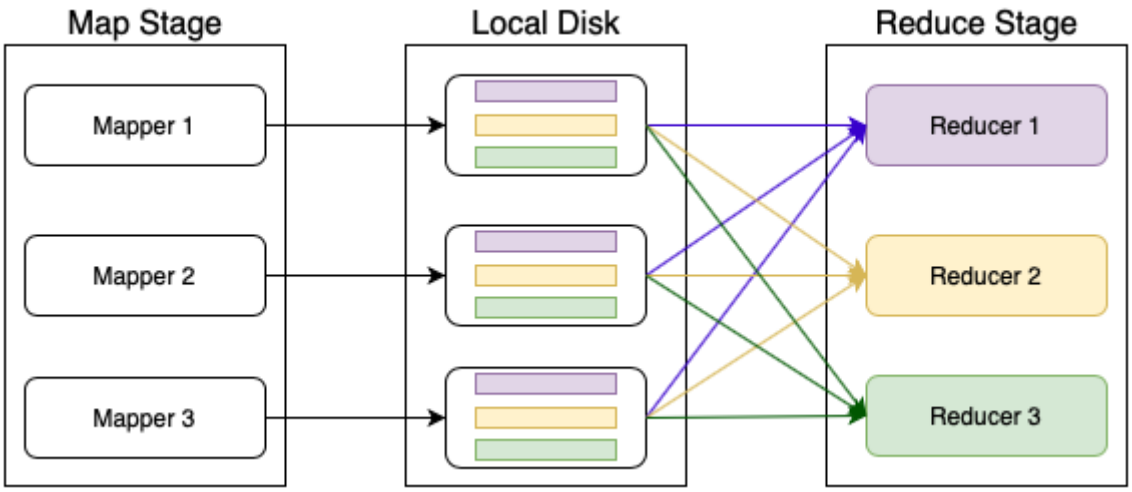

Figure 3: Shuffle Operation for Sort Merge Join

There are two stages to complete the operation, MapStage and ReduceStage (Shen, Zhou, and Singh 2020). The first stage, as shown in the figure 3, writes the shuffle blocks to the local disk in the sorted order of the partition ID. Let's consider the figure above as an example with three partition IDs: purple, yellow, and green. Once

the shuffle blocks are written to the local disk of the executor the reduce stage will start reading the corresponding blocks by querying the application Driver process. Such expensive tasks can be avoided if the datasets are co-located in the first place. This document elaborates more in the next section and Chapter 3.

3.5 Spark SQL

The Spark SQL component is implemented on top of the core RDD abstraction and closes the gap between the procedural RDD interface and the declarative SQL queries. It introduces a structured interface called DataFrame over the underlying RDD interface. DataFrame is a typed dataset, which means it has predefined data types that are used to represent the data in a column-oriented fashion. It can be considered as an in-memory table similar to a data frame in the scripting languages like R and Python.

DataFrame interface introduces some mechanism to mitigate the random read problem. There are two ways to reduce the shuffle overhead:- 1) Using partitionBy interface or 2) Using the bucketBy and sortBy interface along with the Apache Hive (Karau and Warren 2017) as the stable storage.

The first approach helps in grouping the dataset together over a key. Hence, the file scan process will load the data for the dataset grouped by the key together. This still leaves a gap in co-location as the two datasets can have their respective partitions on different worker nodes as shown in figure 4.

First Approach

```
df
.write
.partitionBy(
    "key_column_name"
)
.parquet("dataset_location")
```

Second Approach

```
df
.write
.bucketBy(
    "key_column_name",
    ${number_of_unique_values}
)
.sortBy("key_column_name")
.table("table_name")
```

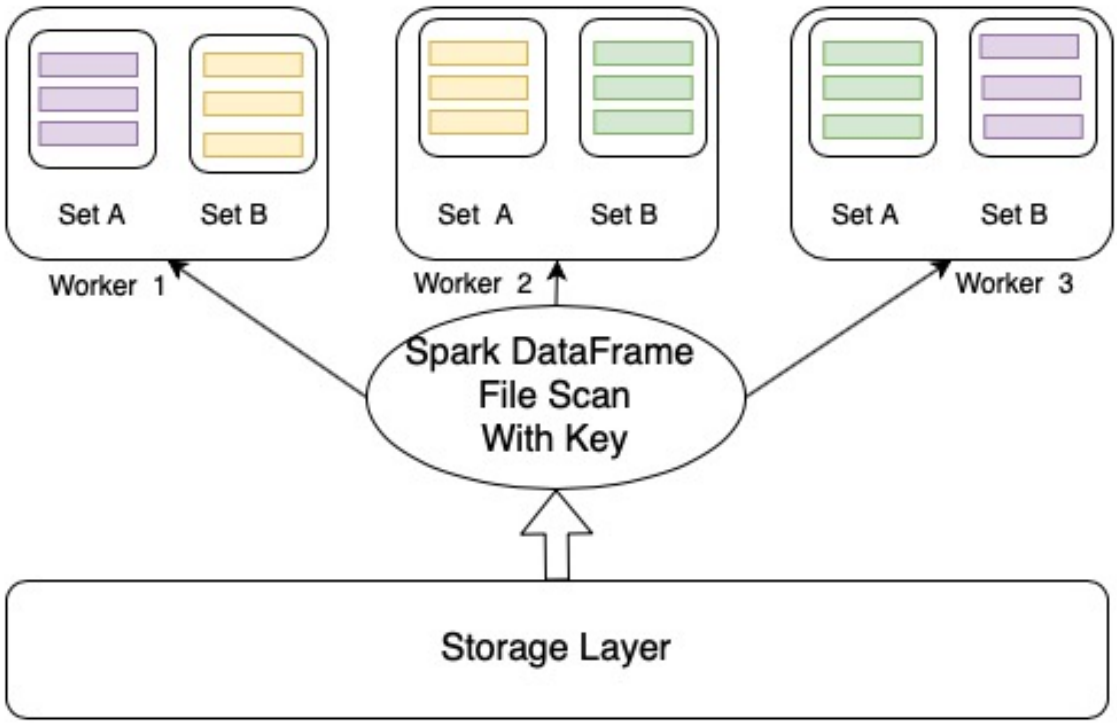

Figure 4: File Scan for a dataset using partitionBy Interface

The second approach can solve the problem with one caveat that it needs to create

the number of buckets equal to the number of unique values over the key used to create the bucket and join the datasets. With the increase in the number of uniques values, the number of buckets will grow and eventually push the system to too many small files problem. Hence, using these approaches are not suitable for all the use case, specifically with large number of unique values in the `key` column.

3.5.1 Catalyst Optimizer

Spark SQL (Armbrust et al. 2015) queries are optimized based on the optimization rules developed using the Scala programming language feature quasiquotes. The language features help in building the Abstract Syntax Tree(AST) and generating the final code for execution.

First, the unresolved plan is analyzed for syntactical errors like resolving the column names and checking the data types at each step. The plan is further optimized based on rules defined within the Catalyst. The most common rule is the filter push-down which helps in the reading as much less data as possible from the stable storage. The optimization at the physical planning step is limited to selecting the broadcast join operator if one the relation is small enough to fit in the memory of a single node. Hyperspace library further extends this step for optimizing the Sort-Merge-Join operator, the details are discussed in the next chapter.

