Join Algorithms using Map/Reduce

Jairam Chandar

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2010

Abstract

Information explosion is a well known phenomenon now and there is a vast amount of research going on into how best to handle and process huge amounts of data. One such idea for processing enormous quantities of data is Google's Map/Reduce.

Map/Reduce was first introduced by Google engineers - Jeffrey Dean and Sanjay Ghemawat [9]. It was designed for and is still used at Google for processing large amounts of raw data (like crawled documents and web-request logs) to produce various kinds of derived data (like inverted indices, web-page summaries, etc.). It is a simple yet powerful framework for implementing distributed applications without having extensive prior knowledge of the intricacies involved in a distributed system. It is highly scalable and works on a cluster of commodity machines with integrated mechanisms for fault tolerance. The programmer is only required to write specialized `map` and `reduce` functions as part of the Map/Reduce job and the Map/Reduce framework takes care of the rest. It distributes the data across the cluster, instantiates multiple copies of the `map` and `reduce` functions in parallel, and takes care of any system failures that might occur during the execution.

Since its inception at Google, Map/Reduce has found many adopters. Among them, the prominent one is the Apache Software Foundation, which has developed an Open-Source version of the Map/Reduce framework called Hadoop [2]. Hadoop boasts of a number of large web-based corporates like Yahoo, Facebook, Amazon, etc., that use it for various kinds of data-warehousing purposes. Facebook for instance, uses it to store copies of internal logs and uses it as a source for reporting and machine learning. See [4] for other companies that use Hadoop.

Owing to its ease of use, installation and implementation, Hadoop has found many uses among programmers. One of them is query evaluation over large datasets. And one of the most important queries are *Joins*. This project explores the existing solutions, extends them and proposes a few new ideas for joining datasets using Hadoop.

Algorithms have been broken into two categories - *Two-Way* joins and *Multi-Way* joins. Join algorithms are then discussed and evaluated under both categories. Options to pre-process data in order to improve performance have also been explored. The results are expected to give an insight into how good a fit Hadoop or Map/Reduce is for evaluating *Joins*.

Acknowledgements

I would like to extend my heartfelt gratitude to my supervisor, Dr. Mary Cryan, for guiding me throughout this project and giving me invaluable advice at times when I needed it the most. I would also like to thank Mr. Chris Cooke who was responsible for making sure the cluster used to run the experiments worked fine.

A special mention to the open-source community for a wonderful product like Hadoop on which this project is based.

And last, but certainly not the least, to my parents and friends who always stood by me.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jairam Chandar*)

To my parents who made me who I am today.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Information explosion is the rapid increase in the amount of published information and the effects of this abundance of data. As the amount of available data grows, the problem of managing the information becomes more difficult, which can lead to information overload." [1]

1.1 Background

The World Wide Web saw a revolution with the advent of Web 2.0. Web applications became more interactive, allowing users the freedom to interact and collaborate over the Internet in ways not possible earlier. Users started to take the role of content creators rather than passive viewer of webpages. Websites started to get swamped with user-generated content from blogs, videos, social media sites and various other Web 2.0 technologies. This had the direct consequence of information stored on servers exploding into sizes not seen earlier. Contributing to this information explosion was the already accumulating business data that was generated everyday across various companies and industries. These were just a couple of reasons that led to the age of *Petabytes* - an age where information stored in data stores reached levels of Petabytes or 1024 TeraBytes or 1048576 Gigabytes. With more and more enterprises using computers and embracing the digital age, more and more information is starting to get digitized and the volume of data published and stored is increasing with each passing day.

Obviously, storing digital data is not sufficient, its needs to be queried as well. But with such huge volumes of data, there is a need to look at query algorithms from a different perspective. For instance, algorithms need to be storage-aware in order

[1] `http://en.wikipedia.org/wiki/Information_explosion`

to load and retrieve data efficiently. There is tremendous amount of research being undertaken towards creating such algorithms, for example, Google's BigTable [8] or Facebook's Cassandra [1] which is now open-source and is maintained by the Apache Software Foundation. Many of the leading Information Technology companies have invested a lot into this research and have come up with a number of innovative ideas and products.

It is well known that more often than not, the most time consuming step in a Database Management System is the Query Evaluation engine, especially when there are large amounts of data involved. Fast and efficient retrieval mechanisms that work within acceptable time frames and that can keep up with the pace of information explosion are essential for effective database systems of the future. Using more processing power, and in particular, exploiting parallel processing is the intuitive solution. While there have been many attempts at parallelizing query evaluation, there is still much scope for research in this area. This project investigates a specific framework for managing parallel processing in the database context. It consolidates work already done and proposes new algorithms and optimizations to existing solutions (discussed in Chapter 3).

One of the most common operations in query evaluation is a *Join*. Joins combine records from two or more tables, usually based on some condition. They have been widely studied and there are various algorithms available to carry out joins. For example, the *Nested-Loops Join*, the *Sort-Merge Join* and the *Hash Join* are all examples of popular join algorithms (see [17]). These algorithms (and more) are used for joining two as well as more datasets. But more often than not, when multiple datasets are involved, *selectivity factor* is exploited to structure the order in which the joins are made. *Selectivity factor* can be defined as the fraction of the datasets involved in the join that will be present in the output of the join. We too will be exploiting it in one of our algorithms later.

All DBMSs support more than one algorithm to carry out joins. But as expected, things start to get complicated and unacceptably slow when data starts to run into very large sizes. Lets Consider a trivial example with a simple algorithm like *Nested-Loops Join*.

Given two datasets R and S, the Nested-Loops Join algorithm joins ($R \bowtie S$) as follows -

for all tuples $r \in R$ **do**
  for all tuples $s \in S$ **do**
    if *JoinCondition* **then** {*JoinCondition* can be like $r.a == s.b$}
      add (r, s) to the result
    end if
  end for
end for

Using this algorithm, two tables containing 100,000 and 50,000 rows will take about an hour and a half to perform an equi-join, in the case where each I/O takes 10 ms, the inputs are clustered and the I/O is done one page at a time [17]. If avoidable, this option is seldom used. Other algorithms like *Hash Join* or *Sort-Merge Join* can perform the same join in about a minute. But they suffer from the drawback that they cannot perform a non-equi join and the only option in that case is the *Nested-Loops Join*. Join Algorithms have been studied extensively with many variants existing for each algorithm. For instance, Hash-Join itself has three different variations - *Simple Hash Join*, *Grace Hash Join* and *Hybrid Hash Join* (all three explained in [17]).

A few years back, engineers at Google introduced a new programming model called *Map/Reduce* [9] (explained in more detail in Chapter 2). Map/Reduce enables easy development of scalable parallel applications to process vast amounts of data on large clusters of commodity machines. Map/Reduce framework hides management of data and job partitioning from the programmer and provides in-built fault-tolerance mechanisms. This lets the programmer concentrate on the actual problem at hand instead of worrying about the intricacies involved in a distributed system. It was designed for processing large amounts of raw data (like crawled documents and web-request logs) to produce various kinds of derived data (like inverted indices, web-page summaries, etc.). It is still a prominently used model at Google for many of its applications and computations (see [9]).

Map/Reduce was not developed for Database Systems in the conventional sense. It was designed for computations that were conceptually quite straightforward, but involved huge amounts of input data. For example, finding the set of most frequent queries submitted to Google's search engine on any given day. It is very different from the traditional database paradigm, in that it does not expect a predefined schema, does not have a declarative query language and any indices are the programmers prerogative. But at the same time, Map/Reduce hides the details of parallelization, has built-in fault tolerance and load balancing in a simple programming framework. The novel idea was

so appealing that it led to an open-source version called Hadoop. Hadoop has become extremely popular in the past few years and boasts of big Web 2.0 companies like Facebook, Twitter and Yahoo! as part of its community.

1.2 Motivation and Aim

Map/Reduce was primarily designed at Google for use with its web-indexing technologies. These included things like keeping track of the web-pages crawled so far, creating inverted-indices from them and summarizing the web-pages for search-result views. Over time, they started considering it for more interesting computations like query processing on the raw or derived data that they already had. This led to its wide-spread adoption and led to uses that were not envisioned at the time of designing. Companies have started using Map/Reduce to manage large amounts of data (see [11] for example). And when there are multiple datasets involved, there will always be a need for joining these datasets. The goal of this project is to test the viability of Map/Reduce framework for joining datasets as part of database query processing. There are four main contributions of this project -

1. Examine and consolidate the existing algorithms for joining datasets using Map/Reduce.
2. Propose new algorithms.
3. Quantitatively evaluate the performance of these algorithms.
4. Analyse the performance and discuss the various factors that come into picture when using Map/Reduce for joining datasets.

1.3 Related Work

A lot of work has already been done on implementing database queries with Map/Reduce and comparing the Map/Reduce model with parallel relational databases. The framework itself comes with a functionality for sorting datasets (explained further in 2). There is also a functionality to join datasets that is available out of the box. This is the Map-Side join which [16] explains in greater detail. This kind of join is also discussed in this thesis in Chapter 3 and Chapter 4. In a very recent publication [6], Blanas et. al. suggested algorithms and compared the performance for joining log

datasets with user datasets. They also compared their work with Yahoo's Pig Project [11] and found that their algorithms performed considerably better than those used in Pig. In [12], Konstantina Palla developed a theoretical cost model to evaluate the I/O cost incurred during the execution of a Map/Reduce job. She also compared this cost model with two popular join algorithms using Map/Reduce, *Reduce-Side Join* and *Map-Side Join*.

In [14] the authors argue that while Map/Reduce has lately started to replace the traditional DBMS technology, Map/Reduce in fact complements rather than competes with with it. They conducted benchmark studies and found that DBMSs were substantially faster (in tasks like query processing) than the Map/Reduce systems once the data is loaded, but that loading the data took considerably longer in the database systems. This lead them to conclude that Map/Reduce is more like an Extract-Transform-Load (ETL) system [1] than a DBMS, as it quickly loads and processes large amounts of data in an ad-hoc manner. Studies conducted by Pavlo et. al. in [13] found that the Map/Reduce model outperformed traditional DBMS systems in terms of scalability and fault tolerance. But they also noted that it suffered from certain performance limitations of the model, particularly in terms of computation time. This is attributed mainly to the fact that the model was not originally designed to perform structured data analysis and lacks many of the basic features that the relational databases routinely provide. On the other hand, advocates of the model, deployed it to investigate its computing capabilities in environments with large amounts of raw data. Abouzeid et al. in their work [5] attempted to bridge the gap between the two technologies, that is, parallel databases and Map/Reduce model, suggesting a hybrid system that combines the best features from both. The acquired results appeared quite promising and pointed out the advantages of an approach that combines the efficiency of relational databases with the scalability and fault tolerance of the Map/Reduce model.

Several research studies also aim to improve the model itself. Yang et al. [19] proposed a Merge component after the reduce function for performing a join operation on two datasets. However, the Map-Reduce-Merge approach introduces an extra processing step that is not there in the standard Map/Reduce framework and therefore will not be found in a standard Map/Reduce deployment. Moreover, the Pig project at Yahoo [11], the SCOPE project at Microsoft [7], and the open source Hive project [15] introduce SQL-style declarative languages over the standard Map/Reduce model in an

[1] An ETL system *Extracts* information from the data source, *Transforms* it using a series of rules and functions and *Loads* the transformed data into a target, usually a Data Warehouse.

attempt to make it more expressive and limit its schema-less disadvantages. All these projects provide ways of declaring join executions over datasets.

While most of the work present currently, like the work done by Blanas et. al in [6], concentrates more on two-way joins, this project extends their work to multi-way joins. Also, we evaluate the results of the algorithms in a different way focussing more on datasets of comparable size.

1.4 Thesis Outline

Chapter 2 introduces the Map/Reduce framework in more detail. It explains the stages involved in the execution of a Map/Reduce program. Section 2.4 talks about Hadoop - the open-source implementation of Map/Reduce - and throws light upon the functionalities of Hadoop that have been used in this project.

Chapter 3 explores the existing join algorithms targeted at Map/Reduce and introduces some new algorithms for multi-way joins. Section 3.2 discusses two-way joins (joins involving two datasets) and section 3.3 addresses multi-way joins (joins involving more than two datasets). The chapter describes the pros and cons of each algorithm, discussing their suitability for a given situation.

