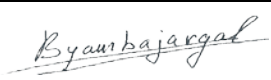




FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study Program: Master Degree Program in Computer Science Specialization: Computer Science	Spring, 2011 Open/Confidential
Author: Byambajargal Byambajav	 (signature of author)
Supervisors: Professor Chunming Rong, (UiS) External supervisor(s): Tomasz Wiktor Wlodarczyk, (UiS)	
Title of Thesis: Methods for Large-scale Semantic Expansion on Hadoop Architecture	
ECTS: 30	
Subject headings: <ul style="list-style-type: none">• Distributed systems• Semantic technologies	Pages: Stavanger, 15/06/2011

Methods for Large-scale Semantic Expansion on Hadoop
Architecture

By

Byambajargal Byambajav

Thesis is submitted in partial fulfillment of the
Requirements for the degree of MASTER DEGREE

In Computer Science

Specialization: Computer Science



FACULTY OF SCIENCE AND TECHNOLOGY

University of Stavanger

2011

Methods for Large-scale Semantic Expansion on Hadoop Architecture



Byambajargal Byambajav

Faculty of Science and Technology Department of Electrical and
Computer Engineering

University of Stavanger, N-4036 Stavanger, Norway

2011 june

Abstract

Volume of publicly available data in biomedicine is constantly increasing. However, this data is stored in different formats on different platforms. Integrating this data will enable us to facilitate the pace of medical discoveries by providing scientists with a unified view of this diverse information. Under the auspices of the National Center for Biomedical Ontology, we have developed the Resource Index a growing, large-scale index of more than twenty diverse Biomedical resources. (13)

The purpose of this thesis is to scaling out the semantic annotation data of NCBO Resource Index (13) that they have implemented on MySQL server on single machine. In order to improve the performance of the computation we implement the algorithms for data-parallel computing and data combining.

We show a time difference of computation performance both user defined function and high-level query languages; furthermore the choice of programming interface has a different effect on the performance of computation. In order to get good performance I need to organize the cluster server and implement the good execution plans that can fit well for such computation and platform.

This thesis evaluates the implementations for performing data combination and computation in several state of the art distributed computing systems: Hadoop(20), HBase(26), Pig(24), MapReduce (2), MySQL servers.

The solution was proposed on the paper Tomasz, Chunming and others (11) "Scaling-out the NCBO Resource Index Processing and Main-

tenance” and I describe implementation details of a number of computation strategies in Hadoop platform and HBase, and present a comprehensive experimental comparison of these techniques on an 11 node Hadoop cluster. The experimental results provide insights that are about the MapReduce platform and comparisons of particular join algorithms on the Hadoop platform.

Acknowledgements

I would like to show my acknowledgements to my supervisor professor Chunming Rong and Tomasz Wiktor Wlodarczyk for his supervision and support during my master's work, and also my gratitude to my co-supervisor Rui Paulo M. P. M. Esteves for this helpful suggestions and guidance. Also I would like to acknowledge Thejas M Nair who works in Yahoo and answering my question regarding the Pig Joins.

At last I should show my thanks to my family and my friends for their support in my living and study in the past two years. Additionally I would like to thank the University of Stavanger for the opportunity to use 11 nodes clusters for my thesis which is a valuable experience for me.

Byambajargal Byambajav

June 14, 2011, Stavanger .

Contents

List of Figures	v
List of Tables	vii
1 Introduction	1
1.1 Motivation	1
1.2 The Contributions of this thesis	3
1.3 What is the NCBO Resource Index	3
1.4 The Thesis outline	6
2 Background	7
2.1 Hadoop	7
2.2 HBase	12
2.3 MapReduce	14
2.4 Apache Pig	16
2.5 Singleton Pattern	16
3 Related work	19
4 Implementation	21
4.1 Hardware configuration	22
4.2 Data sets	22
4.3 Join Algorithms in Pig	23
4.3.1 Replicated Pig Join in HDFS	23

CONTENTS

4.3.2	Parallel Pig Join in HDFS	24
4.3.3	Replicated Pig Join in HBase	26
4.3.4	Parallel Pig Join in HBase	30
4.4	Join Algorithms in MapReduce	33
4.4.1	MapReduce Joins in HDFS use Singleton pattern	33
4.4.2	MapReduce Reduce side join in HDFS	35
4.4.3	MapReduce Join in HBase and HDFS	39
4.4.4	Build Multidimensional structure in HBase	45
5	Discussion on the results	51
6	Conclusions and Future Work	57
6.1	Conclusions	57
6.2	Future Work (11)	58
	References	61

List of Figures

1.1	The Resources Index WorkFlow diagram	5
2.1	Hadoop Work Flow Diagram	10
2.2	Hadoop Data-flow diagram	11
2.3	HBase cluster architecture	13
2.4	MapReduce Dataflow	15
4.1	HBase Regions architecture	28
4.2	Replicated Joins in Pig	30
4.3	Parallel Joins in Pig	32
4.4	MapReduce Joins with Singleton in HDFS	33
4.5	MapReduce Reduce side join in HDFS	36
4.6	MapReduce joins in HDFS	39
4.7	MapReduce join in HDFS and HBase	40
4.8	Hbase Cell structure	42
4.9	Join algorithms in Pig	44
4.10	Join algorithms in MapReduce	44
4.11	ERD (entity relationship diagram)	47
5.1	Joins in obr_wp_annotation	52
5.2	Joins in obr_wp_annotation	53
5.3	Joins in obr_wp_annotation	54
5.4	Joins in MySQL	54
5.5	The Execution time of Joins	55

LIST OF FIGURES

List of Tables

4.1	Data sets	23
4.2	Replicated Pig Join in HDFS.	23
4.3	Parallel Pig Join in HDFS.	26
4.4	Replicated Pig Join in HBase.	29
4.5	Customized configuration of Hbase	31
4.6	Parallel Pig Join in HBase.	32
4.7	MapReduce Joins in HDFS with Singleton pattern.	35
4.8	MapReduce, Reduce side join in HDFS.	39
4.9	Multidimensional Structure of Relation table	41
4.10	MapReduce Joins in HDFS and HBase	43
4.11	Multidimensional index structure in HBase	49

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

The National Center for Biomedical Ontology (NCBO) (13) maintains BioPortal (14), an open library of more than 200 ontologies in biomedicine they use the terms from these ontologies to annotate, or tag, automatically the textual descriptions of the data that resides in diverse public resources. The goal of NCBO is to enable a researcher to browse and analyze the information stored in these diverse resources and it also provide the user interface of BioPortal.

The BioPortal includes the NCBO Resource Index (13), which is a searchable database of semantic annotations for biomedical resources using all BioPortal ontologies. In The context, a biomedical resource is a repository of elements that may contain patient records, gene expression data, scholarly articles, and so on. A data element is unstructured text describing elements in the resource.

The Number of Researchers who are using ontologies extensively to annotate their data, to drive decision support of the systems is increasing constantly. So now the Resource Index currently includes 22 different data resources, comprising over 3.5 million data elements resulting in 16.4 billion annotations stored in a 1.5 terabyte MySQL database. We are ramping-up the system to include nearly 100 different data resources, 50 million data elements, and well over 100 billion annotations as analyzed earlier work on the paper Tomasz, Chunming and others

1. INTRODUCTION

(11) "Scaling-out the NCBO Resource Index Processing and Maintenance".

Running such amount of indexed data and computing from different data resource are reaching storage and processing limits of a single machine. Knowing these storage and performance limitations makes us critical decisions on which systems, platform, algorithm will work best for our needs, or when and how to build something new that can work faster.

Therefore, we approach the scalability problem for a knowledge base of annotations, like the Resource Index, first by examining existing, scalable systems. The goal is to incorporate a large variety of ontologies as well as a large amount of data.

In order to compute Resource index for more than 22 resources we need to scale out from single machine to distributed computing machines. There is possibility of using more powerful machine or parallel databases but it cannot be good solution for a longer term and license of Relational databases. Parallel databases have for some time permitted user-defined selection and aggregation operations that have the same computational expressiveness as MapReduce, although with a slightly different interface. Many data-mining computations have as a fundamental subroutine a Group By Join and Aggregate operation. This takes a dataset; partitions its records into groups according to some key, then performs join, group by, any other computation and an aggregation over each resulting group.

This thesis evaluates the implementations for performing data combination and computation in several state of the art distributed computing systems: Hadoop (20), HBase (26), Pig (24), MapReduce (3) and MySQL Servers. We plan to implement data storage both directly in HDFS and in HBase. We will evaluate relative performance differences and we will create a standard and a custom HBase index and its multidimensional structure which will be also investigated as a way to reduce storage requirements for semantic expansion.

Systems like MapReduce and Hadoop allow programmers to decompose an arbitrary computation into a sequence of maps and reductions, which are written in a full-edged high level programming language (C++ and Java, respectively) using arbitrary complex types.

The resulting systems can perform quite general tasks at scale, but offer a low-level programming interface: even common operations such as database Join

require a sophisticated understanding of manual optimizations on the part of the programmer. Consequently, layers such as Pig Latin and HIVE have been developed on top of Hadoop, offering a SQL-like programming interface that simplifies common data-processing tasks.

1.2 The Contributions of this thesis

In NCBO Resource index(13) there are three main processing steps required to perform the semantic expansion. The first step is concept recognition, which can be classified as embarrassingly parallel and can easily be divided among processing nodes. The second step is term expansion based on semantics, where the most computationally expensive process relies mostly on join operations. During that stage, information on ontological distance between concepts is also utilized. For implementing this step we implement different join techniques and compare a programming models for Resource Indexing in Hadoop, HBase, and MySQL, and show the impact of interface-design choices on optimizations. We describe and implement a general, rigorous treatment of distributed computation in the MapReduce, Pig and HBase system. We use Hadoop to evaluate several optimization techniques for distributed programming in real applications running on a small-sized cluster of several computers.

Finally, we implement an inverted index which is applied on the entire set of associated terms for efficient search.

1.3 What is the NCBO Resource Index

The range of publicly available biomedical data is enormous and is expanding fast. This expansion means that researchers now face a hurdle to extracting the data they need from the large numbers of data that are available. Biomedical researchers have turned to ontologies and terminologies to structure and annotate their data with ontology concepts for better search and retrieval. Using the annotation work flow of the Annotator Web Service NCBO(28) has built a biomedical

1. INTRODUCTION

resources index in which biomedical data is indexed by ontology concepts. The index allows a user to search for biomedical data based on ontology concepts.

The NCBO Resource Index directly run queries in the BioPortal ontology repository: when a user browses a given concept, he has access (link) to the list of resource elements that have been annotated with this concept. A user can also search for resources directly using the 'All resources' tab. The annotations in the index keeps track of the structures of elements that have been annotated i.e., from which part of the element (e.g., title, description) an annotation has been produced. This information is used to score annotations.

The Resources Index system architecture consisting of different levels (See Figure 1.1):

- **Resource level:** Public biomedical resources (such as GEO and PubMed (15)) are composed of elements that represent an abstraction for the unit of storage in those databases.
- **Annotation level:** The System uses a concept recognition tool called mgrep (developed by Univ. of Michigan) to annotate (or tag) resource elements with terms from a dictionary.
- **Index level:** A global index combines all the annotation tables and indexes annotations according to ontology concepts. The index contains information such as: Concept T annotates elements E1, E2,
- **Ontology level:** The system also uses relations provided to expand the annotations. This is the first step of the semantic expansion. Using the `is_a` ontology relation, for each annotation, we create additional transitive closure annotations according to the parentchild relationships subsumed by the original concept.

1.3 What is the NCBO Resource Index

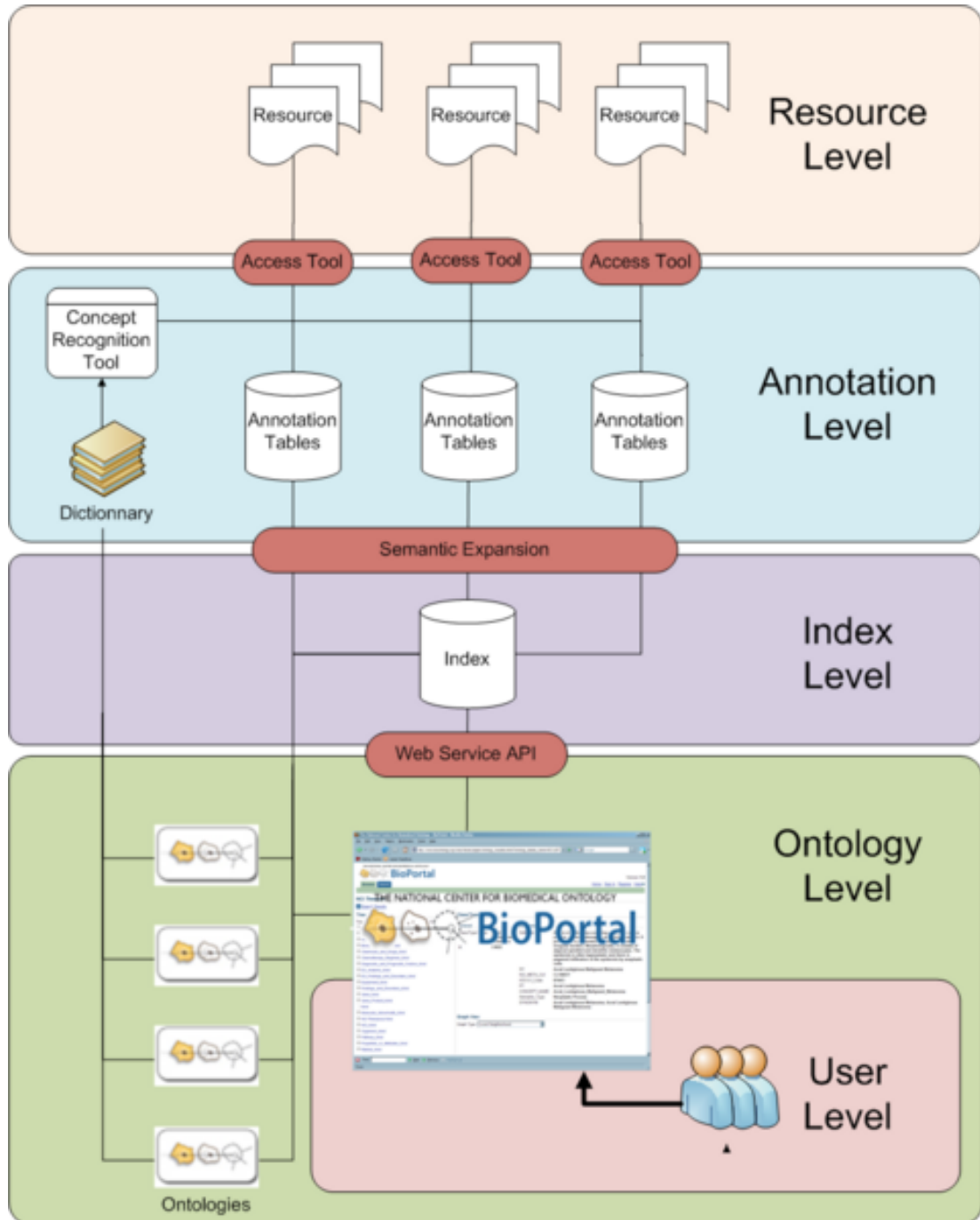


Figure 1.1: The Resources Index WorkFlow diagram - WorkFlow diagram (13)

1.4 The Thesis outline

The following chapters constitute the thesis:

- **Chapter 1** introduces the resent NCBO system and data structure. The scope and contributions of this thesis are summarized.
- **Chapter 2** gives the background of Hadoop based systems such as MapReduce, HBase and Pig. Some NCBO data samples are shown in this chapter.
- **Chapter 3** gives about related work.
- **Chapter 4** shows different methods to compute and combine data. The advantages and disadvantages of these methods are mentioned. The details about our method are discussed and it also shows the implementation of the algorithms and different methods. The results of our methods are presented.
- **Chapter 5** discusses the Hadoop implementation of the Join, Pig joins and Results of our methods are presented.
- **Chapter 6** summarizes the major contributions and conclusions of this work, and suggests the problems for further research.