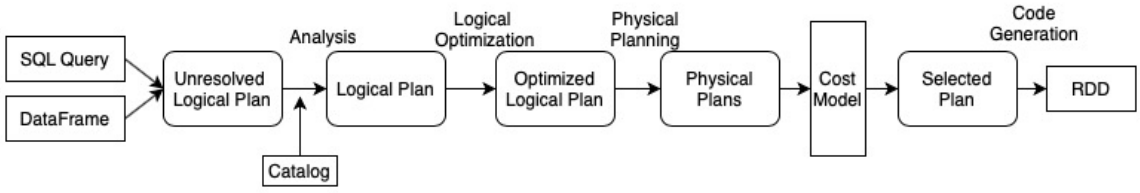

Figure 5: Catalyst Optimization Steps

Chapter 4

PROBLEM: PARTITIONING FOR LAYERED DATA ABSTRACTION

Partitions and data locality of the in-memory abstractions of the modern distributed computing systems are limited to a single running application. As shown in the figure 6 the stable storage is disconnected from the interfaces available to develop application logic.

To reduce the gap between such opaque storage architecture, we have to leverage an external middleware or a meta-data server, which can help in preserving the locality within the system's in-memory abstraction.

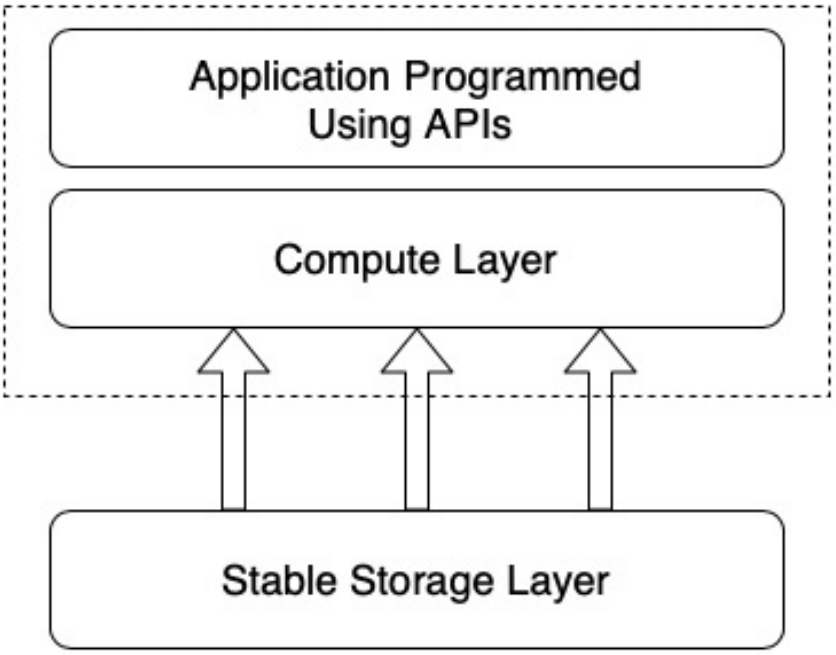

Figure 6: Layered Data Abstractions

Chapter 5

HYPERSPACE

Hyperspace ("Hyperspace Repository" 2021) is the indexing subsystem for Apache Spark developed by Microsoft and open-sourced in June 2020. The users of the Spark SQL (Armbrust et al. 2015) can create indexes over their dataset on the stable storage. These index meta-data can potentially be used to accelerate the workload defined using the SQL interface of Apache Spark. It can be used to accelerate the queries with below mentioned scenarios:-

1. Query which has filter condition over a column with high selectivity.
2. Query with massive shuffle operation for the join condition.

This chapter describes the details about the index structure and the step where the Hyperspace code optimizes the DAG.

5.1 Directories and Index Structure

To create an index over the dataset we need to have a DataFrame reference of the set with the location on the stable storage. The reference and the index configuration are taken as an input and the index structure is created at the location pointed by the Spark configuration `spark.sql.warehouse.dir`. Within the location pointed to the configuration a directory name indexes is created. Within the indexes directory, each index structure is stored under the directory given by its name. Within each index structure, there is a file named `_hyperspace_log` which stores various information like the path of the dataset, the location of all the part files, the type of stable storage used,

modified time, etc. Then there are various directories for each version of the index. Within these directories, the meta-data information is maintained in the `parquet` files.

```
Index Directory Hierarchy on a Stable Storage
${spark.sql.warehouse.dir}
 |--indexes
     |--index_name_1
         |--_hyperspace_log
         |--v__=0
         |--v__=1
     |--index_name_2
         |--_hyperspace_log
         |--v__=0
     |--index_name_3
         |--_hyperspace_log
         |--v__=0
         |--v__=1
         |--v__=2
```

The `_hyperspace_log` stores all the information in the `JSON` formatted file. It is used to decide which index directory to be used. As all the versions of the index over a dataset are used users can specify a particular version for each execution. The next section will describe more about managing the index directories.

The index in this scenario can also be termed as just meta-data, as it does not store the information about the key value and the files in which it is available. Let's

consider the `orders` table from TPC-H data, below is the schema table schema of the table.

```
TPC-H orders table schema
root
 |-- O_ORDERKEY: integer (nullable = true)
 |-- O_CUSTKEY: integer (nullable = true)
 |-- O_ORDERSTATUS: string (nullable = true)
 |-- O_TOTALPRICE: double (nullable = true)
 |-- O_ORDERDATE: date (nullable = true)
 |-- O_ORDERPRIORITY: string (nullable = true)
 |-- O_CLERK: string (nullable = true)
 |-- O_SHIPPRIORITY: integer (nullable = true)
 |-- O_COMMENT: string (nullable = true)
```

For the given schema above if we create an index with the indexing key column as `O_ORDERKEY` and the project columns be `O_CUSTKEY` and `O_ORDERDATE`. The schema of the structure within the corresponding index directory, let's say `v__=0`, is a table structure with the data being sorted by the key column which is the first column of the meta-data structure as shown below.

```
Index schema for orders table
root
 |-- O_ORDERKEY: integer (nullable = true)
 |-- O_CUSTKEY: integer (nullable = true)
 |-- O_ORDERDATE: date (nullable = true)
```

5.2 Creating and Managing Indexes

To create an index over a dataset we need a DataFrame reference to the data on the stable storage. This reference is taken as input to create the index structure and store the location referencing the data. Hence, if any source matching the location of an index is found the library will optimize the computation graph to minimize the shuffle. Below is the example code in Scala to create the index over the `orders` table from the TPC-H benchmark dataset. It will create the directories and the data schema as mentioned in the above section.

```
import com.microsoft.hyperspace._
import com.microsoft.hyperspace.index.Index
import com.microsoft.hyperspace.index.IndexConfig

val ordersDf = spark.read.parquet("/data_location/orders_table")
val hyperspace: Hyperspace = Hyperspace()

val ordersIndexConfig: IndexConfig = IndexConfig(
        "index_name", Seq("O_ORDERKEY"),
        Seq("O_CUSTKEY", "O_ORDERDATE")
    )
hyperspace.createIndex(ordersDf, ordersIndexConfig)
```

Once the index directory is created it can be managed by the various interface provided by the library. The user can list all the available indexes under the currently configured warehouse directory. Indexes can be marked as deleted, this would not physically remove the index but render it inactive and won't be used to optimize the

DAGs. The deleted indexes can be restored to be useful again. This gives the users the flexibility to avoid causing any unnecessary error from the existing indexes or can use some and avoid using the rest of the available information.