Experimental results are evaluated in chapter 4. Detailed analysis is done for each algorithm with different input data sizes and with clusters consisting of different number of participating machines.

Chapter 5 concludes the thesis. It summarizes the various observations made during the project and suggest some future avenues for research.

Chapter 2

Map/Reduce and Hadoop

2.1 What is Map/Reduce?

Map/Reduce [9] is a "programming model and an associated implementation for processing and generating large data sets". It was first developed at Google by Jeffrey Dean and Sanjay Ghemawat. Their motivation was derived from the multitude of computations that were carried out everyday in Google that involved huge amounts of input data. These computations usually happened to be conceptually straightforward. For instance, finding the most frequent query submitted to Google's search engine on any given day or keeping track of the webpages crawled so far. But the input to these computations would be so large that it would require distributed processing over hundreds or thousands of machines in order to get results in a reasonable amount of time. And when distributed systems came into picture, a number of problems like carefully distributing the data and partitioning or parallelizing the computation made it difficult for the programmer to concentrate on the actual simple computation.

Dean and Ghemawat saw a need for an abstraction that would help the programmer focus on the computation at hand without having to bother about the complications of a distributed system like fault tolerance, load balancing, data distribution and task parallelization. And that is exactly what Map/Reduce was designed to achieve. A simple yet powerful framework which lets the programmer write simple units of work as **`map`** and **`reduce`** functions. The framework then automatically takes care of partitioning and parallelizing the task on a large cluster of inexpensive commodity machines. It takes care of all the problems mentioned earlier, namely, fault tolerance, load balancing, data distribution and task parallelization. Some of the simple and interesting computations for which Map/Reduce can be used include (see [9] for details on each) -

- Distributed Grep - finding patterns in a number of files at the same time.
- Count of URL access frequency.
- Reverse Web-Link Graph - Given a list of ⟨target, source⟩ pair of URLs, finding ⟨target, list(source)⟩, i.e., finding all the URLs that link to a given target.
- Construction of *Inverted Indices* from crawled web pages
- Distributed Sort

The idea was soon picked up by the open-source community, the Apache Software Foundation specifically, and developed into an open-source project and subsequently into a full-fledged framework and implementation called Hadoop [2]. Hadoop is free to download and now boasts of a very large community of programmers and enterprises that includes large Web 2.0 corporates like Yahoo. Hadoop is discussed in more detail in the coming sections.

To create a Map/Reduce job, a programmer specifies a **`map`** function and a **`reduce`** function. This abstraction is inspired by the 'map' and 'reduce' primitives present in Lisp and many other functional languages. The Map/Reduce framework runs multiple instance of these functions in parallel. The **`map`** function processes a key/value pair to generate another key/value pair. A number of such **`map`** functions running in parallel on the data that is partitioned across the cluster, produce a set of intermediate key/value pairs. The **`reduce`** function then merges all intermediate values that are associated with the same intermediate key (see figure 2.1).

`map (k1, v1)` $\rightarrow$ **`k2,v2`**
`reduce (k2, list(v2))` $\rightarrow$ **`v3`**

Programs written in this functional style are automatically parallelized by the Map/Reduce framework and executed on a large cluster of commodity machines. As mentioned earlier, the run-time system takes care of the details of data distribution, scheduling the various **`map`** and **`reduce`** functions to run in parallel across the set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any prior experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

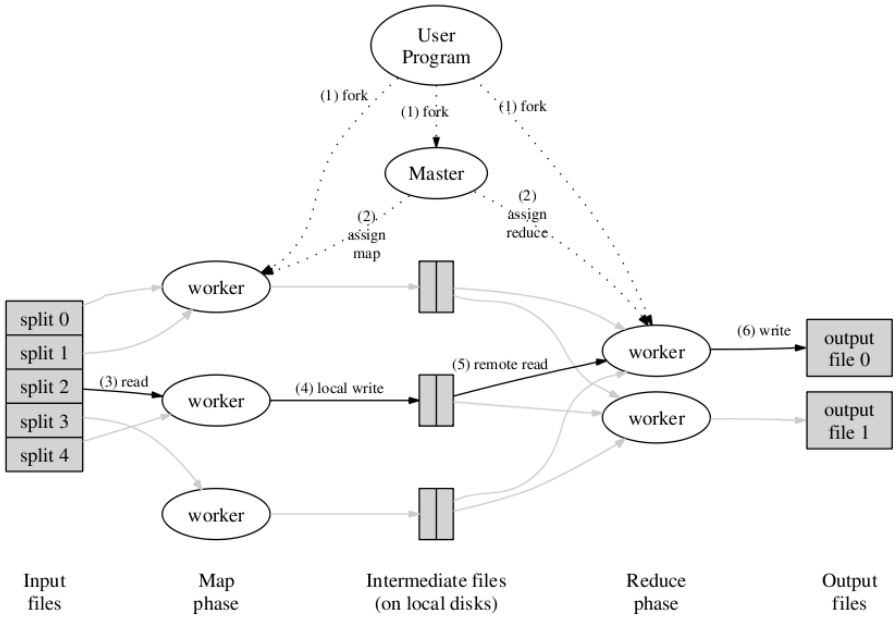

Figure 2.1: Map Reduce Execution Overview [9]

2.2 Map/Reduce Breakdown

Figure 2.1 gives an overview of the Map/Reduce execution model. Before explaining the steps involved in a Map/Reduce job, lets clarify the terminology that will be used from this point on in this thesis (unless specifically specified otherwise) -

- Machine - an actual physical computer, which is part of a distributed cluster
- Task or Worker - a process running on a machine
- Node - This can be thought of as a process handler. This will become more clear when we discuss Hadoop where a Node is basically a *Java Daemon*[1]. Nodes run on the machines that are part of the cluster. Ideally, one machine will correspond to one node.

An entire "Map/Reduce Job" [9] can be broken down into the following steps (reproduced here from [9] with a few modifications) -

1. The Map/Reduce framework first splits the input data files into *M* pieces of fixed size - this typically being 16 megabytes to 64 megabytes (MB) per piece (con-

[1] A *Java Daemon* is a thread that provides services to other threads running in the same process as the daemon thread

trollable by the user via an optional parameter). These M pieces are then passed on to the participating machines in the cluster. Usually there are 3 copies (user controllable) of each piece for fault tolerance purposes. It then starts up many copies of the user program on the nodes in the cluster.

2. One of the nodes in the cluster is special - the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. R is either decided by the configuration specified with the user-program, or by the cluster wide default configuration. The master picks idle workers and assigns each one a map task. Once the map tasks have generated the intermediate output, the master then assigns reduce tasks to idle workers. Note that all map tasks have to finish before any reduce task can begin. This is because the reduce tasks take output from any and every map task that may generate an output that it will need to consolidate.

3. A worker who is assigned a map task reads the contents of the corresponding input split. It parses key/value pairs out of the input data and passes each pair to an instance of the user defined **`map`** function. The intermediate key/value pairs produced by the **`map`** functions are buffered in memory at the respective machines that are executing them.

4. The buffered pairs are periodically written to local disk and partitioned into R regions by the *partitioning* function. The framework provides a default *partitioning* function but the user is allowed to override this function for allowing custom partitioning. The locations of these buffered pairs on the local disk are passed back to the master. The master then forwards these locations to the reduce workers.

5. When a reduce worker is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys (**`k2`** in the definition given earlier for the **`reduce`** function) so that all occurrences of the same key are grouped together. The sorting is needed because typically many different keys map to the same reduce task. If the amount of intermediate data is too large to fit in memory, an external sort is used. Once again, the user is allowed to override the default sorting and grouping behaviours of the framework.

6. Next, the reduce worker iterates over the sorted intermediate data and for each unique intermediate key encountered , it passes the key and the corresponding set of intermediate values to the users **`reduce`** function. The output of the **`reduce`** function is appended to a final output file for this reduce partition.

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the Map/Reduce call in the user program returns back to the user code.

2.3 Map/Reduce Example

To better understand the Map/Reduce, lets consider an example. Given below are the **`map`** and **`reduce`** function for categorizing a set of numbers as even or odd.

```
map(String key, Integer values)
{
//key : File Name
//values : list of numbers
for each v in values:
    if(v%2==0)
      EmitIntermediate("Even", v)
   else
      EmitIntermediate("Odd", v)
}

reduce(String key, Iterator values)
{
//key: Even or Odd
//values : Iterator over list of numbers
//(categorized as odd or even)
String val = ""verbatim
while(values.hasnext())
{
   val=val+","+values.toString()
}
```

```
Emit(key, val)
}
```

So given a list of numbers as $5, 4, 2, 1, 3, 6, 8, 7, 9$, the final output file will look like this -

```
Even 2,4,6,8
Odd 1,3,5,7,9
```

This is a very simple example where both the **map** and **reduce** function do not do anything much interesting. But a programmer has the freedom to write something a lot more complex in these functions, as will be seen in the coming chapters.

2.4 Hadoop and HDFS[1]

2.4.1 Hadoop Overview

Hadoop [2] is the Apache Software Foundation open source and Java-based implementation of the Map/Reduce framework. Hadoop was created by Doug Cutting, the creator of Apache Lucene[2], the widely used text library. Hadoop has its origins in Apache Nutch[3], an open source web search engine, itself a part of the Lucene project.

Nutch was an ambitious project started in 2002, and it soon ran into problems with the creators realizing that the architecture they had developed would not scale to the billions of web-pages on the Internet. But in 2003, a paper [10] was published that described Google's distributed filesystem - the Google File System (GFS). The idea was adopted by the Nutch project and developed into what they called the Nutch Distributed File System or NDFS.

In 2004, [9] was published, introducing the concept of Map/Reduce to the world and the developers at Nutch implemented it for their purposes. Hadoop was born in February 2006, when they decided to move NDFS and Nutch under a separate subproject under Lucene. In January 2008, Hadoop was made its own top level project under Apache and NDFS was renamed to HDFS or Hadoop Distributed File System.

[1] This section borrows heavily from [18]

[2] http://lucene.apache.org/java/docs/index.html

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

Hadoop provides the tools for processing vast amounts of data using the Map/Reduce framework and, additionally, implements the Hadoop Distributed File System (HDFS). It can be used to process vast amounts of data in-parallel on large clusters in a reliable and fault-tolerant fashion. Consequently, it renders the advantages of the Map/Reduce available to the users.

2.4.2 Hadoop Distributed File System or HDFS

"HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters on commodity hardware." [18].

HDFS was designed keeping in mind the ideas behind Map/Reduce and Hadoop. What this implies is that it is capable of handling datasets of much bigger size than conventional file systems (even petabytes). These datasets are divided into blocks and stored across a cluster of machines which run the Map/Reduce or Hadoop jobs. This helps the Hadoop framework to partition the work in such a way that data access is local as much as possible.

A very important feature of the HDFS is its "streaming access". HDFS works on the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. Once the data is generated and loaded on to the HDFS, it assumes that each analysis will involve a large proportion, if not all, of the dataset. So the time to read the whole dataset is more important than the latency in reading the first record. This has its advantages and disadvantages. One on hand, it can read bigger chunks of contiguous data locations very fast, but on the other hand, random seek turns out to be a so slow that it is highly advisable to avoid it. Hence, applications for which low-latency access to data is critical, will not perform well with HDFS.

The feature affected the design of a few of the algorithms discussed in chapter 3. For instance in section 3.2.1, the **`reduce`** function had to buffer data in memory in order to allow random access. This will be explained in detail later.

2.4.3 Hadoop Cluster

A Hadoop cluster consists of the following main components, all of which are implemented as JVM daemons -

- JobTracker

  Master node controlling the distribution of a Hadoop (Map/Reduce) Job across

free nodes on the cluster. It is responsible for scheduling the jobs on the various *TaskTracker* nodes. In case of a node-failure, the *JobTracker* starts the work scheduled on the failed node on another free node. The simplicity of Map/Reduce tasks ensures that such restarts can be achieved easily.

