Eurecom Institute

Master Thesis

A Work Sharing System on Apache Spark

Author:
Quang-Nhat
HOANG-XUAN

Supervisor:
Prof. Pietro MICHIARDI
Academic Supervisor:
Prof. Pietro MICHIARDI
Duy-Hung PHAN

A thesis submitted in fulfilment of the requirements for the degree of Master

in the

Distributed System Group
Network and Security Department

September 2015

Declaration of Authorship

I, Quang-Nhat HOANG-XUAN, declare that this thesis titled, 'A Work Sharing System on Apache Spark' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at Eurecom Institute.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this Institute or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

EURECOM INSTITUTE

Abstract

Network and Security Department
Network and Security Department

Master

A Work Sharing System on Apache Spark

by Quang-Nhat HOANG-XUAN

Large-scale data analysis has grown enormously in recent years. For example, Hadoop MapReduce, the de-facto standard framework to process such huge amount of data, has undergone several evolutions, which materialized in a new framework called Apache Spark.
In large data warehouses, multiple queries can be submitted by users at the same time. In addition, it has been shown that different jobs can operate on the same input file (indeed, some data is "hotter" than other), so there is a high probability for sharing the cost of reading the input file and computation also.
In this project, we propose a system that accepts multi-query workloads and proceeds with work-sharing optimization, for the Apache Spark system. In practice, we will focus on sharing I/O, the system automatically detects and efficiently combine the sharable queries, which are combined into smaller number of jobs that will compute output for all shared queries. The system also provides a scheduling mechanism which allows multi-query workloads to be optimized and executed in an efficient way.

EURECOM INSTITUTE

Abstract

Network and Security Department
Network and Security Department

Master

A Work Sharing System on Apache Spark

by Quang-Nhat HOANG-XUAN

Analyse de données à grande échelle a énormément augmenté ces dernières années. Par exemple, Hadoop MapReduce, le cadre standard de facto pour traiter cette quantité énorme de données, a subi plusieurs évolutions, réalisée dans un nouveau cadre appelé Apache étincelle.
Dans les entrepôts de données volumineuses, plusieurs requêtes peuvent être envoyées par les utilisateurs en même temps. En outre, il a été démontré que des emplois différents peuvent fonctionner sur le même fichier d'entrée (en effet, certaines données sont "plus chaudes" que l'autre), donc il y a une probabilité élevée d'un partage des coûts de lire le fichier d'entrée et calcul aussi.
Dans ce projet, nous proposons un cadre qui accepte les requêtes multiples charges de travail et procède avec partage des tâches optimisation, pour le système d'allumage d'Apache. Dans la pratique, nous concentrer sur le partage des e/s, le framework détecte automatiquement et allie efficacement les requêtes partageables, qui sont combinés en plus petit nombre d'emplois qui calculera la sortie pour toutes les requêtes partagées. Le cadre fournit également un mécanisme de planification qui permet des requêtes multiples charges de travail soit optimisé et exécuté de manière efficace.

Acknowledgements

I am really grateful because I managed to complete the master thesis: ' within the given time.

Firstly, I would like to express my sincere gratitude to my advisor Prof. Pietro MICHIARDI for the continuous support of my master thesis, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor.

The thesis cannot be completed without the great support and cooperation of PhD student Duy-Hung PHAN. I sincerely thank you for all of your help, time, and encouragement which allowed me to give all my best for this thesis.

I thank my labmates Trong-Khoa NGUYEN, Duc-Trung NGUYEN for the stimulating discussions and supports. Also I thank my fellow friends, Cu-Khoi-Nguyen MAC, Hoang-An LE, Quoc-Minh BUI, my friends in EURECOM institute, John von Neumann institute, and University of Science - VNU, for having always been a part of my life.

Last but not the least, I would like to thank my mother and father for supporting me spiritually throughout writing this thesis and my life in general. Thank you for nourishing my dream with me and helping me become the one I am today.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, large-scale data analysis has explosively grown. In the beginning of the big data era, Hadoop MapReduce [1] was the dominant framework which was widely used to process huge amount of data. Under several evolutions, a new framework appeared which is called Apache Spark [4].

In general, writing a MapReduce [10] algorithm in Hadoop and Spark requires exceptional skills, because of the intricacies of the underlying programming model: to overcome such problems, high-level languages (in essence, similar to SQL) have emerged as an alternative method to query data. For example, Apache Pig [11] [3] and Apache Hive [7] [2] bring to users the ability to express their programs in SQL-like by PigLatin [19] and HiveQL [7] , which are then compiled into MapReduce jobs before their execution. Apache Spark has also brought to us its SQL-like framework called SparkSQL [15], which provides users the flexibilites of using Dataframe API and SQL.

In large data warehouses, multiple queries can be submitted by multiple users at the same time. Some files can be used at the input of many queries with different tasks. In other way, these files are "hotter" than the others because they are used more frequently, so there is high potential for sharing the cost of reading the input file and also the computations.

In this project, we propose a system that accepts multi-query workloads and proceeds with work-sharing optimization, for the Apache Spark system. The system automatically detects and efficiently combine the sharable queries, which are combined into smaller number of jobs that will compute output for all shared queries. The system also provides a scheduling mechanism which allows multi-query workloads to be optimized and executed in an efficient way. The system design aims to provide users generalization and extensibility so users can introduce new work sharing optimizations using a simple domain specific language. We take sharing I/O at a usecase for our system to show its efficiency, generality and extensibility.

In order to accomplish the internship, in our work, we:

- Study Apache Spark internals and SparkSQL. We also document them carefully in this report.
- Design and implement a work sharing system which is efficient and easy to extend.
- Take sharing I/O at a usecase to demonstrate the efficiency of our system and its correctness.
- Conduct a preliminary experimental performance evaluation.

Chapter 2

Apache Spark

2.1 Apache Spark

Nowaday, the speed and complexity for data processing have grown fast. Many applications like machine learning or graph processing require using complex algorithms. The need of a new framework which supports more applications is the main reason why apache spark was created. Apache Spark is an open source project originally developed in the AMPLab at UC Berkeley. It is the implementation of Resislient Distributed Datasets (RDDs) [14]. Spark runs program up to 100 times faster in comparison to Apache Hadoop. It also supports a wide range of application and can run everywhere: Hadoop, Mesos, standalone or in the cloud. Spark provides a convenient language-integrated programming interface in the Scala programming language but users can also write applications using Java, Python or R from the API it supports. Apache Spark has fastly grown since it is one of the most active projects of Apache with hundred of contributors. In this chapter, we will mainly focus on:

- An overview of Apache Spark, which mainly focus on Resislient Distributed Dataset and Spark Job Submission process.
- A brief introduction of SparkSQL and its main components.

2.1.1 Resilient Distributed Datasets

Resilient Distributed Datasets (RDD) extends the data flow programming model introduced by MapReduce [10] and Dryad [16], which is the most widely used model for large-scale data analysis today. RDDs are fault-tolerant and parallel data structures which let users explicitly store data in memory or on disk, manage their partitioning, and use them with a rich set of operators. They provide a simple and efficient programming interface that can capture both current specialized models and new applications like streaming, machine learning or graph processing.

An RDD is an immutable, partitioned set of records. An RDD can only be created through a set of operations, which is called transformation, from two sources: storage and other RDDs. Some transformation can be listed such as: map, filter, union...
An RDD always contains the information about how it was created by storing its parents RDDs, which is represented by Dependencies. There are two kinds of dependencies: narrow dependency where each parition of parent RDD is used by at most one child RDD, wide dependency where each partition of parent RDD is used by multiple child RDDs. For example, a filter transformation creates a narrow dependency while an union transformation creates a wide dependency.
RDDs have two fancy features provided for users: persistence and partitioning. Persistence is very helpful when an RDD is reused many times. When persisting an RDD, each node stores any partitions of it that it computes in memory and reuses them in other actions on that dataset. This allows future actions to be much faster. Persisting is a key tool for iterative algorithms and fast interactive use which is widely used among developers and researchers. Partitioning lets users manage their data better, which is necessary for some optimizations required location information, for example, join is a transformation which it would perform better if having information about data location.