Once the index directory is created it can be managed by the various interface provided by the library. The user can `list` all the available indexes under the currently configured warehouse directory. Indexes can be marked as `deleted`, this would not physically remove the index but render it inactive and won't be used to optimize the DAGs. The `deleted` indexes can be `restored` to be useful again. This gives the users the flexibility to avoid causing any unnecessary error from the existing indexes or can use some and avoid using the rest of the available information. `Vacuum` is the interface that can be used to completely remove the created index structure from the stable storage. While creating the Spark session the interfaces `disableHyperspace` or `enableHyperspace` can be used to enable or disable the use of the Hyperspace indexes as shown below ("Microsoft Hyperspace" 2021).

```
// Disable Hyperspace
val spark =  SparkSession.builder().config(conf).disableHyperspace
// Enable Hyperspace
val spark =  SparkSession.builder().config(conf).enableHyperspace
```

5.3 Hyperspace File Scan

The Hyperspace library reads the optimized logical plan from the Catalyst and checks for the Sort-Merge-Join operator. If the meta-data for the source datasets is found the external optimizer tries to identify if the key column used to perform the operation is the same as that of the sorted key in the index structure then it optimizes

the physical planning step for the Sort-Merge-Join operation. As shown in the figure 7, the library alters the physical plan for a computation.

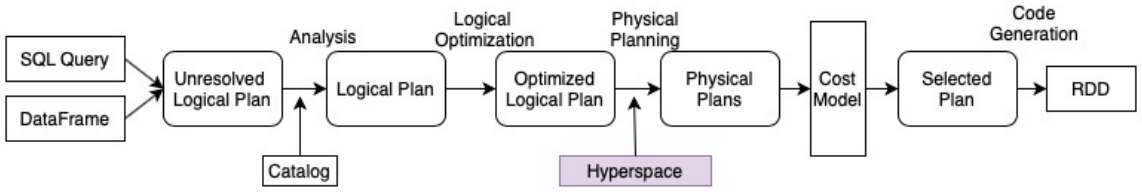

Figure 7: Physical Plan extension

When the hyperspace is enabled for a given spark session, the library identifies the sorted key values from the index structure. The key is then used to dispatch the rows to the required nodes. This initial load of the data from stable storage to the in-memory abstraction is inevitable and cannot be avoided in the case of layered data abstractions. This will describe the Lachesis system in brief, in the next chapter, which has integrated storage and can avoid the initial load shuffle too.

If we consider the same dataset with three unique values; purple, yellow, and green, as shown in the figure above. The scan process is completed in such a way that all the parts of the sets with the same key value from both the datasets are co-located, as shown in the figure 8. Hence, the Hyperspace library is improving the file scan process for Apache Spark.

As the meta-data is available on the stable storage along with the data itself there is no need to manage a meta-data server which can become a bottleneck in the multi-tenant environment. It also opens up the idea for cross system optimizations, there can be a library or a middleware which can make the scanning process for the various in-memory abstractions efficient making the meta-data useful for various

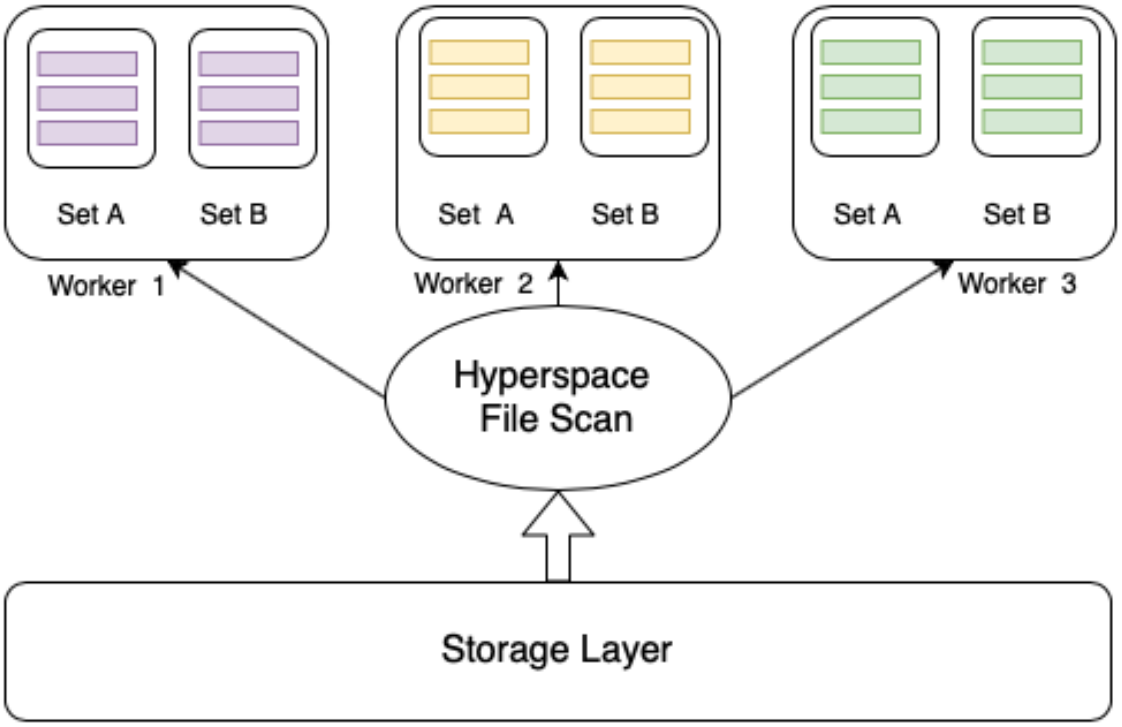

Figure 8: Hyperspace File Scan Process

computing framework. This can give the teams at the organization flexibility to choose a framework for an application while working with a centralized data lake.

Chapter 6

LACHESIS

To solve the complexity involved in the distributed analytics workload, Lachesis (Zou et al. 2020) introduces an automatic partitioning system. The stable storage of the system is integrated with the computation layer. The system is extended from the Pangea (Zou, Iyengar, and Jermaine 2018) system which introduces the idea of pushing the computation to the storage layer and handling various types of data in a unique way. The following chapter describes the overview of the Lachesis system.

6.1 Integrate Stable Storage

As explained in the previous chapters the shuffle during the initial load of the data from the stable storage is fixed and can not be avoided if the stable storage is abstracted. Lachesis is a User Defined Functions (UDF) centric analytics system with integrated storage. The persistent partitions can be identified by the optimizer to check if it is possible to avoid the shuffle operation.

Let's consider the same two datasets as considered before with three unique values for the key column; purple, yellow, and green. If both the datasets under the observation are co-located on the stable storage the join operation can be performed on the same nodes circumventing the shuffle during the join as well as the opening loading process. Figure 9 illustrates the difference in the data reading operation from the stable storage.

The data from the stable storage is loaded into a buffer pool fully managed by the