- NameNode
  Node controlling the HDFS. It is responsible for serving any component that needs access to files on the HDFS. It is also responsible for ensuring fault-tolerance on HDFS. Usually, fault-tolerance is achieved by replicating the files over 3 different nodes with one of the nodes being an off-rack node.

- TaskTracker
  Node actually running the Hadoop Job. It requests work from the *JobTracker* and reports back on updates to the work allocated to it. The TaskTracker daemon does not run the job on its own, but forks a separate daemon for each task instance. This ensure that if the user code is malicious it does not bring down the *TaskTracker*

- DataNode
  This node is part of the HDFS and holds the files that are put on the HDFS. Usually these nodes also work as *TaskTrackers*. The *JobTracker* tries to allocate work to nodes such files accesses are local, as much as possible.

2.4.4 Hadoop Job

Figure 2.2 shows Map/Reduce workflow in a user job submitted to Hadoop.

To run a Map/Reduce or Hadoop job on a Hadoop cluster, the client program must create a *JobConf* configuration file. This configuration file -

- Identifies classes implementing Mapper and Reducer interfaces.
  - *JobConf.setMapperClass(), setReducerClass()*
- Specifies inputs, outputs
  - *FileInputFormat.addInputPath(conf)*
  - *FileOutputFormat.setOutputPath(conf)*

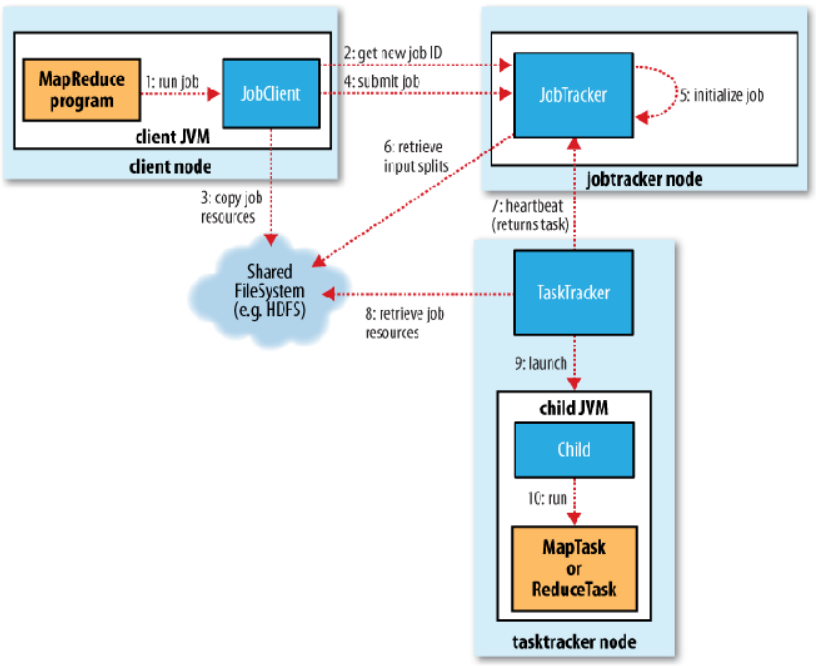

Figure 2.2: Hadoop Cluster [18]

- Optionally, other options too:
  - *JobConf.setNumReduceTasks()*
  - *JobConf.setOutputFormat()*

The *JobTracker* inserts the program (supplied in the form of a .jar file) and the *JobConf* file in a shared location. *TaskTrackers* running on slave nodes periodically query the *JobTracker* for work and retrieve job-specific jar and configuration files. As mentioned earlier, they then launch tasks in separate instances of Java Virtual Machine.

2.4.5 Map and Reduce Functions

The *Map* and *Reduce* classes in Hadoop extend *MapReduceBase* class and the **`map`** and **`reduce`** functions have the signature as shown in figure 2.3.

Hadoop has its own serialization format. Central to this serialization format is the *Writable* interface, which is a part of the signatures shown in figure 2.3. *WritableComparable* is a sub-interface of *Writable* that requires the implementation of a *Comparator* function. *OutputCollector* is a generic class used purely for emitting key/value pairs. *Reporter* is for updating counters and statuses.

```
map(WritableComparable key,
Writable value,
OutputCollector
output, Reporter reporter)
```

(a) **map** function

```
reduce(WritableComparable
key,
Iterator values,
OutputCollector
output, Reporter reporter)
```

(b) **reduce** function

Figure 2.3: Signatures of **map** and **reduce** functions in Hadoop

2.4.6 Partitioning and Grouping

The Hadoop framework takes the output from the *Mapper* and does the following -

1. Partitions the output
2. Sorts the individual partitions
3. Sends relevant partitions to *Reducers*
4. Merges the partitions received from different *Mappers*
5. Groups the tuples in the partition based on key and calls the **reduce** function

Figure 2.4 shows the steps just described in the context of the entire Hadoop (Map/Reduce) job -

Hadoop lets the programmer control the partitioning and grouping of tuples, although it does provide with a default implementation in the absence of one provided by the programmer. This would be required for some of the *Reduce-Side Join* algorithm specified in section 3.2.1 and *Reduce-Side One-Shot Join* and *Reduce-Side Cascade Join* discussed in section 3.3 A 'Partitioner' class can be defined by implementing the *Partitioner* interface of the *org.apache.hadoop.mapred* package. And this class needs to be specified as part of the Job Configuration by using *JobConf.setPartitionerClass()* function. See Appendix B for how this is done.

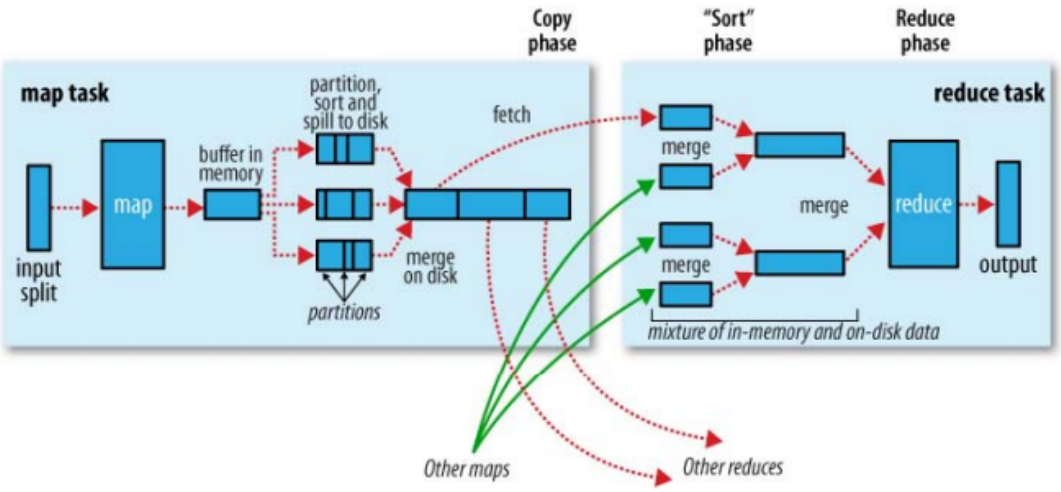

Figure 2.4: Shuffle and Sort in Map/Reduce [18]

To override the grouping done in the *Reducer* one needs to write a class that extends *org.apache.hadoop.io.WritableComparator* class. This class must have functions to compare key values, but the behaviour of the comparison is completely dependent on the programmer. To make sure the *Reducer* uses this class for comparison, it needs to be specified as part of the Job Configuration by using *JobConf.setOutputValueGroupingComparator()* function. The class *FirstComparator* defined inside the *TextPair* class in Appendix A is an example of such a class.

Partitioning and Grouping functions are used as part of the 'Reduce-Side Join', which is described in chapter 3.

2.4.7 Hadoop Counters

When a client program initiates a Hadoop job, the Hadoop framework keeps track of a lot of metrics and counters as part of its logs. These are usually system metrics like time taken, amount of data read, etc. Apart from these, Hadoop also allows users to specify their own user-defined counters which are then incremented as desired in the *mapper* and the *reducer*. For more details on the default counters provided by Hadoop, see [18]. Only user-defined counters are described here, which are used in one of the join algorithms described later in chapter 3. There are two ways of defining a counter-

- Java `enum`

  Users can define any number of Java enums with any number of fields in them. For ex. -

```
enum NumType{
EVEN,
ODD
}
```

These can be incremented in the *mapper* or the *reducer* as -

```
Reporter.incrementCounter(NumType.EVEN, 1)
```

- Dynamic Counters
  Apart from `enums`, users can also create dynamic counters. The user will have to make sure he/she uses the same name every where. An example is -

  ```
  Reporter.incrementCouner("NumType", "Even", 1)
  ```

2.4.8 Hadoop Data Types

Hadoop uses its own serialization format called *Writables*. It is fast and compact. For ease of use, Hadoop comes with built-in wrappers for most of Java primitives. The only one that is used in this project *Text*, which is a wrapper around the Jave *String* object. See [18] for more details.

2.4.9 Compression in Hadoop

Compression can be easily implemented in Hadoop. It has built-in libraries or *codecs* that implement most of the well-known compression algorithms. Compression will be used in section 3.3.3 for optimizing one of the multi-way join algorithms. Shown below are two of the most famous ones used - The interesting thing to note is that some

Compression Format	Hadoop CompressionCodec
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`

Table 2.1: Hadoop compression codecs

of the codecs allow you to split and some dont. For instance, its not possible to read

a gzip file in the middle but it is possible to do so for a bzip2 file. Hence depending on one's needs, they must use the relevant codec. For our purpose, we used the bzip2 codec so that Hadoop can efficiently split the compressed input files. To compress the output of a MapReduce job, the `mapred.output.compress` property is set to true and the `mapred.output.compression.codec` property is set to the classname of the required codec (in our case `org.apache.hadoop.io.compress.GzipCodec`) in the Job Configuration. If the inputs are compressed, Hadoop automatically infers the compression codec using the file's name. Hence it is quite straightforward to implement compression in Hadoop.

Chapter 3

Join Algorithms

Most of the existing work has concentrated on two-way joins leaving the reader to extend the idea for multi-way joins. One such paper is [6]. The *Broadcast Join* algorithm mentioned in section 3.2.3 is a variation of an idea mentioned in it. This project deals with both two-way and multi-way joins. Before moving any further, let us define what these are specifically in context of this project-

- Two-way Joins - Given two dataset P and Q, a two-way join is defined as a combination of tuples $p \in P$ and $q \in Q$, such that $p.a = q.b$. a and b are values from columns in P and Q respectively on which the join is to be done. Please note that this is specifically an 'equi-join' in database terminology. This can be represented as-

$$P \bowtie_{a=b} Q$$

- Multi-way Joins - Given n datasets $P_1, P_2, ..., P_n$, a multi-way join is defined as a combination of tuples $p_1 \in P_1, p_2 \in P_2, ..., p_n \in P_n$, such that $p_1.a_1 = p_2.a_2 = ... = p_n.a_n$. $a_1, a_2, ..., a_n$ are values from columns in $P_1, P_2, ..., P_n$ respectively on which the join is to be done. Notice once again that this is specifically an 'equi-join'. This can be represented as-

$$P_1 \bowtie_{a_1=a_2} P_2 \bowtie_{a_2=a_3} ... \bowtie_{a_{n-1}=a_n} P_n$$

The algorithms thus described in this chapter have been divided into two categories-

1. Two-Way Joins - Joins involving only two tables
2. Multi-Way Joins - Joins involving more than two tables

3.1 Join Algorithms in standard database context

Before jumping on to join algorithms using Map/Reduce (or Hadoop) it might be a good idea to review the currently exiting join algorithms in the standard database context. There are three of them that are very popular. These are -

1. Nested Loops Join
2. Sort-Merge Join
3. Hash Join

Nested Loops Join

We described the *Nested Loops Join* earlier in chapter 1. It is one of the oldest join algorithms and is one of the simplest. It is capable of joining two datasets based on any join condition. It does not suffer from the drawback of the next two algorithms that can only perform equi-joins. Unfortunately, all the algorithms using Map/Reduce that are discussed in this chapter suffer from this very drawback. They all can perform only equi-joins.

