RICE UNIVERSITY

Performance Analysis and Optimization of Apache Pig

by

Ruoyu Liu

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Alan L. Cox, Chair
Associate Professor of Computer Science
Associate Professor of Electrical and
Computer Engineering

T. S. Eugene Ng
Associate Professor of Computer Science
Associate Professor of Electrical and
Computer Engineering

John Mellor-Crummey
Professor of Computer Science
Professor of Electrical and Computer
Engineering

Houston, Texas
June, 2014
ABSTRACT

Performance Analysis and Optimization of Apache Pig

by

Ruoyu Liu

Apache Pig is a language, compiler, and run-time library for simplifying the development of data-analytics applications on Apache Hadoop. Specifically, it enables developers to write data-analytics applications in a high-level, SQL-like language called Pig Latin that is automatically translated into a series of MapReduce computations. For most developers, this is both easier and faster than writing applications that use Hadoop directly.

This thesis first presents a detailed performance analysis of Apache Pig running a collection of simple Pig Latin programs. In addition, it compares the performance of these programs to equivalent hand-coded Java programs that use the Hadoop MapReduce framework directly. In all cases, the hand-coded Java programs outperformed the Pig Latin programs. Depending on the program and problem size, the hand-coded Java was 1.15 to 3.07 times faster. The Pig Latin programs were slower for three reasons: (1) the overhead of translating Pig Latin into Java MapReduce jobs, (2) the overhead of converting data to and from the text format used in the HDFS files and Pig’s own internal representation, and (3) the overhead of the additional MapReduce jobs that were performed by Pig.

Finally, this thesis explores a new approach to optimizing the Fragment-replicated join operation in Apache Pig. In Pig’s original implementation of this operation, an identical in-memory hash table is constructed and used by every Map task. In contrast, under the optimized implementation, this duplication of data is eliminated through the use of a new interprocess shared-memory hash table library. Benchmarks
show that as the problem size grows the optimized implementation outperforms the original by a factor of two. Moreover, it is possible to run larger problems under the optimized implementation than under the original.
Contents

Abstract ii
List of Illustrations vii
List of Tables viii

1 Introduction 1
1.1 Background 2
   1.1.1 MapReduce 2
   1.1.2 Pig 3
1.2 Thesis and Contributions 4
1.3 Thesis Roadmap 5

2 Background 7
2.1 Hadoop 7
2.2 The Pig Latin Language 9
   2.2.1 Pig Latin and SQL 9
   2.2.2 Types in Pig Latin 11
   2.2.3 Operations in Pig Latin 12
2.3 Workflow of Pig 13
   2.3.1 Logical Plan Optimization Phase 14
   2.3.2 Compile Phase 15
   2.3.3 MapReduce Plan Optimization Phase 16
   2.3.4 Execution Phase 17
2.4 Example 17
   2.4.1 Script 17
   2.4.2 Logical Plan 18
   2.4.3 Physical Plan 18
   2.4.4 MapReduce Plan 19
3 Performance Analysis of Pig

3.1 Overall Performance

3.1.1 Experimental Setup

3.1.2 Benchmarks

3.2 Execution Breakdown

3.2.1 Fragment-replicated Join

3.2.2 Order by One field

3.2.3 Order by Multi-field

3.2.4 Multi-store

3.2.5 Distinct Aggregates

3.3 Possible Optimizations

4 Optimization of Fragment-replicated join in Pig

4.1 Shared-memory Hashjoin

4.1.1 Benchmarks

4.1.2 Analysis

4.2 Use Shared Memory Hashjoin in Pig

4.2.1 How Pig Uses the Hadoop Platform

4.2.2 Modification of POFRjoin

4.2.3 Benchmarks

5 Related Work

5.1 Related Systems

5.1.1 Hive

5.1.2 Sawzall

5.1.3 Enterprise Control Language (ECL)

5.2 Analysis of Hadoop and Pig

5.3 Performance Optimization on Hadoop

5.4 Related Work on Join

6 Conclusions and Future Work

6.1 Conclusions

6.2 Future Work
Bibliography
Illustrations

2.1 SQL version ................................................................. 10
2.2 Pig Latin version ......................................................... 10
2.3 Redundant fields ......................................................... 14
2.4 Reorder script ........................................................... 15
2.5 Merge jobs ................................................................. 16
2.6 Example script ........................................................... 18
2.7 Logical plan before optimization ................................... 19
2.8 Logical plan after optimization ..................................... 20
2.9 Physical plan ............................................................. 21
2.10 MapReduce plan before optimization ............................. 22
2.11 MapReduce plan after optimization ............................... 23

3.1 Script of order by one field ........................................... 30
3.2 Script of order by multi-field ......................................... 31
3.3 Script of distinct aggregates ................................. 33
Tables

3.1 PigMix benchmark .................................................. 25
3.2 Fragment-replicated join .......................................... 27
3.3 Order by one field ................................................... 29
3.4 Order by multi-field ................................................ 32
3.5 Multi-store ............................................................. 33
3.6 Distinct aggregates .................................................. 35

4.1 Running time on Hadoop ............................................. 37
4.2 Running time before and after optimization ..................... 40
4.3 FRJoin running time breakdown .................................. 41
Chapter 1

Introduction

Over the last two decades, the Internet and information digitalization have been growing rapidly. We are experiencing an ‘information avalanche’. As a result, a great amount of information from online resources, business and other organizations needs to be processed, analyzed and indexed. This has led to development of new large-scale distributed computing platforms such as Apache Hadoop, an open-source MapReduce framework written in Java.

Hadoop has received a great deal of attention from both academia and industry. More and more companies have begun using the Hadoop platform for data processing not only because it is open-source, but also because it can scale and it is fault tolerant. While it is easy for experienced Java programmers to write Hadoop programs, it is much harder for non-programmers to make use of the power of Hadoop. Making the power of Hadoop accessible to non-programmers requires higher-level programming abstractions. This led to the development of systems that provide abstractions on top of Hadoop.

My work focuses on one of these systems, Pig, which was invented for those users who are familiar with SQL. Though Pig was introduced back in 2008, there has been no detailed analysis of its performance. My work is the first to give a detailed analysis of the performance of Pig. In addition, this thesis explores a new approach to optimizing the Fragment-replicated join operation in Apache Pig. A new interprocess shared-memory hash table library was introduced to save time and
memory from building in-memory hash table in every mapper.

1.1 Background

1.1.1 MapReduce

MapReduce is a programming model announced by Google [1] in 2004. It is used as a tool to process and generate large data sets [1] using a large-scale cluster of machines. The execution of a MapReduce program can be divided into several steps:

- The MapReduce framework first splits input data into several pieces, the size of which can be configured by the user. There is a master machine that takes control of the execution of the program. It assigns tasks to worker machines.
- The workers that have been assigned map tasks are called mappers. The number of mappers is the same as the number of the input pieces. Every mapper reads a piece of input data in the form of key-value pairs calls the user-defined Map function that transfer a key-value pair into a new key-value pair. The intermediate outputs, which are also in key-value pair format, are buffered in the memory and are finally written to the local disk of the mappers.
- The master always keeps track of the locations of the intermediate outputs. When the master notifies the workers to do reduce tasks (reducers), it will send them the location information of the intermediate outputs.
- After the reducers get the location information from the master, they begin to read data from those locations. When a reducer finishes reading all the intermediate data it needs, it begins to sort all the data based on the key so that all the values from the same key are grouped together.
- For each key of the sorted intermediate data, the reducer passes it with its set
of values to the user-defined *Reduce* function. The outputs are directly written into a global file system.

The MapReduce framework is fault tolerant. When a worker fails, the master will reassigns the that worker’s tasks to other workers and restart those tasks. The master writes periodic checkpoints of its status to make it easy to start from a checkpoint when the master node fails.

At the same time that the MapReduce framework provides a distributed computing model with scalability and fault tolerant, it hides the implementation details of the complex system and makes it easy to use. This is the reason for its popularity. Not long after Google revealed MapReduce, Hadoop, an open-source implementation of MapReduce made this idea even more widely-used.