Chapter 2

Background

2.1 Hadoop

Hadoop (20) is the Apache Software Foundation top-level project that holds the various Hadoop sub projects that graduated from the Apache Incubator. The Hadoop framework is open source software that supplies a framework for the development of highly scalable distributed computing and handles the processing details, leaving developers free to focus on application logic. It includes several sub-projects such as: Hadoop Cores is core sub-project and it provides a distributed file system (HDFS)(22) and support for the MapReduce distributed computing. The Hadoop Distributed File System (HDFS) and MapReduce environment provides the user to manage the execution of map and reduce tasks across a cluster of machines. The user is required to specify the following parameters to run a job

- The location(s) in the distributed file system of the job input
- The location(s) in the distributed file system for the job output
- The input format
- The output format
- The class containing the map function

2. BACKGROUND

- Optionally. the class containing the reduce function
- The JAR file(s) containing the map and reduce functions and any support classes

The Hadoop will partition the input into the small chunks, and schedule and execute map tasks across the cluster. If requested, it will sort the results of the map task and execute the reduce task(s) with the map output. The final output will be moved to the output directory, and the job status will be reported to the user.

This framework provides two basic processes that can handle the management of MapReduce jobs;

- Task Tracker executes an individual map and reduces tasks on a compute node in the cluster.
- Job Tracker accepts job submissions, provides job monitoring and control, and manages the distribution of tasks to the Task Tracker nodes.

Generally a cluster have a Master node and several Slave nodes. The JobTrackers and TaskTrackers work on master and slave nodes respectively to handle jobs and tasks. When a MapReduce job is submitted to the master, a Job Trackers divides it into tasks and assigns a task to each TaskTrackers. Following is the sequence of steps for MapReduce work flow on Hadoop (see figure 2.1) :

1. Mapping Phase: The Mapper performs the interesting user-defined work of the first phase of the MapReduce program. Given a key and a value, the `map()` method emits (key, value) pair(s) which are forwarded to the Reducers. Each mapper works on input splits assigned to it by the NameNode. An input split consists of a number of records. The records can be in different formats depending on the InputFormat of the input file. A RecordReader for that particular InputFormat reads each and every record, determines the key and value for the records, and supplies the key-value pairs to map functions where the actual processing takes place (Figure 3). Each mapper applies a user defined function on the key-value pairs and converts them to intermediate key-value pairs. The intermediate results of mappers are written to the local file-system in a sorted order.

- 2. Partitioning Phase:** A partitioner determines which reducer an intermediate key-value pair should be directed to. The default partitioner provided by Hadoop computes a hash value for the key and assigns the partition on the basis of the function: $(\text{hash_value_of_key}) \bmod (\text{total_number_of_partitions})$.
- 3. Shuffling Phase:** Each map process, in its heartbeat message, sends information to the master about the location of the partitioned data. The master informs each reducer about the location of the mapper from which it has to pick its partition. This process of moving data to appropriate reducer-nodes is called shuffling.
- 4. Sorting Phase:** Each reducer, on receiving its partitions from all mappers, performs the sort-merge join to sort the tuples on the basis of the keys. Since keys within partitions were already sorted in each mapper, the partitions have to be merged only such that the similar keys are grouped together.
- 5. Reduce Phase:** A user-defined reduces operation is applied on each group of keys and the result is written to HDFS.

Hadoop Distributed File System (HDFS) is a distributed, scalable, and portable file system written in Java for Hadoop and it designed for use for MapReduce jobs that read input in large chunks of input, process it, and write potentially large chunks of output. Furthermore, data in HDFS are simply mirrored to multiple storage nodes which has a datanodes process. Each node in a Hadoop typically has a single datanode instance.

HDFS services are provided by two processes:

- NameNode handles management of the file system metadata, and provides management and control services.
- DataNode provides block storage and retrieval services

To get an idea of how data flows between the clients interacting with HDFS, the namenode and the datanodes, consider figure 2.2.

Step 1. The client opens the file it wishes to on the file system, which for HDFS is an instance of Distributed File System.

2. BACKGROUND

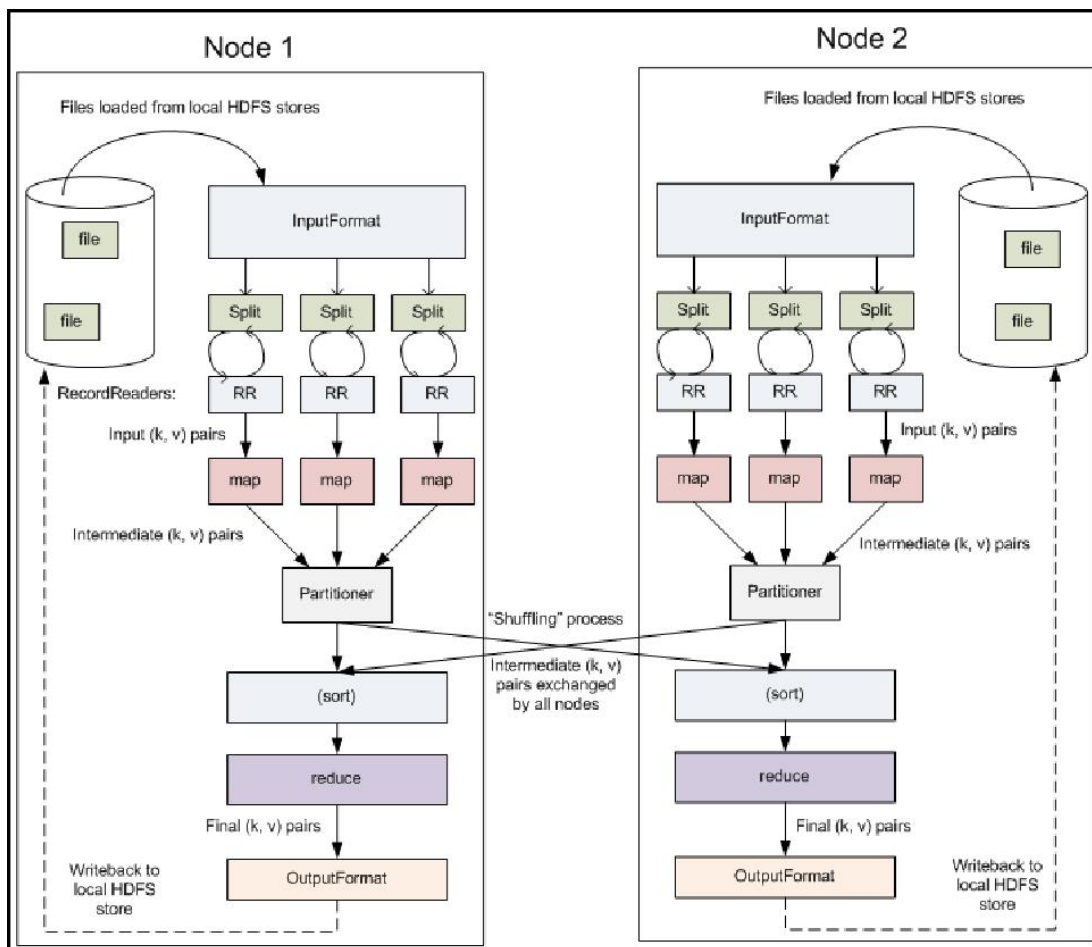


Figure 2.1: Hadoop Work Flow Diagram - Work Flow Diagram (3)

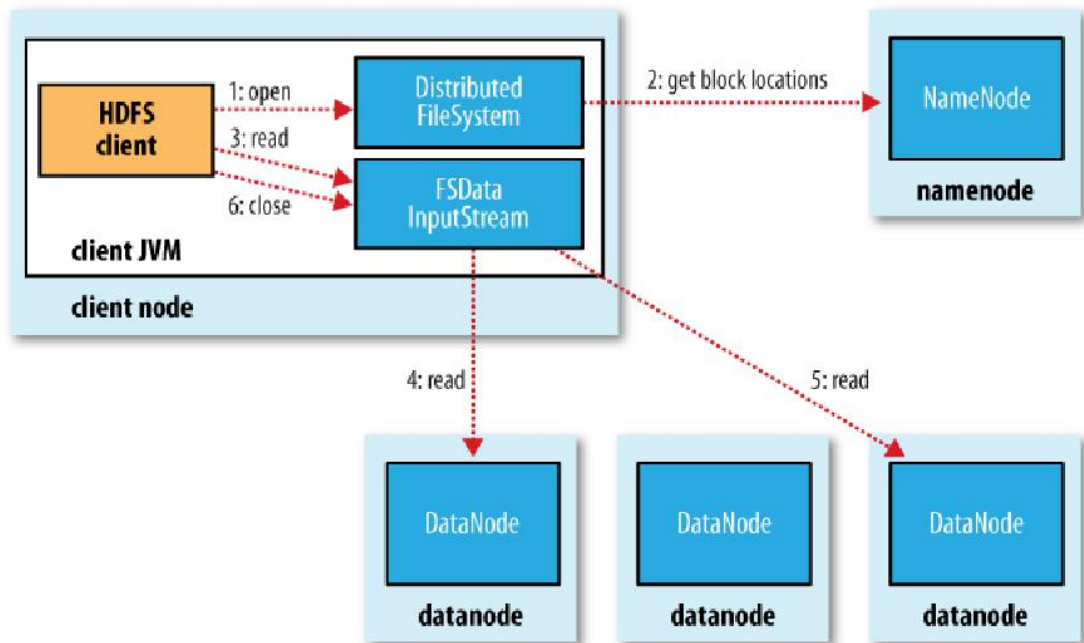


Figure 2.2: Hadoop Data-flow diagram - Data-flow diagram (19)

- Step 2.** Distributed File System calls the namenode, using RPC, to determine the locations of the blocks for the first few blocks in the file.
- Step 3.** The Distributed File System returns an input stream that supports file seeks to the client for it to read data from Input Stream.
- Step 4.** DFS Input Stream, which has stored the datanode addresses for the first few blocks in the file, then connects to the first datanode for the first block in the file. Data is streamed from the datanode back to the client, which calls `read()` repeatedly on the stream.
- Step 5.** When the end of the block is reached, DFS Input Stream will close the connection to the datanode, then find the best datanode for the next block.
- Step 6.** When the client has finished reading, it calls `close()` on the FS Data InputStream

2. BACKGROUND

2.2 HBase

The HBase(26, 27) is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets and it provides a scalable, distributed database. Data modeling is important on the HBase that store data into labeled tables those are made of row and columns. Row columns are grouped into column families. The HBase called a column-oriented storage. see Figure 2.3. **Tables:** HBase tables are like those in an RDBMS, only cells are versioned, rows are sorted, and columns can be added on the fly by the client as long as the column family they belong to preexists. Tables are automatically partitioned horizontally by HBase into **regions**. Rows: Row keys are uninterpreted bytes. Rows are lexicographically sorted with the lowest order appearing first in a table. The empty byte array is used to denote both the start and end of a tables' namespace.

Column Family: Columns in HBase are grouped into column families. All Column family members have a common prefix, so for example in my implementation the columns data: concept_id and data: dictionary_id are both members of the data column family. The colon character (:) delimits the column family from the column qualifier. The qualifying tail, the column family qualifier, can be made of any arbitrary bytes. Column families must be declared up front at schema definition time whereas columns do not need to be defined at schema time but can be conjured on the fly while the table is up a running.

Physically, all column family members are stored together on the file system. Because tunings and storage specifications are done at the column family level, it is advised that all column family members have the same general access pattern and size characteristics.

Cells: A row, column, version tuple exactly specifies a cell in HBase. Cell content is uninterpreted bytes.

Regions: The Tables are automatically partitioned horizontally by HBase into regions and each of them comprises a subset of a tables rows. HBase characterized with an HBase master node orchestrating a cluster of one or more regionserver slaves. The HBase Master is responsible for boot strapping a virgin

install, for assigning regions to registered regionservers, and for recovering region-server failure. HBase keeps special catalog tables named `-ROOT-` and `.META`, which maintains the current list, state, recent history, and location of all regions afloat on the cluster. The `-ROOT-` table holds the list of `.META` table regions. The `.META` table holds the list of all user-space regions. Entries in these tables are keyed using the regions start row. see Figure2.3