Sort-Merge Join

Given two datasets P and Q, the *Sort-Merge Join* algorithm sorts both datasets on the join attribute and then looks for qualifying tuples $p \in P$ and $q \in Q$ by essentially merging the two datasets. [17]. The sorting step groups all tuples with the same value in the join column together and thus makes it easy to identify partitions or groups of tuples with the same value in the join column. This partitioning is exploited by comparing the P tuples in a partition with only the Q tuples in the same partition (rather than with all tuples in Q), thereby avoiding enumeration or the cross-product of P and Q. This partition-based approach works only for equality join conditions.

$p \in P; q \in Q; gq \in Q$
while more tuples in inputs **do**

  while $p.a < gq.b$ **do**
    advance p
  end while
  while $p.a > gq.b$ **do**
    advance gq {a group might begin here}
  end while
  while $p.a == gq.b$ **do**
    $q = gq$ {mark group beginning}
    while $p.a == q.b$ **do**
      Add $\langle p, q \rangle$ to the result
      Advance q
    end while
    Advance p {move forward}
  end while
  $gq = q$ {candidate to begin next group}
end while

It should be apparent that this algorithm will give good performances if the input datasets are already sorted.

The *Reduce-Side Join* algorithm described in section 3.2.1 is in many ways a distributed version of *Sort-Merge Join*. In *Reduce-Side Join* algorithm, each sorted partition is sent to a **`reduce`** function for merging.

Hash Join

The *Hash Join* algorithm consists of a 'build' phase and a 'probe' phase. In its simplest variant, the smaller dataset is loaded into an in-memory hash table in the build phase. In the 'probe' phase, the larger dataset is scanned and joined with the relevant tuple(s) by looking into the hash table. The algorithm like the *Sort-Merge Join* algorithm, also works only for equi-joins.

Consider two datasets P and Q. The algorithm for a simple hash join will look like this-

for all $p \in P$ **do**
  Load p into in memory hash table H
end for
for all $q \in Q$ **do**

  if H contains p matching with q **then**
    add $\langle p, q \rangle$ to the result
  end if
end for

This algorithm usually is faster than the *Sort-Merge Join*, but puts considerable load on memory for storing the hash-table.

The *Broadcast Join* algorithm described in section 3.2.3 is a distributed variant of the above mentioned *Hash Join* algorithm. The smaller dataset is sent to every node in the distributed cluster. It is then loaded into an in-memory hash table. Each **`map`** function streams through its allocated chunk of input dataset and probes this hash table for matching tuples.

3.2 Two-Way Joins

3.2.1 Reduce-Side Join

In this algorithm, as the name suggests, the actual join happens on the Reduce side of the framework. The 'map' phase only pre-processes the tuples of the two datasets to organize them in terms of the join key.

Map Phase

The **`map`** function reads one tuple at a time from both the datasets via a stream from HDFS. The values from the column on which the join is being done are fetched as keys to the **`map`** function and the rest of the tuple is fetched as the value associated with that key. It identifies the tuples' parent dataset and tags them. A custom class called TextPair (See Appendix A) has been defined that can hold two *Text* values (see section 2.4.8 for more details). The **`map`** function uses this class to tag both the key and the value. The reason for tagging both of them will become clear in a while. Listing 3.1 is the code for the **`map`** function.

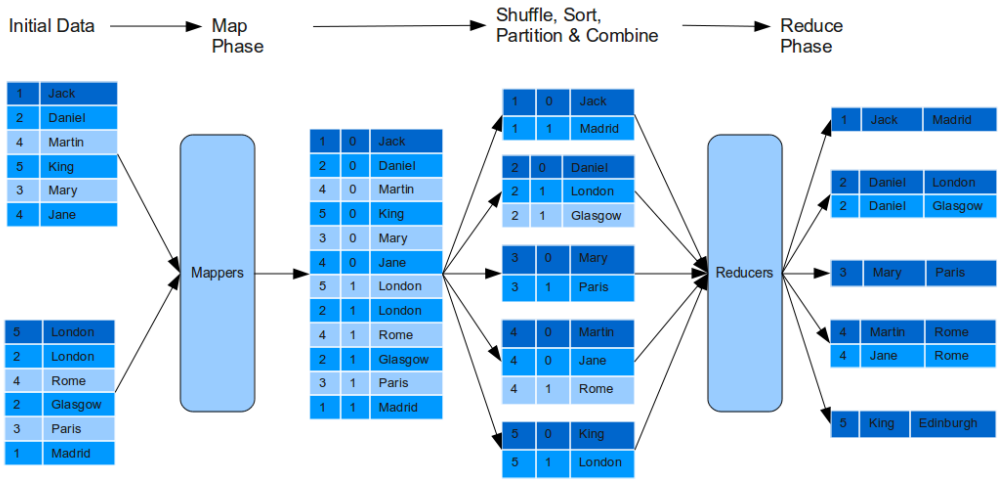

Figure 3.1: Reduce Side Join - Two Way

```
void map(Text key, Text values,
    OutputCollector<TextPair, TextPair> output, Reporter
       reporter) throws IOException {
    output.collect(new TextPair(key.toString(), tag),
          new TextPair(values.toString(), tag));
}
```

Listing 3.1: *Reduce-Side Join* - **map** function

Partitioning and Grouping Phase

The partitioner partitions the tuples among the reducers based on the join key such that all tuples from both datasets having the same key go to the same reducer (section 2.4.6). The default partitioner had to be specifically overridden to make sure that the partitioning was done only on the Key value, ignoring the Tag value. The Tag values are only to allow the reducer to identify a tuple's parent dataset.

```
int getPartition(TextPair key, TextPair value, int
   numPartitions) {
  return (key.getFirst().hashCode() & Integer.MAX_VALUE)
     % numPartitions;
}
```

Listing 3.2: *Reduce-Side Join* - Partitioner function

But this is not sufficient. Even though this will ensure that all tuples with the same key go to the same Reducer, there still exists a problem. The **`reduce`** function is called once for a key and the list of values associated with it. This list of values is generated by grouping together all the tuples associated with the same key (see section 2.4.6). We are sending a composite TextPair key and hence the Reducer will consider (key, tag) as a key. This means, for eg., two different **`reduce`** functions will be called for [Key1, Tag1] and [Key1, Tag2]. To overcome this, we will need to override the default grouping function as well. This function is essentially the same as the partitioner and makes sure the reduce groups are formed taking into consideration only the Key part and ignoring the Tag part. See figure 3.2 to understand better.

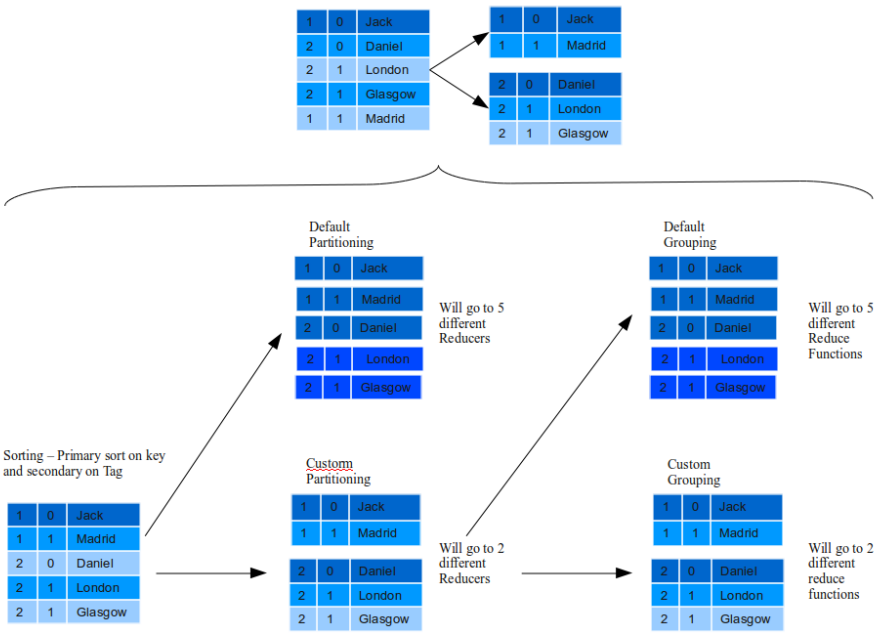

Figure 3.2: Custom Partitioning and Grouping

Reduce Phase

The framework sorts the keys (in this case, the composite TextPair key) and passes them on with the corresponding values to the Reducer. Since the sorting is done on the composite key (primary sort on Key and secondary sort on Tag), tuples from one table will all come before the other table. The Reducer then calls the **`reduce`** function for

each key group. The **reduce** function buffers the tuples of the first dataset. This is required since once the tuples are read from the HDFS stream (see section 2.4.2), we lose access to these values unless we reinitialize the stream. But we need these tuples in order to join them with the tuples from the second dataset. Tuples of the second dataset are simply read directly from the HDFS stream, one at time and joined with all the tuples from the buffer (all the values with one **reduce** function are for the same key) It was mentioned earlier that both key and values need to be tagged. This is because the tag attached with the key is used to do a secondary sort to ensure all tuples from one table are processed before the other. Once the **reduce** function is called, it will only get the first (key, tag) pair as its key (owing to the custom grouping function written which ignores the tags). Hence the values also need to be tagged for the **reduce** function to identify their parent datasets.

```
void reduce(TextPair key, Iterator<TextPair> values,
      OutputCollector<Text, Text> output,  Reporter
         reporter)
      throws IOException {
  ArrayList<Text> T1 = new ArrayList<Text>();
  Text tag = key.getSecond();
  TextPair value = null;
  while(values.hasNext())
  {
    value = values.next();
    if(value.getSecond().compareTo(tag)==0)
    {
      T1.add(value.getFirst());
    }
    else
    {
      for(Text val : T1)
      {
        output.collect(key.getFirst(),
          new Text(val.toString() + "\t"
          + value.getFirst().toString()));
      }
    }
  }
}
```

Listing 3.3: *Reduce-Side Join* - **reduce** function

Note that the this algorithm makes no assumption about the number of times a key repeats in either of the datasets and can handle duplicate keys.

3.2.2 Map-Side Join

The Reduce-Side join seems like the natural way to join datasets using Map/Reduce. It uses the framework's built-in capability to sort the intermediate key-value pairs before they reach the *Reducer*. But this sorting often is a very time consuming step. Hadoop offers another way of joining datasets before they reach the *Mapper*. This functionality is present out of the box and is arguably is the fastest way to join two datasets using Map/Reduce, but places severe constraints (see table 3.1) on the datasets that can be used for the join.

Table 3.1: Limitation of Map-Side Join [16]

Limitation	Why
All datasets must be sorted using the same comparator.	The sort ordering of the data in each dataset must be identical for datasets to be joined.
All datasets must be partitioned using the same partitioner.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.
The number of partitions in the datasets must be identical.	A given key has to be in the same partition in each dataset so that all partitions that can hold a key are joined together.

These constraints are quite strict but are all satisfied by any output dataset of a Hadoop job. Hence as a pre-processing step, we simply pass both the dataset through a basic Hadoop job. This job uses an *IdentityMapper* and an *IdentityReducer* which do no processing on the data but simply pass it through the framework which partitions, groups and sorts it. The output is compliant with all the constraints mentioned above.

Although the individual map tasks in a join lose much of the advantage of data locality, the overall job gains due to the potential for the elimination of the reduce phase and/or the great reduction in the amount of data required for the reduce. This algorithm also supports duplicate keys in all datasets. More can be found out about this kind of join in [16].