### 1.1.2 Pig

Hadoop and MapReduce have been widely used for processing data. Many companies run MapReduce jobs on tens of thousands of machines. But when developers try to create MapReduce programs to compute data, they need to write map and reduce in a procedural programming language like Java. However, not every developer is familiar with procedural programming languages like java. Fortunately, many of the tasks used by non-programmers are basic ETL (Extract-transform-load) operations that use common operators such as filters, projections, and join. Thus it is possible to create a higher-level platform that will automatically generate MapReduce programs. Apache Pig is one such system. It defines a high-level data-flow language and provides an execution framework for parallel computation based on Hadoop. Apache Pig has two parts:

- **Pig Latin**: A SQL-like language that hides the details of writing MapReduce
computations from programmers. Unlike SQL, Pig Latin is procedural, and thus it is easier to express computations with pipelined data flow.

- **Pig Engine**: the platform that compiles, optimizes and executes Pig Latin as a series of MapReduce jobs.

Overall, Pig can be regarded as a higher-level interface. It makes it easier for developers to write complex MapReduce jobs. Instead of writing multiple stages of MapReduce transformations, developers can use the simpler Pig Latin scripting language to accomplish the same thing. Besides, it is flexible and extensible. User can write customized functions with Java, Python and other languages as well.

## 1.2 Thesis and Contributions

This thesis is the first detailed performance analysis of Apache Pig. Although Apache Pig was introduced in 2008 and has been widely used by both academia and industry, to the best of my knowledge, there is no published work analyzing its performance. My work presents a detailed performance analysis of Apache Pig running a collection of simple Pig Latin programs. In addition, it compares the performance of these programs to equivalent hand-coded Java programs that use the Hadoop MapReduce framework directly.

The experimental results in this thesis show that there exists a substantial performance gap between Pig Latin and hand-coded Java MapReduce programs. In all cases, Pig Latin programs were outperformed by hand-coded Java program. Depending on the program and problem size, Pig Latin was 1.15 to 3.07 times slower. Through a detailed breakdown of the program execution, I conclude that there are three reasons for the slowness of Pig Latin programs: (1) the overhead of translating Pig Latin into Java MapReduce jobs, (2) the overhead of converting data to and from
the text format used in HDFS files and Pig’s own internal representation, and (3) the overhead of additional MapReduce jobs that were performed by Pig. In some applications, this gap could be reduced with modest changes in the implementation of Pig.

My work also explores a new approach to optimizing the Fragment-replicated join operation in Apache Pig. In Pig’s original implementation of this operation, an identical in-memory hash table is constructed and used by every Map task. In contrast, under the optimized implementation, this duplication of data is eliminated through the use of a new interprocess shared-memory hash table library. Benchmarks show that as the problem size grows the optimized implementation outperforms the original by up to a factor of two. Moreover, it is possible to run larger problems under the optimized implementation than under the original.

The optimization of Fragment-replicated join in Pig explored in this thesis does not need any changes to other parts of Pig or Hadoop. It is also transparent to users of Pig Latin. In principle, it should be possible to apply this optimization to similar systems such as Hive and Sawzall.

1.3 Thesis Roadmap

This thesis consists of the following chapters:

Chapter 2 introduces Hadoop implementation of the MapReduce programming model and then describes Apache Pig - a system that supports a higher-level notation for expressing computations to be executed as a collection of MapReduce operations.

Chapter 3 provides a detailed performance analysis of several Pig applications.

Chapter 4 describes how I optimized the Fragment-replicated join in Pig using an interprocess shared memory library. In addition, this chapter also gives benchmarks
and explanations of the performance difference between the original and optimized implementation of Fragment-replicated join in Pig.

Chapter 5 discusses related systems like Hive and Sawzall. It also discusses related work on the optimization and performance analysis of Hadoop and join operations.

Chapter 6 summarizes this thesis and discusses possible future work.
Chapter 2

Background

The most attractive feature of Pig is that it can translate all kinds of SQL operations to MapReduce jobs. Pig has several built-in operations from which all operations in Pig are constructed. The system starts from parsing a Pig Latin script and ends with launching a set of MapReduce jobs. This chapter will first introduce the underlying platform of Pig – Hadoop and then focus on the general workflow of Pig.

2.1 Hadoop

Apache Hadoop is an open-source project which develops open-source software for reliable, scalable, distributed computing. Hadoop system consists of four modules:

- **Hadoop Common**: The common utilities that support the other Hadoop modules
- **Hadoop Distributed File System [2]**: A distributed file system that provides high-throughput access to application data
- **Hadoop YARN**: A framework for job scheduling and cluster resource management
- **Hadoop MapReduce**: A YARN-based system for parallel processing of large data sets

The Hadoop system has lots of similarities with Google MapReduce [1] and Google File System [3] and can be considered as an open-source implementation of MapRe-
duce and GFS.

There are five daemons in Hadoop that manage job execution and data storage:

- **NameNode:** The NameNode is the center of HDFS [2]. It keeps the directory tree of all files in the file system and also tracks data movement. NameNode is also the first node clients talk to whenever they want to add/delete/modify any file in the file system.

- **SecondaryNameNode:** The SecondaryNameNode is not the backup of NameNode, insteads, it helps NameNode maintain checkpoints of HDFS.

- **DataNode:** A DataNode stores data in the HDFS. There can be multiple DataNodes in a Hadoop cluster with data replicated across them. On startup, a DataNode connects to the NameNode; spinning until that service comes up. It then responds to requests from NameNode for filesystem operations. When a client make a request to HDFS, it first talks to NameNode to get location information for the data it requests. After that, it talks to the certain DataNode(s) that contains the data it wants.

- **JobTracker:** The JobTracker is where a client talks to when it submits a job to Hadoop cluster. After receiving the job, JobTracker first talks to NameNode to locate the data that will be used in the job and then locates a collection of TaskTrackers that are at or near the data. The JobTracker then submits tasks to those TaskTrackers and monitors and schedules the task execution until the job succeeds or fails. There is only one JobTracker in a Hadoop cluster.

- **TaskTracker:** TaskTrackers are the ones that actually perform the work. A TaskTracker periodically sends heartbeats to JobTracker and will be assigned
tasks whenever needed. When TaskTracker receives a task, it will spawn a separate JVM to do the actual work. It will report to JobTracker when the task succeeds or fails.

Hadoop is adopted in a variety of fields to deal with large-scale data sets. Hadoop’s success is a function of both the popularity of MapReduce in addition to the fact that Hadoop is available as open-source.

2.2 The Pig Latin Language

Pig Latin is a dataflow language \[4\]. It describes how data being processed through different stages. There is at least one input source of the data and at least one output location where the final results should be written.

The syntax of Pig Latin is straightforward. The are only two kinds of statements: a) Generate a new data object from a source or an existing data object by some operations. b) Store or output a data object to screen or to a location. Pig does not support conditional statements or loops. Users can assume the data is processed by the statements in the script in order.

2.2.1 Pig Latin and SQL

Pig Latin is similar to SQL in some ways. However, the central idea of these two languages is different. SQL is a declarative language and it mainly focuses on what results it should get instead of how it gets those results. Pig Latin, on the other hand, is a procedural language and focuses on data pipelines and how we perform in each pipeline. Consider the following example. Figure 2.1 is the SQL version implementation and Figure 2.2 is the Pig Latin implementation. We can see that instead of confusing subqueries, Pig Latin expresses a simple pipeline to perform the
required set of operations. Unlike SQL expressions, each intermediate result is readily available, which makes it easier to reuse data.

There are also other differences between Pig Latin and SQL:

- **Pig Latin is easy to optimize.** In SQL, by the definition of declarative language, users can specify what should be done instead of how it should be done. This forces developers to trust the optimizer instead of controlling the optimization themselves.

In Pig Latin, users have full knowledge of how the data being processed and users can choose to specify the best way to process data. For example, the join operation. Users can specify to use default hash join, merge join, Fragment-replicated join, or any other kind of join operations.
• **Splits in pipelines.** SQL has single result for a certain operation and this make it hard to split a data pipeline into several ones. Pig Latin, on the other hand, can easily split a pipeline into several pipelines, process those pipelines and output several results. A good example is do ‘GROUP BY’ based on different fields on one table. Pig Latin can easily split the data pipeline and operates ‘GROUP BY’ based on different fields. The Pig engine will further optimize this kind of operation to make it more efficient (as we describe in section 2.3.3).