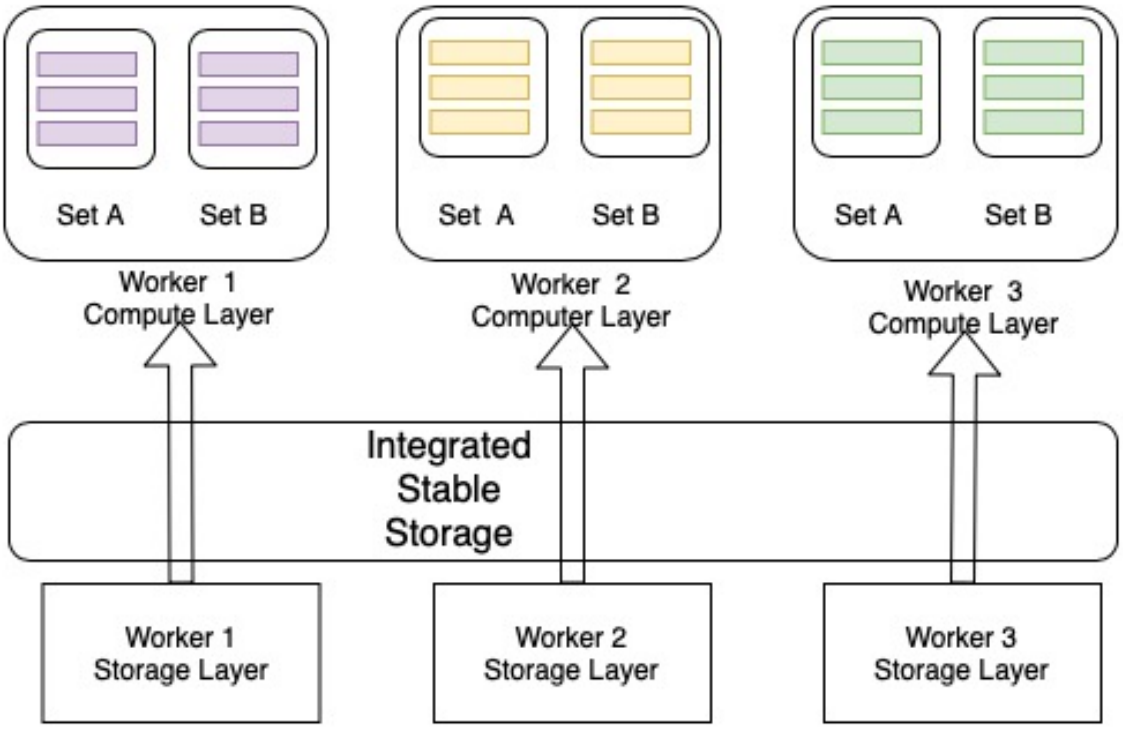

Figure 9: File scanning phase in Lachesis

system (Zou et al. 2020). All the main memory available to the worker nodes and the driver (master node) is managed to have fine-grained control to optimize the memory usage. The worker threads access the data from the buffer pool to process the specific rows. The data files on the stable storage are represented by a sequence of pages. Page iterator can be used by a worker to scan a subset of all of the pages on that node. The applications can be iterate over the data using the set iterator interface. To improve the efficiency further the dataset can be replicated across the cluster with different properties associated with each replica. The replica most suitable for a task is used for execution.

6.2 Partitioning the Data

The process to identify the ideal partitioner candidate is a two-step process. The first step is to enumerate all the possible sub-graphs with the `scan` node and unique leaf node. The unique `leaf` node is defined as the node that connects to a `pair` node (Zou et al. 2020) after the `join` node in a path starting with a `scan` node.

To find an ideal candidate among the list of partition candidates the Lachesis system takes a Deep Reinforcement Learning (DRL) based approach, adopting the actor-critic network (Zou et al. 2020). The feature vector used to define the query comprises of the following set of information; i) `frequency` defines the number of time the graph is executed, ii) `distance` defines the average gap between the recent executions, iii) `recency` defines the latest execution timestamp, iv) `complexity` defines the longest path within the subgraph, v) `selectivity` defines the amount data which will be shuffled in case the candidate is not selected, vi) `key_distribution` defines the number of unique values and the key distribution over the partitioner candidate, vii) `num_copartitioned` defines the number of the sets other than the one under the observation that will be co-located with the current set, and viii) `size_copartitioned` defines the volume of the data which will be co-located including the other sets which will be benefited by choosing the candidate key. The reward for choosing a partitioner candidate is the performance improvement of an application consuming the set under the observation.

6.3 Extending Lachesis

Lachesis benefits from two main components, 1) Integrated Storage, and 2) Automatic creation of persistent partitions. It might be feasible to extend the automatic identification for the partitioner candidate for a dataset in a multi-tenant environment for the frameworks with layered data abstraction using the help of the libraries like Hyperspace. The following chapter presents the benchmark results for analytical queries and matrix multiplication using Hyperspace along with the Spark SQL interface. The results show that manually creating the ideal candidate might not be always possible.

Chapter 7

BENCHMARK RESULTS

7.1 Environment Setup

The following executions were performed using the node with 4 vCPU cores and 30.5 GB RAM each, connected with 10Gbps ethernet links. The `driver` (`master`) node has 192 GB of disk space with 576 IOPS, the worker nodes have 92 GB of disk space with 276 IOPS. In total 5 worker nodes were used for the executions. The software versions used are as follows: 1) Apache Spark - `v2.4.7`, 2) Microsoft Hyperspace - `v0.4.0`, and 3) Apache Hadoop - `v2.7.7`

All the query executions were completed using the Scala API for Spark SQL. One executor instance was running on each worker node with 4 CPU cores and 28 GB executor memory, hence in total, there were 5 executor instances. The executions were completed in the client mode of the execution, which means that the driver was running on the master node.

Scale factor 100 was used for running TPC-H queries. The tables were stored on HDFS in the `parquet` file format. Each table had two copies one over which the Hyperspace index structure was generated and another used to generate default Spark execution results.

For matrix multiplication, random matrices of the given dimensions were generated using the SciPy libraries `sparse.random` class. The results graph shows the dimensions of the left matrix, the dimension of the right matrix is the same as that of the transpose of the left, although it is not an actual transpose.

7.2 TPC-H Results

The evaluations were conducted using two different combinations of the indexing keys. For each table, there can be multiple candidate keys, and identifying the correct combination which might be suitable for all the queries is a challenging task. The following sub-sections will discuss the results with the indexes created using the Hyperspace library while keeping a custom query as the target and query 17 of the TPC-H queries as the target.

7.2.1 Custom Query as the Target

The custom query performs the join operation over the two largest tables in the TPC-H benchmark data, lineitems, and orders. It is a simple query with an aggregation function, making it easier to tune the rest of the tables for the remaining queries. Below is the custom query represented in SQL.

```
SELECT o_custom, SUM(l_discount)
FROM lineitem INNER JOIN orders
ON l_orderkey = o_orderkey
GROUP BY o_custkey;
```

For the above SQL query equivalent, Scala code was used to complete the evaluations. The table 1 defines the indexing key and the projection columns used to create the index structure over the parquet data for the given table. The index overhead is the time in seconds required to create the meta-data structure by the Hyperspace library. The tables `lineitem` and `orders` were indexed over the keys used to perform the join operation.

Table Name	Index Column	Project Columns	Index Overhead (in seconds)
lineitem	L_ORDERKEY	L_DISCOUNT, L_SHIPDATE	144
orders	O_ORDERKEY	O_CUSTKEY, O_ORDERDATE~	60
customer	C_CUSTKEY	C_NATIONKEY	9
part	P_PARTKEY	P_BRAND	19
partsupp	PS_SUPPKEY	PS_PARTKEY	37
supplier	S_SUPPKEY	S_NATIONKEY	7
nation	N/A	N/A	N/A
region	N/A	N/A	N/A

Table 1: Index Combinations Targeted at Custom Query

The results in the figure 10 show that the custom query executed with high-performance gains. Although, the majority of the other queries were not able to take advantage of the specific combination. The total time taken for all the queries with default Spark to execute is `1545 seconds` and the total time for all queries when the Hyperspace library is used, `1436 seconds`. The total time saved is `109 seconds`.