All transformations in Spark are lazy, which means they do not compute their results right away. Instead, they just remember the transformations applied to some base datasets using the Dependencies or the Directed

Acyclic Graph to know what transformations were applied to them. The transformations are only computed when an action requires a result to be returned to the driver program.

To get the result or export the result after processing the data, users call "action". An action is also used to trigger the whole Spark job.

To summarize, each RDD is characterized by five main properties:

- A list of partitions
- A function for computing each split
- A list of dependencies on other RDDs
- A partitioner for key-value RDDs
- A list of preferred locations to compute each split on

2.1.2 Spark Job Lifetime

Exploring the flow of a Spark Job helps us understanding the internal of Spark, how it works. This is important since our SparkSQL Server needs some modifications to the Spark Core so we need to understand how Spark works clearly.
There are four steps in the lifetime of a Spark Job:

- RDD objects creation.
- DAG scheduling.
- Task scheduling.
- Task execution.

Figure 2.1 illustrates those four steps clearly.

While the first three steps happen at the driver, the representation of the connection to the Spark cluster, the last step happens at the Executors.

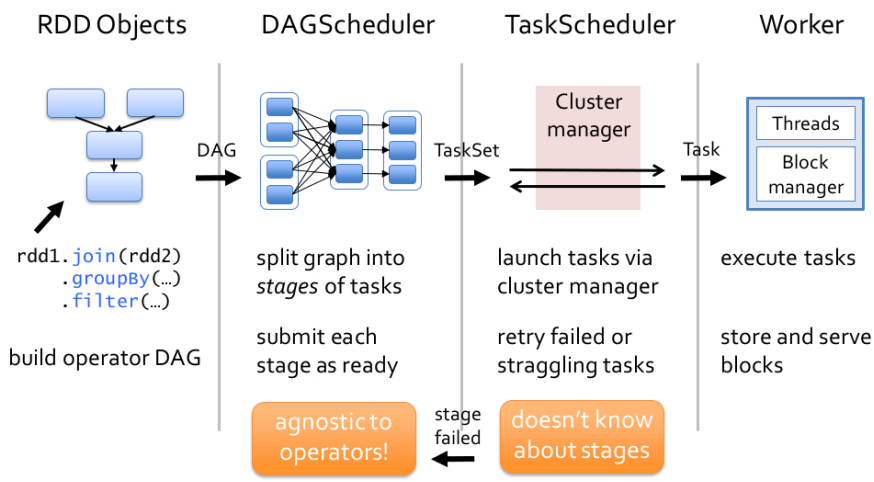

FIGURE 2.1: The lifetime of a Spark job

We go deeper into each step to understand a Spark Job Lifetime.

RDD objects creation Users create RDDs through a set of transformations, at the moment, the RDD is created at the driver.

DAG Scheduling When an action is called on an RDD, the RDD is transfered to the next step: DAG Scheduling. It is the process of splitting the DAG into stages, then submitting each stage as ready. A module called DAGScheduler is in charge of DAG Scheduling. A stage is a set of independent tasks all computing the same function that need to run as part of a Spark job, where all the tasks have the same shuffle dependencies. Each DAG of tasks runs by the scheduler is splitted up into stages at the boundaries where shuffle occurs, and then the DAGScheduler runs these stages in topological order.

Task Scheduling TaskScheduler is the component which receives the stages from DAGScheduler and submits them to the cluster.

Task Execution Spark calls worker as Executor. The backend will receive the worker list from the Cluster Manager, then it will launchTask at the Executor. A BlockManager at each Executor will help it to deal with shuffle data and cached RDDs. New TaskRunner is created at the Executor and it starts the threadpool to process taskset, each task runs on one thread. After finishing the tasks, results are sent back to the driver or saved to disks.

Some information we need to notice are:

- Thanks to the DAGSchedulerEventProcessLoop, DAGScheduler can keep track of stages' statuses and resubmit failed stages.
- TaskScheduler only deals with taskSet after being formed from Stages at DAGScheduler, that's why it doesn't know any thing about stages.
- A Spark program can contain multiple DAGs, each DAG will have one action. So, inside a driver, they are submitted as jobs one by one in order.
- Spark has a nice feature appears from version 1.2: dynamic resource allocation. Spark will base on the workload to request for extra resources when it needs or give the resources back to the cluster if they are no longer used.

2.1.3 The Log Mining Example

Below is an example about log mining. The idea of the example is loading the log, then we keep the error messages in memory and analyze it many times. By using caching, we can reduce the analyzed times. This is just a basic example to demonstrate how to write a Spark program, and the caching feature in action.

```
//base RDD
val lines = spark.textFile("hdfs://...")

//transform to get error RDD, then cache it.
val errors = lines.filter(_.startsWith("ERROR"))
val messages = errors.map(_.split('\t')(2))
val cachedMsgs = messages.cache()

//action 1
cachedMsgs.filter(_.contains("foo")).count
//action 2
cachedMsgs.filter(_.contains("bar")).count
```

Table 2.1 gives us some basic transformations and actions inside Spark.

Firstly, create an RDD by reading an input file. Then we apply a set of transformations: filter and map to get the error logs and preprocessing

them by splitting them by tabluture character. Then, we cache it. At the moment the "cache" command is called, nothing happens. When action 1 is happend, Spark keeps the cachedMsgs in memory, then filters the error logs which contains "foo". When action 2 is called, the cachedMsgs was kept in memory so Spark does not need to process everything from the beginning as the action 1 was called. If there are many actions after action 2 and they also process on cachedMsgs, they also do not need to process the previous part from scratch. This reduces the cost of I/O and computation many times.

2.2 Apache SparkSQL

SparkSQL is a new module of Apache Spark, which provides the flexibilities for users to combine relational programming and functional programming through DataFrame API. It also provides an extensible optimizer called Catalyst, which utilizes the features of Scala language to make it easier to add new rules and to control code generation. The two following subsections brieftly describe these two contributions.

2.2.1 DataFrame API

DataFrames are distributed collection of column-structured records which can be manipulated by both new functional API and produceral API of Spark. DataFrames can be created from various data sources or even existing RDDs.

DataFrame also provides user-defined function, which is very important for database systems. The inline-definition of UDFs also makes it easier for user to write their functions.
Each DataFrame object has a 'lazy' logical plan which means no execution occurs until an action is called, this let the logical plan itself have better chance of optimizations.

Spark SQL supports two different methods for converting existing RDDs into DataFrames:

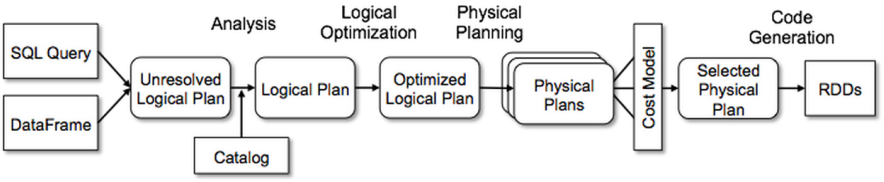

FIGURE 2.2: Catalyst Dataflow

- Using reflection to infer the schema of an RDD that contains specific types of objects. This method provides more concise codes and it works well when we already know about the schema of the object.
- Using a programmatic interface, we can construct a schema and then apply it to an existing RDD. This method is more verbose, it allows us to construct DataFrames when the columns and their types are not known until runtime.

2.2.2 Catalyst Optimizer

Catalyst Optimizer is an extensible optimization engine. It contains a general library for representing trees and applying rules to manipulate them.

Trees Tree is the main data structure in Catalyst, which is formed by many nodes. A node has a type and its childs, which can be none or more than zero nodes. Node objects are read-only and can be manipulated using functional transformations.

Rules Trees can be transformed to another trees by using rules. By utilizing pattern matching, a Scala feature, a part of a tree can be found and replaced by another one.