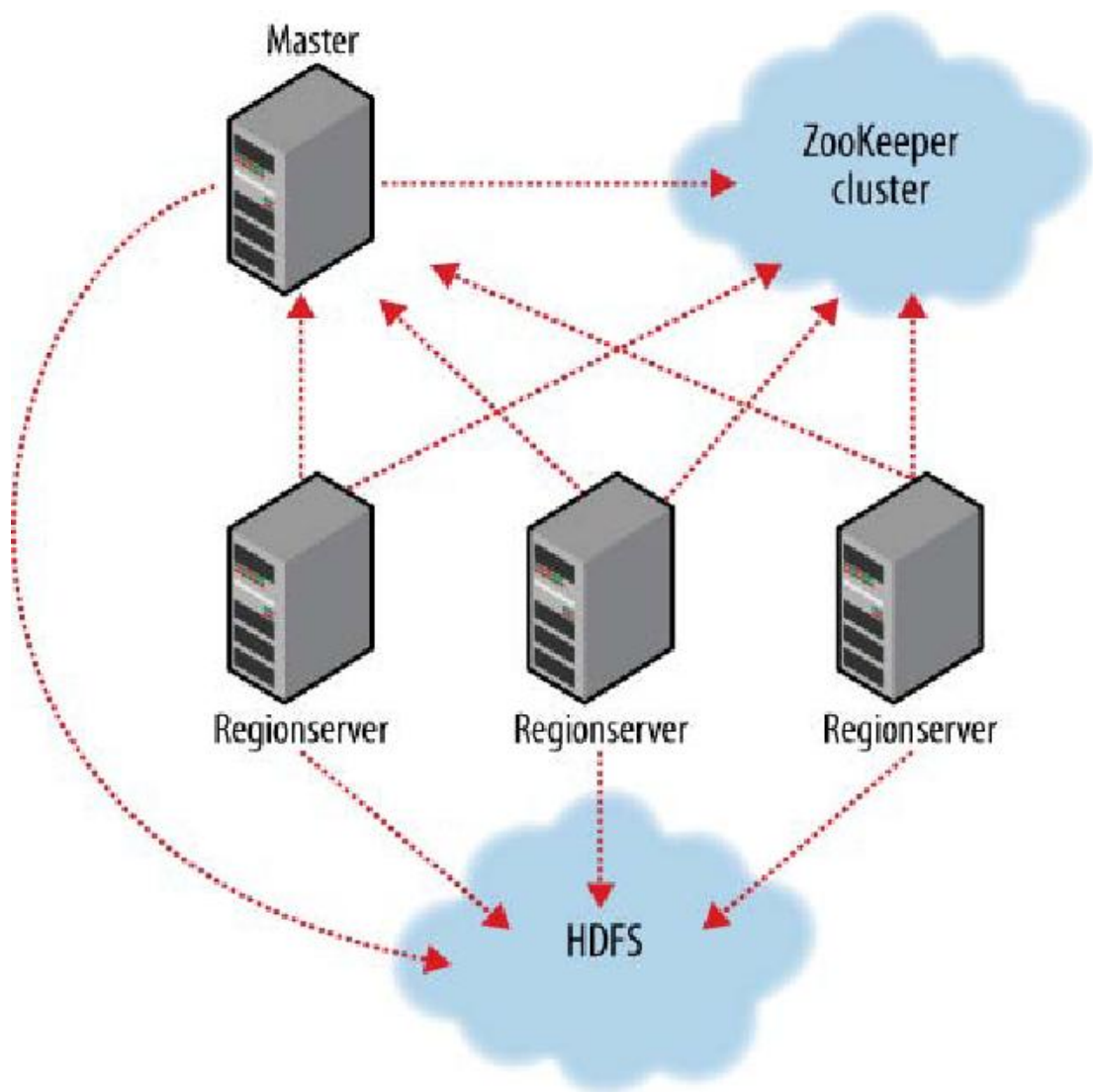


Figure 2.3: HBase cluster architecture - Cluster architecture (19)

2. BACKGROUND

2.3 MapReduce

The advantage of parallel programming in Hadoop is a MapReduce Job that runs it on a cluster of machines. MapReduce allows you the programmer to specify a map function followed by a reduce function, but working out how to fit your data processing into this pattern, which often requires multiple MapReduce stages.

Hadoop provides our query as a MapReduce Job that works by dividing the processing into two phases; each phase has key-value pairs as input and output.

- Map phase: The master node takes the input and chops it up into pieces and distributes those to worker nodes:
 - In the inputs extract keys and record
 - Partition in these outputs by keys of the records
- Reduce phase: The master node then takes the answers to all the pieces and combines them in a way to get the output.
 - Collect and merge all the records with same k

Detailed: The map function merely extracts the keys and values and emits them as its output:

(k1, 0)
(k1, 22)
(k2, 11)
(k4, 111)
(k4, 78)

The output from the map function is processed by the MapReduce framework before being sent to the reduce function. This processing sorts and groups the key-value pairs by key. So, continuing the example, our reduce function sees the following input:

(k1, [0, 22])
(k2, [-11])
(k4, [111, 78])

2.3 MapReduce

All the reduce function has to do now iterate through the list and aggregate a max function:

(k1, 22)
 (k2, 11)
 (k4, 111)

This is the final output: the maximum values recorded in each key. See Figure 2.4.

The HBase support MapReduce job and classes and utilities in the org.apache.hadoop.HBase package facilitate using HBase as a source and/or sink in MapReduce jobs. The **TableInputFormat** class makes splits on region boundaries so maps are handed a single region to work on. The **TableOutputFormat** will write the result of reduce into HBase.

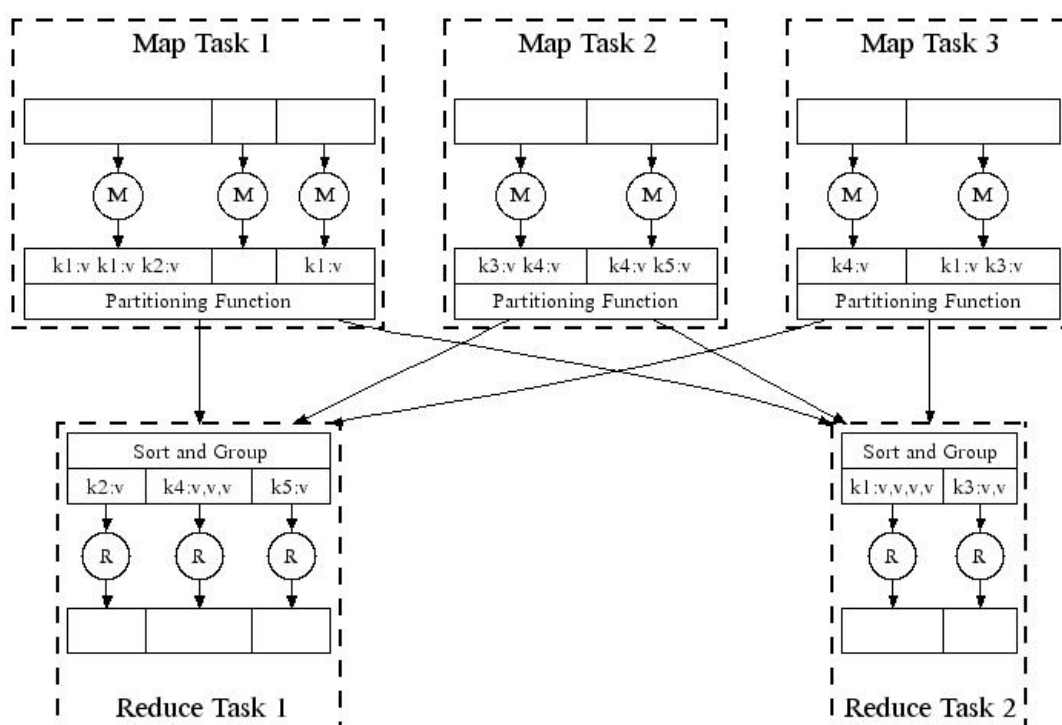


Figure 2.4: MapReduce Dataflow - MapReduce (22)

2. BACKGROUND

2.4 Apache Pig

Apache Pig(24) is a platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs. Pig raises the level of abstraction for processing large datasets. With Pig, the data structures are much richer, typically being multivalued and nested; and the set of transformations you can apply to the data are much more powerful - they include joins, for example, which are not for the faint of heart in MapReduce. Pig is made up of two pieces:

- The language used to express data flows, called Pig Latin.
- The execution environment to run Pig Latin programs.

A Pig Latin program is a high level distributed programming language that consists of a series of operations, or transformations, that are applied to the input data to produce output.

Taken as a whole, the operations describe a data flow, which the Pig execution environment translates into an executable representation and then runs. Under the covers, Pig turns the transformations into a series of MapReduce jobs, but as a programmer you are mostly unaware of this, which allows you to focus on the data rather than the nature of the execution.

Pigs sweet spot is its ability to process terabytes of data simply by issuing half-dozen lines of Pig Latin from the console. The Pig is a Hadoop extension that simplifies Hadoop programming by giving you a high-level data processing language while keeping Hadoops simple scalability and reliability. Yahoo , one of the heaviest user of Hadoop (and a backer of both the Hadoop Core and Pig), runs 40 percent of all its Hadoop jobs with Pig. Twitter is also another well-known user of Pig.

2.5 Singleton Pattern

In software engineering, the singleton pattern is a design pattern used to implement the mathematical concept of a singleton, by restricting the instantiation of a

class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects, but common mistakes can inadvertently allow more than one instance to be created.

The Singleton's purpose is to control object creation, limiting the number to one but allowing the flexibility to create more objects if the situation changes. Since there is only one Singleton instance, any instance fields of a Singleton will occur only once per class, just like static fields.

Singletons often control access to resources such as database connections or sockets. For example, if you have a license for only one connection for your database or your JDBC driver has trouble with multithreading, the Singleton makes sure that only one connection is made or that only one thread can access the connection at a time. If you add database connections or use a JDBC driver that allows multithreading, the Singleton can be easily adjusted to allow more connections. Moreover, Singletons can be stateful; in this case, their role is to serve as a unique repository of state. If you are implementing a counter that needs to give out sequential and unique numbers, the counter needs to be globally unique. The Singleton can hold the number and synchronize access; if later you want to hold counters in a database for persistence, you can change the private implementation of the Singleton without changing the interface. On the other hand, Singletons can also be stateless, providing utility functions that need no more information than their parameters. In that case, there is no need to instantiate multiple objects that have no reason for their existence, and so a Singleton is appropriate.

The Singleton should not be seen as way to implement global variables in the Java programming language; rather, along the lines of the factory design patterns, the Singleton lets you encapsulate and control the creation process by making sure that certain prerequisites are fulfilled or by creating the object lazily on demand.

2. BACKGROUND

Chapter 3

Related work

Since we choose Hadoop based platforms such as HBase, HDFS the data of materialization of semantic expansion is expanded on disk and queries are directly performed on the data. On the NOSQL query a join operation is one of the fundamental, most difficult operations, and hence the most researched query operation. So this is an important operation that makes easy for combining data from two sources on the basis of some common key.

There are some papers related to the distributed computing and join algorithms in parallel and distributed computing such as "A Comparison of Join Algorithms for Log Processing in MapReduce" (1),(2),(4),(5),"MapReduce: Simplified Data Processing on Large Clusters" (6),(7),(8). Also there is a rich history of studying join algorithms in parallel and distributed RDBMSs. The database literature is full of discussions on techniques, performance, and optimization of this operation. Nested loops, sort/merge, and hash join are the most commonly used join techniques. A more recent work (4) proposes extending the current MapReduce interface with a merge function. While such an extension makes it easier to express a join operation, it also complicates the logic for fault-tolerance.

On the paper Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations (8) the approach they have defined six strategies both accumulator- and iterator-based implementations of distributed reduction using the DryadLINQ system to take advantage of a good data reduction, pipelining,

3. RELATED WORK

low memory consumption and multi core cluster computer. The full Sort implementation accumulates all the objects in memory and performs in parallel and it attains an optimal data reduction for each partitions. This strategy adopted by MapReduce and Hadoop. Iterator PartialSort approach close to MapReduce. Idea is to keep only a bounded number of chunks of input records into memory and is processed independently parallel. The bound on memory makes pipelining possible. They had two implementation of MapReduce those one applies the Map faction, sorts the resulting records, while another performs partial aggregation on the sorted records but they have almost same performs.

On the paper A Comparison of Join Algorithms for Log Processing in MapReduce (1) They have been explored a few join methods in the declarative frameworks on MapReduce, like Pig (24), Hive (23) and Jaql. Then they compare their join algorithms to those in Pig. The contribution of this paper is to investigate various parallel/distributed join techniques on the MapReduce platform, thus the insights from our work can be directly used by declarative frameworks like Pig. They had two join strategies such as repartition join and fragment replicate join in pig and the results consistently show a more 2.5X speed up with their implementation than in Pig implementation.

In my work, we choose to modify the existing MapReduce programming interface for join implementation. Nextly we tried to evaluate the preforms of all those approaches. Furthermore, there have been several efforts in designing high level query languages on MapReduce. This includes Pig (24), (23), and HBase client API, all of which are open source. They differ in the underlying data model and query syntax. Moreover this work will provide an experimental result on the performance of these strategies of join, in Pig and in HBase.

Chapter 4

Implementation

There are three main processing steps required to perform the semantic expansion. The first step is concept recognition, which can be classified as embarrassingly parallel it can easily be divided among processing nodes. The second step is term expansion based on semantics, where the most computationally expensive process relies mostly on join operations. During that stage, information on ontological distance between concepts is also utilized. For every recognized term, approximately 14 additional terms are also associated. Finally, an inverted index is applied on the entire set of associated terms for efficient search.

In order to create the Resource index we implement a number of different join algorithms those are provided by MapReduce and Pig in different platforms such as HDFS(22) and HBase(26): map-side, reduce-side, parallel and Replicated join algorithms. In this section, we discuss those algorithms, present our own algorithm, and provide some important implementation-specific details. Finally we will present efficient multidimensional data structure for search index in HBase. We conducted each experiment three times and present the mean of those values here.

4. IMPLEMENTATION

4.1 Hardware configuration

We have a Hadoop cluster of eleven nodes. Out of these eleven nodes, ten nodes are the datanodes and tasktrackers as slave; a node is the secondarynamenode, namenode responsible for managing the distributed file system and assigning the map and reduce tasks to other worker nodes as a master.

Each node is a HP with sex AMD Phenom(tm) II X6 1090T Processor and a 16GB ECC DDR-2 memory chip. The secondary storage of each node is 80GB SATA drive running at 7200rpm. The nodes are connected to an HP ProCurve 2650 at the network bandwidth of 100 BaseTx-FD. The cluster contains two racks with three datanodes in each rack. The racks are connected by a 1Gbps Realtek Semiconductor Co., Ltd. RTL8111/8168B PCI Express Gigabit Ethernet link. On each node are installed Linux version 2.6.18-194.32.1.el5.centos.plus (gcc version 4.1.2 20080704 (Red Hat 4.1.2-48)), Hadoop 0.20.1, pig 0.8.1 and Java 1.6.0-23 for 64 bit. The block size is the default 64MB. The size of the heap memory is increased to 2048MB. The cluster hase:

- 11 Nodes
- 66 Cores
- 176 GB memory
- CentOS 4.12
- 1GB Ethernet link

4.2 Data sets

Currently the NCBO Resource Index(12) currently includes 22 different data resources comprising over 3.5 million data elements resulting in 16.4 billion annotations stored in a 1.5 terabyte MySQL database.

The semantic expansion consumes considerable amount of resources in terms of storage and processing power. In this implementation we got some real data sets from NCBO Resource Index(13) those consist of three different files from annotation table and a file from relation table.

4.3 Join Algorithms in Pig

File name	Size of file	Number of Rows
obr_wp_annotation	1786MB	54039
obr_ct_annotation	5916MB	164808416
obr_pm_annotation	16983MB	42049697
obs_relation	659MB	24153638

Table 4.1: Data sets - Size and Rows

4.3 Join Algorithms in Pig

4.3.1 Replicated Pig Join in HDFS

In certain cases, the performance of inner joins and outer joins can be optimized using replicated. We run the pig in MapReduce mode with 11 nodes cluster. Pig allocates a fix amount of memory to store bags and spills to disk as soon as the memory limit is reached. This is very similar with how Hadoop decides when to spill data accumulated by the combiner. The amount of memory allocated to bags is determined by `pig.cachedbag.memusage` the default is set to 10% of available memory. Note that this memory is shared across all large bags used by the application (25). So In our case the relation file is not bigger than our machine's memory but default memory for Pig is not enough for running the replicated join so we have increased the memory up to 8GB and we use some optimization techniques for speeding up the Pig queries(25). The following table 4.2 shows the result of the experiment.