7.2.2 Query 17 as the Target

Query 17 runs a join operation between a very large table `lineitem` and a considerably small table `part`. The query has very high selectivity as it considers a particular container for a given brand name and calculates the annual average for 7-year data. The index combinations for this execution are defined by table 2. Below is the query 17 represented in SQL.

```
SELECT SUM(l_extendedprice) / 7.0 AS avg_yearly
FROM lineitem, part
WHERE p_partkey = l_partkey
```

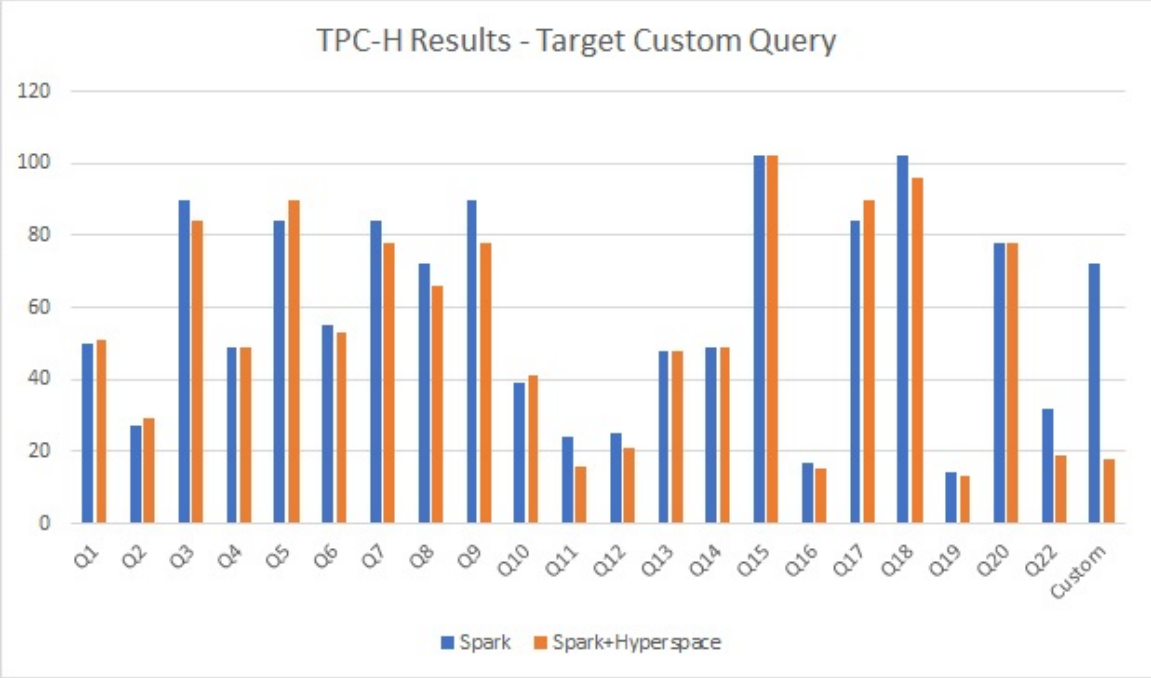

Figure 10: TPC-H Results with Custom query as the target

```
and p_brand = 'Brand#23'
and p_container = 'MED BOX'
and l_quantity < ( SELECT 0.2 * AVG(l_quantity)
                   FROM lineitem
                   WHERE l_partkey = p_partkey
                   );
```

The results in the figure 11 show that query 17 executed with high-performance gains and similar to the first execution for custom query all other queries did not benefit much. The total time taken for all the queries with default Spark to execute is `1557 seconds` and the total time for all queries when the Hyperspace library is used, `1434 seconds`. The total time saved is `123 seconds`.

From the results, we can conclude that there is a need to build an automatic

Table Name	Index Column	Project Columns	Index Overhead (in seconds)
lineitem	L_PARTKEY	L_QUANTITY, L_SHIPDATE	156
orders	O_ORDERDATE	O_CUSTKEY, O_ORDERKEY	38
customer	C_CUSTKEY	C_NATIONKEY	13
part	P_PARTKEY	P_BRAND, P_CONTAINER	9
partsupp	PS_SUPPKEY	S_SUPPKEY	38
supplier	S_SUPPKEY	S_NATIONKEY	7
nation	N/A	N/A	N/A
region	R_NAME	N/A	0.8

Table 2: Index Combinations Targeted at Query 17

recommendation system targeted at reducing the total time saved for all the queries similar to that of the Lachesis (Zou et al. 2020) system.

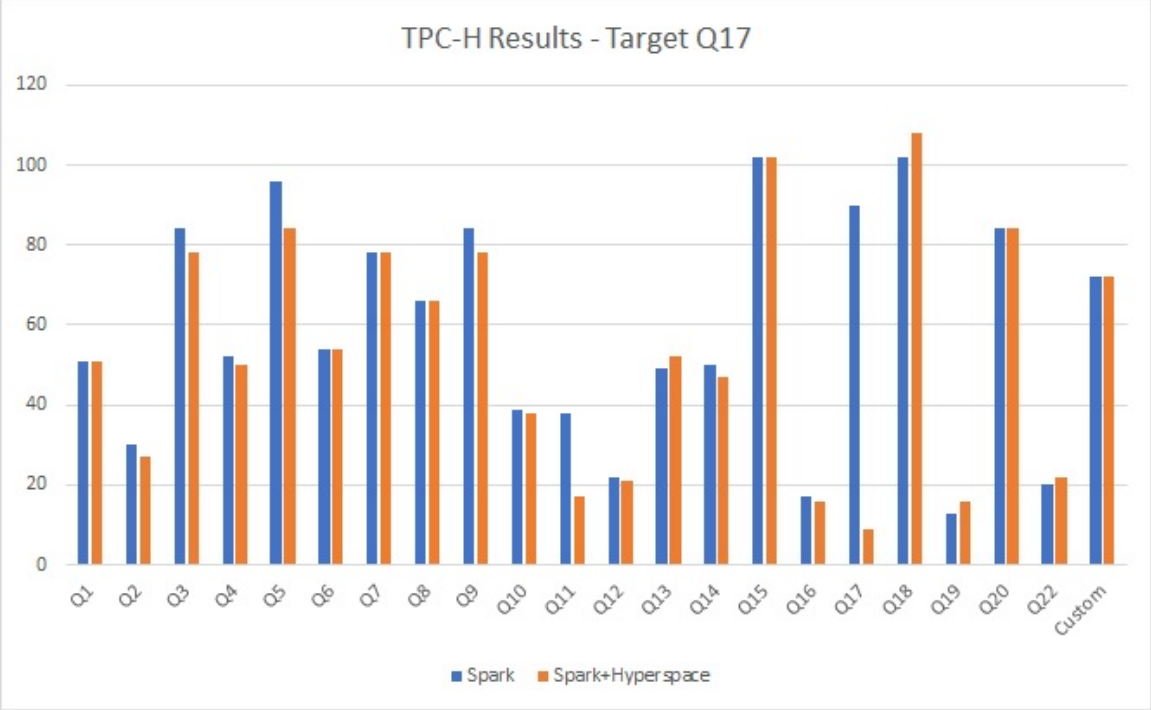

Figure 11: TPC-H Results with Custom query as the target

7.3 Distributed Matrix Multiplication

For matrix multiplication, the matrices were first generated randomly in a text file. These text files are then converted into the parquet formatted files with three columns `rowID`, `columnID`, and `value`. The matrices are represented in tabular format to be useful through the SQL interface. Below is the matrix multiplication represented in SQL.

```
SELECT g.rowID, g.columnID, SUM(m_op)
FROM (SELECT l.rowID, r.columnID, (l.value * r.value) AS m_op
      FROM left AS l INNER JOIN right AS r
      ON l.columnID = r.rowID) AS g
GROUP BY g.rowID, g.columnID;
```

If the matrices are co-located in a way, such that the rows of the left-hand side matrix in the operation are co-located with the columns of the right-hand side matrix within the operation, then the shuffle triggered for the element-wise multiplication step can be avoided. As shown in the figure 12, we will able to observe only two main stages in a distributed environment. It reflects similarly in the results where we can observe in total three stages when the matrices are not co-located as compared to only two stages when the matrices are co-located during the computation.