3.2.3 Broadcast Join

Broadcast Join was studied by Blanas et. al. in [6]. If one of the datasets is very small, such that it can fit in memory, then there is an optimization that can be exploited to avoid the data transfer overhead involved in transferring values from *Mappers* to *Reducers*. This kind of scenario is often seen in real-world applications. For instance, a small users database may need to be joined with a large log. This small dataset can be simply replicated on all the machines. This can be achieved by simply using `-files` or `-archive` directive to send the file to each machine while invoking the Hadoop job. Broadcast join is a Map-only algorithm

Map Phase

The *Mapper* loads the small dataset into memory and calls the **`map`** function for each tuple from the bigger dataset. For each (key, value), the **`map`** function probes the in-memory dataset and finds matches. This process can be further optimized by loading the small dataset into a Hashtable. It then writes out the joined tuples. The below shown *configure* function is part of the *Mapper* class and is called once for every *Mapper*. The below code reads the 'broadcasted' file and loads the same into an in-memory hash table.

```
public void configure(JobConf conf) {
  //Read the broadcasted file
  T1 = new File(conf.get("broadcast.file"));
  //Hashtable to store the tuples
  ht = new HashMap<String, ArrayList<String>>();
  BufferedReader br = null;
  String line = null;
    try{
      br = new BufferedReader(new FileReader(T1));
      while((line = br.readLine())!=null)
      {
        String record[] = line.split("\t", 2);
        if(record.length == 2)
        {
          //Insert into Hashtable
          if(ht.containsKey(record[0]))
          {
            ht.get(record[0]).add(record[1]);
          }
          else
          {
```

```
            ArrayList<String> value = new ArrayList<
               String>();
            value.add(record[1]);
            ht.put(record[0], value);
          }
        }
      }
    }
    catch(Exception e)
    {
      e.printStackTrace();
    }
}
```

Listing 3.4: Loading the small dataset into a Hash table

The next piece of code is the actual **map** function that receives the records from the HDFS and probes the HashTable containing the tuples from the broadcasted file. Notice that it takes care of duplicate keys as well.

```
public void map(LongWritable lineNumber, Text value,
        OutputCollector<Text, Text> output, Reporter
           reporter)
        throws IOException {
  String[] rightRecord = value.toString().split("\t",2);
  if(rightRecord.length == 2)
  {
    for(String leftRecord : ht.get(rightRecord[0]))
    {
        output.collect(new Text(rightRecord[0]), new Text
           (leftRecord + "\t" + rightRecord[1]));
    }
  }
}
```

Listing 3.5: Broadcast Join Map Function

Broadcast Join benefits from the fact that it uses a small 'local' storage on the individual nodes instead of the HDFS. This makes it possible to load the entire dataset into an in-memory hash table, access to which is very fast. But on the down side, it will run into problems when both the datasets are large and neither of them can be stored locally on the individual nodes.

3.3 Multi-Way Joins

The algorithms presented in the previous section on Two-Way joins were mainly work that already existed. This project extends these ideas to Multi-Way joins. This section details them. We compare the performance of this algorithms against the next two algorithms in the next chapter.

3.3.1 Map-Side Join

Map-Side joins are the same when it comes to Multi-Way joins. They can handle as many datasets as long as they conform to the constraints (Table 3.1) mentioned earlier.

3.3.2 Reduce-Side One-Shot Join

This is basically an extension of the Reduce-Side join explained earlier (see 3.2.1). A list of tables is passed as part of the job configuration (passes as `tables.tags`) to enable the *Mapper* and *Reducer* to know how many tables to expect and what tags are associated with which table. For instance, to join n datasets, $T_1, T_2, T_3, .., T_n$, this algorithm in simple relational algebra can be represented as -

$$T_1 \bowtie T_2 \bowtie ... \bowtie T_{n-1} \bowtie T_n$$

Map Phase

The *Mapper* reads `tables.tags` from the job configuration and calls the **`map`** function. The **`map`** function then proceeds to tag the tuples based on the dataset they originate from. This is similar to the map phase in section 3.2.1.

Partitioning and Grouping Phase

The *Partitioner* and the *Grouping* function are the exact same as explained in section 3.2.1. They partition and group based on just the key, ignoring the tag.

Reduce Phase

The reduce phase is slightly more involved than the two sided join. The *Reducer* as usual, gets the tuples sorted on the (key, tag) composite key. All tuples having the same value for the join key, will be received by the same *Reducer* and only one **`reduce`** function is called for one key value. Based on the number of tables passed as part of

the job configuration, the *Reducer* dynamically creates buffers to hold all but the last datasets. The last dataset is simply streamed from the file system. As a necessary evil, required to avoid running out of memory, the buffers are spilled to disk if they exceed a certain pre-defined threshold. This quite obviously will contribute to I/O and runtime. Once the tuples for a particular key are divided as per their parent datasets, it is now a cartesian product of these tuples. Subsequently, the joined tuples are written to the output.

Advantages and Drawbacks

- Advantages
  (a) Joining in one go means no setting up of multiple jobs as will be the case in the next section (3.3.3).
  (b) No intermediate results involved which could lead to substantial space savings.
- Disadvantages
  (a) Buffering tuples can easily run in to memory problems, especially if the data is skewed. In fact, this algorithm failed to complete and gave *OutOfMemory* exceptions a number of times when we tried to run it with datasets having more than 2.5 million tuples in our experiments (see section 4.4.1).
  (b) If used for more than 5 datasets, its highly likely that the buffer will overflow.

3.3.3 Reduce-Side Cascade Join

This is a slightly different implementation of the *Reduce-Side Join* for Multi-Way joins. Instead joining all the datasets in one go, they are joined two at a time. In other words, it is an iterative implementation of the two-way *Reduce-Side Join* The implementation is the same as the *Reduce-Side Join* mentioned in section 3.2.1. The calling program is responsible for creating multiple jobs for joining the datasets, two at a time. Considering n tables, $T_1, T_2, T_3, .., T_n$, T_1 is joined with T_2 as part of one job. The result of this join is joined with T_3 and so on. Expressing the same in relational algebra -

$$(...(((T_1 \bowtie T_2) \bowtie T_3) \bowtie T_4)... \bowtie T_{n-1}) \bowtie T_n$$

Advantages and Drawbacks

- Advantages
  (a) Data involved in one Map/Reduce job is lesser than the algorithm mentioned in section 3.3.2. Hence lesser load on the buffers and better I/O.
  (b) Datasets of any size can be joined provided there is space available on the HDFS.
  (c) Any number of datasets can be joined given enough space on the HDFS.
- Disadvantages
  (a) Intermediate results can take up a lot of space. But this can be optimized to some extent as explained in the next section.
  (b) Setting up the multiple jobs on the cluster incurs a non-trivial overhead.

3.3.3.1 Reduce-Side Cascade Join - Optimization

Simply joining datasets two at a time is very inefficient. The intermediate results are usually quite large and will take up a lot of space. Even if these are removed once the whole join is complete, they could put a lot of strain on the system while the cascading joins are taking place. Perhaps there is a way of reducing the size of these intermediate results. There are two ways this can be achieved -

- Compressing the intermediate results
- Join the datasets in increasing order of the output cardinality of their joins, i.e., join the datasets that produce the least number of joined tuples first, then join this result with the next dataset which will produce the next lowest output cardinality.

Optimization using compression

Compression was explained in section 2.4.9. Compressing the results will not only save space on the HDFS, but in case the subsequent join's *Map* task needs to fetch a non-local block for processing, it will also result in lower number of bytes transferred over the network.

Optimization using Output Cardinality

Deciding the order of joining datasets using the output cardinality is more involved. Lets consider two sample datasets -

(a) Table 1

Key	Value
1	ABC
2	DEF
2	GHI
3	JKL

(b) Table 2

Key	Value
1	LMN
2	PQR
3	STU
3	XYZ

Table 3.2: Sample Datasets

The output of joining these two datasets will be -

Key	Table 1	Table 2
1	ABC	LMN
2	DEF	PQR
2	GHI	PQR
3	JKL	STU
3	JKL	XYZ

Table 3.3: Joined result of the sample datasets

The thing to notice is that the output will have

$$T_1(K_1) * T_2(K_1) + T_1(K_2) * T_2(K_2) ... T_1(K_n) * T_2(K_n)$$

$$= \sum_{i=1}^{n} T_1(K_i) * T_2(K_i)$$

number of rows. where $T_m(K_n)$ represents the number of tuples in table T_m having the key K_n. In our case, it will be

$$1 * 1 + 2 * 1 + 1 * 2 = 5$$

Hence, if we find out the number of keys in each dataset and the corresponding number of tuples associated with each key in those datasets, we can find the number of output tuples in the join, or in other words, the output cardinality of the joined dataset. This

can be very easily done as part of a pre-processing step. This pre-processing is done using counters that were mentioned on page 17. To accomplish this, output cardinalities are found pair-wise for all datasets at pre-processing time and stored in files. When the join is to be executed, the datasets with the lowest output cardinality are chosen and joined first. This intermediate result this then joined with the dataset that has the lowest join output cardinality with either of the two tables joined previously. In the next round, the dataset with the lowest join output cardinality with any of the three datasets joined so far is chosen to join with the intermediate result and so on. The next chapter shows the results of experimental evaluation and compares and contrasts the gains that such optimization leads to.

Chapter 4

Evaluation and Analysis

4.1 Enviroment

Experiments were conducted on 3 different kinds of clusters (see section 4.2.1). There were two types of machines used that were slightly different from each other -

	Type 1	Type 2
CPU	Intel® Xeon™CPU 3.20GHz	Intel® Xeon™CPU 3.20GHz
Cores	4	4
Memory	3631632 KB	3369544 KB
Cache Size	2 MB	1 MB
OS	Scientific Linux 5.5	Scientific Linux 5.5
Linux Kernel	2.6.18-194	2.6.18-194

Table 4.1: Types of machines used in the Experimental Cluster

All machines were running Clouders Inc.'s[3] distribution of Hadoop - version 0.18.3+76. File system used in all the experiments was HDFS.

4.2 Experimental Setup

4.2.1 Cluster Setup

There were three types of clusters used -

	Data Node(s)	Name Node(s)	Job Tracker(s)	Total Node(s)
Type 1	1	1	1	1
Type 2	3	1	1	5
Type 3	6	1	1	8

Table 4.2: Types of clusters used for the experiments

4.3 Two-Way Joins

4.3.1 Experiment 1 : Performance of two-way joins on increasing input data size

A series of experiments were conducted to compare the runtime and scalability of the various algorithms. The first of these involved running the three two-way algorithms mentioned earlier on increasing data sizes using the Type 3 (see Table 4.2.1) cluster. Both datasets involved in the joins had the following characteristics -

- Same number of tuples
- Same key space
- $\#keys = \frac{1}{10}\ \#tuples$

We tested on two kinds of key distributions as shown in Figure 4.1 and Figure 4.2. In the case of skewed data, one of the datasets had a key that was repeated 50% of the times. Which means that given n rows, $^n/_2$ of them were the same key. The other dataset had uniform key distribution

Concentrating first on the Uniform Key Distribution graph (Figure 4.1), it can be seen that all the three algorithms gave comparable numbers at lower input data sizes. But Map-side join always performed the best, even with large input datasets. Broadcast join deteriorated the fastest as the data size increased. This is because a larger file had to be replicated across all the machines in the cluster as the input size increased. Figure 4.2 shows something interesting. Map-side join gave a worse time than the other two algorithms for the input with 5 million rows when then input dataset had skewed key distribution. This can be attributed to the unequal partitioning that will occur due to the skew.