File name	Size/ # of records	Mapper	Reducer	Time(sec)	Relation file
wp_annotation	1786MB/54039	12	0	29	659MB/24153638
ct_annotation	5916MB/164808416	104	0	799	659MB/24153638
pm_annotation	16983MB/42049697	277	0	1794	659MB/24153638

Table 4.2: Replicated Pig Join. - The Execution time

4. IMPLEMENTATION

The replicate join is a special type of join that load the small file (`obs_relation.txt`) into the main memory then the join is done by each map side in the memory. If one or more relations are small enough to compare with another file it can work efficiently. In such cases, Pig can perform a very efficient join because all of the hadoop work is done on the map side and the time for loading file into the memory is negligible for small file. In our case it does not work very well because of our relation file is not small enough. The reason why replicated join performs worse than default join is because of the large number of maps and the large size of the replicated file. Each map task ends up reading and (de)serializing the replicated file (`obs_relation.txt`), and usually that takes bulk of the runtime. In my case $(691\text{MB} \times 277 \text{ (maps)} = \sim) 187\text{GB}$ of replicated input data will be read and (de)serialized by all the map tasks.

Here is the theoretical rule of thumb for replicated join: for replicated join to perform significantly better than default join, the size of the replicated input should be smaller than the block size (or `pig.maxCombinedSplitSize` if property `pig.splitCombination=true` and larger than block size).

This is because for the number of map tasks started are equal to the number of blocks (or `size/pig.maxCombinedSplitSize`) in the left side input of replicated join. Each of these blocks will read the replicated input. If the replicated input read size is few times larger than block size, using replicated join will not save on IO/(de)serialization costs. If I increase the block size of my cluster the time for transferring the block through the network will be increased.

4.3.2 Parallel Pig Join in HDFS

The Parallel join operator is used if all the relation files are too large to fit in memory. In our case the relation file is not small and we have not got a good result from previous experiment so the regular hash join can fit in this case.

The Pig uses a hash join algorithm for default join. Because of their nature, hash join algorithms can easily be parallelized. The difference is between the single processor and multi-processor/parallel variants of these algorithms and the partitions are processed in parallel by multiple processors in the parallel variant. The following section describes how the hash join works on the files

obr_ct_annotation.txt and obs_relation from NCBO Resource index on the cluster machines.

The input obr_ct_annotation.txt file (we call ANNO file) is horizontally divided into 104 partitions (mappers) such that each partition carries approximately $\frac{\text{ANNO}}{104}$ tuples (it is around 64MB that is default block size of Hadoop). Then a hash function F1 is applied on the concept_id as a distribution key. The range of this hash function is from 0 to 103 so that keys can be directed to one of the ten nodes. The 104 partitions of annotation file formed as a result of the hash distribution are written to the disk.

On another side same process is applied for input obs_relation.txt file (we call REL file). It is divided into n partitions, each partition carrying about $\frac{\text{REL}}{n}$ tuples, by applying the same hash function F1. This ensures that a partition x of the REL contains the same join keys as partition x of the REL. The partitions of REL are also written to the disk. Each processor reads in parallel a partition of relation file from the disk. It creates an in-memory hash table for the partition using a hash function F2. A corresponding partition of ANNO is also read in parallel from the disk by each processor. For each tuple in this relation, it probes the in-memory hash table for any match. For each matching tuple, a joined record is outputted to the disk.

Since all the n partitions of a relation are completely independent of each other, parallel processing can be carried out. Each processor handles the corresponding partitions from both the relations and writes the joined records for the matching tuples. Take an advantage of Join we use some optimization techniques: **Use Types:** If types are not specified in the load statement, Pig assumes the type of =double= for numeric computations. Specifying the real type will help with speed of arithmetic computation and it also has an additional advantage of early error detection.

Use the Parallel Features: We can set the number of reduce tasks for the MapReduce jobs generated by Pig using parallel features. (The parallel features only affect the number of reduce tasks. Map parallelism is determined by the input file, one map for each HDFS block.)

In our case I use PARALLEL clause and sets the number of reducers using a heuristic based on the size of input data. When I calculate the number of reduce

4. IMPLEMENTATION

I use following calculation that set the values for these properties:

- `pig.exec.reducers.bytes.per.reducer` - Defines the number of input bytes per reduce; default value is $1024*1024*1024$ (1GB).
- `pig.exec.reducers.max` - Defines the upper bound on the number of reducers; default is 999.

The formula, shown below, helps us to improve the performs. The computed value takes all inputs within the script into account and applies the computed value to all the jobs within Pig script.

- Number of reducers = $\text{MIN}(\text{pig.exec.reducers.max}, \text{total input size (in bytes)} / \text{bytes per reducer})$.
- Number of reducers = $\text{MIN}(999, 6203559140+690981765 / 1073741824) \approx 7$ reducer (around)

But in my case I have already 10 nodes so it was better result on 10 reducer. In the following tables 4.3 shows the results of the Parallel join implementation in Pig.

File name	Size/ # of records	Mapper	Reducer	Time(sec)	Relation file
wp_annotation	1786MB/54039	12	10	32	659MB/24153638
ct_annotation	5916MB/164808416	104	10	350	659MB/24153638
pm_annotation	16983MB/42049697	277	10	454	659MB/24153638

Table 4.3: Parallel Pig Join in HDFS. - The Execution time of join

4.3.3 Replicated Pig Join in HBase

The Algorithm of replicated join in this part is exactly same as previous part only difference is we use HBase data storage instead of HDFS. In the previous part I had mentioned that what is HBase so now I will explain why we needed HBase instead of HDFS and how it works.

4.3 Join Algorithms in Pig

In the HBase data is organized into tables, rows and columns, but a query language like SQL is not supported. Instead, an Iterator-like interface is available for scanning through a row range (and of course there is an ability to retrieve a column value for a specific key).

Any particular column may have multiple values for the same row key. A secondary key can be provided to select a particular value or an Iterator can be set up to scan through the key-value pairs for that column given a specific row key. Furthermore the main reason is the tables are mutable that I can update and edit and delete as single row or multiple rows. The HDFS works great with immutable data such as log file or some text file.

I needed to organize data into tables, rows and columns, but on this experiment I just kept the structure of the NCBO Resource index data such as obr_wp_annotaion, obr_ct_annotaion, obr_pm_annotaion and obs_relation as a tables. Both of those tables use row key that is concept_id as a distribution key that you can see the following structure in the HBase distributed data base system.

- 11885226 column=cf:concept_id, timestamp=1304003760030, value=11885226
- 11885226 column=cf:context_id, timestamp=1304003760030, value=4
- 11885226 column=cf:dictionary_id, timestamp=1304003760030, value=5
- 11885226 column=cf:element_id, timestamp=1304003760030, value=1
- 11885226 column=cf:porsition_from, timestamp=1304003760030, value=1
- 11885226 column=cf:position_to, timestamp=1304003760030, value=17
- 11885226 column=cf:term_id, timestamp=1304003760030, value=26871846
- 11885226 column=cf:workflow_status, timestamp=1304003760030, value=8

HBase automatically partitions the Tables horizontally into regions. Each region comprises a subset of a tables rows. A region is denoted by the table it belongs to, its first row, inclusive, and last row, exclusive. Initially, a table comprises a single region, but as the size of the region grows, after it crosses a

4. IMPLEMENTATION

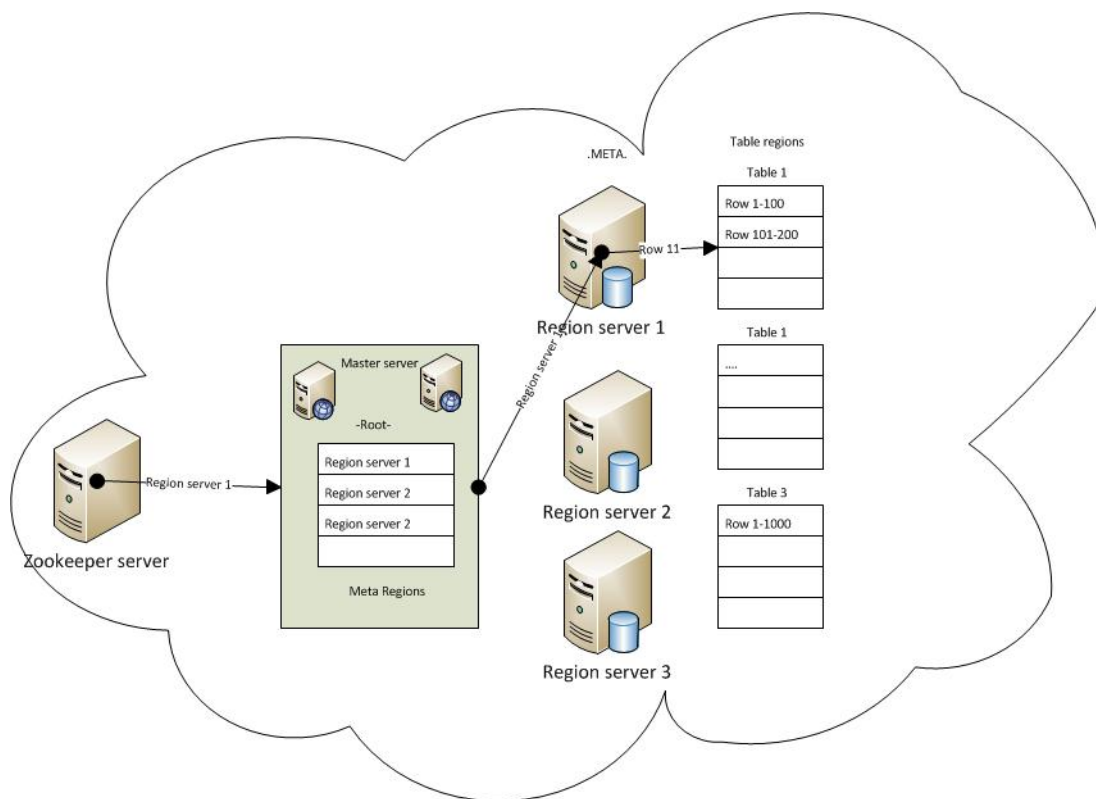


Figure 4.1: HBase Regions architecture - Regions architecture

4.3 Join Algorithms in Pig

configurable size threshold, it splits at a row boundary into two new regions of approximately equal size. see Figure(4.1).

In my case HBase the files obr_wp_annotation, obr_wp_annotation, obr_wp_annotation splits 42, 571 and 1518 regions respectively. Regions are the units that get distributed over an HBase cluster. In this way, a table that is too big for any one server can be carried by a cluster of servers with each node hosting a subset of the tables total regions. This is also the means by which the loading on a table gets distributed. In the following table4.4 you can see the result of the implementation.

File name	Size/ # of records	Mapper	Reducer	Time(sec)	Relation file
wp_annotation	1786MB/54039	43	0	50	659MB/24153638
ct_annotation	5916MB/164808416	613	0	532	659MB/24153638
pm_annotation	16983MB/42049697	1560	0	707	659MB/24153638

Table 4.4: Parallel Pig Join. - The Execution time

We depicted a diagram that shows the execution time for two Replicated join algorithms such as Replicated Join in HDFS and Replicated Join in HBase. We conducted each experiment three times and present the mean of those values here. See Figure 4.2.

4. IMPLEMENTATION

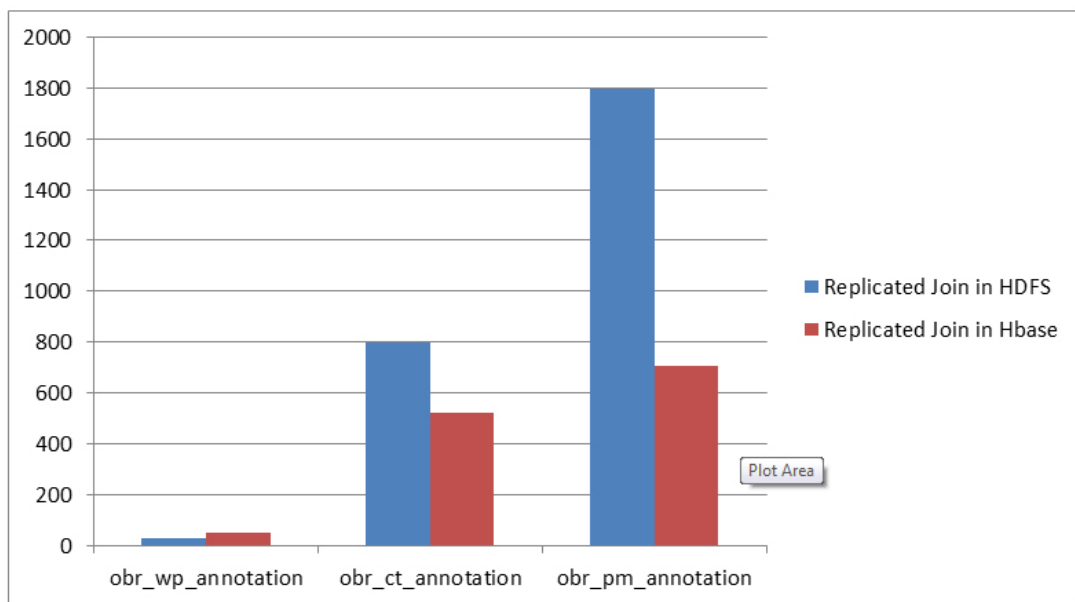


Figure 4.2: Replicated Joins in Pig - Execution time of joins

4.3.4 Parallel Pig Join in HBase

This implementation is almost same as **Parallel** Join in HDFS in previous part. The only difference is that we use HBase distributed database instead of HDFS. The main reason is the tables are mutable that I can update and edit and delete as single row or multiple rows. The HDFS works great with immutable data such as log file or some text file. In order to increase speed of the join in the HBase I need to configure some important parameters in the configuration file of HBase. HBase0.20.0 configuration important parameters of hbasesite.xml.

See Table 4.5

The value of those parameters that I had configured depends on the job and how many mappers can be handled, memory consumption of Hadoop, how many zookeeper server that I have and how many region servers for the tables. The parameter `hbase.client.scanner.caching` is number of rows that will be fetched when calling next on a scanner if it is not served from (local, client) memory.

4.3 Join Algorithms in Pig

Parameter	value	Comments
hbase.cluster.distributed	true	Fullydistributed with unmanaged ZooKeeper Quorum
hbase.regionserver.handler.count	200	Count of RPC Listener instances spun up on RegionServers. Same property is used by the Master for count of master handlers. Default is 10.
hbase.zookeeper.property.maxClientCnxns	1000	Property from ZooKeeper's config zoo.cfg. Limit on number of concurrent connections (at the socket level) that a single client, identified by IP address, may make to a single member of the ZooKeeper ensemble. Set high to avoid zk connection issues running standalone and pseudo-distributed. Default: 30
hbase.hregion.max.filesize	1073741824	Maximum HStoreFile size. If any one of a column families' HStoreFiles has grown to exceed this value, the hosting HRegion is split in two. Default: 256M. Default: 268435456
hfile.block.cache.size	0.4	Percentage of maximum heap (-Xmx setting) to allocate to block cache used by HFile/StoreFile. Default of 0.2 means allocate 20Default: 0.2
hbase.client.scanner.caching	100000	Number of rows that will be fetched when calling next on a scanner if it is not served from (local, client) memory.

