



University of
Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization:

Spring Semester, 2012

Master Degree Program in Computer Science

Open Access

Writer: Ming Hao

.....
(Ming Hao)

Faculty supervisor:

Chunming Rong (UiS)

Tomasz Wiktor Włodarczyk (UiS)

External supervisor(s):

Title of thesis:

Performance Analysis and Optimization of Left Outer Join on Map Side

Credits (ECTS): 30

Key words:

MapReduce

HDFS

Left Out Join

Map Side Join

Performance Analysis and Optimization

Pages: 59

+ enclosure: CD

Stavanger
15th June, 2012

Abstract

Ontologies are representations of the entities and relationships that structure an application area. Ontologies are important for tasks such as data integration, natural-language processing, information retrieval, and decision support.

NCBO Resource Index is a system for ontology based annotation and indexing of biomedical data. With the increasing of its data, a distributed processing method should be implemented, which can store, compute and inquire those large-scale data in an efficient way.

This paper is based on the master thesis of B. Byambajav, *Methods for Large-scale Semantic Expansion on Hadoop Architecture*, and going forward to seek a better solution for process NCBO Resource Index data and forced on performance optimization of left outer join on the Map side. In this paper, we researched and contrasted different kinds of join algorithms.

In order to implement more effective experiments, we studied the characteristics of HDFS and DistributedCache, then an algorithm of left outer join on map side had been implemented on the Hadoop platform, and for the purpose of performance optimization, we inspected several methods to control amount of map task.

Further, according to the result of the experiment, we adjusted critical parameters and we got a lot of valuable conclusions. Based on these conclusions, we found the map side join works well and got a better result in previous works.

Acknowledgments

I would like to express my gratitude to my supervisor Professor Chunming Rong, who is the head of CIPSI with very high achievements in cloud computing, for his supervision. Also I would like to express my gratitude to PostDoc Tomasz Wiktor Włodarczyk. Tomasz is a clever and detailed tutor, gave me many proposals and supports during my master's work.

I would like to express my gratitude to my parents for their support in my living and study in the past two years. And my wife Fei Lin, if there is no support with her, I could not accomplish well with my study.

Additionally I would like to thank the University of Stavanger for the opportunity of learning the master study program, which gave me a lot of experiences and memories of life.

Ming Hao

June 15, 2012, Stavanger

Contents

ABSTRACT	2
ACKNOWLEDGMENTS	3
CONTENTS	4
LIST OF FIGURES	6
LIST OF TABLES	7
CHAPTER 1. INTRODUCTION	8
1.1 Motivation	8
1.2 The Thesis Outline	8
CHAPTER 2. BACKGROUND	9
2.1 NCBO Resource Index.....	9
2.1.1 OBR Resources	10
2.1.2 OBS Database	11
2.2 HDFS and DistributedCache	12
2.2.1 HDFS Features	12
2.2.2 DistributedCache in Brinf	13
2.3 Map and Reduce.....	14
2.3.1 MapReduce Programming Model.....	14
2.3.2 MapReduce Behaviors in A Cluster.....	15
2.4 Hadoop Job Scheduling	15
2.5 HashMap.....	16
CHAPTER 3. RELATED WORK	18
3.1 Classic Join on Hadoop	18
3.1.1 Default Join	18
3.1.2 Map Side Join	18
3.1.3 Pig and Hive Join	18
3.1.4 Other Joins	19
3.2 Previous Work	19
CHAPTER 4. IMPLEMENTATION	20
4.1 Datasets and Cluster Environment.....	20
4.1.1 Data Sets Used	20
4.1.2 Cluster Specification	21
4.1.3 Hadoop Configurations	21
4.2 Algorithm of Map Side Join	22
4.3 Control the Number of Mapper	23
4.4 Performance Experiments.....	24
4.4.1 Block Size and Split Size Test.....	24

4.4.2 Replication Number Test	27
4.4.3 Relation Table Size Test	29
4.4.4 Split Table Size Test.....	30
4.4.5 DistributedCache Test.....	32
4.4.6 A Model Assumption.....	34
CHAPTER 5. DISCUSSION ON THE RESULTS	36
5.1 Strengths and Weaknesses of Map Side Join	36
5.2 The Usefulness of DistributedCache	37
5.3 Load Balancing in HDFS	37
CHAPTER 6. CONCLUSIONS AND FUTURE WORK	39
6.1 NCBO Resource Index Solution	39
6.2 Future Work.....	39
REFERENCE	41
APPENDIX	43
I. HDFS Common Commands.....	43
II. Usage of the Programs	45
III. Generation Methods of NCBO Resource Index.....	46
IV. Hadoop Configuration in Our Experiment	47
V. Source Codes	51

List of Figures

Figure 1: NCBO Resource Index workflow	9
Figure 2: Population of the OBR tables.....	10
Figure 3: Population of the OBS database	11
Figure 4: HDFS Architecture	13
Figure 5: A general flow diagram of MapReduce	14
Figure 6: Job scheduling flow chart	16
Figure 7: Time components of different split size	25
Figure 8: Job cost of different split size	26
Figure 9: Speed of Join	27
Figure 10: A comparison of time cost by different number of replication.....	28
Figure 11: The cluster block distribution	29
Figure 12: Time cost on different size of the relation table	30
Figure 13: Time cost in scale of exponential distribution.....	31
Figure 14: Time cost in scale of linear distribution.....	31
Figure 15: Time cost by different size of split table.....	32
Figure 16: Speed of Join	33
Figure 17: A Comparison of time cost by using and without using DistributedCache	33
Figure 18: Expected and actual time cost for different split size	35
Figure 19: Time cost comparison of the different Join methods	36
Figure 20: RAID 0 mode	40

List of Tables

Table 1: Original files from NCBO Resources Index	20
Table 2: Generated R tables.....	20
Table 3: Generated S Tables by Exponential Growth	20
Table 4: Generated S Tables by Linear Growth	21
Table 5: Time cost of different split size	25
Table 6: Time cost of 1GB S Table.....	26
Table 7: Time cost by different size of split table	32
Table 8: Time contrast of expected and actual for different split size	34

Chapter 1. Introduction

1.1 Motivation

What the NCBO Resource Index situation is, data is in the continuous growth, traditional DBMS ways to storage and to process data is limited by the capacity of the single machine. A scale-out solution, which fully uses of the power of cloud computing should be utilized for the constantly changed biological world. [1]

The purpose of this thesis is trying to find a better situation to solve this problem, which is processing the date of NCBO Resource Index.