Apart from testing for runtimes, we also checked for the amount of data transferred over the network from the machines running the *Map* tasks to the machines running

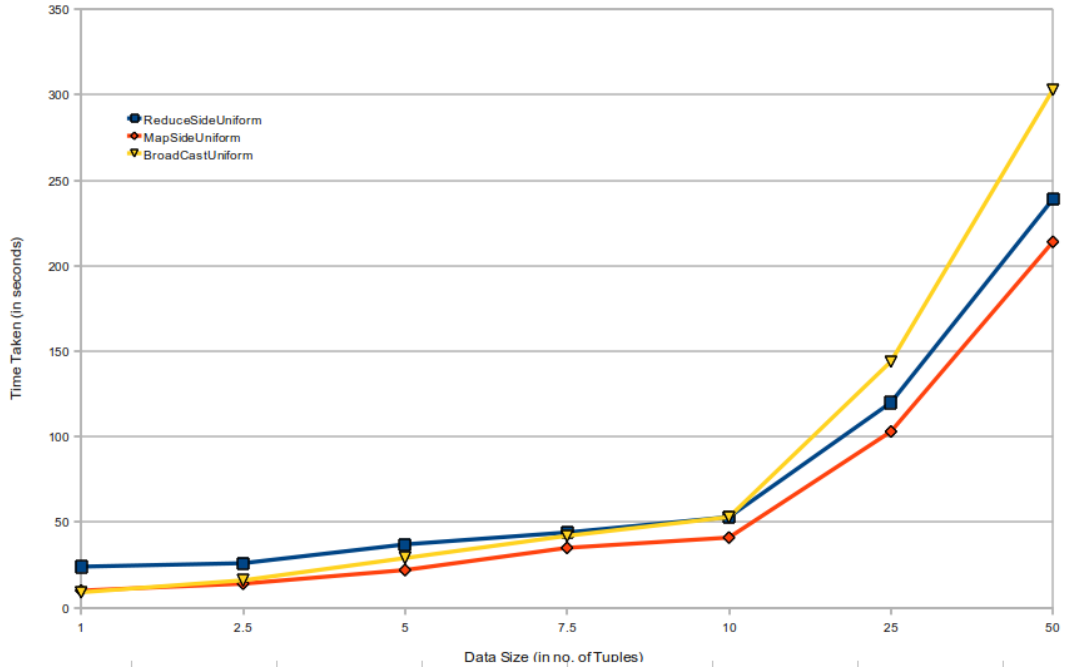

Figure 4.1: Experiment 1 : Uniform Key Distribution

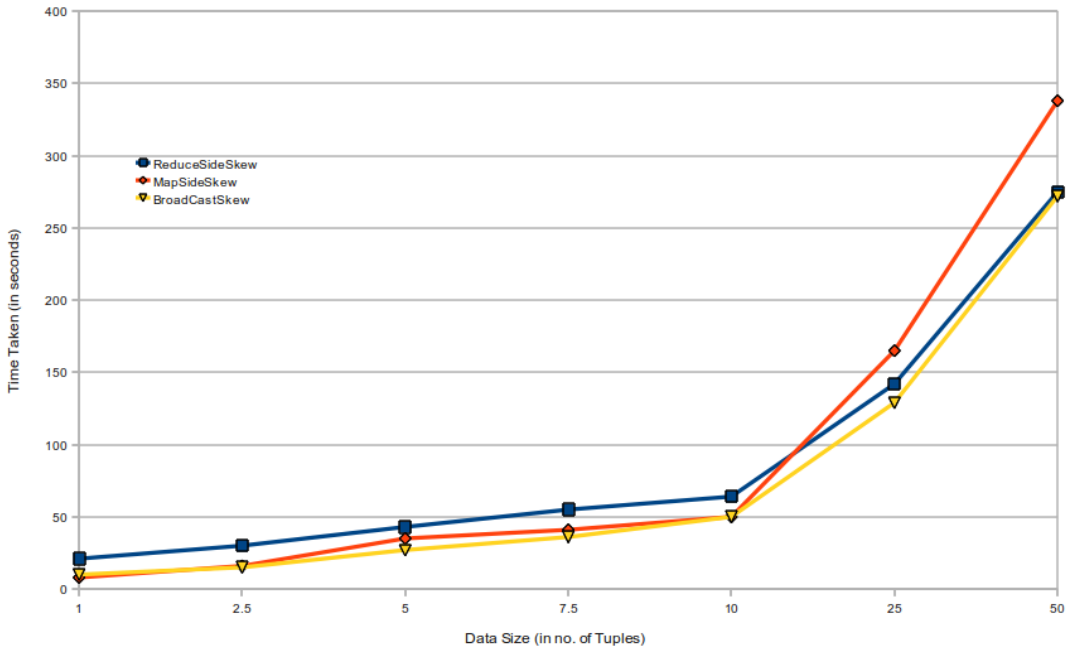

Figure 4.2: Experiment 1 : Skewed Key Distribution

the *Reduce* tasks. A simple number like the number of bytes transferred might not be

very meaningful. Hence we took the ratio -

$$\frac{Bytes\ transferred\ over\ the\ network}{Initial\ Data\ Size}$$

We call this the *Data Shuffle* ratio. This ratio is interesting because it gives an idea of how well *Hadoop* achieves data locality and whether it improves with bigger data sizes. Table 4.3 and Table 4.4 show the actual run-times of the different algorithms shown in Figure 4.1 and Figure 4.2. Also given are the number of bytes shuffled across machines from the *Map* phase to the *Reduce* phase in the *Reduce-Side* Join. Note that *Map-Side* Join and *Broadcast* Join do not have a reduce side and hence no data is shuffled across machines between the *Map* phase and the *Reduce* phase. The sixth column shows the time taken for pre-processing in the case of the *Map-Side* Join algorithm.

Tuples (x100000)	Reduce-Side Join	Data Shuffled (bytes)	Data Shuffle Ratio	Map-Side Join	Pre-Processing	Broadcast Join
1	24s	9178000	1.150	9s	38s	10s
2.5	26s	23278000	1.148	14s	45s	16s
5	37s	46778000	1.147	22s	46s	29s
7.5	44s	70278000	1.147	35s	47s	42s
10	53s	93778000	1.147	41s	49s	53s
25	120s	237778000	1.144	103s	56s	144s
50	239s	477778000	1.144	214s	71s	303s

Table 4.3: Experiment 1 : Uniform Key Distribution

Tuples (x100000)	Reduce-Side Join	Data Shuffled (bytes)	Data Shuffle Ratio	Map-Side Join	Pre-Processing	Broadcast Join
1	21s	8194793	1.1.27	8s	41s	10s
2.5	30s	23243259	1.148	16s	42s	15s
5	43s	46805826	1.147	35s	42s	27s
7.5	55s	70118029	1.147	41s	46s	36s
10	64s	93806021	1.147	50s	46s	50s
25	142s	238055988	1.144	165s	54s	129s
50	275s	478056409	1.144	338s	76s	272s

Table 4.4: Experiment 1 : Skewed Key Distribution

There are two main things worth noticing in Table 4.3 and Table 4.4. The first is that the *Data Shuffle* ratio remains around 1.14 for both uniform and skewed key distribution. This is because as the data size increases, Hadoop is able to better manage data distribution and data locality. The other thing is that though Map-Side performed best most of the times, it took an average time of 50.29 seconds (uniform key distribution)

and 49.57 seconds (skewed key distribution) for the pre-partitioning phase. This may not seem a lot, but is likely to be quite high if the input datasets are much larger than the ones used in this experiment.

4.3.2 Experiment 2 - Two-way Join algorithms across different clusters

Instead of just testing the scalability of an algorithm in terms of the input data, we decided to find out the behaviour of these algorithms with different kinds of clusters as well. For this, the algorithms were run against all the 3 kinds of clusters mentioned in Table 4.2.1 with the data having the following characteristics -

- Both tables contained 1 million tuples.
- Both tables contained the same number of keys - $\#keys = \frac{1}{10}\ \#tuples$ spread across the same key space.

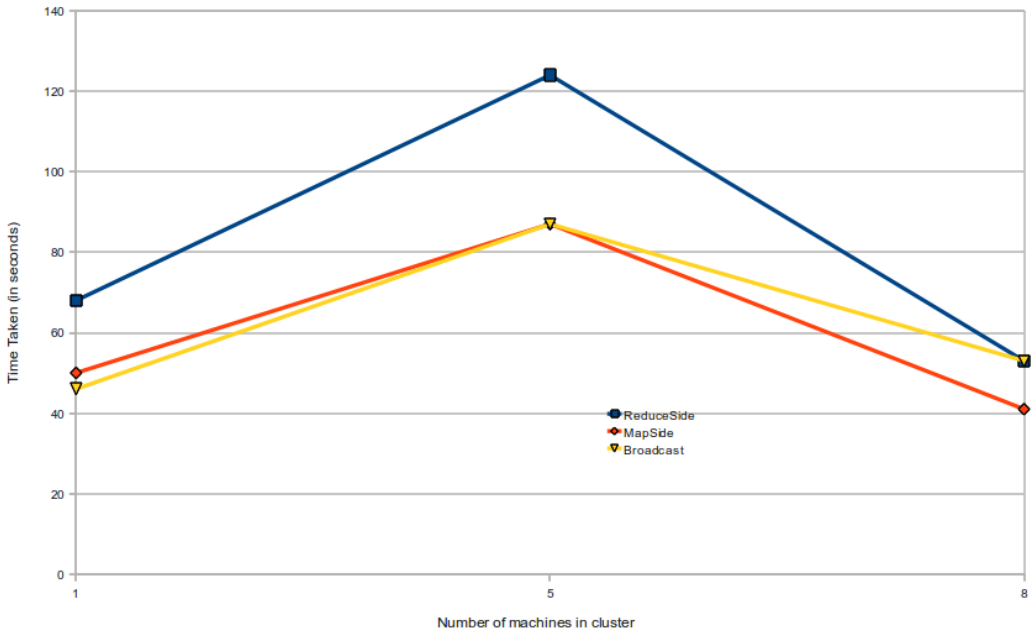

Figure 4.3: Experiment 2 : Two-way Join algorithms across different clusters

We noticed something that we were not expecting to see. The 1-machine cluster, performed better than the 5-machine cluster for all the three algorithms. The performance then improved again when the algorithms were run on the 8-machine cluster.

On checking the experiment logs, we concluded that this was because on a 1-machine cluster, all data was present locally and there was no data transfer that happened across the network. In the case of the *Reduce-Side Join* algorithm the time taken to transfer data from the machines running the *Map* task to the machines running the *Reduce* tasks was considerable in the 5-machine cluster, whereas all the data was local in the case of the 1-machine cluster. In case of *Map-Side Join* as well, the input data was all present locally on the machine running the job and no input had to be fetched for other data nodes. Similarly for the *Broadcast Join* the file is not broadcast across the machines in the cluster, but is just broadcast from the master-node to the task-node, both running on the same machine. The overall performance improved in the case of the 8-machine cluster because there were more machines available for processing the task in a well partitioned distributed fashion. The individual *Map* tasks performed smaller units of work and finished much faster.

No. of Nodes	Reduce-Side Join	Data Shuffled (bytes)	Data Shuffle Ratio	Map-Side Join	Pre-Processing	Broadcast Join
1	68s	0	0.000	50s	169s	46s
5	124s	93777932	1.147	87s	55s	87s
8	53s	93778000	1.147	41s	49s	53s

Table 4.5: Experiment 2 : Two-way Join algorithms across different clusters

Table 4.5 shows the actual numbers recorded across the three different kinds of clusters. Once again, in the case of *Reduce-Side Join*, we found the value of the *Data Shuffle* ratio to analyse data transferred from *Mappers* to *Reducers* and how well *Hadoop* achieves data locality with increasing number of nodes. Notice that once again the ratio remained close to 1.14. Also notice that this ratio is 0 for the 1-machine cluster since, obviously, there is only 1 machine and there is no data transferred across the network.

4.3.3 Experiment 3 - Performance of *Broadcast Join*

The major use-case of the *Broadcast Join* is when one of the datasets is small in comparison to the other. So we decided to test this scenario by keeping one of the datasets of constant size and increasing the size of the other. The dataset characteristics were as follows -

- The larger dataset size was fixed at 1,000,000 tuples

- The size of the smaller dataset was varied from 100,000 tuples to 1,000,000 tuples

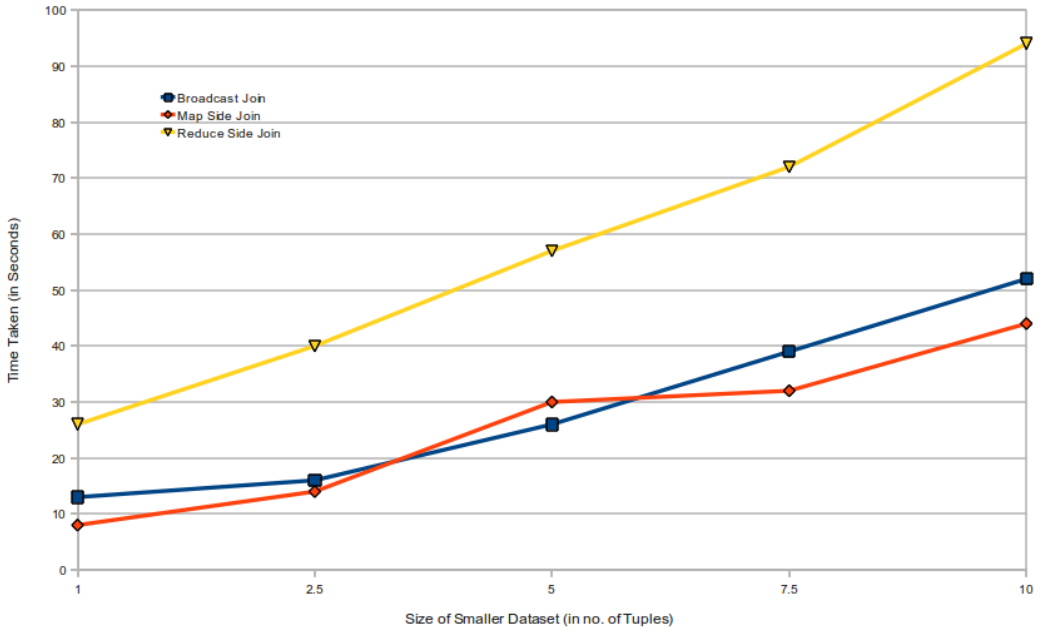

Figure 4.4: Experiment 3 : *Broadcast Join* Performance

Broadcast Join gave performances that were very close to the ones by *Map-Side Join*, but without any pre-processing step involved. Performance of *Reduce-Side Join* was the worst when compared to the other two algorithms.