Table 4.5: Customized configuration of Hbase - HBase0.20.0 hbase site.xml

4. IMPLEMENTATION

As long as HBase depends on memory and in the pig join we need to fetch data as much as we can, so that value is as maximum as our cluster can handle. The parameter `hbase.zookeeper.property.maxClientCnxns` is limit on number of concurrent. In our case we have 613 mappers for `obr_ct_annotation` file so we have to handle more than 100 connections one time. The following the tables 4.6 shows the results of the Parallel join implementation in HBase.

File name	Size/ # of records	Mapper	Reducer	Time(sec)	Relation file
wp_annotation	1786MB/54039	43	10	43	659MB/24153638
ct_annotation	5916MB/164808416	613	10	482	659MB/24153638
pm_annotation	16983MB/42049697	1560	10	583	659MB/24153638

Table 4.6: Parallel Pig Join HBase. - The Execution time of Join

We depicted a diagram that shows the execution time for two parallel join algorithms such as Parallel Join in HDFS and Parallel Join in HBase. We conducted each experiment more than three times and present the mean of those values here. See Figure 4.3.

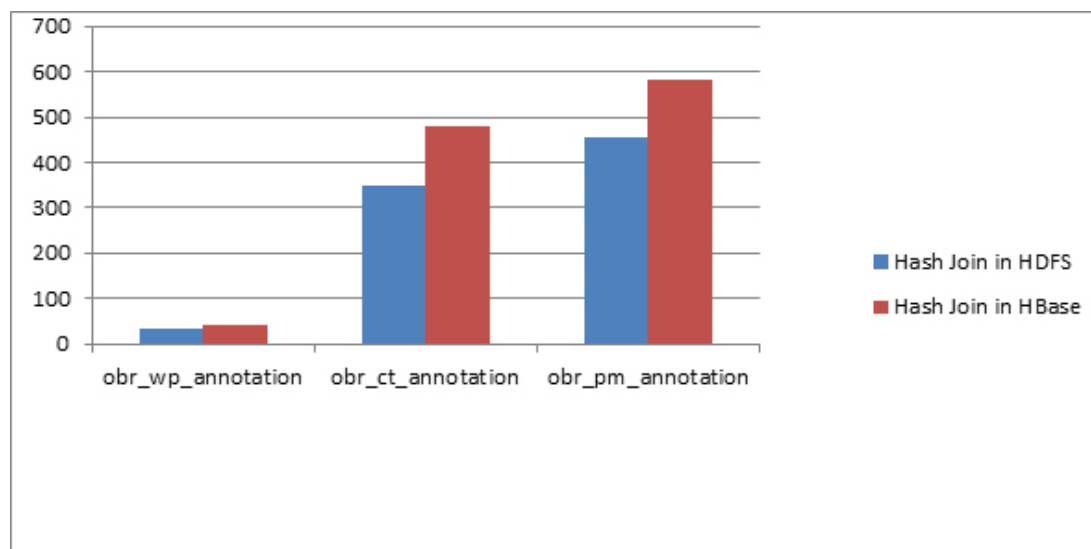


Figure 4.3: Parallel Joins in Pig - Execution time of join

4.4 Join Algorithms in MapReduce

4.4.1 MapReduce Joins in HDFS use Singleton pattern

On This Join we implement the Singleton pattern that we have mentioned in the background part.

How we implement the join depends on how large the datasets are and how they are partitioned. If one dataset is large but the other one is small enough to be distributed to each node in the cluster, then the join can be affected by a MapReduce job that brings the records into the every task trackers.

The mapper or reducer uses the smaller dataset to look up the relation metadata for a Join key, so it can be written out with each record. In our case the relation file is not so small for distributing the data to all the tasktrackers so we needed to implement other efficient way that is the Singleton pattern can fit our requirement. We use Singleton object for the relation dataset to look up the metadata for the Join key so every node of cluster has an instance of Singleton object that contains a Hash Map object. In each node have two reducers and those two reduce access through a Singleton object for looking up the join key (concept_id). see Figure 4.4

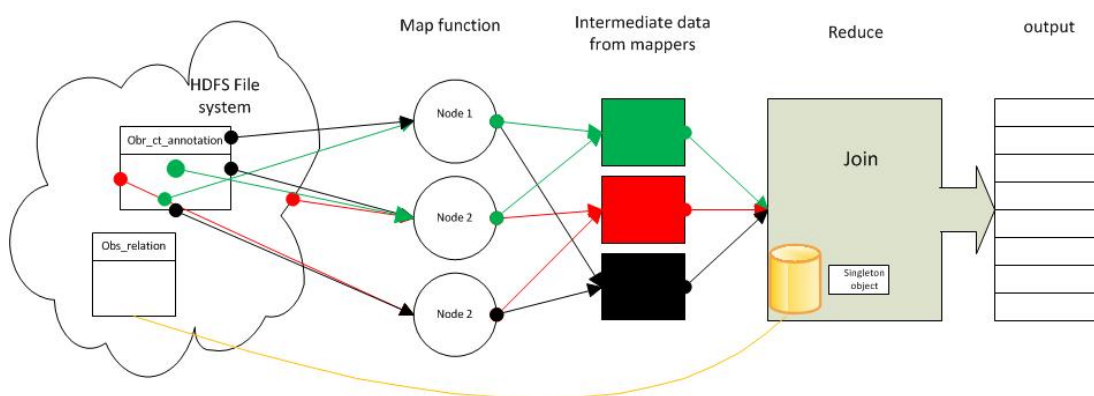


Figure 4.4: MapReduce Joins with Singleton in HDFS - Data flow diagram

Implementing Singletons: There are a few ways to implement Singletons. Although Singleton can be like behavior with static fields and methods, you gain

4. IMPLEMENTATION

more flexibility by creating an instance. With Singletons implemented as single instances instead of static class members, we can initialize the Singleton lazily, creating it only when it is first used. Likewise, with a Singleton implemented as single instance, you leave open the possibility of altering the class to create more instances in the future. With some implementations of the Singleton, you allow writers of subclasses to override methods polymorphically, something not possible with static methods.

We implement a Singleton in Java by having a single instance of the class as a static field. We can create that instance at class-loading time by assigning a newly created object to the static field in the field declaration.

Implementation details: In our case we consider two relations `obr_ct_annotation` (ANNO) and `obs_relation` (REL) in HDFS those have to be joined together and emit the output into HDFS. I will explain the sequential steps of the algorithms:

1. **Datasets:** we have two datasets as `obr_ct_annoation` and `obs_relation` those are in HDFS as a text file. The input file `obr_ct_annoation` is split by Hadoop into chunks those are assigned to a collection of map processes.
2. **Mapping:** The Map function carry out only on the `obr_ct_annotation` file. When map function run it read key-value pairs from the input file (`obr_ct_annotation`) and create intermediate key -value pairs such that in each pair, the generated key is the join key. In case of relation `obr_ct_annotation`, a map process will turn each tuple (K1, V1) from the input of ANNO file into a key-value pair.the
3. **Partitioning:** In our case the Mapper was producing (key, value) pairs, partitioning is to be done so that pairs with the same join key reach the same reducer.
4. **Shuffling:** The intermediate key-value pairs of mappers are shuffled across the network such that each reducer gets the key-value pairs of its partition.
5. **Grouping:** Each reducer groups the keys within a partition and presents each group to a separate reduce process.

4.4 Join Algorithms in MapReduce

6. **Reducing:** The Each groups (K1, [(V1) ,(V2) ,(V3)]) and (K2, [(V1) ,(V2) ,(V3)]) are provided to separate reduce processes those has single instance of Singleton object (obs_relation) and the method for the values by associated with a join key from the Hash Map. In each reduce process, a join is computed between the values grouped as K1 with values fetched as K1 from Hash Map (Singleton object). Then the joined records produced as a result of this cross-product are written to the output part file of the reducer.

The following the tables 4.7 shows the results of the MapReduce Joins in HDFS use Singleton pattern

File name	Size/ # of records	Mapper	Reducer	Time(sec)	Relation file
obr_wp_annotation	1786MB/54039	110	10	19	659MB/24153638
obr_ct_annotation	5916MB/164808416	110	10	69	659MB/24153638
obr_pm_annotation	16983MB/42049697	228	10	138	659MB/24153638

Table 4.7: MapReduce Joins in HDFS with Singleton pattern. - The Execution time of Join

4.4.2 MapReduce Reduce side join in HDFS

In this part we implement the Reduce side join that is applied on the reduce nodes and the join is more general than a other join, in that the input datasets do not have to be a structured in any particular way, but it is less efficient as both datasets have to go through the MapReduce shuffle process.

The main idea is that the mappers tags each record from the input dataset in the HDFS with its source information and generate key value pairs such that the join key as the map output key, so that the records with the same key are brought together in the reducer. Then key-value pairs in the output of Mapper are shuffled across the network and each reducer gets the emitted key-value pairs

4. IMPLEMENTATION

from both data sets. The join is carried out by each reducer between the records of the two datasets and the joined records are outputted.

In practice all of the input is interpreted by a single input format and a single Mapper function. In our case we need multiple inputs and different dataset those are different format. The input sources for the datasets have different formats, in general, so we need to implement the **MultipleInputs** class to separate the logic for parsing and tagging each source. For instance, one might be tab-separated plain text and the other a binary sequence file. Even if they are in the same format, they may have different representations and, therefore, need to be parsed differently.

Implementation details: In our case we consider two relations obr_ct_annotation (ANNO) and obs_relation (REL) that have to be joined together. Both relations are stored in separate text files in the HDFS. These two relations have to be joined on the concept_id as key in MapReduce that is not essentially the same as in relational databases. The Map/Reduce keys are not unique. They are just the attributes used to distribute data among reduce processes. To keep things simple, we consider an inner-join between ANNO and REL.

In the following figure 4.5 you can see the Join obr_ct_annotation and obr_relation using MapReduce.

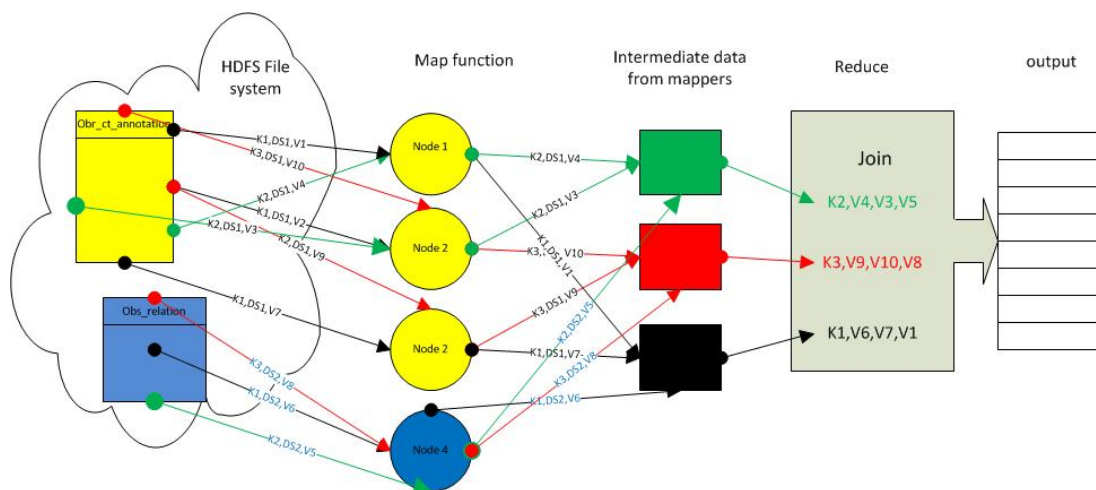


Figure 4.5: MapReduce Reduce side join in HDFS - Data flow diagram

As you see on the figure 4.5 I will explain the sequential steps of the algorithms:

1. **Datasets:** we have two datasets as obr_ct_annoation and obs_relation those are yellow and blue color for input. The input files of both the datasets are split by Hadoop into chunks those are assigned to a collection of map processes.
2. **Secondary sort:** the reducer will see the records from both sources that have same key, but they are not guaranteed to be in any particular order. However, to perform the join, it is important to have the data from one source before another.
3. **Mapping:** When map function run it read key-value pairs from the input file (obr_ct_annotation and obs_relation files) and create intermediate key-value pairs such that in each pair, the generated key is the join key. It also tags the intermediate key-value pairs with information about their source relation. Furthermore, the intermediate key-value pair emitted by a mapper consists of the join key tagged with data source (DS1 and DS2 on the figure) and a value.

In case of relation obr_ct_annotation, a map process will turn each tuple (K1, V1) from the input of ANNO file into a key-value pair with composite key-value and the indication of data source (K1, DS1,V1). Similarly, for obs_relation file, each map process will turn tuple (K1, V4) from REL into a key-value pair with key and the indication of data source (K1, DS2, V4). The indication DS1, DS2 that indicates the source of data. Instead of tagging a key-value pair with full name of the dataset, we use here this abridged notation in order to save some bytes since the tag has to be included with each key-value pair. This indication of data source has another use in sorting the pairs during the reduce phase as well. Also this tagging is important because in the reduce phase, we want tuples from ANNO and REL to be joined together and having an identifier for each relation helps distinguish the tuples according to their data sources and hence tuples from different relations are joined.

4. **Partitioning:** In Hadoop regular MapReduce job do a partitioning automatically after the map part. In our case the Mapper was producing

4. IMPLEMENTATION

(composite_key, value) pairs, partitioning is to be done so that pairs with the same join key reach the same reducer. But if we partition the key-value pairs on the basis of the composite key, values belonging to keys (K1,DS1) and (K1, DS2) will be reached to different reducers. We want the partitioning on the basis of just the join key K1 in our case.

Since the output of the map phase is a composite key (join key, data source), we need to implement our own partitioner that extracts the join key and assigns an appropriate partition number to the tuple depending on the hash value of this join key.

In our implementation of Partitioner class we overwrite the getPartition() function of this custom partitioner class, we extract the join key from the composite key and on the basis of the hash code of this join key, we assign it a particular partition number.

5. **Shuffling:** The intermediate key-value pairs of mappers are shuffled across the network such that each reducer gets the key-value pairs of its partition.
6. **Grouping:** Each reducer groups the keys within a partition and presents each group to a separate reduce process. But in our case we use composite key so we want the key-value pairs with same join attribute should be received by one reduce process. This can be accomplished by grouping the keys according to the join attribute. To achieve this, we implement our custom group comparator derived from WritableComparator class of Hadoop which considers only the join attribute for grouping.
7. **Reduction:** The groups (K1, DS2,[V6,V7,V1]) and (K2, DS2,[V9,V10,V8]) in the example above are provided to separate reduce processes. In each reduce process, a join is computed between the values tagged as DS1 with values tagged as DS2 after decoupling the tags from the (value, tag) pairs. The joined records produced as a result of this cross-product are written to the output part file of the reducer.