A Catalyst tree transformation process is composed by four phases. Figure 2.2 illustrates this process. We explain each process below:

- **Analysis** Its input is a relation to be computed, which is created from SparkSQL Parser to form an Abstract Syntax Tree (AST) or from using DataFrame API. The relation can contain unresolved attribute references or relations, which means we do not know its type or have not matched it to an input table (or an alias). With

Catalyst Rules and a Catalog, an object which is used to keep track of created tables, SparkSQL resolves those attributes.

- **Logical Optimizations** This phase applies rule-based optimizations to the logical plan which is outputed from Analysis phase. By utilizing the nature of Scala language,users can simply add rules to fulfill their needs.
- **Physical Planning** In this phase, Spark SQL tries to generate as much physical plans as possible from one optimized logical plan, by using physical operators that match the Spark execution engine. By associating to a cost model, it can select the best plan.
- **Code Generation** This final phase of query optimization involves generating Java bytecode to run on each machine. By utilizing a special feature of Scala, which is quasiquotes, Catalyst can make code generation simpler. Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. Again, due to the simplicity of quasiquotes, users can add their own rules for new types of expressions.

2.2.3 An application written using SparkSQL

Below is a simple example of SparkSQL to demonstrate the combination of relational programming and functional programming of SparkSQL.

```
case class Person(name: String, age: Integer)
val input = sc.textFile("examples/src/main/resources/people.txt")
val people = input.map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))
val peopleDF = people.toDF()
peopleDF.registerTempTable("people")
val teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")
val result = teenagers.map(t => "Name: " + t(0)).collect.foreach(println)
```

We can declare our data structure, then read the input file with that format and store into an RDD. We create a DataFrame from that RDD and create a table from it. We are using the method of reflection to interoperate between DataFrame and RDD. An SQL query is passed to SparkSQL. After the optimization and translation happened, a normal

TABLE 2.1: Some transformatios and actions in Spark

Transformation
map(f : $T \Rightarrow$U) : $RDD[T] \Rightarrow$RDD[U]
filter(f : $T \Rightarrow$Bool) : $RDD[T] \Rightarrow$RDD[T]
flatMap(f : $T \Rightarrow$Seq[U]) : $RDD[T] \Rightarrow$RDD[U]
sample(fraction : Float) : $RDD[T] \Rightarrow$RDD[T]
groupByKey() : $RDD[(K, V)] \Rightarrow$RDD[(K, Seq[V])]
reduceByKey(f : $(V, V) \Rightarrow$V) : $RDD[(K, V)] \Rightarrow$RDD[(K, V)]
union() : $(RDD[T], RDD[T]) \Rightarrow$RDD[T]
join() : $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow$RDD[(K, (V, W))]
cogroup() : $(RDD[(K, V)], RDD[(K, W)]) \Rightarrow$RDD[(K, (Seq[V], Seq[W]))]
crossProduct() : $(RDD[T], RDD[U]) \Rightarrow$RDD[(T, U)]
mapValues(f : $V \Rightarrow$W) : $RDD[(K, V)] \Rightarrow$RDD[(K, W)]
sort(c : Comparator[K]) : $RDD[(K, V)] \Rightarrow$RDD[(K, V)]
partitionBy(p : Partitioner[K]) : $RDD[(K, V)] \Rightarrow$RDD[(K, V)]
Actions
count() : $RDD[T] \Rightarrow$Long
collect() : $RDD[T] \Rightarrow$Seq[T]
reduce(f : $(T, T) \Rightarrow$T) : $RDD[T] \Rightarrow$T
lookup(k : K) : $RDD[(K, V)] \Rightarrow$Seq[V]
save(path : String) : Outputs RDD to a storage system, e.g., HDFS

Spark Job is created with a Directed-Acyclic Graph inside. An action would trigger all these things.

Chapter 3

A Work Sharing System

3.1 Work Sharing

In large data ware houses, many queries are usually submitted at the same time by multiple users. In the context of big data, a query would take long time to run. There are some data that would be used more frequently than the others, so we can avoid the redundant computations by sharing works among those queries and reduce the total amount of execution time.

The idea of work sharing originally comes from Multiple Query Optimization (MQO) [20]. There are many works on traditional database systems [9] [8] [12] [21] and also on Hadoop MapReduce have been published. Olston et al. worked on a system called CoScan [18], a system for sharing data and processing costs among multi-step MapReduce workflows which are executed in Hadoop. In Apache Pig, a number of work sharing optimizations [5] have been used with the idea of MQO. The current state-of-the-art work in this field is MRShare[17] on Apache Hadoop. With the grouping technique, it provides the map input sharing to share scan of input files and map output sharing for saving cost of communication.

3.2 SparkSQL Server

All the works above are only supported for Hadoop MapReduce and they are focused only on a part of work sharing, then they are hard to extend when users want to have new sharing techniques. To the best of our knowledge, there is no published work about Multi Query Optimization on Apache Spark. In our work, we present a new system called SparkSQL Server, which is focused on work sharing among multiple queries.

SparkSQL Server is an open source project and hosted at our team's Github.

SparkSQL Server supports queries written in SQL by using DataFrame API of SparkSQL or in produceral languague supported by SparkCore itself.

The system does not only care about work sharing among multiple queries but also the scheduling mechanisms to associate with the sharing techniques or to fullfil the users' requirements.

The system is generalized so it can support many types work sharing. All the modules of SparkSQL Server are easy to extend, this lets users and developers easily plug their own implementations into the system.

3.2.1 System Design

The design of our system aims at the generalization and extensibility so users and developers can easily plug their own implementations. Figure 3.1 expresses the design of our system.

Overall, our system has two sides: client side and server side.
Client side The client submits the query and needed information to SparkSQL Server so SparkSQL Server can reconstruct exactly the query happened at client side. The query can be written by using produceral

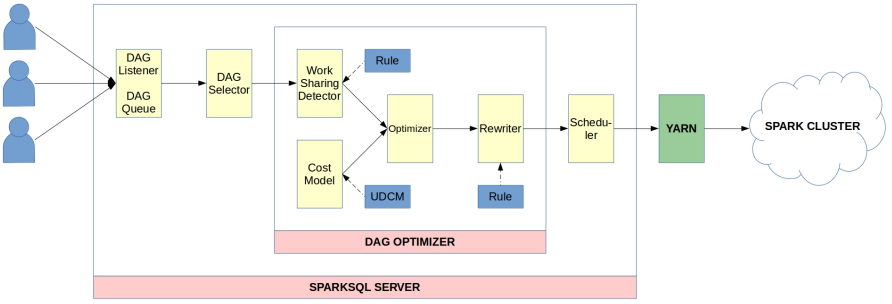

FIGURE 3.1: The design of SparkSQL Server

operations in Spark or relational operations in SparkSQL or the combination of both types of operations.

Server side The SparkSQL Server contains three main components:

- The Listeners, which listen for client connections and communicate with them.
- The DAG Optimizer is the heart of our system. All the detections, optimizations, and transformations happen here. The DAG Optimizer is composed of:
  - WorkSharing Detector: this module detects the sharing opportunities among a batch of jobs which has just received from clients. The detector uses rule-based mechanism to detect the sharable jobs. Users can easily write their own rules to detect sharable jobs with many types of sharing.
  - CostModel: it calculates the cost of each execution plan. Users provide their own cost model by the User-defined Cost Model.
  - Optimizer: it receives the input from the output of WorkSharing Detector and uses the CostModel to pick the best DAG or set of DAGs to execute.
  - Rewriter: the Rewriter transforms the original execution plans into the optimized execution plans and then submits them to the Schedulers. The transformation is also based on the rule-based mechanism. It is very easy to define a new rule to support another type of sharing.

Again, all of these four modules are extendable so users can plug their owns into these modules.

- The Schedulers, which are used to fulfill users' requirements and to associate with the sharing techniques. These Schedulers are aslo extensible so users can plug their own shechuling strategies.

3.2.2 Implementations

Similar to the above subsection, we present the implementation of our system on both client side and server side. The order of the explaination is the same as the flow in figure 3.1.