• **User-defined function (UDF).** Pig allows users to include their own implementations at any point in the processing. Users can specify how the data is loaded, processed and stored. Since the data is processed in a pipeline and each statement corresponds to one stage in the pipeline, users can specify their own ways in each stage.

### 2.2.2 Types in Pig Latin

Pig’s data types can be divided into two categories: *scalar* types, which contain a single value, and *complex* types, which contain other types [4].

**Scalar Types**

The scalar types in Pig are similar to simple types in other programming languages including *int, long, float, double, chararray* and *bytearray*. All the types except *bytearray* are all represented in Pig interfaces by *java.lang*, making it easy to write UDFs.

**Complex Types**

In order to process complex types as *key-value* pairs, Pig maintains three complex types: maps, tuples and bags. These three types can contain data of any type includ-
ing themselves.

- **Map** is a mapping from a *chararray* to any Pig type.

- **Tuple** is a ordered collection of Pig data fields with fixed length. It can corresponds to a row in SQL.

- **Bag** is an unordered collection of tuples. Pig does not provide collection types like list or set in Java. Bag can be used as a substitution.

### 2.2.3 Operations in Pig Latin

There are two categories of built-in operations in Pig Latin: Input and Ouput operations, Relational operations. For the first category, there are three operations: **Load**, **Store** which load and store data, and **Dump** which print the data to console.

There are 8 built-in Relational operations:

- **Foreach** takes a set of expressions and applies them to every record in the data pipeline.

- **Filter** retains the records that satisfied the filter statement in the data pipeline.

- **Group** collects records with the same key into a **Bag**.

- **Order By** sorts the data as another table.

- **Distinct** removes duplicated records.

- **Join** joins two tables by certain fields.

- **Limit** truncates the output to certain lines.

- **Sample** samples the data according to a certain percentage.

Pig users can also define their own operations and use them in Pig Latin.
2.3 Workflow of Pig

The execution of a Pig program contains several phases:

- **Parse Phase:** In this phase, the system will read in Pig Latin script and build an initial logical plan. This phase has two main functions:
  - **Verification:** Verify the correctness of the script as well as the legality of user-defined functions.
  - **Parse:** The parser will parse each statement in the Pig Latin script into a logical operator. At the end of parsing, the parser will return a DAG as the logical plan.

- **Logical Plan Optimization Phase:** The logical plan generated by previous phase is passed to a logical plan optimizer where optimizations like rearrangement and projection pushdown are been executed. (More examples will be introduced in 2.3.1)

- **Compile Phase:** In this phase, the system will compile the optimized logical plan into a series of MapReduce jobs.

- **MapReduce Plan Optimization Phase:** The MapReduce Plan generated by previous phase is passed through a MapReduce plan optimizer. Optimizations may be performed in different ways as will be mentioned in 2.3.3

- **Execution Phase:** In this phase, the system will take optimized MapReduce plan and generate MapReduce jobs and let Hadoop execute then while monitoring the process.

In the rest of this section, I describe some of these phases in detail.
2.3.1 Logical Plan Optimization Phase

Optimization of the logical plan aims mainly to reduces the amount of data that being shuffled and written as temporary files. There are several ways to obtain these targets:

- Adjust the sequence of operators. Sometimes the sequence of the some operators will not affect the final results of the job. For example the ‘Foreach’ operator followed by the ‘Filter’ operator. It will not change the final results if we only swap the two operators. However, by doing so we can reduce the size of data that ‘Foreach’ will take to process.

- Cut out fields. Not every user of Pig will write efficient script. It may happen sometimes that some fields in the script are never used. If we cut out those fields, we can reduce the size of data being processed by much. The example in Figure 2.3 shows this senario. We can cut out field ‘uname’ and ‘time’ since those are never used during the process.

- Put limit operators as early as possible. If we can reorder a plan to put a limit operator right after a load operator, we can apply the limitation while loading in data which will reduce the input data as well as following intermedia data.

```java
data = LOAD 'dataset' AS (id, uname, time, type_machine) 
group = GROUP data BY id 
result = FOREACH order GENERATE id, COUNT(type_machine) AS cnt 
STORE result INTO 'result'
```
```sql
data = LOAD 'dataset' AS (id, uname, time, type, machine)
newdata = FOREACH data GENERATE id#, name, time, type, machine
first10 = LIMIT group 10
STORE first10 INTO 'result'
```

Figure 2.4: Reorder script

The example in Figure 2.4 shows this scenario. We can put the limit operator right after load operator since we only need first 10 records of the data.

- Put operators that generate data as late as possible. For example, the cross product will generate lots of data. If we can postpone this operation to the end of the job, the intermediate data size before that will be less.

### 2.3.2 Compile Phase

In the compile phase, the logical plan is first translated into a physical plan. The physical plan is called for the name of the physical operators in it. These operator will be executed during the running process. A logical operator will be translated to one or a combination of several physical operators. For example, the ‘GROUP’ logical operator can be divided into three physical operators: ‘Local Rearrange’, ‘Global Rearrange’ and ‘Package’.

The physical plan is then translated into MapReduce plan, based on which the MapReduce jobs are generated. The work done here is to put different physical operators into one or more MapReduce jobs. In the meantime, some physical operators are implemented by Hadoop framework, so it is natural to use them as the function
```sql
-- Load dataset
data = LOAD 'dataset' AS (field0, field1, field2, field3)

-- Group data by field0 and store into 'group0'
group0 = GROUP data BY field field0
STORE group0 INTO 'group0'

-- Group data by field1 and store into 'group1'
group1 = GROUP data BY field field1
STORE group1 INTO 'group1'

-- Group data by field2 and store into 'group2'
group2 = GROUP data BY field field2
STORE group2 INTO 'group2'

-- Group data by field3 and store into 'group3'
group3 = GROUP data BY field field3
STORE group3 INTO 'group3'
```

Figure 2.5: Merge jobs

unit of physical operators. A good example is ‘LOAD’, we need not to do anything else but let Hadoop framework do the loading for us.

### 2.3.3 MapReduce Plan Optimization Phase

The optimization in MapReduce plan is mainly aiming to reduce the number of MapReduce jobs and to decide the use of combiners.

The reason for reducing the number of MapReduce jobs is that the overhead of a starting a MapReduce job can be as long as 10 seconds and if we can reduce the number of MapReduce jobs, it means we can reduce the time we read in data. The example in Figure 2.5 shows one scenario that we can reduce the number of MapReduce jobs. There are four STORE statement here and should lead to four MapReduce jobs. However, since these four GROUP By operators in the four MapReduce jobs are using the same data ‘data’. The optimizer will schedule only one MapReduce job instead of four.

The use of combiners are also one possible optimization during this phase. It can
utilize the combiner stage to perform early partial aggregation for those algebraic functions.

\section*{2.3.4 Execution Phase}

In the execution phase, the optimized MapReduce plan is presented as a DAG of MapReduce jobs. These jobs are then topologically sorted and processed in that order. The system will first generate a JobControl for each job containing all the configuration information of that job. The system will also generate a Jar file which includes all the mappers and reducers Pig implements and PhysicalOperator classes which implement the functions in Pig (basically every PhysicalOperator has its own class except those whose function are covered by Hadoop framework). Pig will monitor the whole process of execution and log information as needed.

\section*{2.4 Example}

In this section, I will present an example Pig Latin script and use this script to present how Pig Latin scripts are being executed.

\subsection*{2.4.1 Script}

The script in Figure \ref{fig:script} is the Pig Latin script I want to use as an example.

The first and second statements are load operations. In the first statement, by ‘Using PigStorage(,)', the system will use a comma as field separator. Users can also define their own functions for loading data as the second statement shows. The third statement is a join operation. Instead of using the default strategy for join, user can assign certain strategy of join by USING statement. The potential strategies are: merge join, skew join and Fragment-replicated join. The fourth and fifth statements
Figure 2.6 : Example script

are filter statements. The last four statements perform the group by operation and store the results into different places.

2.4.2 Logical Plan

The logical plan produced by the parser is shown by Figure 2.7 Nine statements correspond to nine logical operators. The number in each unit corresponds to the line number in the script. However, just as stated in 2.3.1, if we put those two filters right after the load operation, we can reduce the amount of data being processed. Figure 2.8 shows the logical plan after optimization.