The following table 4.8 shows the results of MapReduce Reduce side join in HDFS

4.4 Join Algorithms in MapReduce

File name	Size/ # of records	Mapper	Reducer	Time(sec)	Relation file
obr_wp_annotation	1786MB/54039	200	20	36	659MB/24153638
obr_ct_annotation	5916MB/164808416	210	20	108	659MB/24153638
obr_pm_annotation	16983MB/42049697	379	20	248	659MB/24153638

Table 4.8: MapReduce, Reduce side join in HDFS. - The Execution time of join

We depicted a diagram that shows the execution time for two MapReduce join algorithms such as MapReduce, Reduce side Join in HDFS and MapReduce join with Singleton. We conducted each experiment more than three times and present the mean of those values here.

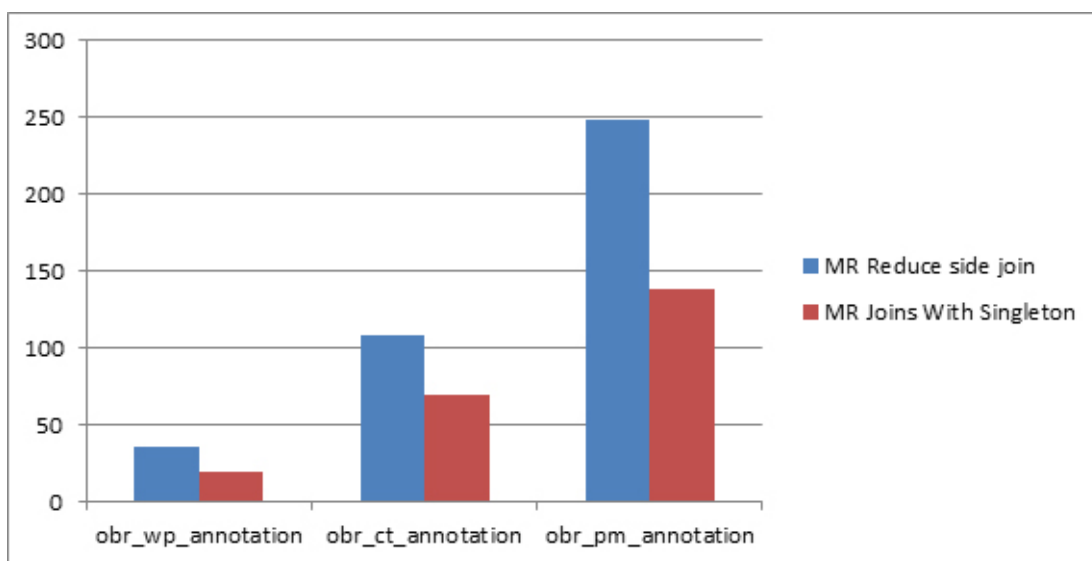


Figure 4.6: MapReduce joins in HDFS - The execution time of joins

4.4.3 MapReduce Join in HBase and HDFS

In this part we implement the MapReduce join that is applied on the reduce nodes and input datasets are in HBase and HDFS, in that the input datasets

4. IMPLEMENTATION

have to be structured in HBase as structure way.

The main idea is that the mappers tags each record from the bigger input dataset obr_ct_annotation in the HDFS and generate key value pairs such that the join key as the map output key, so that the records with the same key are brought together in the reducer. Then key-value pairs in the output of Mapper are shuffled across the network and each reducer gets the emitted key-value pairs from the big data sets. The smaller dataset obs_relation is in the HBase as a table, the join key is as row key on the table. In the Reducer side reducer function load the smaller dataset from the HBase to look up the relation meta data for a Join key. The join is carried out by each reducer between the records of the two datasets and the joined records are emitted. See Figure 4.7.

In our case the relation file is not so small for distributing the data to all the task trackers and I will be difficult to load into the memory of machines as Singleton implementation, so that is why we need to implement in the HBase. The main Advantage of this method is the tables (in HBase) are mutable that I can insert, fetch, update and edit and delete as single row or multiple rows. The HDFS works great with immutable data such as log file or some text file. In order to increase speed of the join I needed to make a multidimensional structure of obs_relation table.

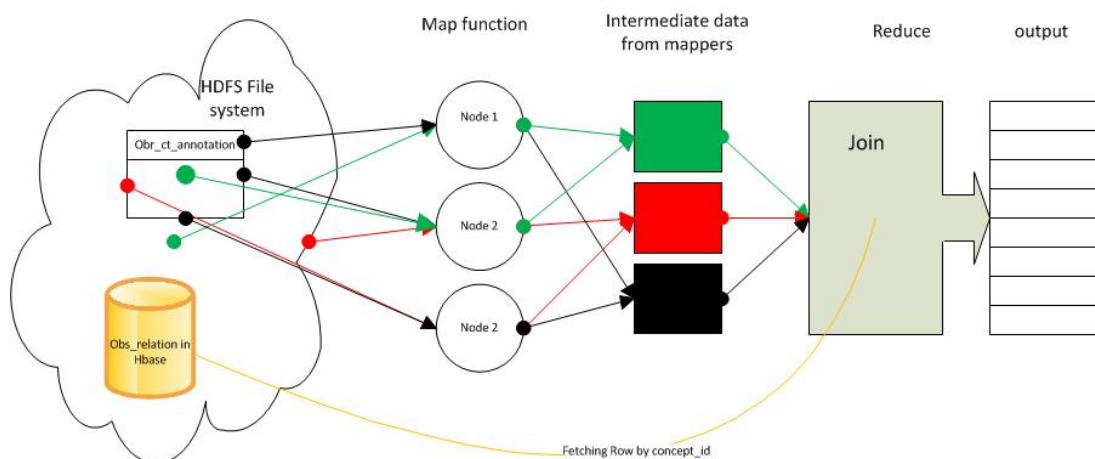


Figure 4.7: MapReduce join in HDFS and HBase - Data flow diagram

Implementation details: In our case we consider two relations obr_ct_annotation

(ANNO) in HDFS and obs_relation (REL) in HBase those have to be joined together and emit the output into HDFS. I will explain the sequential steps of the algorithms:

1. **Datasets:** we have two datasets as obr_ct_annoation and obs_relation those are in HDFS as a text file and HBase as a table respectively. The input file obr_ct_annoation is split by Hadoop into chunks those are assigned to a collection of map processes.

2. **Make a multidimensional structure:** As I mentioned before I have the obs_relation file that has 24,153,638 lines (row) and 690,981,765 bytes (658 MB). In order to get an advantage of HBase we need to reorganize the structure of the dataset. So I have changed the id by concept_id and I have added a dimension parent_concept_id as a column family. After I implement the multidimensional structure I can decrease the number of row up to 2257402 and it means a row has it is much easy to fetch row from HBase. It means more chances of hitting a single region to fetch the needed data. In the following table 4.9 is showed the implantation of multidimensional structure of obs_relation table.

Table name	Row key	Family	Attributes
obs_relation table	concept_id	metainfo:id	Always contains the column keys id. It should be IN-MEMORY.
		parent_concept_id:	Column keys are written like parent_concept_id and the values are written as a parent_level

Table 4.9: Multidimensional Structure of Relation table - obs_relation table

The table means a row has multiple columns as parent_concept_id under column family parent_concept_id and the values are written as a parent_level. See Figure 4.8.

4. IMPLEMENTATION

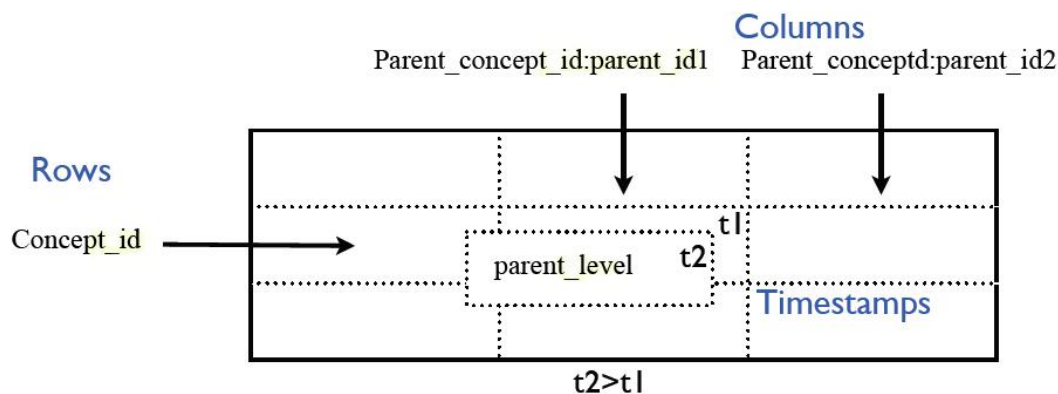


Figure 4.8: Hbase Cell structure - Cell diagram

3. **Mapping:** When map function run it read key-value pairs from the input file (obr_ct_annotation) and create intermediate key -value pairs such that in each pair, the generated key is the join key. In case of relation obr_ct_annotation, a map process will turn each tuple (K1, V1) from the input of ANNO file into a key-value pair.
4. **Partitioning:** In our case the Mapper was producing (key, value) pairs, partitioning is to be done so that pairs with the same join key reach the same reducer.
5. **Shuffling:** The intermediate key-value pairs of mappers are shuffled across the network such that each reducer gets the key-value pairs of its partition.
6. **Grouping:** Each reducer groups the keys within a partition and presents each group to a separate reduce process.
7. **Reducing:** The Each groups (K1, [(V1) ,(V2) ,(V3)]) and (K2, [(V1) ,(V2) ,(V3)]) are provided to separate reduce processes those has an instance of HTable (obs_relation) and the method for getting a row by associated with a key. In each reduce process, a join is computed between the values grouped as K1 with values fetched as K1 from **HTable**. Then the joined

4.4 Join Algorithms in MapReduce

records produced as a result of this cross-product are written to the output part file of the reducer.

In the following the tables 4.10 shows the results of the MapReduce Join in HBase and HDFS

File name	Size/ # of records	Mapper	Reducer	Time(sec)	Relation file
obr_wp_annotation	1786MB/54039	110	10	41	659MB/24153638
obr_ct_annotation	5916MB/164808416	210	10	2780	659MB/24153638
obr_pm_annotation	16983MB/42049697	228	10	3650	659MB/24153638

Table 4.10: MapReduce Joins in HDFS and HBase - The Execution time of Join

We depicted a diagram 4.9 that shows the execution time for join algorithms in Pig and the execution time for MapReduce join algorithms 4.10. We conducted each experiment more than three times and present the mean of those values here.

4. IMPLEMENTATION

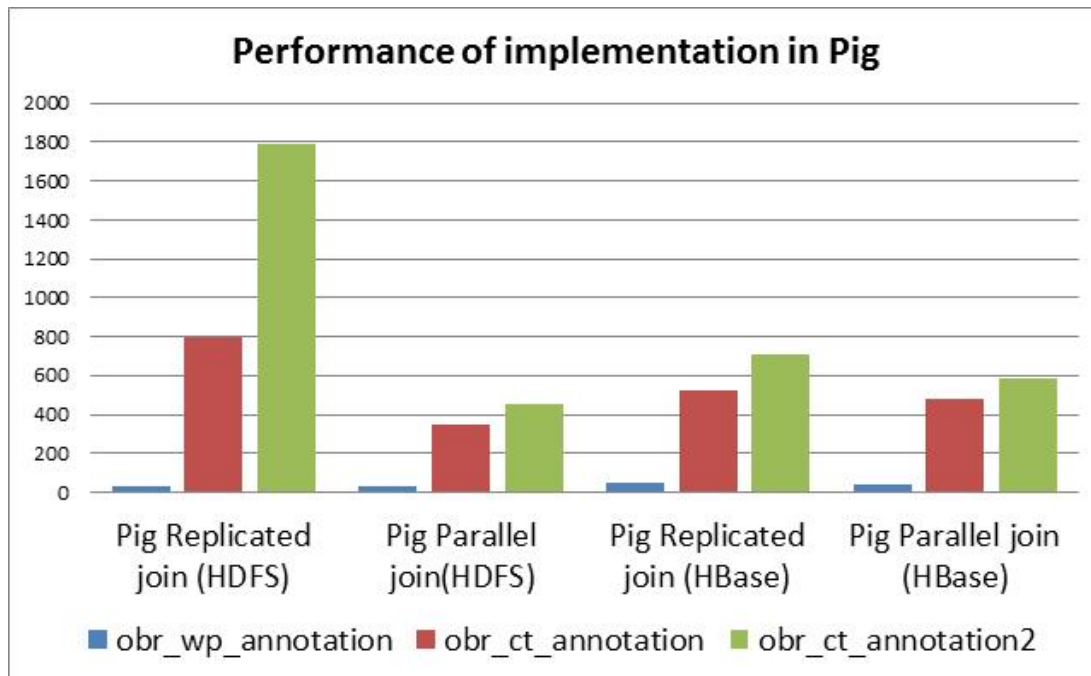


Figure 4.9: Join algorithms in Pig - The Execution time of Joins

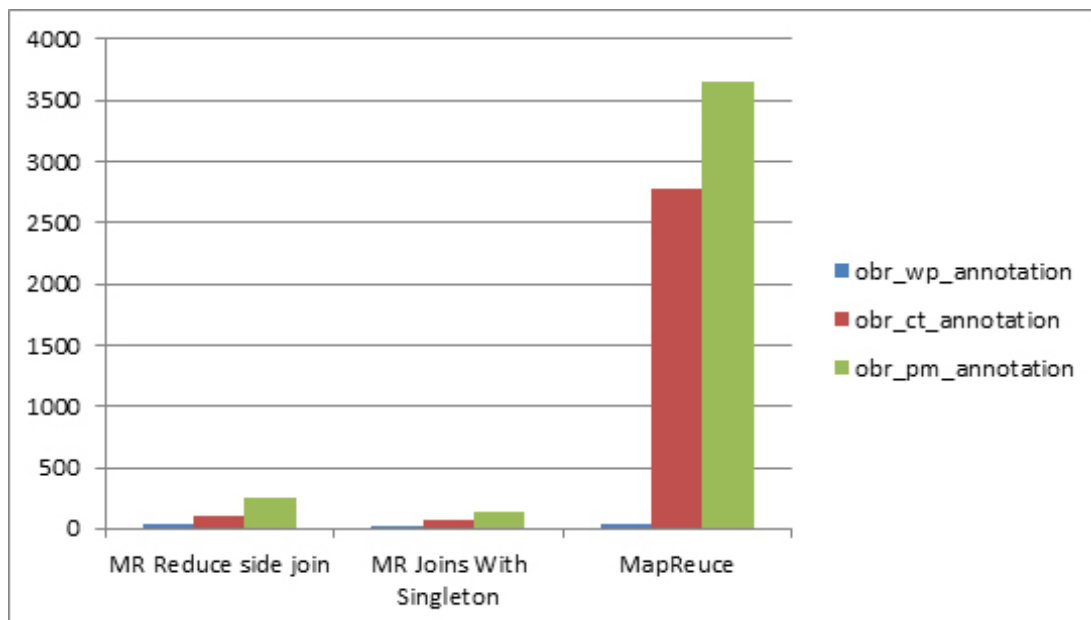


Figure 4.10: Join algorithms in MapReduce - The Execution time of Joins

4.4.4 Build Multidimensional structure in HBase

In NCBO BioPortal (14) they are currently working on expanding the Resource Index to include more resources. The goal is to index up to 100 public resources, including PubMed (15), which provides access to all research articles in biomedicine. They have already encountered limitation of storage and processing, with the original workow taking too long to process each resource. The most computationally expensive process is term expansion based on semantics, which relies mostly on join operations.