Our project is based on the master thesis of B. Byambajav, *Methods for Large-scale Semantic Expansion on Hadoop Architecture*, and going forward to analyze and optimize the performance of left outer join, especially in Map side join.

In our project, we are going to study the classic join methods on Hadoop, the features of HDFS, and we are going to explore Hadoop job scheduling, the MapReduce behaviors in the cluster. For various aspects of experiments, we are going to implement algorithms of Map side join, generate sufficient number of data which based on the NCBO Resource Index and we will contrast the results.

1.2 The Thesis Outline

Chapter 1 is the introduction of our motivation and the thesis outline.

Chapter 2 is the background of resources and principles we used in our thesis.

Chapter 3 is the theories and previous work that related in our thesis.

Chapter 4 is the implementation of our project, which includes the algorithm, hypothesis and experiments. Based on the experimental results, we got much valuable knowledge.

Chapter 5 is the discussions of the work, we talked about the advantage and disadvantage of our algorithm, also some experiences in our experiments.

Chapter 6 is the conclusions and future work. We proposed some proposals which could be a good solution for the NCBO Resource Index.

Chapter 2. Background

2.1 NCBO Resource Index

The National Center for Biomedical Ontology Resource Index is a system for ontology based annotation and indexing of biomedical data. The fundamental functionality of the system is to enable users to locate biomedical data resources related to particular concepts. That functionality is based on semantically enhanced search. 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. [3]

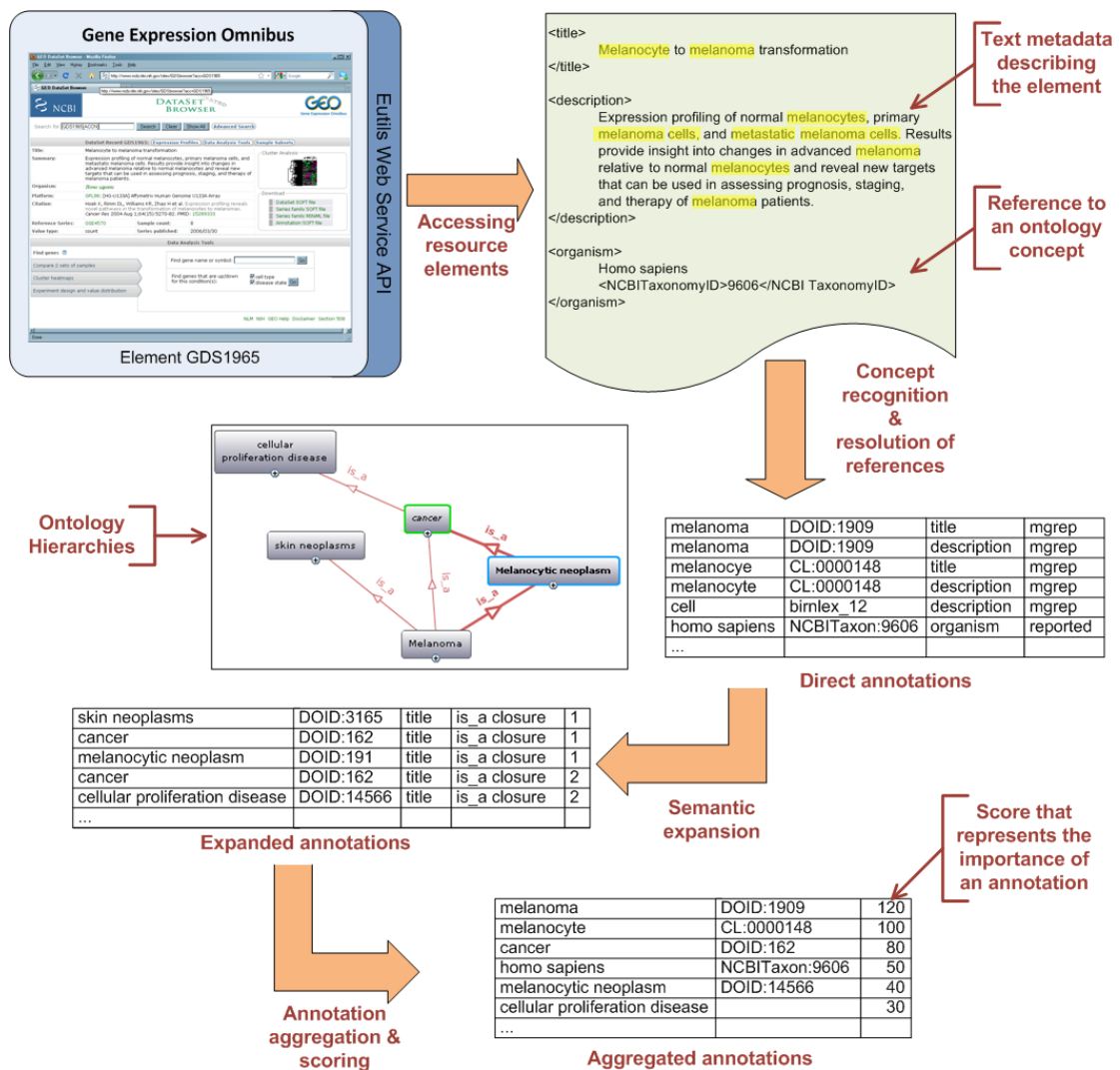


Figure 1: NCBO Resource Index workflow

Population of OBR and OBS tables using properties files and shell script is described in document [Technical_Instructions_for_Configuring_OBR_Workflow](#). [18]

2.1.1 OBR Resources

OBR Resources is the workflow executions in the biomedical resources.