The following document presents the results for two executions, first with both the matrices being dense; second, both the matrices are sparse. The matrix dimensions shown in the figures are of the left matrix, the dimension for the right matrix is similar to that of a transpose. If the dimension for a left matrix is `p x q` then the dimensions for the right matrix is `q x p`. The sparsity of the sparse matrices is `5%`, taken from the

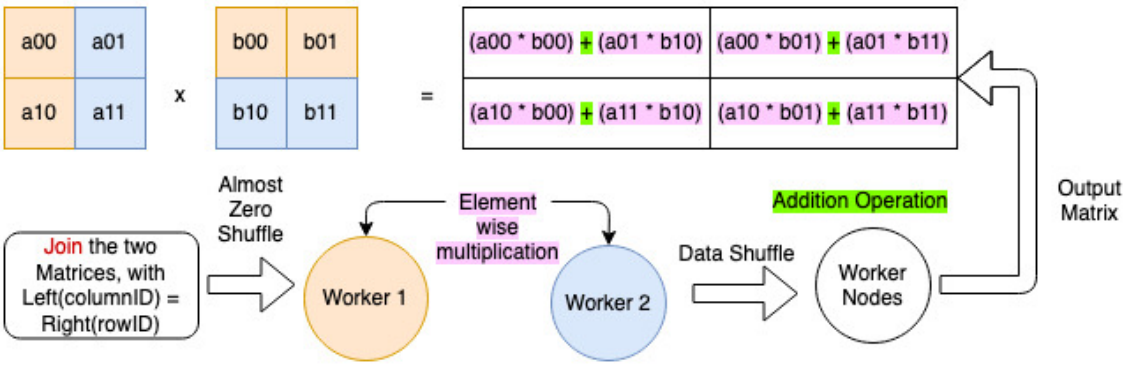

Figure 12: Distributed Matrix Multiplication

Movielens ("GroupLens Research" 2021) dataset collected by the GroupLens Research group.

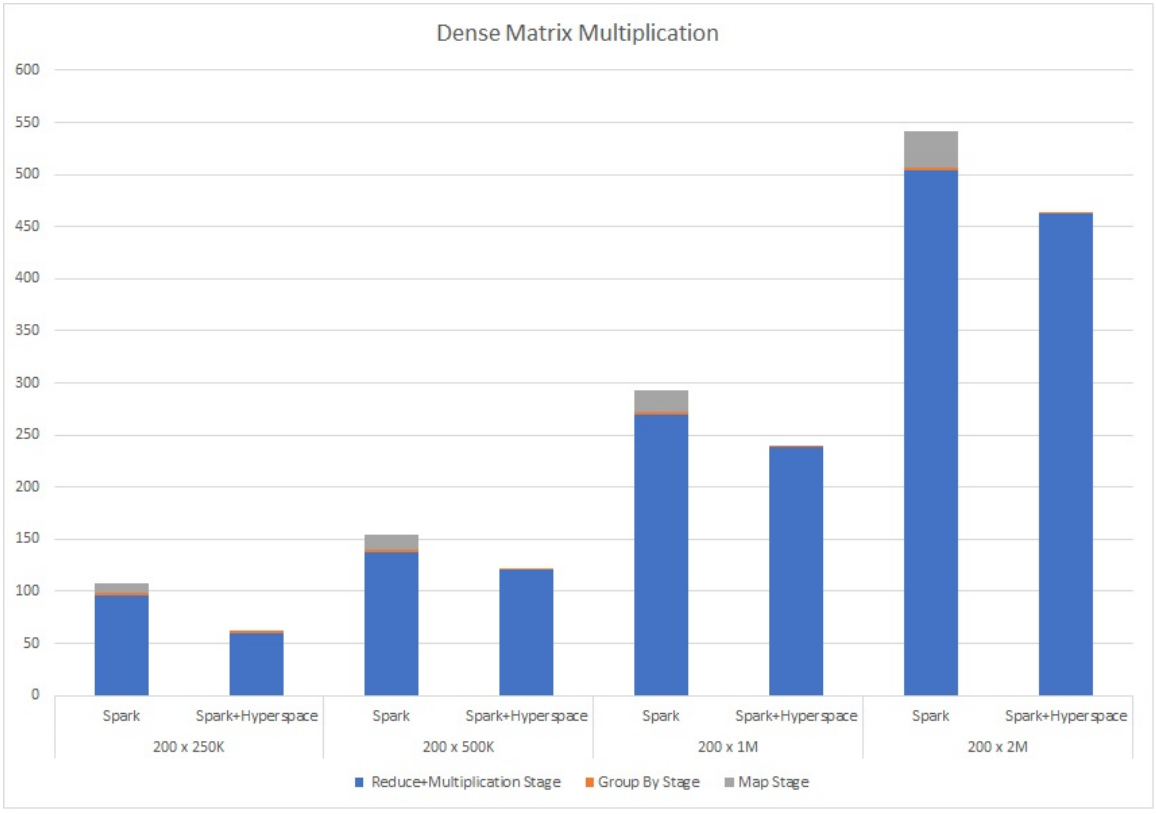

Figure 13: Dense Matrix Multiplication

Figure 13 shows the results for the dense matrix multiplication over the SQL interface. For dense matrices, there is approximately 20% gain observed. Whereas for the sparse matrices, shown in figure 15, the gains are approximately 45%. Most of the recommendations dataset have the used sparsity level, hence if the inference phase for recommendation algorithms is implemented on top of the SQL operators it can benefit from such acceleration methods and reduce the latency for serving the ML model.

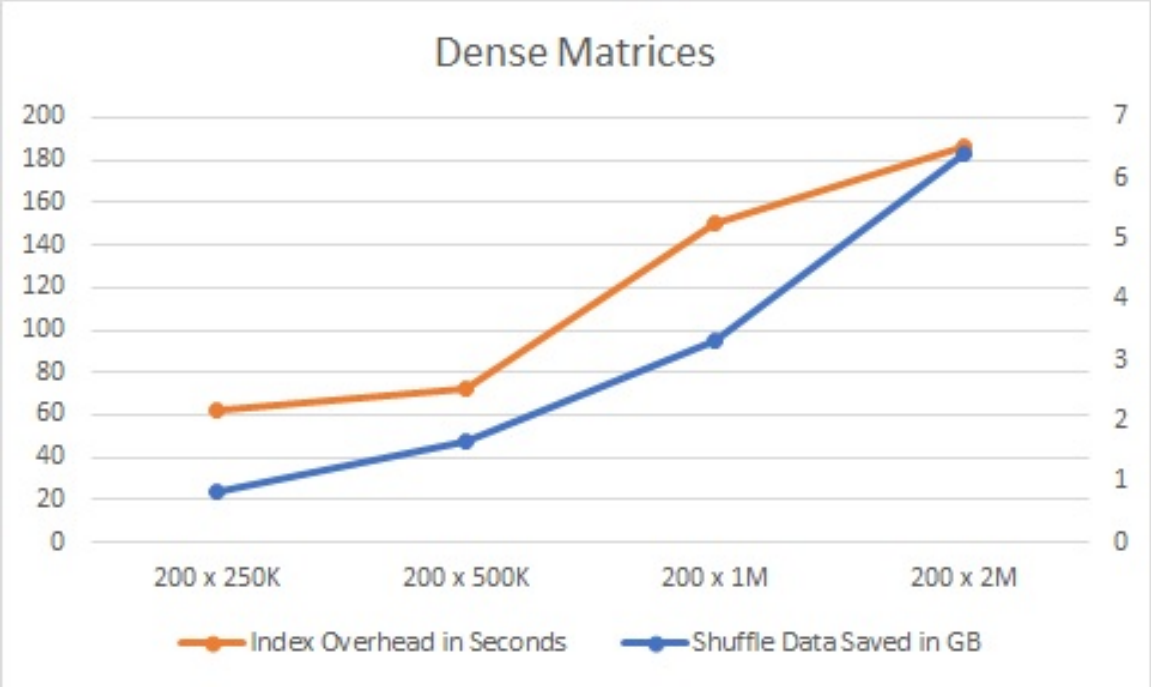

Figure 14: Shuffled data saved compared to the Index Overhead

The overhead to create an index structure increases in parallel with the amount of data volume being saved from the shuffle. Once the index structure is created the overhead will diminish as the calls to the multiplication operation increase. The figures 14 and 16 show the comparison of the overhead to create the index structure to that of the amount of data moved over the network is saved during the multiplication operation for dense and sparse matrices respectively.