The NCBO Resource Index that is used for search purposes is a product of several complex processes. Resource Index has to be available for search and it would be beneficial to introduce the update as soon as possible. We need special architecture that will allow for concurrency of updates and searches. That will allow for continuous updates to Resource Index without taking the search offline. Moreover, such architecture should support many concurrent requests without noticeable decrease in performance.

So I have analyzed the resent structure of the database in order to re-structure the database; this restructuring will be enabled us to reduce the processing time for one of the larger datasets from one week to few hour and no need time for join queries.

Implementation details: Getting high scalability from your relational database is not done by simply adding more machines because its data model is based on single-machine architecture. HBase can be reduced to a Map [byte[], Map [byte[], Map [byte[], Map [Long, byte[]]]]]. The first Map maps row keys to their column families. The second maps column families to their column keys. The third one maps column keys to their timestamps. Finally, the last one maps the timestamps to a single value. The keys are strings, the timestamp is a long and the value is an array of bytes. The column key is always preceded by its family and is represented like this: family:key. Since a family maps to another map, this means that a single column family can contain a theoretical infinity of column keys. So, to retrieve a single value, the user has to do a get using three keys:

- row key+column key+timestamp = value

4. IMPLEMENTATION

1. **Rows:** In our case we chose the row key as `concept_id` that is treated by HBase as an array of bytes but it must have a string representation. A special property of the row key Map is that it keeps them in a lexicographical order. To keep the integers natural ordering, the row keys have to be left-padded with zeros. Getting a row we can collect the all parent concept ids with their elements information for a concept quickly and the advantage of making this structure is all related data will be saved on the a region in on a Region server. In order get a row data we use `Get` class of HBase client API. When I get a row I will have maps those contain keys as column families (position_from, metainfo, element_id etc in the table.) and values as maps those contain keys as column keys.
2. **Column Families:** A column family regroups data of a same nature in HBase and has no constraint on the type. The families are part of the table schema and stay the same for each row; what differs from rows to rows is that the column keys can be very different from each other. For example, row key "20080702" (`concept_id`) may have in its "content_id:" family the following column keys as `element_id` or `parent_concept_id`:

```
content_id:321321
content_id:43215432
content_id:1
```

A column family is a dimension so in my implementation have 11 column families it means a table with 11 dimensions.

3. **Timestamps:** The values in HBase may have multiple versions kept according to the family configuration. By default, HBase sets the timestamp to each new value to current time in milliseconds and returns the latest version when a cell is retrieved. In our case I left the timestamp for latest data time for fetching the values.

In the figure 4.11 shows the resent database structure of MySQL :

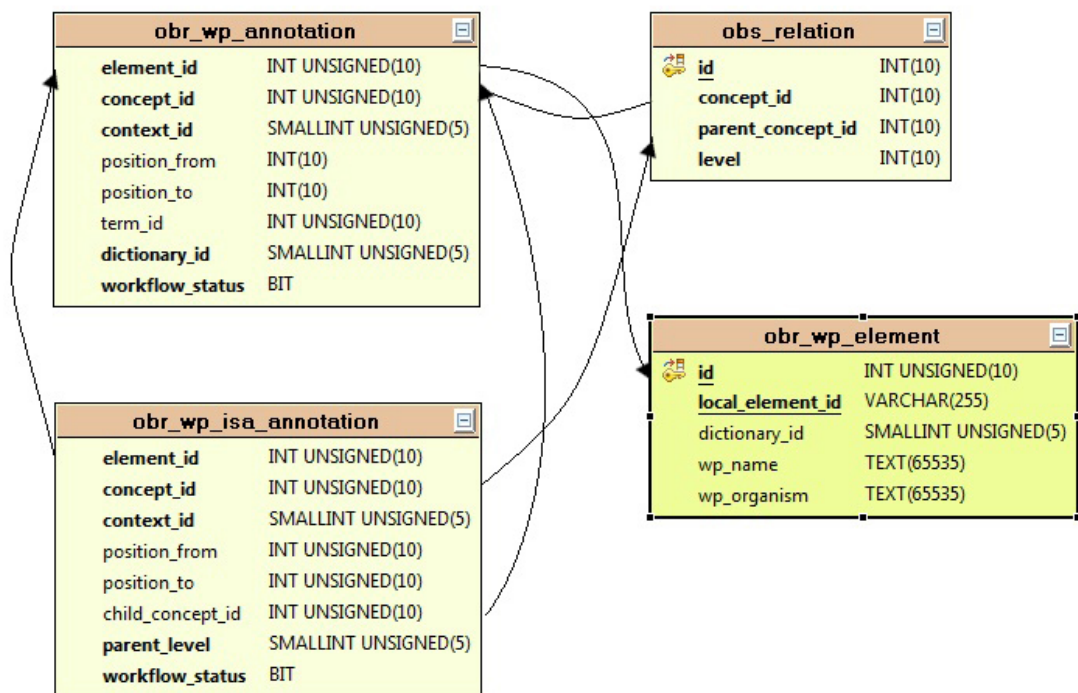


Figure 4.11: ERD (entity relationship diagram) - ERD (entity relationship diagram) of Database

4. IMPLEMENTATION

Here you can see the Multidimensional index structure in HBase 4.11

Also you can see 4.11 that the one-to-many relationship between obr_wp_annotation and obr_wp_element is handled by putting each attributes of the obr_wp_element as a family in Annotation table and by using it an element_id as a column key, all elements are already sorted. One advantage of this design is that when we fetch a row we can get the all the elements and parents.

4.4 Join Algorithms in MapReduce

Table name	Row key	Family	Attributes
Annotation table	concept_id	metainfo:	Always contains the column keys dictionary_id and workflow
		content_id:	Column keys are written like element_id that is from the obr_wp_annotation table. The values are written as a content_id.
		context_id:	Column keys are written like element_id and the values are written as a context_id
		position_from:	Column keys are written like a element_id and the values are written as a position_from.
		position_to:	Column keys are written like element_id and the values are written as a context_id
		term_id:	Column keys are written like element_id and the values are written as a position_to.
		element_id:	Column keys are written like element_id and the values are written as a element_id
		parent_concept_id:	Column keys are written like parent_concept_id and the values are written as a parent_level
		local_element_id:	Column keys are written like element_id and the values are written as a local_element_id
		wp_name: wp_organism:	Column keys are written like element_id and the values are written as a local_element_id

Table 4.11: Multidimensional index structure in HBase - HBase Resource index

4. IMPLEMENTATION

Chapter 5

Discussion on the results

We would like to see following results of our experiments, it is quite evident from the results presented in Figures 5.1,5.2,5.3,5.4 those for combination of five different distributed join algorithms and two different distributed file systems such as HDFS and HBase.

Theoretically Replicated Join should be faster than Hash Join but in my case Regular Hash join was faster than replicated join. The reason behind this is because for the number of map tasks started are equal to the number of blocks in the left side input of replicated join. Each of these blocks will read the replicated input. If the replicated input read size is few times larger than block size, using replicated join will not save on IO/ (de)serialization costs. In the Replicated Join in Pig we have got only better performance than other joins in Pig when the input data is small enough in our case of obs_wp_annotation file. The Replicated join does not finish when the build relation does not fit in the main memory. In the experiments conducted for the comparison of the join algorithms, fortunately the relation file is fitting in the main memory, but more practical, real datasets consisting of trillions of records are huge enough to exceed the size of the main memory. Hence, the Replicated join will be out of the competition for future work.

The MapReduce join with Singleton pattern performs efficiently to handle the situations in all tree different input dataset. Except MapReduce join with Singleton pattern, among the remaining algorithms, the reduce-side join performs

5. DISCUSSION ON THE RESULTS

better in case of obr_ct_annotation and obr_pm_annotation. This is because all other algorithms require partitioning of datasets on the join key to accumulate similar keys of both datasets into partitions with same partition number. The two corresponding partitions of both datasets are then joined in the join stage. Since the reduce-side join skips this phase of partitioning, its performance is better than the other algorithms.

It is evident from the results that in case of MapReduce join in HDFS and HBase and join MySQL. The algorithm MapReduce join in HDFS and HBase takes a longer time than others joins but it was faster than implementation of join in MySQL it show in case of our situation using cluster is more efficient than Relational database management system (RDBMS). In our experiments running join query is possible in case of obr_wp_annotation and obr_ct_annotation files and it was not able to complete the query in case of obr_pm_annotation file but more practical, real datasets consisting of trillions of records are huge enough to exceed the size of the single machine so running RDBMS server is expansive and difficult to create the index.

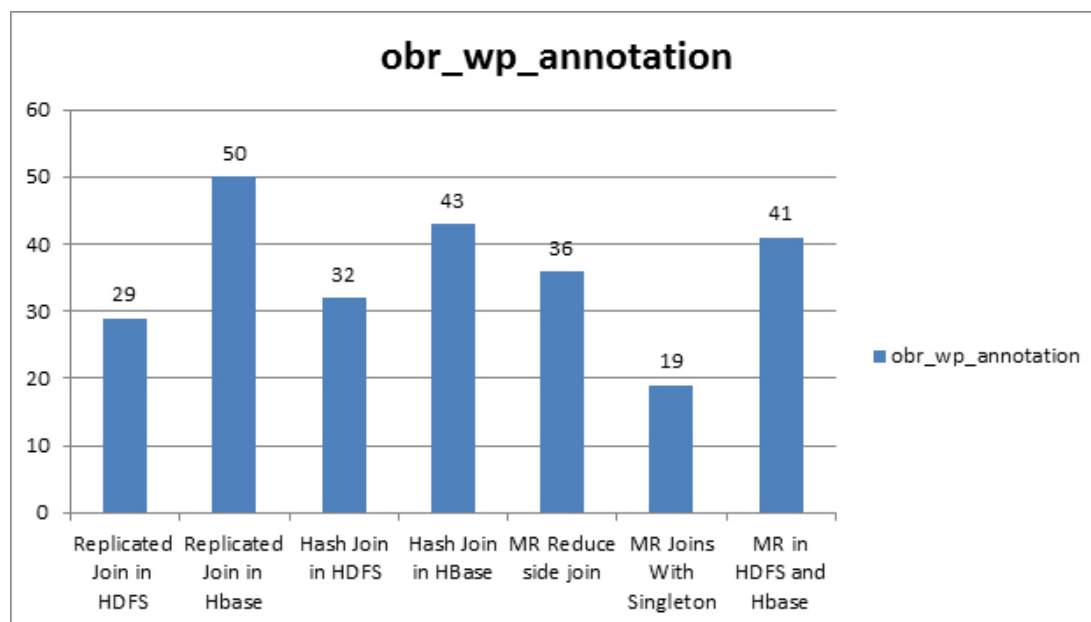


Figure 5.1: Joins in obr_wp_annotation - The Execution time of obr_wp_annotation file

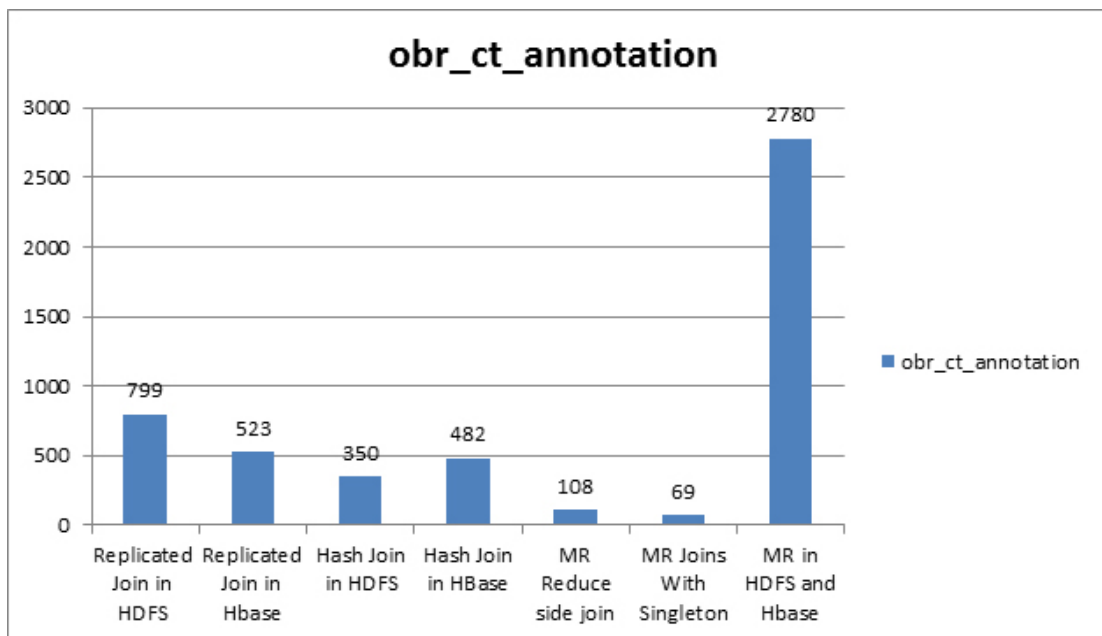


Figure 5.2: Joins in obr_wp_annotation - The Execution time of obr_ct_annotation file