2.4.3 Physical Plan

In the physical plan, each logical operator is presented as one or more physical operators. For example, the JOIN operator is presented as four physical operators: Local Rearrange, Global Rearrange, Package and Foreach. As Figure 2.9 shows, there are 18 physical operators generated from nine logical operators.
2.4.4 MapReduce Plan

In the MapReduce plan, the physical operators in physical plan are grouped with bounds for MapReduce jobs. Some of the operators are implemented by Hadoop framework and thus omitted. Figure 2.10 shows the MapReduce plan before optimization. The first map phase should contain three kinds of physical operators. However, the ‘LOAD’ operator is implemented by Hadoop framework, so the first map phase contains only ‘FILTER’ and ‘LOCAL REARRANGE’. There is no ‘GLOBAL REARRANGE’ operator in MapReduce plan because it is implemented by the merge and shuffle phase in Hadoop framework. The first reduce phase contains the remain-
ing operators of ‘JOIN’ operation. After the ‘JOIN’, there are two ‘GROUP BY’ operations which should correspond to two MapReduce jobs. Since there is no dependence between these two jobs, I put them in the same level under the proceeding ‘JOIN’ operation. Like the ‘JOIN’ operation, there is no ‘GLOBAL REARRANGE’ operator when converted into MapReduce plan. Whether a physical operator is in map phase or reduce phase is not fixed, but depends on the surrounding operators and operations. For example, the Fragment-replicated join is located in either a map phase or a reduce phase since it only makes use of half of a MapReduce job.

The MapReduce plan level optimization combines the last two MapReduce jobs together as one as shown by Figure 2.11.
Figure 2.9 : Physical plan
Figure 2.10: MapReduce plan before optimization
Figure 2.11: MapReduce plan after optimization
Chapter 3

Performance Analysis of Pig

3.1 Overall Performance

To evaluate the overall performance of Pig jobs, I use a set of benchmarks, PigMix, which use a set of queries to test performance of Pig to test the overall performance of Pig. PigMix also provides a set of Hadoop Java programs to run equivalent map reduce job directly without Pig. The idea of PigMix is to evaluate the performance gap between Pig and hand-coded Hadoop programs in Java.

3.1.1 Experimental Setup

The experiments were done on a 3-node Hadoop cluster on campus. The Jobtracker runs on a node with 4 cores (8 hardware threads) and 12GB memory. The Tasktrackers run on nodes with 2 cores (2 hardware threads) and 4 GB memory. Java 6 and Hadoop 1.2.1 are used running Pig 0.13.0.

3.1.2 Benchmarks

The benchmarks from the experiments are shown in Table 3.1. We define multiplier as the ratio between Pig running time and Java running time. As we can see, Pig programs are constantly outperformed by Java MapReduce programs. In some operations, the multiplier is as high as 3.07. In the next section, I introduce the detailed execution breakdown of several operations to discuss the overhead introduced by Pig.
Table 3.1: PigMix benchmark

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Pig running time (s)</th>
<th>Java running time (s)</th>
<th>Multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flatten data</td>
<td>96</td>
<td>59</td>
<td>1.63</td>
</tr>
<tr>
<td>FR join</td>
<td>85</td>
<td>33</td>
<td>2.58</td>
</tr>
<tr>
<td>Join</td>
<td>115</td>
<td>58</td>
<td>1.98</td>
</tr>
<tr>
<td>Nested distinct</td>
<td>76</td>
<td>47</td>
<td>1.62</td>
</tr>
<tr>
<td>Anti-join</td>
<td>67</td>
<td>48</td>
<td>1.40</td>
</tr>
<tr>
<td>Group by large key</td>
<td>76</td>
<td>46</td>
<td>1.65</td>
</tr>
<tr>
<td>Nested split</td>
<td>72</td>
<td>42</td>
<td>1.71</td>
</tr>
<tr>
<td>Group all</td>
<td>77</td>
<td>47</td>
<td>1.64</td>
</tr>
<tr>
<td>Order by one field</td>
<td>146</td>
<td>66</td>
<td>2.21</td>
</tr>
<tr>
<td>Order by multi-field</td>
<td>230</td>
<td>75</td>
<td>3.07</td>
</tr>
<tr>
<td>Distinct and Union</td>
<td>245</td>
<td>163</td>
<td>1.50</td>
</tr>
<tr>
<td>Multi-store</td>
<td>87</td>
<td>49</td>
<td>1.78</td>
</tr>
<tr>
<td>Outer join</td>
<td>67</td>
<td>47</td>
<td>1.43</td>
</tr>
<tr>
<td>Merge join</td>
<td>111</td>
<td>54</td>
<td>2.06</td>
</tr>
<tr>
<td>Distinct aggregates</td>
<td>76</td>
<td>53</td>
<td>1.43</td>
</tr>
<tr>
<td>Accumulative mode</td>
<td>67</td>
<td>44</td>
<td>1.52</td>
</tr>
<tr>
<td>Wide key group</td>
<td>152</td>
<td>109</td>
<td>1.39</td>
</tr>
</tbody>
</table>

3.2 Execution Breakdown

For every execution of Pig Latin program, there are always two steps: First, Pig compiles the script and translates it to a set of MapReduce jobs. The second step is execution. However, the first step takes as short as one to two seconds and can be omitted when execution time is long enough.

Each job execution has its own overheads. Pig spends three to four seconds creating the Jar file containing all the class files that may be used during job execution and pass it to DistributedCache of HDFS. Also, each job can be further divided into
three or four phases: Setup, Map, Reduce (if the job is not Map-only job) and Cleanup. Map tasks start after the Setup is finished. Cleanup starts after all the Map and Reduce tasks are finished.

In this section, I demonstrate execution breakdowns on several operations that reveal different types of overheads brought by Pig.

### 3.2.1 Fragment-replicated Join

Fragment-replicated join is Pig’s version of hash join. The smaller table, also known as ‘lookup table’ is replicated on every work node. The larger table, also known as ‘join table’ is fragmented into splits. Each of the mapper will take the ‘lookup table’ to build a hash table in memory and stream in one split of the ‘join table’ to do the join operation.

Table 3.2 presents time usage and percentage of different parts of the execution. From the data we can see that the actual time used to do the join operation (Map-only tasks for Job2) is 32.6s which occupies only 38.4% of the total running time. The 32.6s running time is reasonable since Table 3.1 shows that the execution time of Java Hadoop is 33. However, it is not acceptable to spend 61.6% of total time on overheads. Here are some analysis on the two main overheads:

**Extra Job**

There are two MapReduce jobs for Fragment-replicated join: a) a Map-Only job that reads in the replicated table(s) using Pig’s own read in format, and b) a Map-Only job that perform the join operation. The first job reads all the input data and writes them back to HDFS as a modest number of large files. The first job reduces the burden of the Namenode when there are too many tiny files used as replicated table.
Table 3.2: Fragment-replicated join

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subphases</th>
<th>Time (s)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>Connect to Hadoop</td>
<td>0.7</td>
<td>0.8%</td>
</tr>
<tr>
<td></td>
<td>Compile to MapReduce jobs</td>
<td>1.1</td>
<td>1.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.8</td>
<td>2.1%</td>
</tr>
<tr>
<td>Job1</td>
<td>Create Jar for Job1</td>
<td>3.4</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job1</td>
<td>0.8</td>
<td>0.9%</td>
</tr>
<tr>
<td></td>
<td>Setup Job1</td>
<td>11</td>
<td>12.9%</td>
</tr>
<tr>
<td></td>
<td>Map-only task for Job1</td>
<td>4.3</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job1</td>
<td>3.3</td>
<td>3.9%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22.8</td>
<td>26.8%</td>
</tr>
<tr>
<td>Communicate</td>
<td>Communicate between Jobs</td>
<td>1.4</td>
<td>1.6%</td>
</tr>
<tr>
<td>Job2</td>
<td>Create Jar for Job2</td>
<td>3.4</td>
<td>4%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job2</td>
<td>1.1</td>
<td>1.3%</td>
</tr>
<tr>
<td></td>
<td>Setup Job2</td>
<td>13.7</td>
<td>16.1%</td>
</tr>
<tr>
<td></td>
<td>Map-only tasks for Job2</td>
<td>32.6</td>
<td>38.4%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job2</td>
<td>5.2</td>
<td>6.1%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>56</td>
<td>65.9%</td>
</tr>
<tr>
<td>Final synchronization</td>
<td>Final synchronization</td>
<td>3</td>
<td>3.5%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>85</td>
<td>100%</td>
</tr>
</tbody>
</table>

Not every case needs the extra job. The test case we use, for example, only contains a single file for replicated table. However, there is no way to avoid the first job which occupies 26.7% of the total time in the current implementation.