Table 4.6 gives the actual numbers that were recorded for the graph shown in Figure 4.4. Once again, notice that the *Map-Side Join* involved an average pre-processing time of 45.6 seconds which is not required for the *Broadcast Join* algorithm.

No. of tuples (x100000) in smaller relation	Broadcast Join	Map Side Join	Pre-Processing	Reduce Side Join
1	13	8	41	26
2.5	16	14	51	40
5	26	30	43	57
7.5	39	32	46	72
10	52	44	47	94

Table 4.6: Experiment 3 : Broadcast Join Performance

4.4 Multi-Way Joins

Similar experiments were run for the multi-way joins as well. The experiments were run for 3 datasets at a time. The deductions can easily be extrapolated for higher number of input datasets.

Please note that unless otherwise specified, *Reduce-Side Cascade Join* includes the optimizations explained in section 3.3.3.

4.4.1 Experiment 4 : Performance of multi-way joins on increasing input data size

Like experiment 1 (section 4.3.1, this experiment involves running the multi-way algorithms discussed in section 3.3 on increasing sizes of input datasets. The experiment was run with three datasets of the same size, all of which had the following characteristics -

- Same number of tuples
- Same key space
- $\#keys = \frac{1}{10}\ \#tuples$

Figure 4.5 and figure 4.6 show the results of the experiment. Figure 4.5 shows the results when the experiment was conducted with datasets that had uniform key distribution. Figure 4.6 shows the results when two of the input datasets had a skewed key distribution and the third one had a uniform key distribution. Skewed datasets, as in experiment 1 (section 4.3.1) had one of the keys occurring in half of the tuples in the datasets. The keys exhibiting the skew were randomly chosen for each dataset. Optimization involving output cardinality of the joins (mentioned for *Reduce-Side Cascade Join* in section 3.3.3) were not tested with uniform input since the output cardinality of both the smaller joins would be the same and cannot be exploited in this case. The optimizations were tested for skewed input and as expected, showed improvement against test-runs with no optimization.

The graphs show that the *Reduce-Side One-Shot Join* algorithm and *Map-Side Join* algorithm gave performances that were very close to each other relative to the times taken by *Reduce-Side Cascade Join* algorithm. Though this may lead one to think that *Reduce-Side Cascade Join* algorithm is a bad choice for joining datasets, this in

fact is not true. If you notice, the largest size of the datasets used for this experiment contained 2,500,000 tuples. The choice was intentional since *Reduce-Side One-Shot Join* failed to complete most of the times when used with datasets of higher sizes and hence we could not compare the times when datasets having more than 2,500,000 tuples were used. Though we do not claim that this will be the case when clusters of higher sizes are used, this certainly was the case with our cluster of 8 machines. The system gave *OutOfMemory* exceptions a number of times for *Reduce-Side One-Shot Join* algorithm when used with datasets that contained more than 2.5 million tuples each. In contrast, the *Reduce-Side Cascade Join* algorithm was able to handle dataset with each of them containing 5 million rows, albeit it took a very long time to complete (one of the test-runs took 9427 seconds, which is over 2 and half hours)

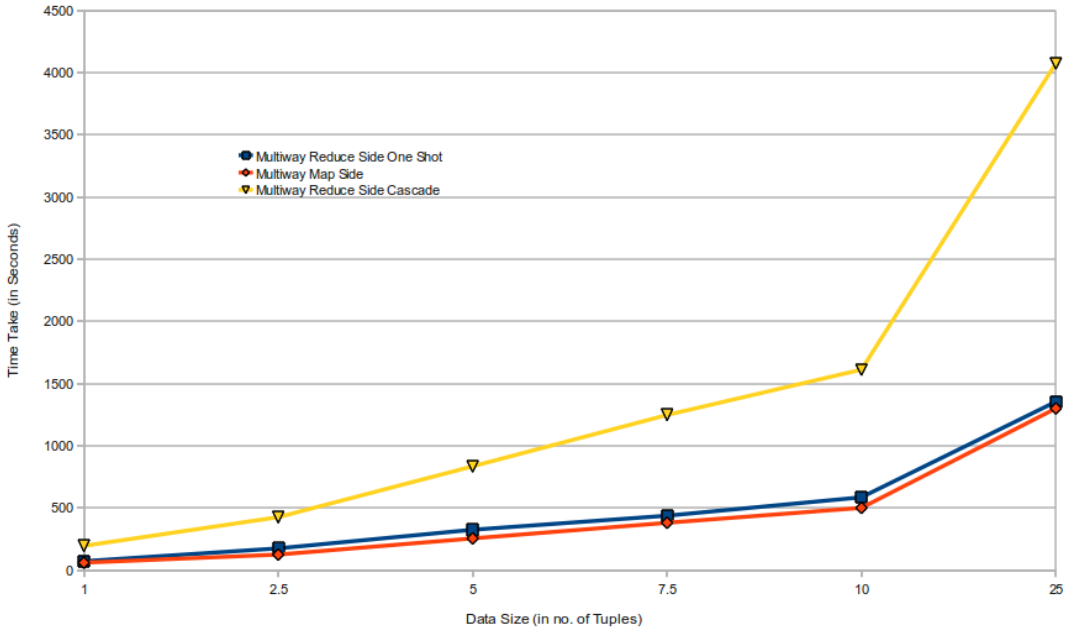

Figure 4.5: Experiment 4 : Uniform Key Distribution

Tables 4.7 and 4.8 show the actual values that were recorded to construct graphs in Figures 4.5 and 4.6. Even here, like in earlier experiments, we can notice that the *Data Shuffle* ratio remains close to 1.14. This means that with increasing data, Hadoop is able to achieve better data locality. Table 4.6 also shows the times for test-runs when no optimization involving output-cardinality was used for *Reduce-Side Cascade Join*.

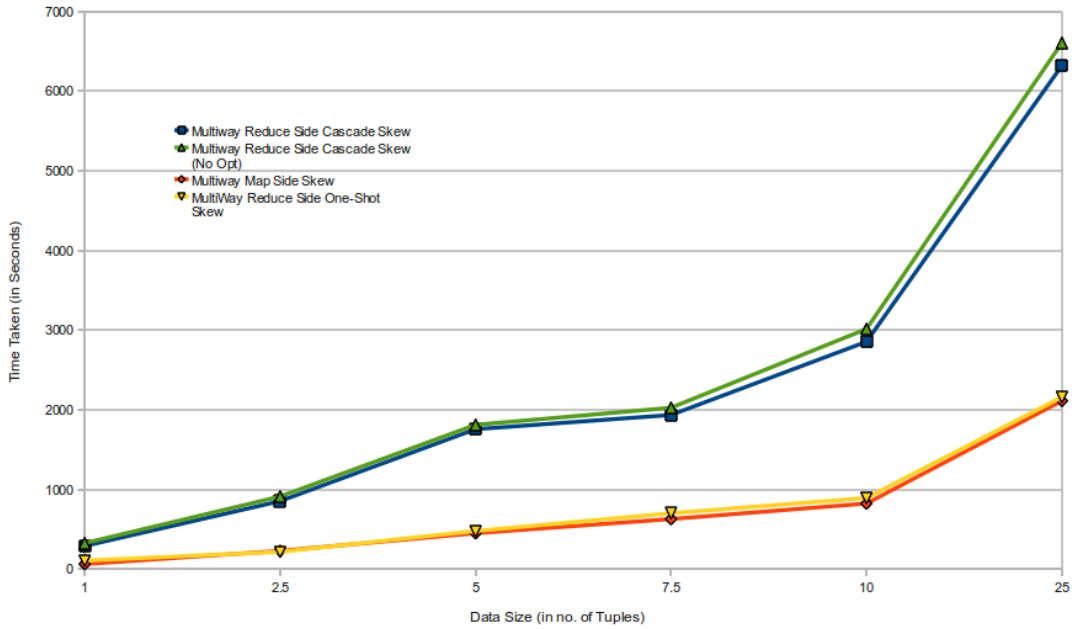

Figure 4.6: Experiment 4 : Skewed Key Distribution

Tuples (x100000)	Reduce-Side One-Shot Join	Data Shuffle Ratio	Map-Side Join	Reduce-Side Cascade Join
1	71s	1.150	59s	194s
2.5	174s	1.148	124s	425s
5	324s	1.147	254s	834s
7.5	437s	1.147	380s	1247s
10	585s	1.147	500s	1612s
25	1354s	1.144	1300s	4076s

Table 4.7: Experiment 4 : Uniform Key Distribution

Tuples (x100000)	Reduce-Side One-Shot Join	Data Shuffle Ratio	Map-Side Join	Reduce-Side Cascade Join	Reduce-Side Cascade Join (No-Opt)
1	106	1.150	62	289	325
2.5	216	1.148	228	852	910
5	477	1.147	449	1755	1814
7.5	700	1.147	627	1933	2024
10	891	1.147	822	2853	3013
25	2158	1.144	2111	6319	6604

Table 4.8: Experiment 4 : Skewed Key Distribution

4.4.2 Experiment 5 : Two-way Join algorithms across different clusters

For the multi-way join algorithms as well, like in experiment 2 (section 4.3.2) for two-way join algorithms, we ran the three algorithms on different types of clusters (see Table 4.2.1). The datasets that we used had -

- 1 million tuples each.
- The same number of keys - $\#keys = \frac{1}{10}\ \#tuples$ spread across the same key space.

Figure 4.7 shows the results that were observed. The results were similar to the unexpected ones observed in experiment 2 (section 4.3.2). 1-machine cluster performed better than the 5-machine cluster with the performance then improving again for the 8-machine cluster. On checking the logs, we found the very same reasons for this anomaly. There was no network traffic involved in the case of 1-machine cluster and this was a dominant factor in case of the 5-machine cluster. The 8-machine cluster performed better since the smaller units of work that were well distributed across the cluster took lesser time to complete and brought down the overall time.