5. DISCUSSION ON THE RESULTS

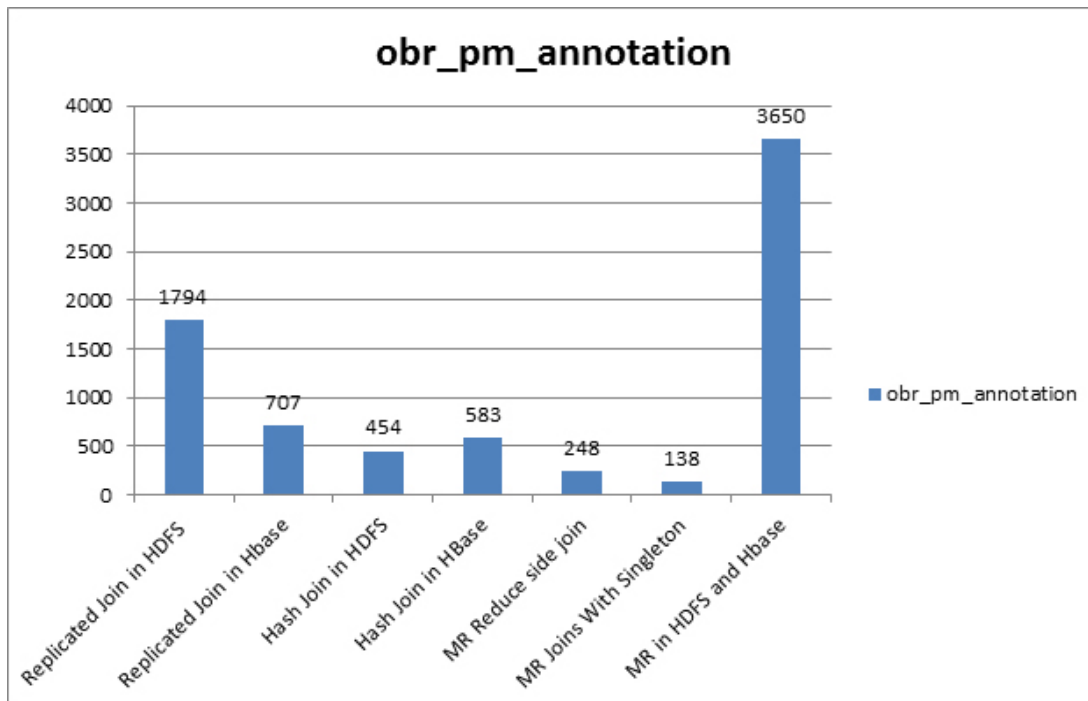


Figure 5.3: Joins in obr_wp_annotation - The Execution time of obr_pm_annotation file

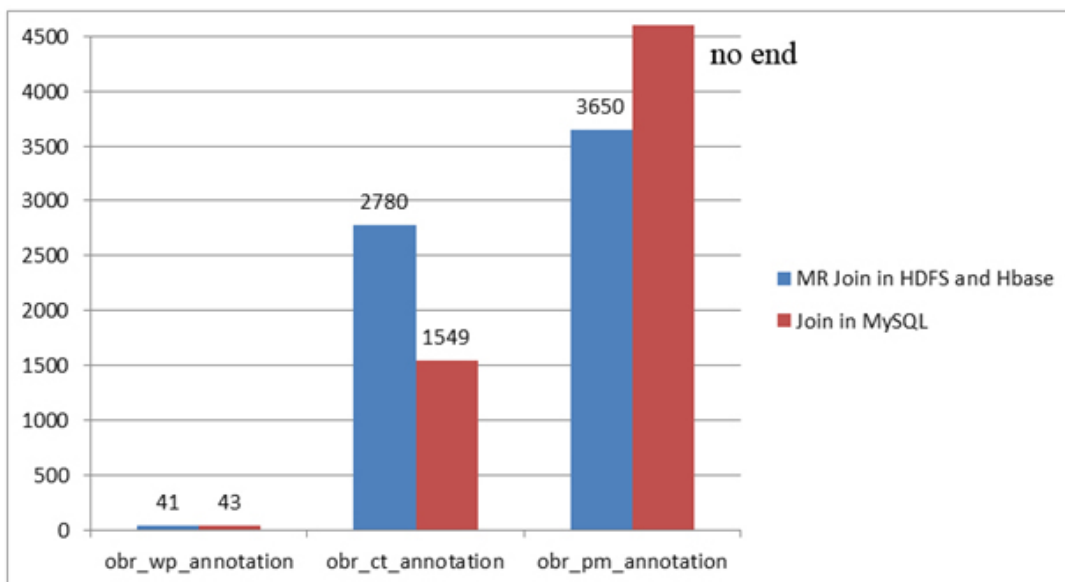


Figure 5.4: Joins in MySQL - The Execution time in MySQL

Number	Time	Mappers	Reducers	File	Join Technique	Storage	Language
1	29	12	0	wp	Replicated	hdfs	Pig
2	799	104	0	ct	Replicated	hdfs	Pig
3	1794	277	0	pm	Replicated	hdfs	Pig
4	32	12	10	wp	Parallel Hash	hdfs	Pig
5	350	104	10	ct	Parallel Hash	hdfs	Pig
6	454	277	10	pm	Parallel Hash	hdfs	Pig
7	50	43	0	wp	Replicated	hbase	Pig
8	532	613	0	ct	Replicated	hbase	Pig
9	707	1560	0	pm	Replicated	hbase	Pig
10	43	43	10	wp	Parallel Hash	hbase	Pig
11	482	613	10	ct	Parallel Hash	hbase	Pig
12	583	1560	10	pm	Parallel Hash	hbase	Pig
13	19	110	10	wp	MR Singleton	hdfs	MR
14	69	110	10	ct	MR Singleton	hdfs	MR
15	138	228	10	pm	MR Singleton	hdfs	MR
16	41	110	10	wp	Join in HDFS and Hbase	hbase,hdfs	MR
17	2780	210	10	ct	Join in HDFS and Hbase	hbase,hdfs	MR
18	3650	228	10	pm	Join in HDFS and Hbase	hbase,hdfs	MR
19	36	200	20	wp	MR Reduce side join	hdfs	MR
20	108	210	20	ct	MR Reduce side join	hdfs	MR
21	248	379	20	pm	MR Reduce side join	hdfs	MR
22	42	0	0	wp	MySQL Join	MySQL	SQL
23	1549	0	0	ct	MySQL Join	MySQL	SQL
24	no end	0	0	pm	MySQL Join	MySQL	SQL

Figure 5.5: The Execution time of Joins - The Execution time of Joins

5. DISCUSSION ON THE RESULTS

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis we presented methods related with scaling-out the semantic annotation data of NCBO Resource Index (13) those they have implemented on MySQL sever on single machine. In order to improve the performance of the computation we implement the algorithms for data-parallel computing and data combining.

We have presented a time difference of computation performance both user defined function and high-level query languages, furthermore the choice of programming interface has a different effect on the performance of computation. In order to get good performance I had organized the cluster server and implemented the good execution plans that can fit well for such computation and platform.

This thesis evaluates the implementations for performing data combination and computation in several state of the art distributed computing systems: Hadoop (20), HBase (26), Pig (24), MapReduce (1) and MySQL Servers. We had stored data in storage in the Hadoop Distributed File System (HDFS) (22), which is abstracts data distribution across the cluster nodes and HBase, which is distributed database system based on the HDFS. The Pig and The Map/Reduce framework facilitates parallel processing of the data distributed among processing nodes in a computing cluster.

We have implemented several methods in two different storage HDFS and

6. CONCLUSIONS AND FUTURE WORK

HBase, the execution of all the Join in HDFS was faster than in HBase, furthermore As every kind of Processing task benefits from parallelization, so does the parallel joining of datasets. The processing stage was initially implemented in Pig but we needed to compare it with pure MapReduce jobs to potentially leverage a custom HBase index. Implementing join in Pig was easier than join in MapReduce.

In this project, we presented a join algorithm that is a MapReduce Join with Singleton pattern and the MapReduce reduce-side joins and those are capable of handling in the any input datasets. The MapReduce join with Singleton pattern performs efficiently to handle the situations of our case.

The hash join in Pig performs well in the case of big data which is bigger than memory of machine and the Replicated join has a better performance when the input data is small enough in our case of obs_wp_annotation file, our algorithm dynamically selects an appropriate partitioning strategy on the basis of the characteristics of the input data.

I have analyzed the resent structure of the database in order to create a custom HBase index as a multidimensional structure; this restructuring enabled us to reduce the processing time for one of the larger datasets from one week to few hours and we do not need a time for join queries. The multidimensional structure of HBase will be also investigated as a way to reduce storage requirements for semantic expansion. This allowed scaling-out the NCBO Resource Index to cover all the required resources and to perform better.

6.2 Future Work (11)

Processing part: The project can be extended in future to apply several different data processing languages those base on top of Hadoop such as Hive (23) is a SQL-like query language and Cascalog (31) is a Clojure-based query language for Hadoop inspired by Datalog. Furthermore can be used for data storage HyperTable (32) can be an alternative it has its own HyperTable Query Language with a SQL-like syntax.

Maintaining Part: The big part of future work is maintaining the Resource Index that includes two main tasks: updating and making it available for rapid search and those both tasks should work in parallel. In order to maintain the Resource index that we implement in HBase it can be implemented client API of HBase. Cassandra and Voldemort (30) can both be considered here as they focus on fast data serving in contrast with bulk processing in, for example, HBase. They can also handle parallel read and writes well. However, they do not offer any dedicated mechanism for handling the connection with the bulk processing backend. Though, it can be constructed. The most recent project that emerged is ElephantDB (29). It consists of two integrated components where one is dedicated to creating the Resource Index and the other to serving it.

6. CONCLUSIONS AND FUTURE WORK

References

- [1] SPYROS BLANAS, JIGNESH M. PATEL, VUK ERCEGOVAC, JUN RAO, EUGENE J. SHEKITA, YUANYUAN TIAN. **A Comparison of Join Algorithms for Log Processing in MapReduce.** *Computer Sciences Department University of Wisconsin-Madison, IBM Almaden Research Center.* 19, 20, 57
- [2] JEFFREY DEAN AND SANJAY GHEMAWAT. **MapReduce: Simplified Data Processing on Large Clusters.** *Google, Inc.* 2004. ii, 19
- [3] YAHOO. **MapReduce**. <http://developer.yahoo.com/hadoop/tutorial/module4.html> 2011. 2, 10
- [4] HUNG-CHIH YANG, ALI DASDAN RUEY-LUNG HSIAO, D. STOTT PARKER. **Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters.** *Yahoo! Sunnyvale, CA, USA, Computer Science Department, UCLA Los Angeles, CA, USA.* 19
- [5] GANG LUO LIANG DONG. **Optimizing Joins in a Map-Reduce Environment.** *NDuke University Durham, NC 27705.* 19
- [6] FOTO N. AFRATI JEFFREY D. ULLMAN. **Optimizing Joins in a Map-Reduce Environment.** *National Technical University of Athens, Greece Stanford University, USA.* 19
- [7] HERODOTOS HERODOTOU, HAROLD LIM, GANG LUO, NEDYALKO BORISOV, LIANG DONG, FATMA BILGEN CETIN, SHIVNATH BABU. **Starfish: A Self-tuning System for Big Data Analytics.** *erodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, Shivnath Babu.* 19
- [8] YUAN YU PRADEEP KUMAR GUNDA MICHAEL ISARD. **Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations.** *Microsoft Research 1065 La Avenida Ave. Mountain View, CA 94043.* 19
- [9] CLEMENT JONQUET, MARK A. MUSEN, AND NIGAM SHAH. **A System for Ontology-Based Annotation of Biomedical Data.** *Stanford Center for Biomedical Informatics Research Stanford University School of Medicine.*
- [10] PAEA LEPENDU, NATALYA F. NOY, CLEMENT JONQUET, PAUL R. ALEXANDER, NIGAM H. SHAH, AND MARK A. MUSEN. **Optimize First, Buy Later: Analyzing Metrics to Ramp-up Very Large Knowledge Bases.** *Stanford University, Stanford, California USA.*
- [11] TOMASZ WIKTOR WLODARCZYK, PAEA LEPENDU, NIGAM SHAH, CHUNMING RONG. **Scaling-out the NCBO Resource Index Processing and Maintenance.** *University of Stavanger, 4036, Stavanger, Norway Stanford University, USA.* 2011 ii, iv, 2, 58, 59
- [12] CLEMENT JONQUET, PAEA LEPENDU, SEAN M. FALCONER, ADRIEN COULET, NATALYA F. NOY, MARK A. MUSEN, AND NIGAM H. SHAH. **NCBO Resource Index: Ontology-Based Search and Mining of Biomedical Resources.** *Stanford Center for Biomedical Informatics Research, Stanford University, US.* 22
- [13] NATIONAL CENTER FOR INTEGRATIVE BIOMEDICAL INFORMATICS. **NCBO Resource Index.** <http://www.bioontology.org/resources-index/>, 2011. ii, 1, 3, 5, 22, 57
- [14] NATIONAL CENTER FOR INTEGRATIVE BIOMEDICAL INFORMATICS. **NCBO BioPortal.** <http://bioportal.bioontology.org/resources/>, 2011. 1, 45
- [15] NATIONAL CENTER FOR INTEGRATIVE BIOMEDICAL INFORMATICS. **NCBO BioPortal.** <http://www.ncbi.nlm.nih.gov/pubmed/>, 2011. 4, 45
- [16] FAY CHANG, JEFFREY DEAN, SANJAY GHEMAWAT, WILSON C. HSIEH, DEBORAH A. WALLACH MIKE BURROWS, TUSHAR CHANDRA, ANDREW FIKES, ROBERT E. GRUBE. **Bigtable: A Distributed Storage System for Structured Data.** *Google, Inc.,* 2006.
- [17] CHUCK LAM. **Hadoop in Action.** *Manning Publications Co,* 2011.
- [18] JASON VENNER. **Pro Hadoop.** *Apress,* 2010.
- [19] TOM WHITE. **Hadoop: The Definitive Guide, Second Edition.** *O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.,* 2011. 11, 13
- [20] THE APACHE SOFTWARE FOUNDATION. **Apache Hadoop.** <http://hadoop.apache.org/>, 2011. ii, 2, 7, 57
- [21] THE APACHE SOFTWARE FOUNDATION. **Apache Hadoop HDFS.** <http://hadoop.apache.org/hdfs/>, 2011. 7, 15, 21, 57

REFERENCES

- [22] YAHOO. **Yahoo Hadoop Tutorial**. . <http://developer.yahoo.com/hadoop/tutorial/>, 2011. 7, 15, 21, 57
- [23] THE APACHE SOFTWARE FOUNDATION. **Apache Hive**. <http://hive.apache.org/>, 2011. 20, 58
- [24] THE APACHE SOFTWARE FOUNDATION. **Apache Pig**. <http://pig.apache.org/>, 2011. ii, 2, 16, 20, 57
- [25] THE APACHE SOFTWARE FOUNDATION. **Pig Cookbook**. <http://pig.apache.org/docs/r0.8.1/cookbook.html>, 2011. 23
- [26] THE APACHE SOFTWARE FOUNDATION. **Apache HBase**. <http://hbase.apache.org/>, 2011. ii, 2, 12, 21, 57
- [27] THE APACHE SOFTWARE FOUNDATION. **Apache HBase Book**. <http://hbase.apache.org/book/book.html>, 2011. 12
- [28] U.S. NATIONAL LIBRARY OF MEDICINE NATIONAL INSTITUTES OF HEALTH. . <http://portal.ncbi.org/gateway/>, 2011. 3
- [29] BACKTYPE. . <http://tech.backtype.com/introducing-elephantdb-a-distributed-database>, 2011. 59
- [30] VOLDEMORT. . <http://project-voldemort.com/>, 2011. 59
- [31] NATHAN MARZ. **Cascalog**. <http://nathanmarz.com/blog/introducing-cascalog-a-clojure-based-query-language-for-hado.html>, 2011. 58
- [32] . **HyperTable**. <http://www.hypertable.org/>, 2011. 58