One potential solution to this problem is to add a decision branch when generating MapReduce plan when the number of files being that hold the contents of the replicated table is small.
Setup For Jobs

Setup for the two MapReduce jobs occupies 29% of total time, which is the largest overhead in this breakdown. Most of the time for setup is spent on unpacking the jar file passed to every worker node by DistributedCache. It takes every MapReduce job started by Pig eight seconds to unpack 4867 class files from the job’s jar file while it only takes a hand-coded Java MapReduce job 0.6 second to generate 355 class files from the job’s jar file.

There are two options to solve the problem. a) modify Hadoop so that each work node only downloads and unpack the same jar file once, or b) modify Pig so that it only packs necessary class files into job’s jar file.

Inefficiency in Fragment-replicated Join  My experiments on Fragment-replicated join showed that this operation is inefficient in both time and memory usage. This inefficiency prevents it from scaling well when data size grows. In the next chapter, I introduce an optimization that addresses this problem.

3.2.2 Order by One field

This operation reorders the input the data according to a single field.

Table 3.3 presents time usage and percentage of different parts of the execution. From the data we can see that the actual time used to do the operation (Map and reduce tasks for Job2) is 66.8s which occupies 45.6% of the total running time. The overhead of setup for both jobs are the same as Fragment-replicated join. Here is the analysis on the other overhead in this operation:
Table 3.3: Order by one field

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subphases</th>
<th>Time (s)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>Connect to Hadoop</td>
<td>1.1</td>
<td>0.75%</td>
</tr>
<tr>
<td></td>
<td>Compile to MapReduce jobs</td>
<td>1.1</td>
<td>0.75%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.2</td>
<td>1.5%</td>
</tr>
<tr>
<td>Job 1</td>
<td>Create Jar for Job 1</td>
<td>3.5</td>
<td>2.4%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job 1</td>
<td>1.2</td>
<td>0.8%</td>
</tr>
<tr>
<td></td>
<td>Setup Job 1</td>
<td>13.4</td>
<td>9.15%</td>
</tr>
<tr>
<td></td>
<td>Map and reduce tasks for Job 1</td>
<td>33.3</td>
<td>22.75%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job 1</td>
<td>2.8</td>
<td>1.9%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>54.2</td>
<td>37%</td>
</tr>
<tr>
<td>Communicate</td>
<td>Communicate between Jobs</td>
<td>0.6</td>
<td>0.4%</td>
</tr>
<tr>
<td>Job 2</td>
<td>Create Jar for Job 2</td>
<td>3.3</td>
<td>2.25%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job 2</td>
<td>0.9</td>
<td>0.6%</td>
</tr>
<tr>
<td></td>
<td>Setup Job 2</td>
<td>13.6</td>
<td>9.3%</td>
</tr>
<tr>
<td></td>
<td>Map and reduce tasks for Job 2</td>
<td>66.8</td>
<td>45.6%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job 2</td>
<td>3</td>
<td>2.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>88.3</td>
<td>60.3%</td>
</tr>
<tr>
<td>Final synchronization</td>
<td>Final synchronization</td>
<td>1.8</td>
<td>1.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>146.4</td>
<td>100%</td>
</tr>
</tbody>
</table>

**Extra Job**

Similar to Fragment-replicated join, there are also two MapReduce jobs involved in Order-by-one-field. The extra job (the first job) is used to sample the input data to get the distribution of the input data so that the default partitioner used by Pig can assign data evenly to reducers in the second MapReduce job.

The hand-coded Java MapReduce program, on the other hand, only uses one MapReduce job with a customized partitioner. However, the hand-coded Java MapRe-
A = load 'part0' using org.apache.pig.test.udf.storefunc.
   PigPerformanceLoader()
   as (user, action, timespent, query_term, ip_addr, timestamp,
       estimated_revenue, page_info, page_links);
B = order A by query_term;
store B into 'L9out';

Figure 3.1: Script of order by one field

duce program is able to do this because the programmer has the knowledge of the type
of the field so that a customized comparator is easy to implement. Pig’s partitioner,
on the other hand, could not handle all data types without instructions.

A potential solution of this problem is to customize one partitioner for each of the
data type in Pig and include them in MapReduce jobs.

3.2.3 Order by Multi-field

This operation reorders input data according to multiple fields.

Table 3.4 shows the running time breakdown of order by multiple keys. From
the data we can see that the actual time used to do the operation (Map and reduce
tasks for Job3) is 74.5s which occupies only 32.4% of the total running time. The
setup overhead and the overhead of second job (which samples the input data) are
the same as Order by one field. Here is the explanation on the other overhead in this
operation:
A = load 'part0' using org.apache.pig.test.udf.storefunc.
    PigPerformanceLoader()
    as (user, action, timespent: int, query_term, ip_addr, timestamp,
        estimated_revenue: double, page_info, page_links);
B = order A by query_term, estimated_revenue desc, timespent;
store B into 'L9out';

Figure 3.2: Script of order by multi-field

Extra Job

The extra job is related to the loading statement of the script. Figure 3.1 shows the
script of Order by one field while Figure 3.2 shows the script of Order by multi-field.
We can see that the latter one has two type casts for field timespent and estimated_revenue. This triggers an extra MapReduce job only to cast the type.

As I modified the script of Order by multi-field to delete the type cast, the total running time became 155 seconds, which was 74.8 seconds faster.

A potential solution of this problem is to make the loader of Pig more sophisticated
and change the logic of MapReduce plan to make use of the new loader.
Table 3.4: Order by multi-field

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subphases</th>
<th>Time (s)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>Connect to Hadoop</td>
<td>0.8</td>
<td>0.3%</td>
</tr>
<tr>
<td></td>
<td>Compile to MapReduce jobs</td>
<td>0.9</td>
<td>0.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.7</td>
<td>0.7%</td>
</tr>
<tr>
<td>Job1</td>
<td>Create Jar for Job1</td>
<td>3.5</td>
<td>1.5%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job1</td>
<td>0.9</td>
<td>0.4%</td>
</tr>
<tr>
<td></td>
<td>Setup Job1</td>
<td>10.0</td>
<td>4.4%</td>
</tr>
<tr>
<td></td>
<td>Map-only tasks for Job1</td>
<td>45.2</td>
<td>19.7%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job1</td>
<td>2.8</td>
<td>1.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>62.4</td>
<td>27.2%</td>
</tr>
<tr>
<td>Communicate</td>
<td>Communicate between Jobs</td>
<td>2.1</td>
<td>0.9%</td>
</tr>
<tr>
<td>Job2</td>
<td>Create Jar for Job2</td>
<td>3.4</td>
<td>1.5%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job2</td>
<td>0.8</td>
<td>0.3%</td>
</tr>
<tr>
<td></td>
<td>Setup Job2</td>
<td>16.6</td>
<td>7.2%</td>
</tr>
<tr>
<td></td>
<td>Map and reduce tasks for Job2</td>
<td>36.9</td>
<td>16.1%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job2</td>
<td>2.8</td>
<td>1.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>60.5</td>
<td>26.3%</td>
</tr>
<tr>
<td>Communicate</td>
<td>Communicate between Jobs</td>
<td>3.8</td>
<td>1.7%</td>
</tr>
<tr>
<td>Job3</td>
<td>Create Jar for Job3</td>
<td>3.3</td>
<td>1.4%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job3</td>
<td>0.8</td>
<td>0.3%</td>
</tr>
<tr>
<td></td>
<td>Setup Job3</td>
<td>14.5</td>
<td>6.3%</td>
</tr>
<tr>
<td></td>
<td>Map and reduce tasks for Job3</td>
<td>74.5</td>
<td>32.4%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job3</td>
<td>3.1</td>
<td>1.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>96.2</td>
<td>41.9%</td>
</tr>
<tr>
<td>Final sync.</td>
<td>Final synchronization</td>
<td>3.1</td>
<td>1.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>229.8</td>
<td>100%</td>
</tr>
</tbody>
</table>
### Table 3.5 : Multi-store

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subphases</th>
<th>Time (s)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>Connect to Hadoop</td>
<td>0.9</td>
<td>1.0%</td>
</tr>
<tr>
<td></td>
<td>Compile to MapReduce jobs</td>
<td>1.2</td>
<td>1.4%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2.4%</td>
</tr>
<tr>
<td>Job1</td>
<td>Create Jar for Job1</td>
<td>3.5</td>
<td>4.0%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job1</td>
<td>0.9</td>
<td>1.0%</td>
</tr>
<tr>
<td></td>
<td>Setup Job1</td>
<td>12.4</td>
<td>14.3%</td>
</tr>
<tr>
<td></td>
<td>Map and reduce tasks for Job1</td>
<td>60.9</td>
<td>70.2%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job1</td>
<td>2.9</td>
<td>3.3%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>82.8</td>
</tr>
<tr>
<td></td>
<td>Final synchronization</td>
<td>1.9</td>
<td>1.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>86.8</td>
</tr>
</tbody>
</table>

#### 3.2.4 Multi-store

This Pig Latin program involves several Store operations of intermediate results and is used to test the performance of Store in Pig.