Client side We modify the spark-submit command so that it does not send the application to the cluster manager but to the SparkSQL Server, using this option: –sparksql-server

Each client starts it own driver with its own SparkContext. Then, the client can generate its DAG, the DataFrame creation and send those information: DAG, DataFrame generation, SQL Query to the SparkSQL Server.

The client needs to send the jar file, because it contains all user defined functions, classes which are very important to reassemble the DAG at server side. It also needs to send needed information so the SparkSQL Server can reconstruct the DAG and the query. Don't forget that the query has already been optimized by Catalyst at the client side.

Server side

- **DAG Listener and DAG Queue** DAG Listener accepts the clients' connections and receives DAGs, queries and other information they send. Then, it passes to the DAG Queue, the DAG Queue accepts the information the clients send at the FIFO order. In this component, the full DAG of each user will be built based on the initial DAG, the DataFrame creation information and the query. The DAG Queue has a window with fixed size, after reaching the size of the windows, DAG Queue will send a batch of DAGs (and queries also) into the next components. The window size right now is fixed as a constant, it will be changed when we take care of scheduling.

- **DAG Selector** This component is something called a pre-scheduler, which will based on the constraint attached with each query to fulfill user's requirements. For example: job submitted with a deadline. Right now, it just uses the simple FIFO strategy.

- **Worksharing Detector** This component will detect which DAGs have the opportunity for worksharing, which DAGs haven't. Remember that worksharing here is not only sharing scan, it can also have other variants such as sharing group-by or join. It works as rule-based mechanism to detect the worksharing opportunity. The Worksharing Detector bases on the associated rule to find which DAGs have the common part that can be shared. For example, with scan sharing rule, since we always have a pattern of a DAG is that when reading a file from disk, it always contains the HadoopRDD with the attached inputfile path, the rule tries to find each inputfile path of each DAG and puts into an array, then it intersects all the arrays to find out which DAG is shareable.

- **Cost Model** It provides an interface so that users can plug their own cost model, which is called User defined cost model, into the system. We can have many types of Cost Model. The cost model can give score, which is calculated from functions, on each DAG or on a batch of DAG.

- **Optimizer** With a bag from the output of the previous and a cost model associated with its sharing type, the Optimizer component will do the job: pick the plan with the lowest cost. The Optimizer and the CostModel work closely to find out the best result. For example, the Optimizer can generate many combinations of shareable DAGs and give them to the CostModel, the CostModel evaluates them with scores and gives back to the Optimizer. The Optimizer then choose the combination with the best score.

- **Rewriter** This component has many families of rewrite. Each family will have many rewrite rules. It will generate a rewritten DAG which can be optimized, and use the rule that Optimizer picked to transform the original DAGs into the new ones. Each sharing technique can have one or more than one rewriter rules.

For example, with sharing scan, we can have the rewrite technique using MRShare or caching.

There are some special data structures to wrap a DAG after going through a component:

- DAGContainer: use to wrap a DAG when it is received at DAGListener. It contains a DAG and metadata that it brings together.
- AnalysedBag: use to wrap an array of DAGContainers after it went through the WorkSharing Detector. It contains an Array of sharable DAGContainer. It also contains a label to indicate which type of sharing it is since there are many types of sharing.
- OptimizedBag: use to wrap an array of DAGContainers after it went through the Optimizer. It contains an array of DAGContainers, which is the best combination of DAG after going through the Optimizer. It also contains a label to indicate its sharing type and the Rewritten Rule to indicate which rule it will use to rewrite. A no-sharable DAG is also wrapped into an OptimizedBag with the Rewritten Rule equals to Null.
- RewrittenBag: use to wrap a rewritten OptimizedBag after the optimized bag went through the Rewriter.

3.2.3 An Example on SparkSQL Server

This section gives us an example of SparkSQL Server in action so we can understand how SparkSQL Server works clearly. Three examples below are similar to the example in section 2.2.3 and we focus only on scan sharing in this example. We also use the MRShare technique, which will be described in details in next chapter, for the Optimizer and Rewriter. The explaination has three parts for each components: Input, Output, and we relate them to the example.

User1

```
case class Teacher(name: String, age: Integer)
val input = sc.textFile("hdfs://A.txt")
val people = input.map(_.split(",")).map(p => Teacher(p(0), p(1).trim.toInt))
```

```
val peopleDF = people.toDF()
peopleDF.registerTempTable("people")
val teacher = sqlContext.sql("SELECT name FROM people WHERE age <= 50")
val result = teacher.map(t => "Name: " + t(0)).collect.foreach(println)
```

User2

```
case class Student(fullname: String, age: Integer)
val input = sc.textFile("hdfs://A.txt")
val people = input.map(_.split(",")).map(p => Student(p(0), p(1).trim.toInt))
val peopleDF = people.toDF()
peopleDF.registerTempTable("people")
val student = sqlContext.sql("SELECT name FROM people WHERE age >= 19")
val result = student.map(t => "Name: " + t(0)).collect.foreach(println)
```

User3

```
case class Person(name: String, age: Integer)
val input = sc.textFile("hdfs://B.txt")
val people = input.map(_.split(",")).map(p => Persion(p(0), p(1).trim.toInt))
val peopleDF = people.toDF()
peopleDF.registerTempTable("people")
val adults = sqlContext.sql("SELECT name FROM people WHERE age >= 25")
val result = adults.map(t => "Name: " + t(0)).collect.foreach(println)
```

Client Side

- Input: Users' applications (jar files)
- Output: Jarfile, DAGs, DataFrame creation information, SQL queries, mapOutputRatio (the ratio between map output size and input file size) for MRShare are sent to SparkSQL Server. In those example, we have DAG1, DAG2, DAG3.

Server Side
DAG Listener and DAG Queue

- Input: DAGs and queries from clients
- Output: Array of DAGContainers, the metadata is updated with the mapOutputRatio for each DAGContainer.
- In the example, let assume the window size of DAG Queue is 3, so the output will be DAG1, DAG2, DAG3. If there is user4 submits the job, it will be packed with another two jobs.

DAG Selector

- Input: Array of DAGContainers
- Output: Array of DAGContainers based on scheduling strategies (FIFO at the moment).
- In the example, the input and the output is the same as we use FIFO strategy.

Worksharing Detector

- Input: Array of DAGContainers
- Output: Array of AnalysedBags
- In the example, we got two Bags: DAGBag1: (DAG1, DAG2) with the label: "SCAN-SHARING"; DAGBag2: DAG3 with the label: "NO-SHARING".

Cost Model

- Input: a combination of DAGContainer which is grouped from MRShare Optimizer
- Output: cost belong to each group, in MRShare, it is a number which is computed through GS function.
- In the example, we use the MRShare Cost Model so it returns a score for each combination. Let's assume the score of group DAG1, DAG2 is the best.

Optimizer

- Input: Array of AnalysedBags
- Output: Array of OptimizedBags after choosing the best score generated by the Cost Model.
- In the example, we got OptimizedBag1: (DAG1, DAG2), OptimizedBag2: (DAG3)

Rewriter

- Input: Array of OptimizedBags
- Output: Array of RewrittenBags
- In the example, since MRShare uses the simultaneous pipeline technique to merge jobs, we got RewrittenBag1: (DAG12), RewrittenBag2: (DAG3)

PostScheduler

- Input: Rewritten Bags of DAGs
- In the example, we got 2 Bags of DAGs. Since the execution order does not affect these jobs or in other way, they are independent on each other, so we just use FIFO strategy to submit to the cluster.

The example is only for scan sharing and just to show the dataflow of SparkSQL Server but the system is generalized and extensible so other sharing techniques can be implemented into the system. For example, in join sharing, similarly to scan sharing, we can detect which join operation of a DAG can share together. Then, with an Optimizer and a Cost Model to select the best combination of shareable DAGs, the Rewritter can use a rule to rewrite the bag of DAGs.

Chapter 4

SparkSQL Server in Practice: Scan Sharing

4.1 Scan Sharing

In large data warehouses, companies, users submit their queries at the same period of time. There are some files which are "hotter" since they are used more frequently than the others, then reading those files multiple times from disk can be redundant by reading each file only one time and many others users can use them. Sharing scan does not only help to reduce the reading cost but also the communication cost, then the total execution time of the queries is reduced.