The figure shows below is the population of the OBR tables. [18]

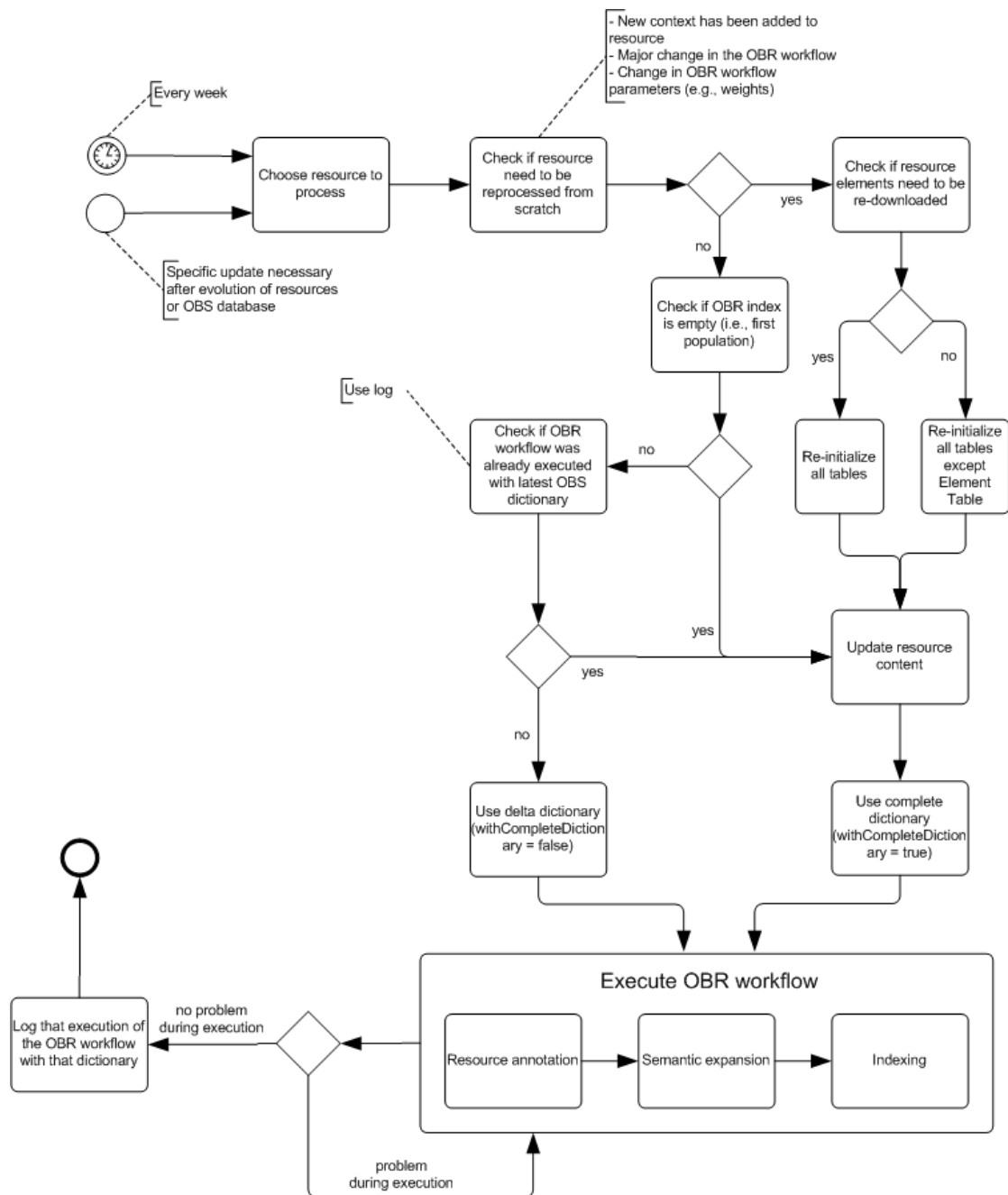


Figure 2: Population of the OBR tables

2.1.2 OBS Database

The OBS database contains the data used by the OBA/OBR workflow. It contains the concepts and terms used for direct annotations as well as the relations between concepts used during the semantic expansion step. [21]