The Multi-store is run as single MapReduce job in Pig and thus does not have the overhead of extra job. As Table 3.5 shows, the actual time spending on the execution is 60.9s which occupies 70.2% of total execution time. The hand-code Java MapReduce program, however, takes 59s to execute the job. The difference here is caused by converting data to and from the text format used in the HDFS files and Pigs own internal representation.
Figure 3.3 : Script of distinct aggregates

3.2.5 Distinct Aggregates

This Pig Latin program is more than one operation. Instead, it consists of a series of operations including nested distinct aggregates. The Pig Latin script is shown by Table 3.3.

As Table 3.6 shows, Pig is able to execute the complex logic in a single MapReduce. The biggest overhead in this program is the setup for the job which takes 18% of the total execution time.
Table 3.6: Distinct aggregates

<table>
<thead>
<tr>
<th>Phase</th>
<th>Subphases</th>
<th>Time (s)</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preparation</td>
<td>Connect to Hadoop</td>
<td>1.3</td>
<td>1.82%</td>
</tr>
<tr>
<td></td>
<td>Compile to MapReduce jobs</td>
<td>0.4</td>
<td>0.56%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.7</td>
<td>2.38%</td>
</tr>
<tr>
<td>Job1</td>
<td>Create Jar for Job1</td>
<td>3.5</td>
<td>4.9%</td>
</tr>
<tr>
<td></td>
<td>Communicate and submitting Job1</td>
<td>0.9</td>
<td>1.26%</td>
</tr>
<tr>
<td></td>
<td>Setup Job1</td>
<td>12.9</td>
<td>18.07%</td>
</tr>
<tr>
<td></td>
<td>Map and reduce tasks for Job1</td>
<td>48.1</td>
<td>67.36%</td>
</tr>
<tr>
<td></td>
<td>Cleanup Job1</td>
<td>3</td>
<td>4.2%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>68.4</td>
<td>95.8%</td>
</tr>
<tr>
<td>Final</td>
<td>Final synchronization</td>
<td>1.3</td>
<td>1.82%</td>
</tr>
<tr>
<td></td>
<td>synchronization</td>
<td>71.4</td>
<td>100%</td>
</tr>
</tbody>
</table>

3.3 Possible Optimizations

As we discussed in 3.2.1, the Fragment-replicated join does not scale well. Duplicated work and memory inefficiency cause the scaling loss. We noticed that in Fragment-replicated join, the execution of a job contains two parts: a) Build the hash table for the replicated table and b) Do the join operation. We also found that Pig’s Fragment-replicated join builds the hash table for every mapper. Since the hash table for a Fragment-replicated join does not change throughout execution, the hash table could be constructed once and accessed in a read-only fashion after that. In addition, as the size of replicated table grows, the memory on the nodes will be consumed up by copies of the table, which can slow down the execution.

In the next Chapter, I introduce how one can implement Fragment-replicated join using Shared-memory that making Fragment-replicated join faster, more efficient and more available.
Chapter 4

Optimization of Fragment-replicated join in Pig

4.1 Shared-memory Hashjoin

As mentioned in 3.3, we can build the hash table once and make use of it throughout the execution of hashjoin. In this way, the use of both time and space can be reduced. For each node that run mappers on, we build an in-memory file from the small table in hashjoin and use a class to wrap the data as a shared-memory HashMap. Then we can stream the input split and do the join operation.

The advantage of shared-memory hashjoin lays in that it need to build the in-memory lookup table once for each node which saves both time and space. In contrast, when doing ordinary hashjoin on Hadoop, every mapper has to build its own lookup table even the small table never changes.

4.1.1 Benchmarks

The experiment that compares performance of shared-memory hashjoin and ordinary hashjoin is done on a 2-node Hadoop cluster. Each node has 8-core CPU and 24GB memory. Java 6 and Hadoop 1.0.3 are used. The maximum number of concurrent mappers is 8 and the trunk size is 8MB. The maximum heap size of child is 2GB. The breakdown of the execution time for the experiment is shown by Table 4.1.
Table 4.1 : Running time on Hadoop

<table>
<thead>
<tr>
<th>Lookup table size (MB)</th>
<th>Join table size (MB)</th>
<th>Hashjoin time (s)</th>
<th>Sharedmem-Hashjoin time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.7</td>
<td>1.7</td>
<td>30.9</td>
<td>30.2</td>
</tr>
<tr>
<td>1.7</td>
<td>17</td>
<td>29.9</td>
<td>30.2</td>
</tr>
<tr>
<td>1.7</td>
<td>82</td>
<td>51.0</td>
<td>32.2</td>
</tr>
<tr>
<td>1.7</td>
<td>163</td>
<td>79.0</td>
<td>75.3</td>
</tr>
<tr>
<td>1.7</td>
<td>326</td>
<td>126.2</td>
<td>127.5</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>60.1</td>
<td>33.3</td>
</tr>
<tr>
<td>17</td>
<td>82</td>
<td>88.2</td>
<td>57.4</td>
</tr>
<tr>
<td>17</td>
<td>163</td>
<td>144.4</td>
<td>90.6</td>
</tr>
<tr>
<td>17</td>
<td>326</td>
<td>148.7</td>
<td></td>
</tr>
<tr>
<td>82</td>
<td>82</td>
<td>103.0</td>
<td>110.9</td>
</tr>
<tr>
<td>82</td>
<td>163</td>
<td>142.6</td>
<td>140.5</td>
</tr>
<tr>
<td>82</td>
<td>326</td>
<td>267.7</td>
<td>211.7</td>
</tr>
<tr>
<td>163</td>
<td>163</td>
<td>586.7</td>
<td>303.3</td>
</tr>
</tbody>
</table>

4.1.2 Analysis

In the experiments, there is a big performance gap between hashjoin and shared-memory hashjoin when the size of data grows. When I looked at the garbage collection time and found out ordinary hashjoin spends more than 70% of its total processing time on garbage collection while the shared memory hashjoin spends less than 15%. This means that the ordinary hashjoin places more load on the memory as each process creates a replica of the small table while using shared-memory hashjoin reduces the replicated data.
4.2 Use Shared Memory Hashjoin in Pig

4.2.1 How Pig Uses the Hadoop Platform

Pig can run in local and MapReduce mode. Pig make use of Hadoop with the following steps:

- Implement different kinds of mapper and reducer (Map-Only jobs and Map-Reduce jobs)
- Mappers and reducers implemented by Pig point to PhysicalOperator classes which implement the functions of Pig
- Set mapper class and reducer class to point to Pig’s own implementation when generating JobControl of Map-Reduce jobs
- Pack compiled class file into a jar file and pass to Hadoop through distributed cache

4.2.2 Modification of POFRjoin

The PhysicalOperator POFRjoin is where Fragment-replicated join is implemented. Thus, we want to modify it and some related files to convert Pig’s Fragment-replicated join to use shared memory.

The mapper calls the function `getNextTuple()` for each input key-value pair. There is a boolean value in POFRjoin that marks whether the HashMap inside has been built or not. Thus, the function `setUpHashMap()` will be called the first time the mapper called `getNextTuple()`, which does the job of loading the lookup table into a HashMap in the JVM heap.