We instantiate the sharing scan technique into our system by using two approaches, one from MRShare [17] works and the other is a feature of Spark itself. Both techniques are easily to implement and integrate into our system, which proves the generalization and extensibility of our system.

4.1.1 Current State-of-the-Art

As described before, on Apache Spark, there is no published work on Work Sharing. The current State-of-the-art of Scan Sharing on Apache Hadoop MapReduce is MRShare.

MRShare is a sharing framework, which provides two techniques of sharing: map input sharing and map output sharing. The idea of MRShare is group the jobs which can share map input or map output into groups, with the constrains of maximizing the total saving by defining their own cost model.

Grouping jobs into one is not always beneficial since the reading cost is decreased but the amount of data shuffled over networks may increased. Using dynamic programming and a cost model, MRShare evalutates possible groups and decides which groups give it the maximum saving.
The technique that MRShare uses to merge multiple jobs into one called simultaneous pipeline. It attaches a tagging in each tuple to identify which job it belongs to.

4.1.2 Caching in Apache Spark

There is an important feature on Apache Spark which is Caching. This feature utilizes the memory storage to keep data in memory and uses it multiple times. When multiple jobs in a Spark Application use the same input file, we can cache the partitions which are hold the input file at the first job. When the second job wants to do some transformations on those partitions, it does not need to recompute from scratch to get them, it just checks if they are in memory or not, if they are in memory, it gets them and applies transformations on them.

Caching is a costly operation itself. If the cost of caching is larger than the cost of reading input files multiple times, so caching is not beneficial. In addition, if memory storage of the cluster is not large enough to cache the whole input files, it may write a part of the input files to disk and read them when needed, which also increases the I/O cost.

4.2 Scan Sharing in SparkSQL Server

We integrate two techniques of scan sharing: using MRShare and using caching.
At client side, since MRShare requires some pre-defined parameters about the jobs, when users submit their Spark applications through the spark-submit command, they need to pass those parameters too.

At server side, we only discuss about the components that we need to plug our own implementations for scan sharing:

- WorkSharing Detector: we need to add our own Rule which is "ScanSharing" to detect which jobs can be shared scan.
- CostModel: for MRShare, we implement the cost model that was published in the paper. For caching technique, currently, there is no pulished works about cost model relates to caching data in memory, it is going to be a part of our future work, currently, we do not add the cost model for caching.
- Optimizer: for MRShare, the dynamic programming algorithms which are described in the original paper are implemented with some modifications to adapt and can be integrated into our system. Since we do not have the cost model for caching technique, currently, we just output the same batch of jobs as the same as the input batch of jobs.
- Rewriter: We add two Rewrite rules. For MRShare, we add the simultaneous rewrite rule. For caching, we add the caching rewrite rule. The details of the implementations of simultaneous pipeline technique is described in the next section. The rewrite rule for caching is simple since we cache scan RDD of the first job in the batch, then replace the scan RDD of the rest jobs in the batch with the cached scan RDD.
- PostScheduler: At the moment, we use FIFO strategy for MRShare technique, while in caching technique, it should be noticed that the first job which does the caching of scan RDD needs to be executed first, then we apply FIFO strategy for the rest jobs in the batch.

4.3 Implementations

Caching is a feature of Apache Spark which is widely used. Besides that, the cost model and dynamic programming algorithms are well described in the MRShare paper. In this section, we mainly discuss about how we implement the simultaneous pipeline technique used in MRShare. In addition, we also briefly describe the MRShare dynamic programming and how we implement it.

4.3.1 MRShare Cost Model and Its Algorithms

The general idea of MRShare for scan sharing is to merge a batch of MapReduce jobs into one job which is called metajob. MRShare Cost Model shows that greedily merging jobs is not always beneficial since it also adds the extra sorting costs and also the shuffle bytes over the network.

The optimization problem of MRShare is formulated as: *Given a set of jobs J = J1, ..., Jn, groups the jobs into S non-overlapping groups G1, G2, ... , Gs, such that the overall sum of savings is maximized.* The authors prove that this is an NP-hard problem so they consider a relaxed version when the extra sorting overheads relies only on the job that contains the highest map output ratio in a group. They realize that if all the jobs in a group are sorted into a list according to the map output ratio, the optimal grouping will consist of consecutive jobs in the list. The problem is become as splitting the sorted list of jobs into sublists which give the maximized savings.

The authors propose an dynamic, exact programming algorithm called *SplitJob*: suppose in the optimial arrangement, the last sublist starts from job Ji, we need to find out the optimal arrangement for the previous i-1 jobs. It will search all the possible cases for job Ji-1. The pseudo code is provided below.

1. Compute GAIN(i,l) for $1 \leq i \leq l \leq n$.
2. Compute GS(i,l) for $1 \leq i \leq l \leq n$.
3. Compute c(l) and source(l) for $1 \leq l \leq n$.
4. Return c and source.

Algorithm 1: SplitJob(J1, ... , Jn)

The map output ratio is measured from processed-jobs and kept in history so user can pass them through the modified version of spark-submit command that we provide. Since the output of *SplitJob* can contain group of single jobs, so a refinement algorithm called *MultiSplitJob* is proposed. The main idea of *MultiSplitJob* is to put those single jobs from *SpiltJob* into a new job list and apply again *SplitJob* on it. The pseudo code is provided below.

$J \leftarrow J1, ..., Jn$ (input jobs)
$G \leftarrow \emptyset$ (output groups)
while *($J \neq \varnothing$)* **do**
  compute δj for each $Jj \in J$
  ALL = SplitJobs(J)
  $G \leftarrow G \cup ALL.getNonSingletonGroups()$
  $SINGLES \leftarrow ALL.getSingletonGroups()$
  if *($|SINGLES| < |J|$)* **then**
    $J \leftarrow SINGLES$
  else
    Jx = SINGLES.theSmallest()
    $G \leftarrow G \cup Jx$
    $J \leftarrow SINGLES \setminus Jx$
  end
end
return G as the final set of groups

Algorithm 2: MultiSplitJob(J1, ... , Jn)

4.3.2 Simultaneous Pipeline Technique

In general, this technique is used when merging multiple jobs read the same input file into one meta job. The technique has been used in Multi Query Optimization of Apache Pig. In MRShare, they proposed this technique on Hadoop MapReduce as an automatic module. On Apache Spark, to the best of our knowledges, all the query optimizations are currently focusing on single query, not multi query. We reintroduce the simultaneous pipeline technique and implement it on Apache Spark.

There are 4 new RDDs that we create to implement the technique.

- MuxRDD: The MuxRDD is used to buffer the input record and multiplex it into multiple pipelines, which represent multiple jobs.

- LabellingRDD: The LabellingRDD is used to attach the label into each tuple before it is shuffled over the network. The label is an integer which is attached to the key of the tuple. So, the old tuple [Key, Value] becomes [(Label, Key), Value].
- DispatchRDD: The DispatchRDD is used to route the tuple to the right pipeline by using the label attached inside it.
- PullRDD: Because Spark is based on Pull Mechanism, which means the child RDD asks its parent RDDs to give it the tuple. In default, the saveAsTextFile action has the puller to trigger the job. To keep Spark at it is, we create our new puller called PullRDD.

We also modify some parts of the shuffle component on Apache Spark:

- ShuffledRDD: the ShuffledRDD contains only one Aggregator, the object holds the aggregate function. In our case, to merge multiple jobs which have different Aggregators, we create a list of Aggregators to hold their Aggregators.
- ExternalSorter: before writing to files and shuffling them over the network, Spark does a local aggregation (it is a combiner in Hadoop MapReduce), so we also need to apply the correct aggregate function to the tuple by checking the label attached into it.
- AppendOnlyMap: when the aggregation happens in the other stage after getting the data over the network (in Hadoop MapReduce, it is the Reduce phase), again, we need to apply the correct aggregate function to the list of tuple by checking the label attached into it.