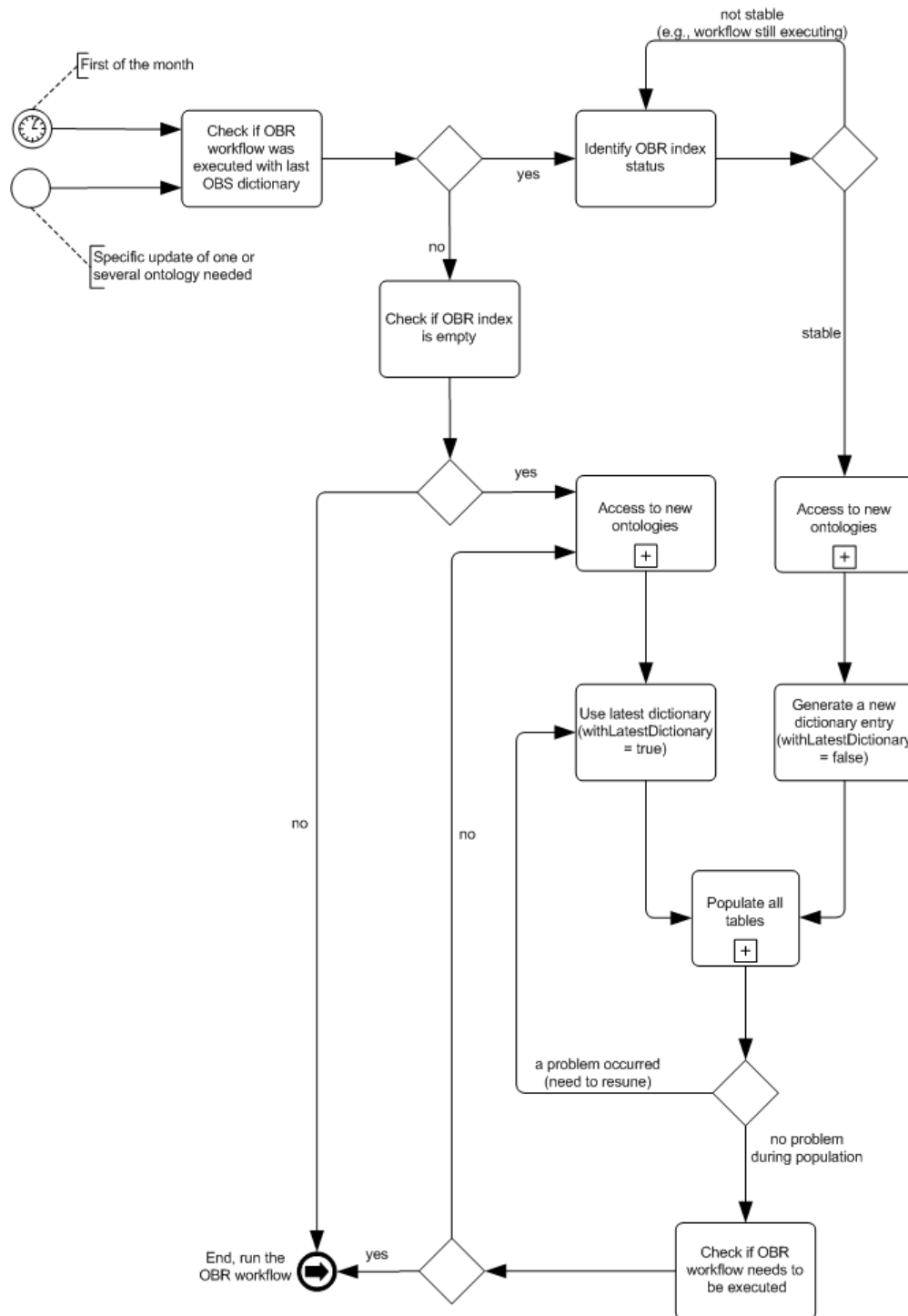


Figure 3: Population of the OBS database

2.2 HDFS and DistributedCache

2.2.1 HDFS Features

HDFS is designed to support large data sets, same as the programs. Write data only once, read data once or several times for a better streaming read speed demand. The feature is for batch processing, rather than for the user-interactive. Hence, the application should access the data set in a streaming way. It is not a typical operation in a conventional file system because HDFS is focused on data throughput, rather than data access response time. So it will not be necessary enforced rigidly demand by POSIX. [9] [11] [17]

Due to the rule of accessing HDFS files is Write-Once Read-Many, and HDFS file is strict compliance with only one write operation in any time, once a file is created and written, the file will not be modified after the closure. This rule can simplify the data consistency and make high throughput data access.

HDFS can support hundreds of nodes in a cluster of millions of files and large data sets. A typical HDFS file may have the size of GBs to TBs. It should reliably store a large number of files in a large number of nodes, by the form of a block sequence for each file. Except for the last block, other blocks have a same size. One should be noted is that the block size and replication number can be specified for each file in the file created or later.

Each block can locate in different data nodes. The strategy of how to choice the replication placement seriously affects the reliability and performance of HDFS, because HDFS tries to satisfy a reading operation from the nearest replication. If there is replication on the same rack with the read node, it will directly read it. The HDFS can read replications in the cluster across racks. Nodes communication in two different racks will go through the switch, in most cases, bandwidth is worse than transfer in the same rack. [11]

Actually in the ideal situation, the data blocks uniformly distributed in every node can make clients loading evenly. A great influence on the performance will occur if the unbalanced load. It will ultimately affect the efficiency of the program.

Once a client request to create a file, it will not immediately request to the NameNode. The fact is that the HDFS client cached data in the local file at the first, the application will write to the temporary local file until the accumulative size of local files is reached to the HDFS block size, the client contact NameNode and then refresh it. [11]

The application requires streaming write files. If the client is written directly to the remote file system, without a local buffer, will have a considerable impact on the speed and network throughput. When the client needs to write data to the HDFS,

like said before, the data first write to a local file, assuming that the HDFS replication factor is 3, when the accumulation of a local file up to the block size, the client requests the list of DataNodes. This list describes the DataNodes which take over the blocks copy. The client is to refresh the data block to a DataNode. The first DataNode start to receive data to a small position (4kb), write them to the local disk, and transfer to the list in the second DataNode, so turn to the second data node, the second DataNode transfer data to the third. A DataNode can accept data from the former DataNode, but also the data flow transfer to the next DataNode, therefore, the data flowing style is passed from one data node to the next. [17]