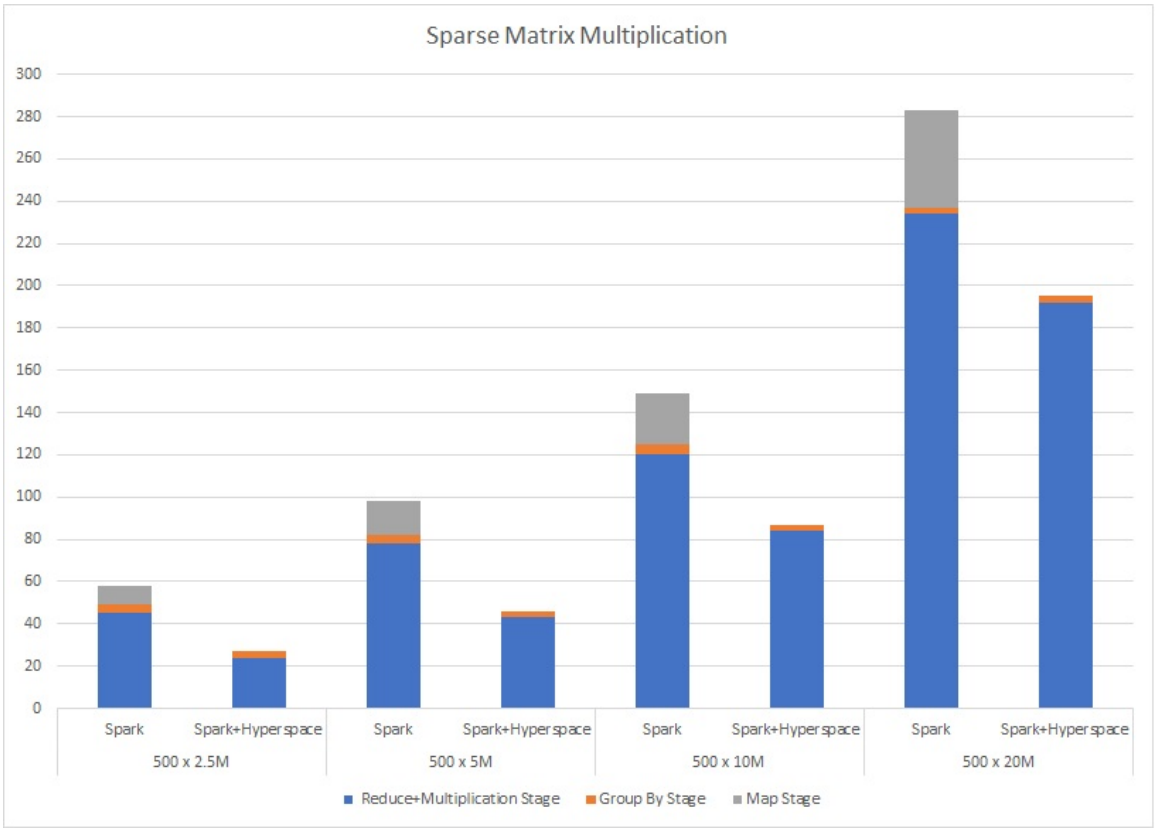

Figure 15: Sparse Matrix Multiplication

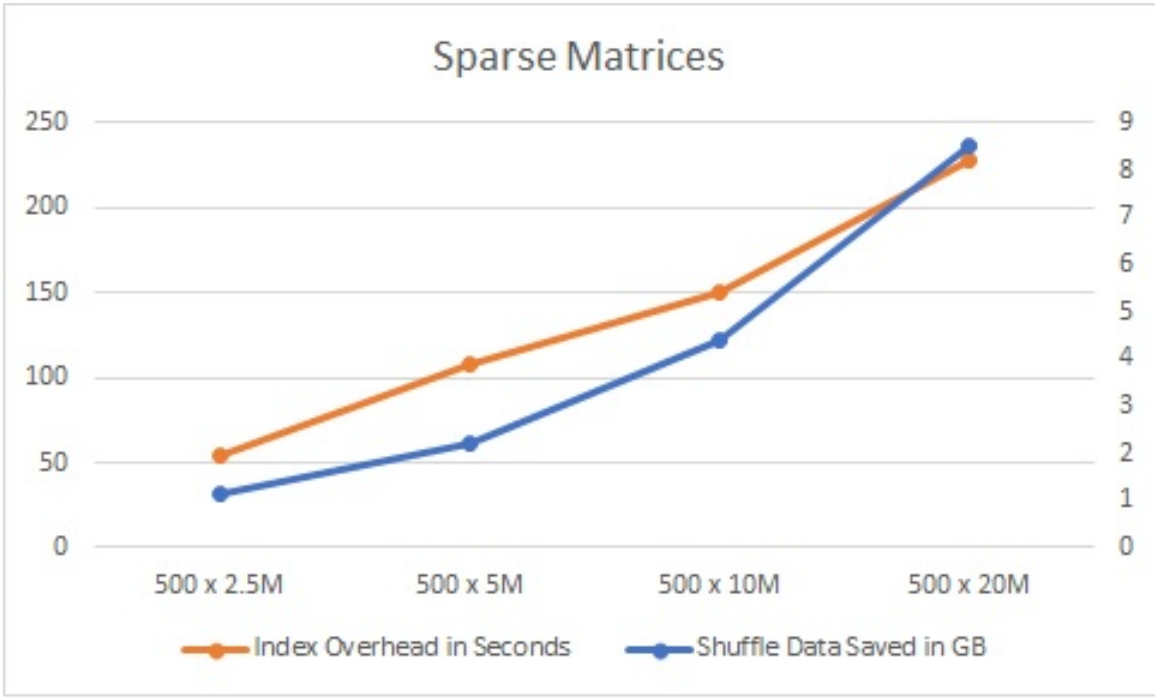

Figure 16: Shuffled data saved compared to the Index Overhead

Chapter 8

CONCLUSION AND FUTURE SCOPE

The flexible interface of the Apache Spark framework and astounding in-memory abstraction makes it easier for the users to define the complicated logic on top of the RDD data abstractions. Stable storage is abstract from the system so that it can work out of the box for the available datasets and just get the user started with the implementation. The fast prototyping phase and significant performance make it a very attractive framework. While it is suitable for iterative algorithms, it is not ideal for the algorithms running only once over data from the stable storage. Persistent partitions with integrated stable storage like Lachesis might reduce the latency for the complex analytical queries and ML inference phase.

The systems with integrated storage like Lachesis need the users to push all of their data into the underlying stable storage before developing applications on top of the datasets. Such a system can take full advantage of the persistent partitions reducing the shuffle overhead to its minimum.

The data locality meta-data information for the in-memory abstraction can be stored along with the data itself on the stable storage to be used while reading the data back into the framework's memory abstraction. The Microsoft Hyperspace library helps users to maintain such information and manage it using the interfaces developed by the Hyperspace team. The idea can be extended to create a middleware that can optimize the execution graph across the frameworks using layered data abstractions like Apache Flink, Spark, and Presto.