We provide two examples to demonstrate how simultaneous pipeline technique works. Two jobs are described at two figures below. For each figure, on the left are two original DAGs and on the right is the meta DAG which is merged from those two DAGs.

- Two jobs, each job has only one ShuffleRDD From figure 4.1, we can see the metajob is a DAG which is mainly composed from the two original DAGs. The MuxRDD is attached right after the scan RDD, it does the multiplexing the input into two pipelines.

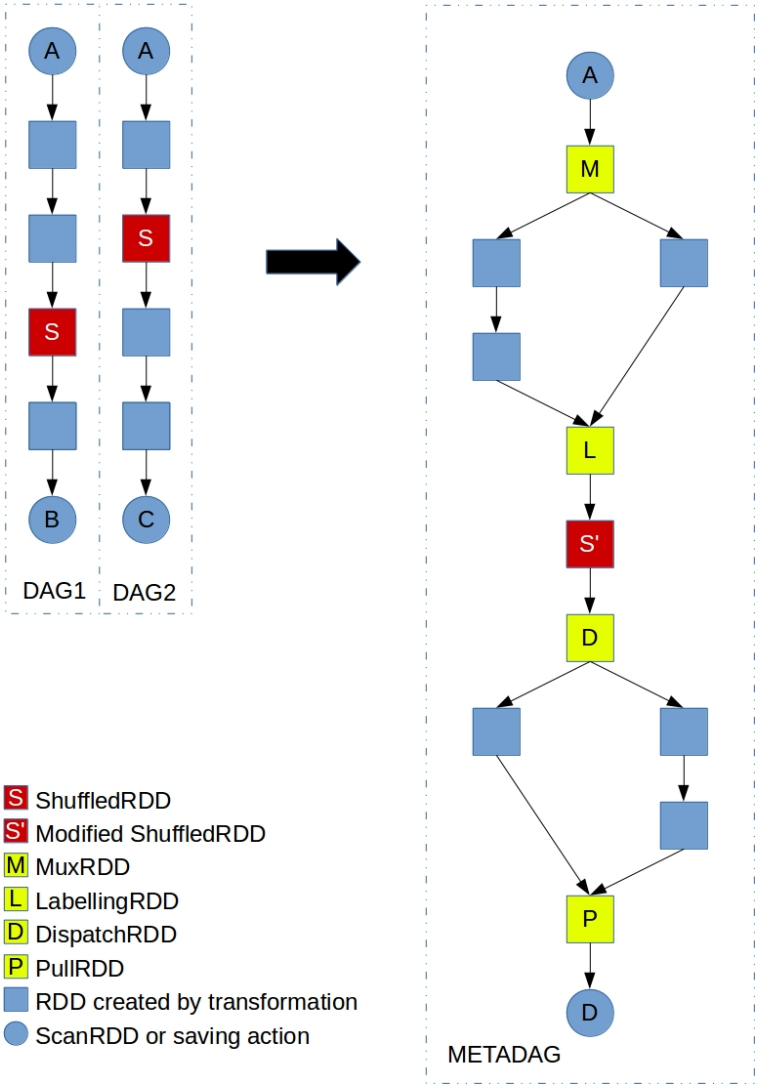

FIGURE 4.1: Transformation process of a simple case

Before going to the ShuffleRDD, a LabellingRDD is inserted right before the ShuffleRDD to attach the label into each tuple. In this example, we have two jobs, so the label has two values which is 0 and 1. The ShuffledRDD is modified so it contains two aggregate functions. The aggregation before and after shuffling happens on the correct tuple with the correct aggregate function due to the label attached into each tuple. The DispatchRDD is attached after the ShuffledRDD and routes the correct tuple to the associated

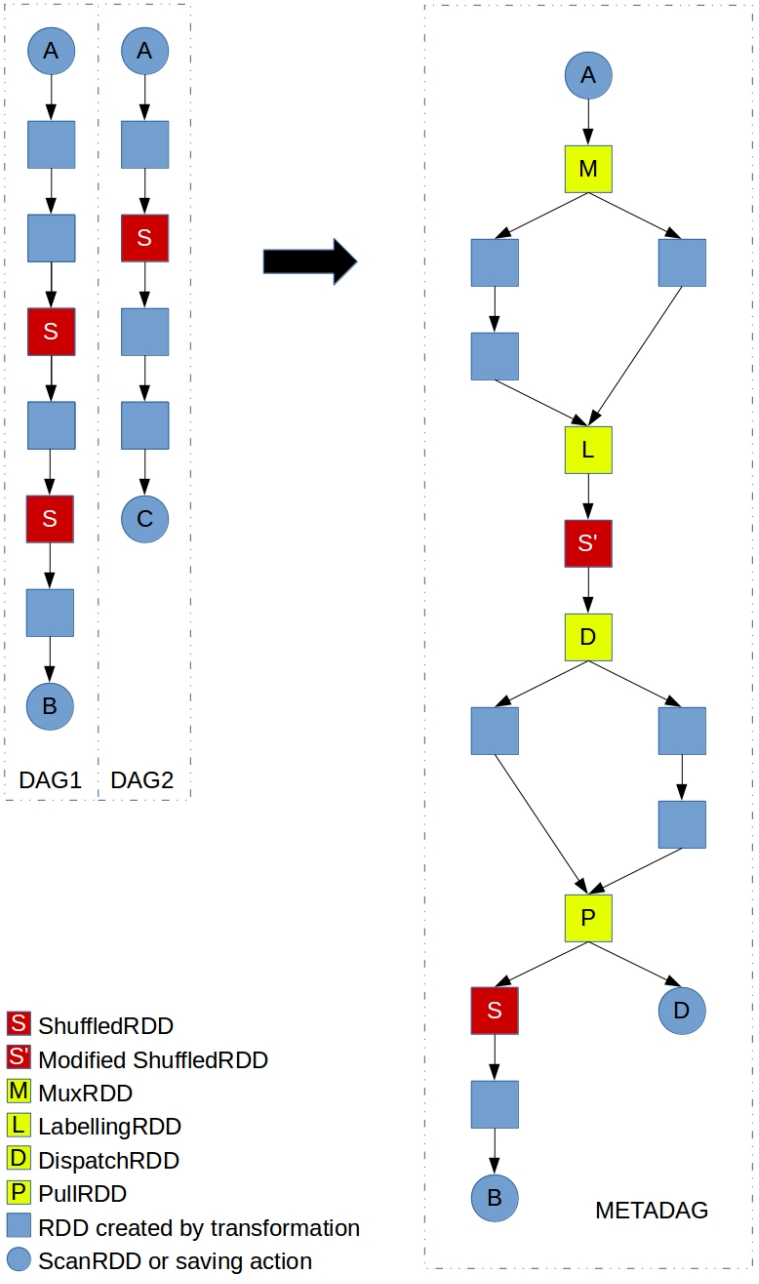

FIGURE 4.2: Transformation process of a complex case

pipeline by the label attached into each tuple. The PullRDD is inserted at last to do the pull and trigger the whole job, then saves the output to disk storage.

- Two jobs, one job has two ShuffleRDDs The part of DAG1 from the beginning up to the second ShuffledRDD and DAG2 are merged as the same way as the case above. The difference is that the PullRDD just saves a part of data comes from the pipeline of DAG2, PullRDD is also followed by the rest part of DAG1 and saves to disk storage when finishes. The process is described in figure 4.2

This technique is embedded into Spark Source Code and is an automatic feature of SparkSQL Server. In those two examples, we just discuss about two simple cases that can happen when using MRShare and simultaneous pipeline technique, but the more complex DAGs can also be merged into one meta job automatically.

In order to implement the simultaneous pipeline technique, we need to have a deep understanding of Spark internal and its dataflow. A modified Spark is needed to make our system work properly, so if user want to use our system, they need to use our provided Spark. Details are descibed in A

Chapter 5

Experimental Evaluation

This chapter's purpose is to verify the correctness and the efficiency of SparkSQL Server. Due to the limitation of time, in its current state the experiment is designed to be simple. We design the experiments on sharing scan with an input file of 10GB and various Window size of the DAG Queue at SparkSQL Server on two techniques: caching of Spark and sharing map input of MRShare.
From these experiments, we evaluate the performance of caching technique and also verify the performance of sharing map input of MRShare, which originally works on Apache Hadoop Mapreduce, on Apache Spark.