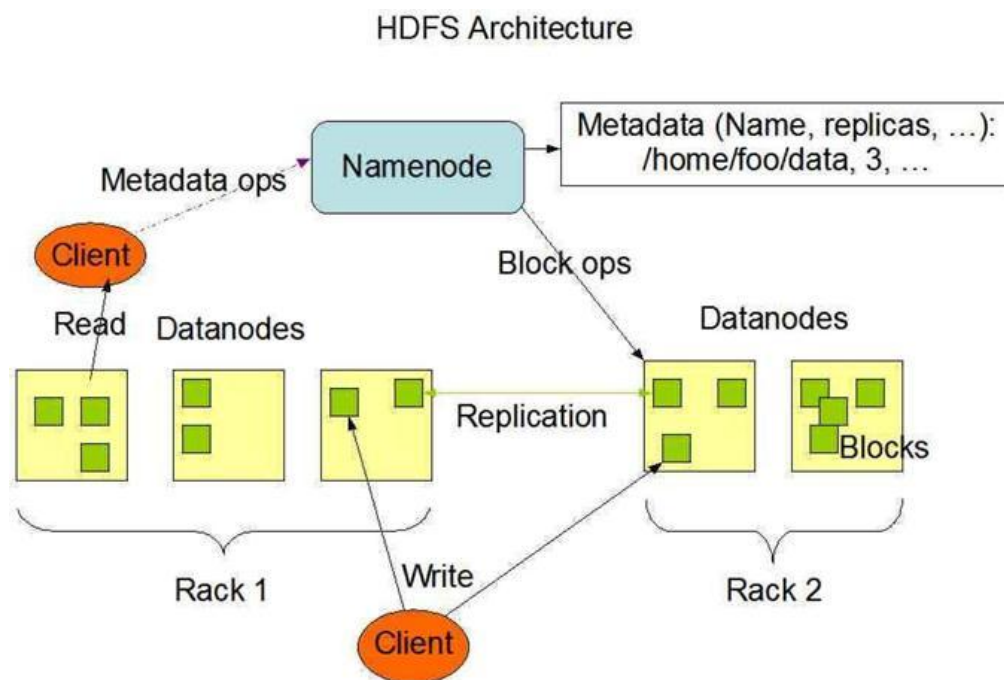


Figure 4: HDFS Architecture

2.2.2 DistributedCache in Brinf

DistributedCache can download the file to the local machine from HDFS. Using DistributedCache to copy files to each node could be a potential improvement in the performance. [6]

The use of the DistributedCache is in the Appendix.

Like the Hadoop's slogan, "Moving Computation is Cheaper than Moving Data". The main purpose is to reduce congestion caused by the data transmit in the Map phase, and as much as possible to calculate the arrangement at this stage. The final copy of the data process postponed to the Reduce phase.

However, in our Map side join, it will not have the data transmitted during the data processing, so the distributed cache will not be reflected in its advantage.

We got some instances to prove our viewpoint in Chapter 4.

2.3 Map and Reduce

MapReduce is a distributed programming model for large-scale data processing, which has been proposed by Google in OSDI '04 (Operating Systems Design and Implementation).

It abstracts the large-scale distributed data processing to one platform and two user-defined functions, Map and Reduce. The Map function is responsible for processing sub-data sets and produce intermediate results, and the Reduce function is responsible for reduction of the intermediate results and generates the final results of processing. The platform is responsible for scheduling, fault tolerance and data management. [10]

2.3.1 MapReduce Programming Model

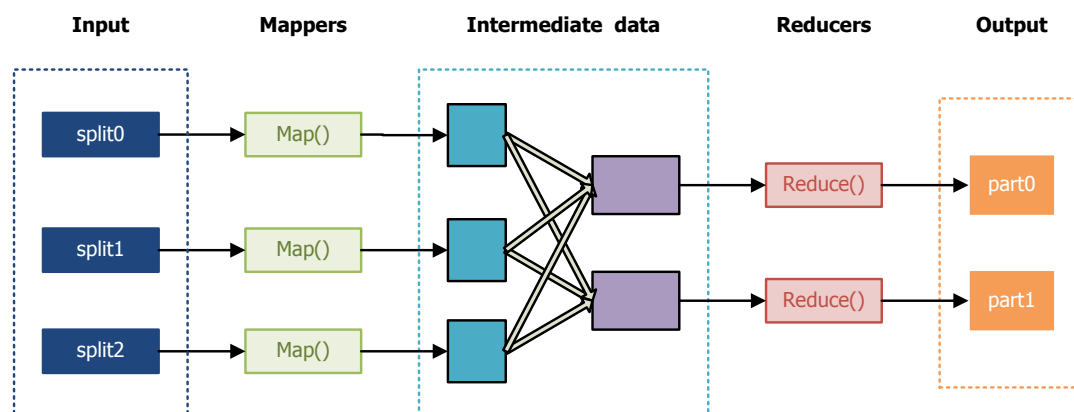


Figure 5: A general flow diagram of MapReduce

From the figure we can see the core of the model is function map and reduce. There are both user defined, which can be customized for their own needs. Transforming the input $\langle \text{key}, \text{value} \rangle$ to output, which is one or a group of $\langle \text{key}, \text{value} \rangle$.

In Map phase, MapReduce Framework segmented the input data to the fixed-size splits. Then each of the split further decomposed as key/value pairs. After that, Hadoop established a map task for every split, which is Mapper we called, to execute the user-defined map function. The function used $\langle K1, V1 \rangle$ in split as input, calculated the intermediate result $\langle K2, V2 \rangle$. Sorted the intermediate results by using $K2$, and generated a new list which the value had the same key, $\langle K2, \text{list}(V2) \rangle$. Finally,

these tuples are grouped by the range of key value, and distributed to the different Reduce tasks.

In Reduce phase, Reducer sorted the received data, which came from the different Mappers, then invoked the user defined reduce function, to process the input $\langle K2, \text{list}(V2) \rangle$. Then calculated and generated $\langle K3, V3 \rangle$. Finally, output them and stored in HDFS.

2.3.2 MapReduce Behaviors in A Cluster

A MapReduce job includes one JobTracker and several TaskTrackers. The JobTracker as the governor of all TaskTrackers, which responsible for scheduling and management, running on the master node as usual. JobTracker assigned the Mappers and Reducers to the idle TaskTrackers. The TaskTrackers performed these tasks so they should be run on the DataNode. DataNode stored the data also calculated the data. If any of the TaskTracker failure, the JobTracker assigned another TaskTracker to reprocess the data.

One of the most important features of MapReduce is the local computing. One DataNode is the node of data storing also the data processing. The framework of MapReduce made the greatest efforts to store and process data on the same node. This approach can effectively reduce the transmission of data in the network, in order to reduce the demand of the network. " Moving Computation is Cheaper than Moving Data ". [17]

Because of this, size of split is equal to or less than the size of HDFS, in order to prevent one split stored crossed the two nodes.

Another most important feature of MapReduce is the output of the Mappers was stored in the local disks, rather than HDFS, under normal circumstances. Because in the model, the output of mapper is the intermediate result, and will be deleted after the task finished. It will lead to loss of performance if the output stored in HDFS, due to the backup mechanism of HDFS.