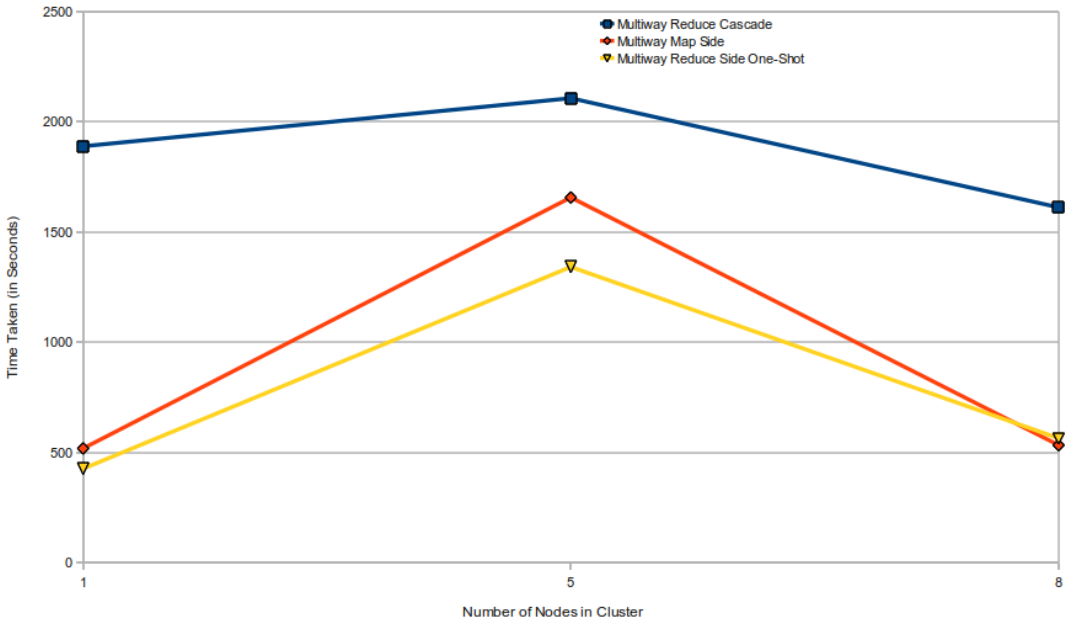

Figure 4.7: Experiment 5 : Multi-way Join algorithms across different clusters

Table 4.9 shows the actual numbers that were recorded for the graph in figure 4.7.

No. of Nodes	Reduce-Side One-Shot Join	Data Shuffle Ratio	Map-Side Join	Pre-Processing	Reduce-Side Cascade Join	Pre-Processing
1	427	0	519	83	1889	85
5	1342	1.147	1657	243	2107	96
8	565	1.147	534	71	1612	32

Table 4.9: Experiment 5 : Multi-way Join algorithms across different clusters

Chapter 5

Discussion and Conclusion

5.1 Overview

Though an elegant solution, Map/Reduce has its pros and cons like any other interesting idea. Quite obviously there have been proponents and opponent of this ideology.

In this project, we set out to test the viability of the Map/Reduce framework for joining datasets as part of database query processing. The motivation stems from the fact that the rate at which information is exploding in today's world has forced us to look at database query processing from a different perspective. To keep up with its pace, use of distributed processing no longer seems to be an option, but seems to be a necessity. Map/Reduce was a novel idea that was developed at Google for simple computations. These computation, though simple, required very large inputs and thus necessitated the use of distributed processing to get results in reasonable amounts of time. Over a period of time, Map/Reduce started getting used for myriad of different application and it was a matter of time before data warehousing specialists started using it for the purposes of query processing over database systems.

It was soon picked-up by the open source community and developed into an open-source framework and implementation called Hadoop - the framework and the Map/Reduce implementation we used in this project. One of the main queries performed over a database (or a dataset) is a *Join*. We specifically explored the options available for joining datasets using Map/Reduce (particularly Hadoop in our case). The aim was to consolidate and extend the existing algorithms, propose new ones and quantitatively evaluate and analyse their performances.

The algorithms were broken into two categories with three algorithms discussed in both categories each -

- Two-way joins - joins involving only 2 datasets.
  - Reduce-Side Join
  - Map-Side Join
  - Broadcast Join
- Multi-way joins - joins involving more than 2 datasets.
  - Map-Side Join (for multi-way joins)
  - Reduce-Side One-Shot Join
  - Reduce-Side Cascade Join

We also suggested optimizations for *Reduce-Side Cascade Join* exploiting the output cardinality of smaller joins involved in the algorithm. Subsequently we performed quantitative performance tests by changing input data sizes and number of machines participating in the cluster.

5.2 Conclusion

The first thing that impressed us while undertaking this project was the ease with which one could program a distributed application using Map/Reduce or Hadoop. Without any considerable prior knowledge about distributed systems we were able to easily install and configure Hadoop on on our development machines. Being an open-source produce and free to download, it is an inexpensive solution for someone looking to harnessing the power of a distributed system for solving problems. It is indeed an inexpensive and simple, yet powerful solution for exploiting the potential of parallel processing.

We found some quite interesting results from our experiments on Join algorithms using Map/Reduce. Even though our clusters were of relatively small sizes, we could notice from our experiments how the Hadoop framework exploited the cluster architecture when more nodes were added (as was seen with the *Data Shuffle* ratio that remained the same throughout our experiments). This gave us a first-hand feel of why Hadoop seems to be enjoying such a high popularity among data warehousing specialists.

The experiments on the two-way join algorithms concurred with the results observed by Blanas et. al. in [6]. We found that Map-Side joins performs the best when

the datasets have relatively uniform key distribution. But this comes at the cost of a pre-processing step. *Broadcast Join* is a better option when one of the datasets is much smaller compared to the other and the dataset is small enough to be easily replicated across all the machines in the cluster. It deteriorated the fastest when the input data size was increased. *Reduce-Side Join* gave almost the same performance for both uniform and skewed key distributions whereas *Map-Side Join* seemed to give slightly unpredictable results with skewed data. We also noticed that the 1-machine cluster gave better performances than the 5-machine cluster since there was no data transfer involved over the network in case of the 1-machine cluster. But the performance started to improve again with 8-machine cluster.

The same behaviour of the 1-machine cluster was observed in the case of multi-way joins as well while being tested on different clusters. When we tested the multi-way algorithms with increasing data sizes, we successfully showed that our optimization for *Reduce-Side Cascade Join* using output cardinality improved the performance of the algorithm. Although this came at the cost of pre-processing step and was the slowest of the algorithms, it was successful in joining datasets that contained over 2.5 million tuples whereas *Reduce-Side One-Shot Join* gave *OutOfMemory* exception a number of times.

5.3 Future Work

We explored the practicality of using Map/Reduce for joining datasets and found it quite easy to use and implement over different sizes of cluster. The largest cluster we had at our disposal contained only 8 machines. The first interesting thing would be to test all the algorithms discussed on clusters of much larger size, with much larger data sizes. Especially the multi-way joins since the authors in [6] did conduct experiments on two-way joins on a cluster containing 100 nodes. Another idea would be to try and use a combination of the algorithms for better performance. For instance, in the case of *Reduce-Side Cascade Join*, the intermediate result is suitable for input to a *Map-Side Join*. It might be possible to run a pre-processing step in parallel to the first join such that the rest of the datasets are ready for joining by the time the first join produces the intermediate result. This may involve exploring the scheduling mechanisms of Hadoop or any other implementation of Map/Reduce. Another possible area to explore could be the use of indices to speed-up the join queries. It might be possible to construct data structures to hold information about the HDFS blocks holding the datasets such

that the user program will know which block a particular tuple from a dataset corresponds to. This project looked at query evaluation. There is always the aspect of query optimization that chooses the join algorithm best suited for a particular situation.

Appendix A

TextPair Class

The code below has been reproduced from [18]

```
//cc TextPair A Writable implementation that stores a
   pair of Text objects
//cc TextPairComparator A RawComparator for comparing
   TextPair byte representations
//cc TextPairFirstComparator A custom RawComparator for
   comparing the first field of TextPair byte
   representations
//vv TextPair
import java.io.*;
import org.apache.hadoop.io.*;

public class TextPair implements WritableComparable<
   TextPair> {

  private Text first;
  private Text second;

  public TextPair() {
    set(new Text(), new Text());
  }

  public TextPair(String first, String second) {
    set(new Text(first), new Text(second));
  }

  public TextPair(Text first, Text second) {
    set(first, second);
  }

  public void set(Text first, Text second) {
    this.first = first;
```

```
    this.second = second;
  }

  public Text getFirst() {
    return first;
  }

  public Text getSecond() {
    return second;
  }

  @Override
  public void write(DataOutput out) throws IOException {
    first.write(out);
    second.write(out);
  }

  @Override
  public void readFields(DataInput in) throws IOException
      {
    first.readFields(in);
    second.readFields(in);
  }

  @Override
  public int hashCode() {
    return first.hashCode() * 163 + second.hashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof TextPair) {
      TextPair tp = (TextPair) o;
      return first.equals(tp.first) && second.equals(tp.
          second);
    }
    return false;
  }

  @Override
  public String toString() {
    return first + "\t" + second;
  }

  @Override
  public int compareTo(TextPair tp) {
```

```
    int cmp = first.compareTo(tp.first);
    if (cmp != 0) {
      return cmp;
    }
    return second.compareTo(tp.second);
  }
  // ^^ TextPair

  // vv TextPairComparator
  public static class Comparator extends
     WritableComparator {

    private static final Text.Comparator TEXT_COMPARATOR
       = new Text.Comparator();

    public Comparator() {
      super(TextPair.class);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1,
        byte[] b2, int s2, int l2) {

      try {
        int firstL1 = WritableUtils.decodeVIntSize(b1[s1
           ]) + readVInt(b1, s1);
        int firstL2 = WritableUtils.decodeVIntSize(b2[s2
           ]) + readVInt(b2, s2);
        int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1
           , b2, s2, firstL2);
        if (cmp != 0) {
          return cmp;
        }
        return TEXT_COMPARATOR.compare(b1, s1 + firstL1,
           l1 - firstL1,
            b2, s2 + firstL2, l2 - firstL2);
      } catch (IOException e) {
        throw new IllegalArgumentException(e);
      }
    }
  }

  static {
    WritableComparator.define(TextPair.class, new
       Comparator());
  }
```

```
  // ^^ TextPairComparator

  // vv TextPairFirstComparator
  public static class FirstComparator extends
     WritableComparator {

    private static final Text.Comparator TEXT_COMPARATOR
       = new Text.Comparator();

    public FirstComparator() {
      super(TextPair.class);
    }

    @Override
    public int compare(byte[] b1, int s1, int l1,
        byte[] b2, int s2, int l2) {

      try {
        int firstL1 = WritableUtils.decodeVIntSize(b1[s1
           ]) + readVInt(b1, s1);
        int firstL2 = WritableUtils.decodeVIntSize(b2[s2
           ]) + readVInt(b2, s2);
        return TEXT_COMPARATOR.compare(b1, s1, firstL1,
           b2, s2, firstL2);
      } catch (IOException e) {
        throw new IllegalArgumentException(e);
      }
    }

    @Override
    public int compare(WritableComparable a,
       WritableComparable b) {
      if (a instanceof TextPair && b instanceof TextPair)
          {
        return ((TextPair) a).first.compareTo(((TextPair)
            b).first);
      }
      return super.compare(a, b);
    }
  }
  // ^^ TextPairFirstComparator

  //vv TextPair
}
//^^ TextPair
```

Listing A.1: TextPair Class

Appendix B

Partitioner Class

The code below has been reproduced from [18]

```
public static class KeyPartitioner implements
   Partitioner<TextPair, TextPair> {
    @Override
    public int getPartition(TextPair key, TextPair
       value, int numPartitions) {
      return (key.getFirst().hashCode() & Integer.
         MAX_VALUE) \% numPartitions;
    }

  @Override
  public void configure(JobConf conf) {}
  }
```

Listing B.1: Partitioner Class

Bibliography

[1] Apache cassandra.

[2] Apache hadoop. Website. `http://hadoop.apache.org`.

[3] Cloudera inc. `http://www.cloudera.com`.

[4] Hadoop wiki - poweredby. `http://wiki.apache.org/hadoop/PoweredBy`.

[5] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, and A. Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2(1):922–933, 2009.

[6] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *SIGMOD '10: Proceedings of the 2010 international conference on Management of data*, pages 975–986, New York, NY, USA, 2010. ACM.

[7] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, 2008.

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.

[11] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110, New York, NY, USA, 2008. ACM.

[12] K. Palla. A comparative analysis of join algorithms using the hadoop map/reduce framework. Master's thesis, School of Informatics, University of Edinburgh, 2009.

[13] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 165–178, New York, NY, USA, 2009. ACM.

[14] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. Mapreduce and parallel dbmss: friends or foes? *Commun. ACM*, 53(1):64–71, 2010.

[15] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, 2009.

[16] J. Venner. *Pro Hadoop*. Apress, 1 edition, June 2009.

[17] S. Viglas. Advanced databases. Taught Lecture, 2010. `http://www.inf.ed.ac.uk/teaching/courses/adbs`.

[18] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1 edition, June 2009.

[19] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1029–1040, New York, NY, USA, 2007. ACM.