5.1 Experimental Setup

Our experimental evaluation is done on a YARN cluster with 17 nodes, each has 4 vcores and 6GB of RAM. Our Apache Spark is modified from version 1.3.1. All results shown in the following are the average of 5 runs: the standard deviation is smaller than 2.5 "%", hence – for the sake of readability – we omit error bars from our figures.
The input files we use are taken from Project Gutenberg, which provides a collection of full texts of public domain books. The size of the input file is 10GB. We take the common used WordCount program for both caching experiment and grep-WordCount for MRShare technique experiment since the paper on MRShare also used grep-WordCount to evaluate their works. In our experiment, with the grep-WordCount, we filter the

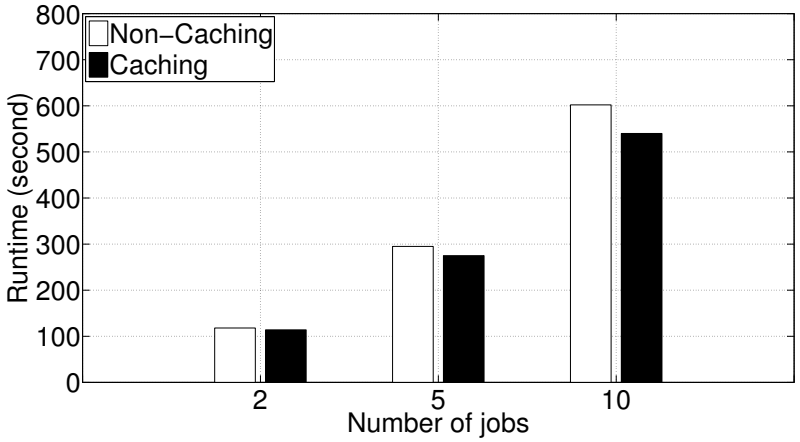

FIGURE 5.1: Performance comparison with jobs submitted by Spark-SQL Server and by users, using Caching

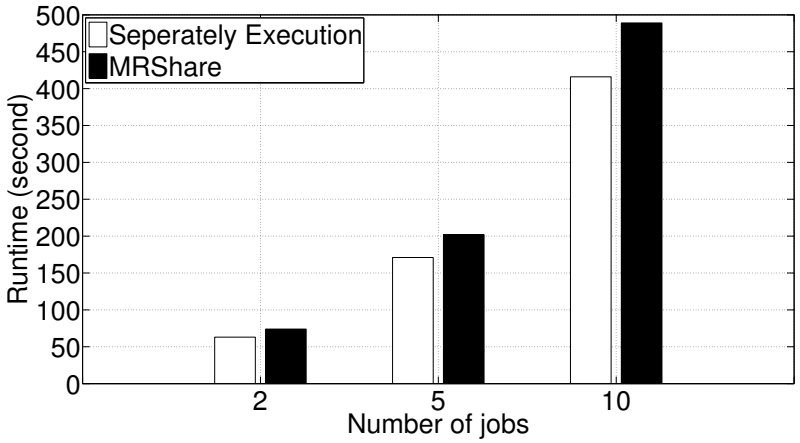

FIGURE 5.2: Performance comparison with job submitted by Spark-SQL Server and by users, using MRShare

length of each word to control the map output size of each job. We choose the Window Size of the Queue at SparkSQL Server with 2 then 5 and 10.

We compare the total runtime of batch of jobs after going through Spark-SQL Server and that batch of jobs which is executed one by one without using scan sharing optimization.

5.2 Results and Evaluations

With caching technique, as illustrated in figure 5.1 for number of jobs varies from 2 to 10, batch of jobs going through SparkSQL Server has

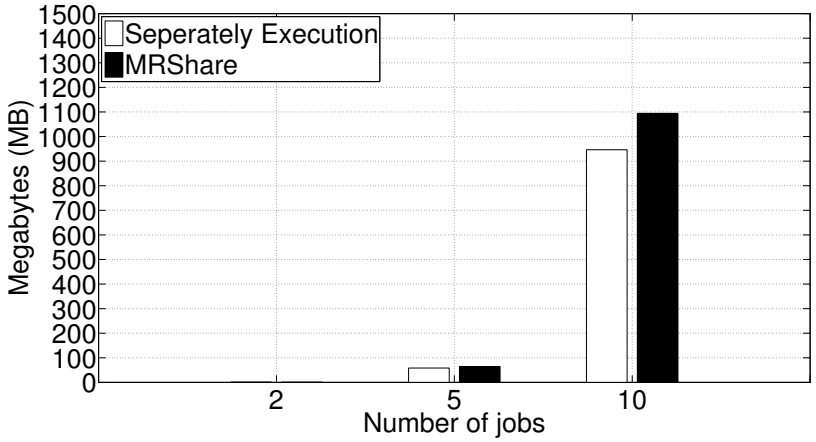

FIGURE 5.3: Shuffle bytes of seperately execution and MRShare

total runtime smaller than those jobs which are executed seperately. The saving using caching is not large in comparison to the total amount of time. This means that the sharing scan benefits do not dominant the sharing of computations. The SparkSQL Server functions well and gives the better performance rather than running those jobs submitted by users seperately.

Figure 5.2 describes the performance of jobs submitted by SparkSQL Server and by users, using MRShare sharing technique. With sharing map input technique of MRShare. The total runtime of batch of jobs submitted by SparkSQL Server is larger than the total runtimes of those jobs submitted separately by users. The larger number of jobs are merged into a group, the larger total runtime it is. There are two reasons to explain this result and they are not related to the SparkSQL Server:

- In Apache Spark, a tuple is immutable. So, when we do the tagging, we can not modify directly on the original tuple but need to generate a new tuple. We create too many objects and the Garbage Collector needs to take more time to do it job. MRShare produces 1 meta jobs for 2 jobs submitted, 2 meta jobs which are created by merging 2 and 3 seperated jobs, and 3 meta jobs which are created by merging 3, 3 and 4 seperated jobs. We sum the average of garbage collector runtime of those meta jobs and also sum the average of garbage collector runtime of seperated jobs. Table 5.1

TABLE 5.1: Garbage Collector Average Runtime

Number of jobs	Seperately executed	MRShare
2	20 ms	35 ms
5	85 ms	235 ms
10	622 ms	1369 ms

shows the average of Garbage Collector runtime between MRShare technique and executing jobs separately. The more jobs are merged into a single meta job, the larger the average of garbage collector runtime is. In Apache Hadoop MapRedue, the record is mutable so it can reduce the cost of Garbage Collector.

- The tagging operation is also a costly operation since larger data is shuffled over the network. Figure 5.3 illustrates the different of shuffle bytes over the network of seperately job execution and MRShare technique. Each tuple also carries a label inside it so the shuffle bytes of MRShare technique is always larger. For 2 jobs, the shuffle bytes difference between two techniques is not so much, but for 5 jobs and 10 jobs, the shuffle bytes difference can affect the performance.

We make a simple experiment by creating a Spark application that we manually replicate the input tuple and tag the labeling into it. The number of replications is also varied from 2 then 5 and 10. The total runtime of MRShare technique is equal or smaller than the total runtime of executing jobs seperately, which means that the tagging technique is a costly operation. The original works of MRShare does not include the tagging cost into their cost model although through experiments, we realize that the cost of tagging is not negligible.

The result of MRShare technique shows that the MRShare does not work as well as it does on Hadoop MapReduce since it is originally designed for MapReduce. In other words, the simultaneous pipeline technique that MRShare for Hadoop MapReduce uses is not suitable on Apache Spark due to the nature of each framework.

For caching approach, the results we obtain are not attractive since we just reduce the total execution time from 2 % to 10 %. For MRShare approach, the results are even worser due to the reasons we explain above. There is another reason which can explain our result: as a study on performance in data analytics frameworks [13] shows that for many simple jobs on Apache Spark, they are often bottlenecked on CPU and not I/O. That would explain more about the results we obtain. In addition, due to the limitation of time, we did not experiment enough on many kinds of workloads. We let this as a part of our future works.