2.4 Hadoop Job Scheduling

The JobTracker is running on the master node, and the TaskTrackers are running on the slave nodes.

The flow chart below shows the whole Hadoop Job Scheduling. [16]

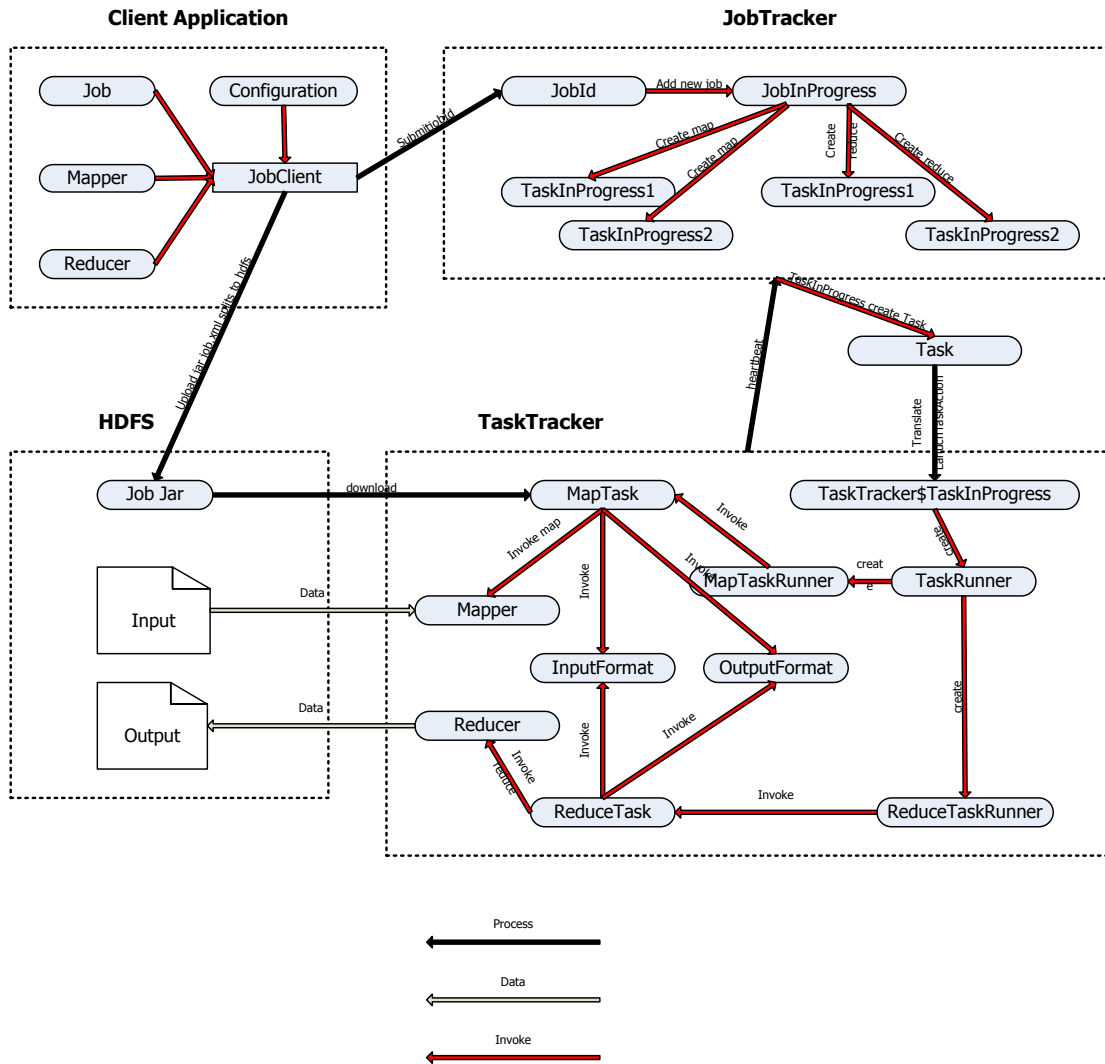


Figure 6: Job scheduling flow chart

2.5 HashMap

Hash map is Hash table based implementation of the Map interface. This implementation provides all of the optional map operations, and permits null values and the null key. (The HashMap class is roughly equivalent to Hashtable, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time. [7]

This implementation provides constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. Iteration over collection views requires time proportional to the "capacity" of the HashMap instance (the number of buckets) plus its size (the

number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important. [7]

Due to the size of relation set is small enough, we can store it in each single node's memory, however we cannot just copy the relation set into memory because it is low efficiency to search based on text.

Depending on the characteristics of data and architecture, we keep it as key/value pairs. In here, we choosing *HashMap* as the carrier of relation file is based on its property.

As far as we know, an instance of hash map has two parameters that affect its performance: *initial capacity* and *load factor* . To avoid rehash, we set *initial capacity* = 25000000 and *load factor* = 0.99.

```
//build a Hash Map
private Map<Integer,Integer> obs_relation =
    new HashMap<Integer,Integer> (25000000, 0.99f);
//...
```

So, in our application the hash map instance will not rehash until the capacity is full, in here is $25000000 * 0.99 = 24750000$. And in our relation file, the number of row is 24153638.

Chapter 3. Related Work

3.1 Classic Join on Hadoop

In traditional databases such as MySQL, the join-operation is very common and very time consuming. Similarly, it is common and time consuming in Hadoop.

3.1.1 Default Join

Default join, or map reduce join, or simply said reduce join, is a two steps join which can work well in most situations. It implements the MapReduce spirit very well.

For example, given two tables, R and S. In here, R is implied the relation table, and S is the table for segmentation. In the map phase, Mappers read two types of table and adding a custom tag, in order to distinguish the two types of table. The main task is tagging the data files. In reduce phase, different Reducers got the <key, value> pairs which had corresponding tag. Then, data join (Cartesian product) for those have the same key. Join operation is in the reduce phase. [4] [10]

The default join can work well for most situations. One of the exceptions is that both R and S tables are huge. It will cause a plenty of data there are lots of data transmission over network from Mappers to Reducers. The network transfer will be the bottleneck in this case.