The matrix multiplications results show that the application can take significant

advantage of the persistent partitions. The training phase of the ML application is an iterative process whereas the inference phase computes the graph only once. Using persistent partitions may help in reducing the latency to serve the ML model.

REFERENCES

Abouzeid, Azza, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. 2009. "HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads." *Proceedings of the VLDB Endowment* 2 (1): 922–933.

Agrawal, Sanjay, Vivek Narasayya, and Beverly Yang. 2004. "Integrating vertical and horizontal partitioning into automated physical database design." In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data,* 359–370.

"Apache Flink." 2020, August. https://flink.apache.org.

"Apache Hadoop." 2020, October. https://hadoop.apache.org.

"Apache Spark." 2021, February. http://spark.apache.org.

Armbrust, Michael, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, et al. 2020. "Delta lake: high-performance ACID table storage over cloud object stores." *Proceedings of the VLDB Endowment* 13 (12): 3411–3424.

Armbrust, Michael, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. "Spark sql: Relational data processing in spark." In *Proceedings of the 2015 ACM SIGMOD international conference on management of data,* 1383–1394.

Brin, Sergey, and Lawrence Page. 1998. "The anatomy of a large-scale hypertextual web search engine." *Computer networks and ISDN systems* 30 (1-7): 107–117.

Cafarella, Michael J, and Christopher Ré. 2010. "Manimal: Relational optimization for data-intensive programs." In *Procceedings of the 13th International Workshop on the Web and Databases,* 1–6.

Curino, Carlo, Evan Philip Charles Jones, Yang Zhang, and Samuel R Madden. 2010. "Schism: a workload-driven approach to database replication and partitioning."

Dean, Jeffrey, and Sanjay Ghemawat. 2008. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51 (1): 107–113.

Dittrich, Jens, Jorge-Arnulfo Quiané-Ruiz, Alekh Jindal, Yagiz Kargin, Vinay Setty, and Jörg Schad. 2010. "Hadoop++ making a yellow elephant run like a cheetah

(without it even noticing)." *Proceedings of the VLDB Endowment* 3 (1-2): 515–529.

Eadon, George, Eugene Inseok Chong, Shrikanth Shankar, Ananth Raghavan, Jagannathan Srinivasan, and Souripriya Das. 2008. "Supporting table partitioning by reference in oracle." In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data,* 1111–1122.

Eltabakh, Mohamed Y, Yuanyuan Tian, Fatma Özcan, Rainer Gemulla, Aljoscha Krettek, and John McPherson. 2011. "CoHadoop: flexible data placement and its exploitation in Hadoop." *Proceedings of the VLDB Endowment* 4 (9): 575–585.

George, Lars. 2011. *HBase: The Definitive Guide.* 1st ed. O'Reilly Media. http://www.amazon.de/HBase-Definitive-Guide-Lars-George/dp/1449396100/ref=sr_1_1?ie=UTF8&qid=1317281653&sr=8-1.

"GroupLens Research." 2021, March. https://grouplens.org/datasets/movielens/.

Hilprecht, Benjamin, Carsten Binnig, and Uwe Roehm. 2019. "Learning a partitioning advisor with deep reinforcement learning." *arXiv preprint arXiv:1904.01279.*

Hilprecht, Benjamin, Carsten Binnig, and Uwe Röhm. 2020. "Learning a partitioning advisor for cloud databases." In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data,* 143–157.

"Hyperspace Repository." 2021, February. https://github.com/microsoft/hyperspace.

Idreos, Stratos, Martin L Kersten, Stefan Manegold, et al. 2007. "Database Cracking." In *CIDR,* 7:68–78.

Jiang, Dawei, Beng Chin Ooi, Lei Shi, and Sai Wu. 2010. "The performance of mapreduce: An in-depth study." *Proceedings of the VLDB Endowment* 3 (1-2): 472–483.

Karau, Holden, and Rachel Warren. 2017. *High Performance Spark: Best Practices for Scaling and Optimizing Apache Spark.* 1st. O'Reilly Media, Inc.

Kumar, K Ashwin, Abdul Quamar, Amol Deshpande, and Samir Khuller. 2014. "SWORD: workload-aware data placement and replica selection for cloud data management systems." *The VLDB Journal* 23 (6): 845–870.

Li, Guoliang, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. "Qtune: A query-aware database tuning system with deep reinforcement learning." *Proceedings of the VLDB Endowment* 12 (12): 2118–2130.

Lu, Yi, et al. 2017. "AdaptDB: adaptive partitioning for distributed joins." PhD diss., Massachusetts Institute of Technology.

Maier, David, and Jacob Stein. 1986. "Indexing in an object-oriented DBMS." In *Proceedings on the 1986 international workshop on Object-oriented database systems,* 171–182.

"Microsoft Hyperspace." 2021, March. https://docs.microsoft.com/en-us/azure/synapse-analytics/spark/apache-spark-performance-hyperspace.

Nehme, Rimma, and Nicolas Bruno. 2011. "Automated partitioning design in parallel database systems." In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data,* 1137–1148.

Olston, Christopher, Benjamin Reed, Adam Silberstein, and Utkarsh Srivastava. 2008. "Automatic Optimization of Parallel Dataflow Programs." In *USENIX Annual Technical Conference,* vol. 21.

Pavlo, Andrew, Carlo Curino, and Stanley Zdonik. 2012. "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems." In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data,* 61–72.

Rao, Jun, Chun Zhang, Nimrod Megiddo, and Guy Lohman. 2002. "Automating physical database design in a parallel database." In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data,* 558–569.

Richter, Stefan, Jorge-Arnulfo Quiané-Ruiz, Stefan Schuh, and Jens Dittrich. 2014. "Towards zero-overhead static and adaptive indexing in Hadoop." *The VLDB journal* 23 (3): 469–494.

Romero, Oscar, Victor Herrero, Alberto Abelló, and Jaume Ferrarons. 2015. "Tuning small analytics on Big Data: Data partitioning and secondary indexes in the Hadoop ecosystem." *Information Systems* 54:336–356.

Sattler, K-U, Eike Schallehn, and Ingolf Geist. 2005. "Towards indexing schemes for self-tuning dbms." In *21st International Conference on Data Engineering Workshops (ICDEW'05),* 1216–1216. IEEE.

Shaikhha, Amir, Yannis Klonatos, and Christoph Koch. 2018. "Building efficient query engines in a high-level language." *ACM Transactions on Database Systems (TODS)* 43 (1): 1–45.

Shen, Min, Ye Zhou, and Chandni Singh. 2020. "Magnet: push-based shuffle service for large-scale data processing." *Proceedings of the VLDB Endowment* 13 (12): 3382–3395.

Shvachko, Konstantin, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. "The hadoop distributed file system." In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST),* 1–10. Ieee.

Vuppalapati, Midhul, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. "Building an elastic query engine on disaggregated storage." In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20),* 449–462.

Zaharia, Matei, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." In *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12),* 15–28.

Zhang, Ji, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. "An end-to-end automatic cloud database tuning system using deep reinforcement learning." In *Proceedings of the 2019 International Conference on Management of Data,* 415–432.

Zou, Jia, Pratik Barhate, Amitabh Das, Arun Iyengar, Binhang Yuan, Dimitrije Jankov, and Chis Jermaine. 2020. "Lachesis: Automated Generation of Persistent Partitionings for Big Data Applications." *arXiv preprint arXiv:2006.16529.*

Zou, Jia, Arun Iyengar, and Chris Jermaine. 2018. "Pangea: monolithic distributed storage for data analytics." *arXiv preprint arXiv:1808.06094.*