My optimization replaces the HashMap with SharedHashMap and sets flags to mark both the start and end of building data in the shared memory file system. Only
the first mapper on each node needs to build the HashMap and the following mappers use the existing HashMap to do the join operation.

4.2.3 Benchmarks

The experiment setup is the same as 3.1. Table 4.2 shows the running time of the original and optimized versions of Fragment-replicated join in Pig. The column Lookup is the size of lookup table (small table) while column Join is the size of join table (big table). The number in the racket in the Lookup column is the size of the data in memory. Since the data is stored as objects in memory, there are extra fields that been stored along with the real data. The size is as large as four times of original text raw data. Column Building time is the time used by optimized version to build the shared data in memory file system. Column Optimized time and column Original time show the running time of optimized version of Fragment-replicated join and the original version respectively.

From the results we can see, when lookup table size is relatively small, the optimized version does not show much performance boost and even outperformed by the original version when join table size is also relatively small. However, when join table size grows, the execution time of original version of Fragment-replicated join increased much quicker than the optimized version. The reason for the difference is that, every mapper has to build the data structure in memory to hold the data from the lookup table in the original implementation while only the first mapper on each node need to bother doing the similar thing in the optimized version. Generally, optimized version shows its benefits in the low increasing rate of running time when the join table size being increased. It will outperform the original implementation at some point. Such point comes earlier as the lookup table size grows. For the
Table 4.2: Running time before and after optimization

<table>
<thead>
<tr>
<th>Lookup table size (MB)</th>
<th>Building time (s)</th>
<th>Join table size (MB)</th>
<th>Optimized time (s)</th>
<th>Original time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 (80 in memory)</td>
<td>4.7</td>
<td>100</td>
<td>76.5</td>
<td>66.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>86.4</td>
<td>71.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>91.5</td>
<td>81.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800</td>
<td>112.7</td>
<td>116.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1600</td>
<td>156.7</td>
<td>166.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3200</td>
<td>252.4</td>
<td>272.0</td>
</tr>
<tr>
<td>40 (160 in memory)</td>
<td>7.5</td>
<td>100</td>
<td>86.4</td>
<td>81.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>91.5</td>
<td>86.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>101.4</td>
<td>96.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800</td>
<td>121.6</td>
<td>126.9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1600</td>
<td>170.2</td>
<td>176.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3200</td>
<td>267.1</td>
<td>297.0</td>
</tr>
<tr>
<td>80 (320 in memory)</td>
<td>15.7</td>
<td>100</td>
<td>106.7</td>
<td>86.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200</td>
<td>111.7</td>
<td>96.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>116.4</td>
<td>115.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800</td>
<td>141.8</td>
<td>166.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1600</td>
<td>196.8</td>
<td>221.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3200</td>
<td>331.8</td>
<td>381.8</td>
</tr>
<tr>
<td>200 (800 in memory)</td>
<td>49.9</td>
<td>200</td>
<td>201.5</td>
<td>211.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>400</td>
<td>216.5</td>
<td>281.7</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800</td>
<td>248.3</td>
<td>362.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1600</td>
<td>312.3</td>
<td>582.2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3200</td>
<td>472.1</td>
<td>1032.1</td>
</tr>
<tr>
<td>400 (1600 in memory)</td>
<td>147.0</td>
<td>400</td>
<td>497.2</td>
<td>OutOfMemory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>800</td>
<td>524.3</td>
<td>OutOfMemory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1600</td>
<td>564.2</td>
<td>OutOfMemory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3200</td>
<td>721.6</td>
<td>OutOfMemory</td>
</tr>
</tbody>
</table>
Table 4.3 : FRJoin running time breakdown

<table>
<thead>
<tr>
<th></th>
<th>Original (s)</th>
<th>Optimized (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Setup</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Job1</td>
<td>43</td>
<td>48</td>
</tr>
<tr>
<td>Job2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Job Setup</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Building Lookup-table Time</td>
<td>26</td>
<td>46</td>
</tr>
<tr>
<td>Average Map Task Time</td>
<td>90</td>
<td>53</td>
</tr>
<tr>
<td>Median Map Task Time</td>
<td>70</td>
<td>26</td>
</tr>
<tr>
<td>Average GC Time</td>
<td>67</td>
<td>1</td>
</tr>
<tr>
<td>Total time</td>
<td>362</td>
<td>249</td>
</tr>
</tbody>
</table>

lookup table size of 200MB, the optimized version outperforms original version from the smallest join table size in the test cases and at last achieves a 2X performance promotion. When lookup table size is 400MB or more, the original implementation could not work due to OutOfMemory error while the optimized implementation still works and scales properly.

To show the differences between these two, I choose one of the experiments and breakdown the running times of both original and optimized versions. From Table 4.3 we can see that, the running times consumed before the second MapReduce job of two versions are similar. However, though optimized version takes 20 more seconds to build the hash-based data structure for lookup table, the average map task time of it is 37 seconds less than the original version. There are two reasons for this. First, every map task in the original version needs to build the data structure for lookup table while the optimized version only needs to invest time once on each node and makes use of that for the following map tasks. Second, since each map task in the original version builds a HashMap inside its heap space which occupies lots of memory, there
are more garbage collections during the execution, which consume more time than the optimized version.

Table 4.3 also includes average time spending on garbage collection (GC) for mappers in the join job. The total number of Full GC in original implementation is 347 while there is no Full GC during the join job. The total number of all GC number of original version is comparatively smaller than the optimized version, but it is the Full GC that pauses the execution and also slows down the execution severely.
Chapter 5

Related Work

5.1 Related Systems

Pig is not the only system that provides its own scripting language to abstract MapReduce jobs. In fact, there are some other systems that provide similar functions. In this section, I briefly introduce Hive, Sawzall and HPCC and discuss possibilities of transplanting our optimization in Pig to these systems.

5.1.1 Hive

Hive is a system very similar to Pig. It also has its own scripting language and executes scripts as sets of MapReduce job. However, the philosophy of these two or the use cases of these two are slightly different. Pig is more like a ‘data factory’ since it uses the concept of pipeline to process data. The system loads in raw data and processes it through a list of pipelines and finally gives out the product data.

Hive, on the other hand, is more suitable to be used as a data warehouse. It has interface for traditional database management system like ODBC and JDBC and also has its own script type [8] which covers a subset of SQL. A data warehouse stores data that is ready for users. Users only need to select the data by their demands instead of caring about the data processing pipeline.

However, Pig and Hive are similar when it comes to implementation. They both have their compilers which first compile scripts into logical plans and then physical
plans and finally MapReduce plans. They all use a DAG to present the sequence of MapReduce jobs and execute following that sequence. What is more, they all use different operator implementations to implement different kind of operations. Hive Similar architecture suggests that the optimization on Pig’s Fragment-replicatd join is likely to be transplanted to Hive’s MapJoin which is very similar.

5.1.2 Sawzall

Sawzall is a procedural language for parallel analysis of very large data sets developed at Google [9]. Instead of specifying how to process the entire data set, a Sawzall program describes the processing steps for a single data record independent of others. In Google, they use Sawzall on top of a stack of systems and tools to do parallel data processing. Google’s protocol buffer is the underlying data format being used; S2l is the compiler used to compile Sawzall language; GFS [3] is where the data stored at; Workqueue is the computing resource scheduling system; Finally the MapReduce [1] is the framework that actually runs the job.

Unlike Hive and Pig, the Sawzall program is compiled multiple times on different machines. When a job request is received by the system, the Sawzall processor first checks the grammar correctness of the script and then passes the source code to Workqueue machines for execution. The number of machines used in this job is determined by the size of the input and the number of input files, with a default upper limit proportional to the size of the Workqueue. This number can also be overridden by explicitly giving out by user.

The execution of a Sawzall job consists of two parts: a)Map side: Each Workqueue machine operates on each input piece individually and outputs intermediate data. b) Reduce side: The intermediate data is sent to machines running the aggregators,
which use different kinds of aggregation method to process the data and produce final results and write them onto GFS.

Sawzall has a similar idea with Pig to abstract complicated MapReduce code into procedural scripts (It was released earlier than Pig). The differences between Pig and Sawzall mainly exist in implementation details. As Sawzall is not an open source system, people outside Google cannot modify it. However, we believe that our optimizations for Pig can be transplanted to Sawzall since they share similar ideas and both have highly modular operators.