3.1.2 Map Side Join

Map side join is one step join, which removed eliminating the reduce phase and thus eliminating the transfer of data over the network between map phase and reduce phase. Map join aims to use only the map phase so no data will be transferred on network. It cannot be widely used based on the size of the R table, which is limited by the size of memory.

We gave more details in Chapter 4.

3.1.3 Pig and Hive Join

Pig and hive are the frameworks which build on the top of Hadoop. The Java codes are not required. Instead, users should write declarative (for Hive) or procedural (for Pig) queries to perform tasks.

However, neither of these tools has addressed the problem of join. Pig has implemented fragment-duplicate join (known as Map side join), and also skew join that can handle skewed tables. The user may need to give some hints to the

compiler, indicating what the join method the system should be used. This is not a good way to handle the problem because the user may don't know what is the Map Side Join. Furthermore, the user may give a wrong hint which could hurt the performance. [4]

3.1.4 Other Joins

Semi Join

In Semi join, only the joined data will transmission in the network. It actually is the map side join, which the R table is huge. Hence extracted the R table to a small and can be totally stored in the memory. [19]

JDBM-based map join

JDBM is a transactional persistence engine for Java. It aims to be a fast and simple persistence engine, can be used to store a mix of objects and BLOBs, and all updates are done in a transitionally safe manner. As the name suggests, JDBM-based map join will utilize JDBM to store the hash table so that memory wouldn't be an issue. [20]

The question to ask about JDBM-based map join is the efficiency of look-ups in a hash table that could reside on the disk. Furthermore, introducing a between-layer will decrease the control over the performance of such a join plan. [4]

3.2 Previous Work

The previous paper *Methods for Large-scale Semantic Expansion on Hadoop Architecture* is to scale out the semantic annotation data of NCBO Resource Index that they have implemented on MySQL server on single machine. [2] [3]

And in order to improve the performance of the computation the author implemented some algorithms for data-parallel computing and data combining.

In that thesis, several tests had been implemented. Included MapReduce Join and Pig Join in HDFS and HBase, also the normal left join in MySQL.

In the thesis he said, the experimental results provide insights that are about the MapReduce platform and comparisons of particular join algorithms on the Hadoop platform. [2]

Chapter 4. Implementation

4.1 Datasets and Cluster Environment

4.1.1 Data Sets Used

Original files from NCBO, they have the data size from MBs to GBs. In the NCBO Resource Index, they include two types of table for join, OBS Database and OBR Resources. In this paper, we note these two types are type R table and type S table. R is for Relation and S is Split.

NCBO Resource Index	Table Type	File Name	Number of Tuples	File Size
OBS Database	R	obs_relation.txt	24 M	658 MB
OBR Resource	S	obr_wp_annotation.txt	54 K	1.79 MB
OBR Resource	S	obr_ct_annotation.txt	165 M	6,073 MB
OBR Resource	S	obr_pm_annotation.txt	442 M	17.4 GB

Table 1: Original files from NCBO Resources Index

In order to facilitate the experiment and get more accurate results, we generated the following data based on the original NCBO Resource Index files.

Generation method is in the Appendix.

obs_relation.25%.txt	6,038,410 lines	157 MB	25% of obs_relation.txt
obs_relation.50%.txt	12,076,819 lines	316 MB	50% of obs_relation.txt
obs_relation.75%.txt	18,115,229 lines	485 MB	75% of obs_relation.txt
obs_relation.txt	24,153,638 lines	658 MB	100% of obs_relation.txt

Table 2: Generated R tables

obr.1k.txt	1,000 lines	31.3 KB	1k tuples
obr.10k.txt	10,000 lines	328.58 KB	10 k tuples
obr.100k.txt	100,000 lines	3.28 MB	100 k tuples
obr.1M.txt	1,000,000 lines	33.99 MB	1m tuples
obr.10M.txt	10,000,000 lines	339.95 MB	10m tuples
obr.100M.txt	100,000,000 lines	3.32 GB	100m tuples
obr.1B.txt	1,000,000,000 lines	33.2 GB	1b tuples

Table 3: Generated S Tables by Exponential Growth

obr.100M.txt	100,000,000 lines	3.32 GB	100m tuples
obr.200M.txt	200,000,000 lines	6.64 GB	200m tuples
obr.300M.txt	300,000,000 lines	9.96 GB	300m tuples
obr.400M.txt	400,000,000 lines	13.28 GB	400m tuples
obr.500M.txt	500,000,000 lines	16.6 GB	500m tuples
obr.600M.txt	600,000,000 lines	19.92 GB	600m tuples
obr.700M.txt	700,000,000 lines	23.24 GB	700m tuples
obr.800M.txt	800,000,000 lines	26.56 GB	800m tuples
obr.900M.txt	900,000,000 lines	29.88 GB	900m tuples
obr.1B.txt	1,000,000,000 lines	33.2 GB	1b tuples

Table 4: Generated S Tables by Linear Growth

4.1.2 Cluster Specification

Experiment was performed on a cluster of eleven nodes. One node is the Master node and other ten nodes are slave nodes.

Each node was a HP server with one six-core AMD Phenom™ II X6 1090T Processor and a 16GB ECC DDR-2 memory chip.

The storage of each node is 1TB SATA-2 drive running at 7200rpm.

Nodes are connected by an isolated HP Pro Curve 2650 100BaseTx-FD switch.

Each node is running Linux version 2.6.18 - 194.32.1.el5.centos.plus (gcc version 4.1.2 20080704 (Red Hat 4.1.2-48)) and Java 1.6.0_23 for 64 bit.

The assignment of 11 nodes in the cluster is in the below.

haisen1: NameNode, JobTracker.

From haisen2 to haisen11: DataNode, TaskTracker.