From both experiments, the SparkSQL Sever proves its correctness. It also shows the extensibility of the system since we can easily add new sharing technique into the system.

Chapter 6

Contributions and Future Works

6.1 Contributions

The work presented in the thesis report, and the contributions that we made can be summarized as followed:

- We provide an overview of Apache Spark and Apache SparkSQL, insight a Spark Job Lifetime: by studying the fundamental components of Apache Spark and Apache SparkSQL, and the internal of Spark through the Spark Job Lifetime, we provide information about them and the documentation on those components, processes. Since documents about internal Apache Spark and SparkSQL are still limited, the information we provide would be helpful for many developers and researchers.
- We propose a new work sharing system. To the best of our knowledges, it is the first work sharing system on Apache Spark to deal with multiple query optimization. The advantages of our system is that it is generalized and extendable so users, developers can plug their own implementations for their work sharing types that they want.
- We provide the simplest form of work sharing which is sharing scan as an usecase with two techniques: caching of Spark and sharing

map input of MRShare. We implement the simultaneous pipeline technique which is widely used in MapReduce but no works have been found on Apache Spark.

- We conduct a preliminary evaluation to verify that the SparkSQL Server can funtion properly and also to verify the performance of MRShare technique on Apache Spark. Though the evaluation is far from being complete, it gives us the proof about the correctness and extensibility of the SparkSQL Server, which encourages the community to contribute new sharing techniques for various sharing types.

6.2 Future Works

In scope of the internship, due to the limitation of time, there are still some mixing points that we need to provide in the future.

- Better experiments and evaluation: due to time and resource limitations, our experimental results are not as comprehensive as they could be. We plan to run experiments that benchmark the performance in various aspects: the cluster size, the size and distribution of data, many kinds of workloads, the type of work sharing and the types of rewritten techniques.

- At client side, we need to stop the execution of the action that users call in their applications and send their DAGs to our SparkSQL Server. At server side, we integrate simultaneous pipeline technique into Spark. So, at both client side and server side need the modified Spark. This is a limitation of the project and we want to find out a solution to solve this problem.

- Propose a solution to improve the performance of sharing scan using MRShare. Some additions to the cost model would be needed since tagging cost is not negligible. In Apache Spark version 1.5, there is a big improvement on memory management. We want to port the whole project to use the latest Apache Spark to see if the MRShare technique is better or not.

- Caching is a nice feature and widely used in Apache Spark. Caching is not only useful for sharing scan but also useful for sharing computations. However, it is not always beneficial since in some cases when the memory can not hold the whole input file, the cost of reading from and writing to disk is not negligible. There is currently a work from my colleague [6] on improving caching performance. Instead of caching only the input file, we try to cache the most common parts of the jobs.

- Scan sharing is the simplest form of work sharing. Since scan sharing is only beneficial when the job is I/O intensive. There is a recently work [13] claims that on Apache Spark the bottleneck is not I/O but CPU. There are many kind of work sharing and each kind also has many techniques. Due to the time limitation, we just provide sharing scan as an usecase and use only two techniques for sharing scan. We plan to provide many work sharing kind such as: grouping set, join...

- In large data warehouses, the latency is an important metric which is closely related to the scheduler and scheduling strategies. In this internship, we just provide the simplest form of scheduling strategy which is FIFO. We plan to provide other kinds of scheduling strategies to improve the performance of our system or to fulfill the users' requirements.

- We plan to open source this project so many developers and researchers can contribute to the project.

Bibliography

[1] http://hadoop.apache.org.

[2] http://hive.apache.org.

[3] http://pig.apache.org.

[4] http://spark.apache.org.

[5] http://wiki.apache.org/pig/PigMultiQueryPerformanceSpecification.

[6] http://www.trongkhoanguyen.com/2015/09/work-sharing-famework-for-apache-spark-introduction.html.

[7] Thusoo Ashish, Sen Sarma Joydeep, Jain Namit, Shao Zheng, Chakka Prasad, Anthony Sureshm, Liu Hao, Wyckoff Pete, and Murthy Raghotham. Hive - A Warehousing Solution Over a MapReduce Framework. 2009.

[8] M.A Bayir, I.H Toroslu, and A Cosar. Genetic algorithm for the multiple query optimization problem. In *IEEE Transactions on Systems, Man, and Cybernetics*, 2007.

[9] A. Cosar, E.P Lim, and J Srivastava. Multiple query optimization with depth first branch-and-bound and dynamic query ordering. In *In Proceedings of the second international conference on Information and knowledge management*, 1993.

[10] Jeffrey Dean and Sanjay Ghemawat. MapReduce : Simplified Data Processing on Large Clusters. In *ACM OSDI*, 2004.

[11] AF Gates, Olga Natkovich, and Shubham Chopra. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. 2009.

[12] Giannikis Georgios, Alonso Gustavo, and Kossmann Donald. SharedDB: Killing One Thousand Queries With One Stone. 2012.

[13] Ousterhout Kay, Rasti Ryan, Ratnasamy Sylvia, Shenker Scott, and Chun Byung-Gon. Making Sense of Performance in Data Analytics Frameworks. 2015.

[14] Zaharia Matei, Chowdhury Mosharaf, Das Tathagata, Dave Ankur, Ma Justin, McCauley Murphy, J. Franklin Michael, Shenker Scott, and Stoica Ion. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. 2012.

[15] Armbrust Michael, S. Xin Reynold, Lian Cheng, Huai Yin, Liu Davies, K.Bradley Joseph, Meng Xiangrui, Kaftan Tomer, J. Franklin Michael, Ghodsi Ali, and Zaharia Matei. Spark SQL: Relational Data Processing in Spark. 2015.

[16] Isard Michael, Budiu Mihai, Yu Yuan, Birrell Andrew, and Fetterly Dennis. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. 2007.

[17] T et al Nikiel. MRShare: Sharing across multiple queries in MapReduce. 2010.

[18] C et al Olston. CoScan: Cooperative Scan Sharing in the Cloud. 2011.

[19] Christopher Olston, Benjamin Reed, and U. Srivastava. Pig latin: a not-so-foreign language for data processing. In *ACM SIGMOD*, 2008.

[20] T.K Sellis. Multiple-query optimization. 1988.

[21] Harizopoulos Stavros, Shkapenyuk Vladislav, and Ailamaki Anastassia. QPipe: A Simultaneously Pipelined Relational Query Engine. 2005.

Appendix A

How To Use SparkSQL Server

SparkSQL Server is an open source project so everyone can get it at the Github repository of our group [1]. The repository have two main folders:

- **spark-assembly**: the modified Spark source codes, we integrate necessary classes to make our SparkSQL Server work properly. We need to build the source codes to get the assembly Spark jar file.
- **sparksql-server**: the source code of SparkSQL Server. We also need to build the project to get the jar file. We can modify the parameters of SparkSQL Server inside class ServerConstant.

To bring SparkSQL in action, we need to know how to submit a Spark application to SparkSQL Server and how to deploy the server on the cluster:

- **At client side** Users write their Spark application normally as they often do. In stead of using the default Spark Library of Apache, users need to use our Spark Library from the Github repository. We need to modify the spark-submit command to provide information about SparkSQL Server. The new submit command that users need to use is:

[1] http://github.com/DistributedSystemsGroup/sparksql-server

```
./bin/spark-submit \
  --class <main-class>
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  --sparksql-server <ip,sparksql-server-port,jar-server-port>
  ... # other options
  <application-jar> \
  [application-arguments]
```

Listing A.1: bash version

The value for sparksql-server argument is splitted by comma, with the order of sparksql-serve IP, sparksql-server Port, jar-server Port.

- **At server side** SparkSQL Server is also a Spark Application, we just submit it as usual. The difference of SparkSQL Server to a normal Spark Application is it does not submit the applications immediately to the cluster but it waits until getting enough applications from users to do the optimizations.