5.1.3 Enterprise Control Language (ECL)

The systems discussed in the earlier part in this chapter are all based on MapReduce (Hive is based on Hadoop and Sawzall is based on Google MapReduce). ECL, however, is based on a system called High Performance Computing Cluster (HPCC). HPCC is developed by LexisNexis as an integrated system environment which excels at both Extract, Transform, Load (ETL) tasks and complex analytics, and at efficient querying of large datasets using a common data-centric parallel processing language [10]. The overall target of HPCC is to meet all the requirements of data-intensive computing applications in an optimum manner. The developers of HPCC design and implement the system in two distinct processing environments.

The first is called Data Refinery which generally processes massive volumes of raw data for data cleansing and hygiene, ETL processing of the raw data (extract, transform, load), record linking and entity resolution, large-scale ad-hoc analysis of data, and creation of keyed data and indexes to support high-performance structured queries and data warehouse applications [11]. In HPCC, such system is called Thor which is similar to Hadoop MapReduce platform.
The second is called Data Delivery Engine in LexisNexis which is designed as an online high-performance structured query and analysis platform or data warehouse. Roxie is the name of the system.

Both Thor and Roxie make use of the same ECL programming language for implementing. ECL is a high-level, declarative, non-procedural dataflow-oriented language. It allows users to define what the processing result should be and the transformations to achieve the result. ECL is compiled into optimized C++ code that can be executed on the HPCC system platforms which is very similar to the execution mode of Pig, Hive and Sawzall. Also, ECL allows inline C++ functions to be incorporated into ECL programs. External services written in C++ and other languages can also be used by generating DLLs. This is similar to UDFs in Pig and Hive.

Thus, ECL on HPCC shares the same idea of Pig, Hive and Sawzall. The difference in between is that ECL is not a separate system on top of another platform. Since ECL also support modular implementation of different functions, we believe that our optimizations on Pig can also fit into ECL.

5.2 Analysis of Hadoop and Pig

Hadoop has been a hot topic in both industry and academic since it came out. There has been lots of analysis on Hadoop about its scalability, performance and potential optimizations.

Some performance models for Hadoop has been introduced in [12]. In this paper, performance models are created with regard to all the sub-phases in both Map and Reduce tasks. Though it does not include any experimental details, it is the first one that builds up detailed performance models for Hadoop. while my work is the first to provide detailed performance analysis and breakdown on Apache Pig, a platform
on top of Hadoop.

There are some other works \cite{13} that discuss the performance of Hadoop. Tan \cite{13} discusses some influence of different configurations of Hadoop, JVM, OS and even BIOS. In this paper, the author claims to observe up to 4.2x speed-up by tuning those configurations. The technical report \cite{14} \cite{15} from VMware interestingly compare the running time between running Hadoop on native machines with running Hadoop on VM(s) on top of the same machines. They come to the conclusion that it is more efficient to virtualize Hadoop. However, none of the above provide detailed breakdown to explain the reason behind performance differences as I do.

5.3 Performance Optimization on Hadoop

Beside performance analysis on Hadoop, there are some previous works introducing some method to improve performance on Hadoop. Tan \cite{16} addresses possible job starvation caused by strong dependence between map and reduce tasks and presents a resource-aware scheduler for Hadoop, which couples the progresses of mappers and reducers, and jointly optimize the placements for both of them. LATE \cite{17} also focuses on scheduling. In the paper, severe performance degradation was shown in heterogeneous environments such as Amazon’s Elastic Compute Cloud (EC2). The scheduling algorithm presented is claimed to be robust to heterogeneity.

Herodotou \cite{18} provides a cost-based optimizer for simple to arbitrarily complex MapReduce programs. The optimizer is applicable to optimize not only MapReduce jobs directly submitted by users but also jobs from a high-level system like Pig or Hive.

Some other previous works, e.g. \cite{19} \cite{20}, try to analyze MapReduce programs and apply appropriate data-aware optimizations by automating the setting of tuning
parameters for MapReduce programs.

However, none of those above optimize from a common use case aspect and address duplicated computation and memory usage. My work, on the other hand, focuses on a small portion but gives a thorough understanding and addresses a common use case.

5.4 Related Work on Join

Before MapReduce [1] was introduced, join-related work all focus on relational database systems since join is one of the fundamental query operations in relational database. Mishra [21] concludes and surveys the join processing in relational databases. Manegold [22] discussed CPU and memory issues during join operation. Some evaluations and analyses on different kinds of join operations were also introduced like Dewitt [23] and Tong [24]. There are also optimizations on different kinds of joins. For example, Lin [25] discussed large join optimizations on hypercube multiprocessors, Swami [26] used heuristics and combinatorial techniques to optimize large join queries.

One of the most common technicals is the hash join. Various hash-based join operations [27] [28] [29] have been invented since 1984. Since then, many optimizations targeting hash join appeared. Chen [30] introduced a way to improve hash join performance through prefetching; Wolf [31] addressed the problem of data skew by introducing a parallel hash join algorithms; Lo [32] examined how to apply the hash join paradigm to spatial joins, and define a new framework for spatial hash joins. Some other works [33] [34] discussed hybrid hash join algorithms.

Since MapReduce was introduced and implemented by the open source community, it has been increasingly used to analyze large volumes of data. As one of the most important operations in data processing, join has been an important part of research.
Blana \cite{35} provides a comparison of join algorithms in MapReduce. This paper studied the join algorithms originally from relational database systems on MapReduce without modifying the framework itself and revealed many details that can make those join algorithms more efficient on MapReduce. Join optimizations are still a hot topic on MapReduce. Afrati \cite{36} and Afrati \cite{37} both address some optimization of join on MapReduce. However, they have not addressed the inefficiency of memory usage in map-side hash join. Zhang \cite{38} addresses this problem by letting multiple mappers run in a single Java Virtual Machine (JVM) by modifying Hadoop MapReduce implementation. My work, on the other hand, solves the inefficiency of map-side hash join without any change to the Hadoop framework.
Chapter 6

Conclusions and Future Work

6.1 Conclusions

Nowadays, as big data processing becomes more and more important, platforms and tools for processing large-scale data in a distributed fashion are widely-used. Apache Pig is one of the most popular tool that simplify the development of parallel data processing applications. However, the exists a substantial performance gap between Pig programs and hand-coded Java MapReduce programs.

In this thesis, I first performed a detailed breakdown of Pig execution and showed the reason why Pig programs are slower than hand-code Java MapReduce programs. The micro benchmarks showed that Pig could bring in extra MapReduce jobs in some context in which Pig could not fit the logical into only one MapReduce job in the existing implementation. There are other overhead including Pig’s default setup and cleanup for every MapReduce job and compiling the script, all of which are constant and become insignificant when data size grows.

During the execution time breakdown experiments of Pig, an inefficiency of Fragment-replicated join operation was observed: in original implementation, every mapper has to take in the replicated table and build a HashMap inside its JVM heap space. Since JVMs does not share data with each other, there are numbers of copies of same piece of read-only data inside memory. Thus, original implementation can achieve neither memory efficiency nor time efficiency. I present a novel idea to use
shared memory to optimize Fragment-replicated join. Instead of letting every mapper
build its own HashMap, I make the first mapper on each compute node to create an
in-memory file and all the mapper on that node will use a Hash-based data structure
that use the data of this file. In this way, the copies of the same data in memory
stays constant and so does the time spending on creating HashMap for replicated
table. The benchmarks show that the optimized version of Fragment-replicated join
outperforms original implementations by more than 2X when data size grows and
provides more availability for larger data sets.

6.2 Future Work

Apache Pig is a successful project that has been widely used for many years. How-
ever, there is still room for optimizations especially on performance side. Firstly, we
can have a richer set of parameterizable operator implementations to let users have
more control over the executions. Secondly, we can have a stronger compiler that
can improve a query plan by selecting and parameterizing these operators to avoid
unnecessary MapReduce jobs.

On the other hand, The Shared-memory concept fits into a common paradigm
and can be applied to multiple popular systems including Hive and Sawzall. Trying
to apply the Shared-memory idea to both systems will be another part of my future
work.
Bibliography