4.1.3 Hadoop Configurations

The HDFS typical block size is 64MB. It could be modified up to 1GB. We found that HDFS does not support blocks to 2GB or greater, it had been confirmed as a bug in Hadoop 0.20.203.0 or previous versions. The size of the heap memory is 1.74 GB and the maximum use of memory for child JVMs is 10,240 MB. Hadoop version in our cluster is 0.20.203.0.

Configuration direction is in the Appendix.

4.2 Algorithm of Map Side Join

The NCBO Resource Index currently includes 22 different data resources and comprising over 3.5 million data elements in 16.4 billion annotations stored in a 1.5 TB MySQL database. [2]

And we can simply use SQL Syntax to get left outer join result. It is like this:

```
SELECT
  obs_relation_mem.parent_concept_id AS concept_id
  obs_relation_mem.level AS parent_level,
  obr_ct_annotation_cut.concept_id AS child_concept_id,
  obr_ct_annotation_cut.element_id AS element_id
FROM
  ncbo_test_1.obr_ct_annotation_cut
LEFT OUTER JOIN
  ncbo_test_1.obs_relation_mem
ON
  obr_ct_annotation_cut.concept_id = obs_relation_mem.concept_id
WHERE
  obs_relation_mem.parent_concept_id IS NOT NULL;
```

With the increasing of date, it will not be computed in an acceptable time on a single machine. When scaling from 22 resources to 100 or more, the limitation of one single machine will be appeared. A distributed processing method should be implemented. Hence we are going to use a scale out method to solve this problem. [2] [3] [12]

As far as we know, left join operation can be implemented in several ways depending on the characteristics of data and architecture. One of such scenarios consists of one main large dataset and one or more small datasets.

Map side join in our experiment aims to use only the map phase for a better performance than map-reduce join. It is based on the experience that to integrate the reduce phase function in map phase so there will no data transferred on the network between the map and the reduce side.

In our algorithm, each map task is initialized for every split parts in S table, then the job could include a plenty of map tasks, depends on the amount of split of S table.

One single task component is from task lunched to task finished, includes four phases.

1. Map task set up,
2. Reading relation set from HDFS and build a HashMap in memory,
3. Reading split table line by line and seeking in the HashMap, output results,

4. Map task clean up.

In here, we keep the R table, or *obs_relation_mem*, in memory by using hash map, which has the best searching efficiency.

Each map task will follow the steps bellow.

1. The algorithm divided the *obs_relation_mem* as two part. One part is related to the *obr_ct_annotation_cut*, which is *concept_id* in here. Another part is not related, in there are *parent_concept_id* and *level*.
2. Then we stored the irrelevant part as key/value pairs in the hash map in memory. In here, we using *concept_id* as the keys and *parent_concept_id + level* as the corresponding values.
3. The initialization of Hash Map
4. After the initialization of Hash Map, a Map task reads one line of *Annotation_set_2* or, *obr_ct_annotation_cut*, then get the *obr_ct_annotation_cut.concept_id* which is equal with *obs_relation_mem.concept_id*. It can probe the value in the Hash Map, combine the corresponding values of *obr_ct_annotation_cut* with the same key.
5. Then, the task can figure out all data in *obr_ct_annotation_cut* one by one. Assemble useful fields and get output.

4.3 Control the Number of Mapper

We can use *mapred.tasktracker.map.tasks.maximum* to control the maximum map task slots on one node.

This value is depends on the hardware, such as the number of CPU and the size of memory, also the demands of the map task itself.

In our experiment, we set the value is 2, it means one node can lunch two synchronous map task, which is following the configuration of the previous thesis.

Based on 10 nodes we have, and each node can lunch 2 tasks. Theoretically optimal is 20 as the maximum map tasks in the cluster. We can use *mapred.map.tasks* to set this value. Therefore we have 20 map task slots in the cluster.

However, in some cases, the number of split block will be greater than the slots in a cluster. Actually, there have a configuration method *JobConf.setNumMapTasks(n)* in program to control the amount of map task. But it cannot work well because it is just a hint to the Hadoop framework. The fact is the amount of map task is affected by number of input split.

The default input split is 64 MB, same with block size of HDFS. It is not applicable if input data was very big. In that case, hundreds of thousands of map tasks will cause

the network of cluster congestion and to bear much pressure to Job Tracker. And each split needs a map task to process, but the task initialization time is long. So minimized the amount of map task is our primary task.

In another aspect, we also need to consider the best split size for the map. We will get some test next.

To find the way to control the amount of map task, we inspected the Hadoop API [6], in *FileInputFormat.java*, source code of Hadoop, we found that,

```
//..  
splitSize = Math.max(minSize, Math.min(goalSize, blockSize));  
//..
```

and we found that,

```
//..  
goalSize = totalSize / (numSplits == 0 ? 1 : numSplits);  
minSize = Math.max(job.getLong("mapred.min.split.size", 1),  
                  minSplitSize);  
//..
```

It means:

```
goalSize = totalSize / mapred.map.tasks  
minSize = max (mapred.min.split.size, minSplitSize)  
splitSize = max (minSize, min(goalSize, dfs.block.size))
```

Hence, there have three ways to control the input data size for map task.

1. Change the HDFS block size.
2. Change the MapReduce minimal split size.
3. Overwrite a custom MapReduce FileInputFormat to split the input data.

In this paper, we mainly focus on the first two methods, and we will find out which is more suitable for our algorithm.

4.4 Performance Experiments

4.4.1 Block Size and Split Size Test

The default *dfs.block.size* is 64 MB, also same with the default *mapred.min.split.size*.

Generally, the amount of map task is equals the number of splits. And the number of split is determined by *dfs.block.size* and *mapred.min.split.size* as we talked before.

In here, we test different block size and split size from 64 MB to 1024 MB.

The graph showing below is the average task time components in each different split size. In order to get a more meticulous result, we use 3 slots in 3 nodes in the test.

The data sets used are obs_relation.txt and obr_ct_annotation.txt

The following figure, we tested the composition of the different split size of the average map task execution.

Each task includes three parts, task set up and clean up, read R table and initialized to the HashMap in memory, left out join.

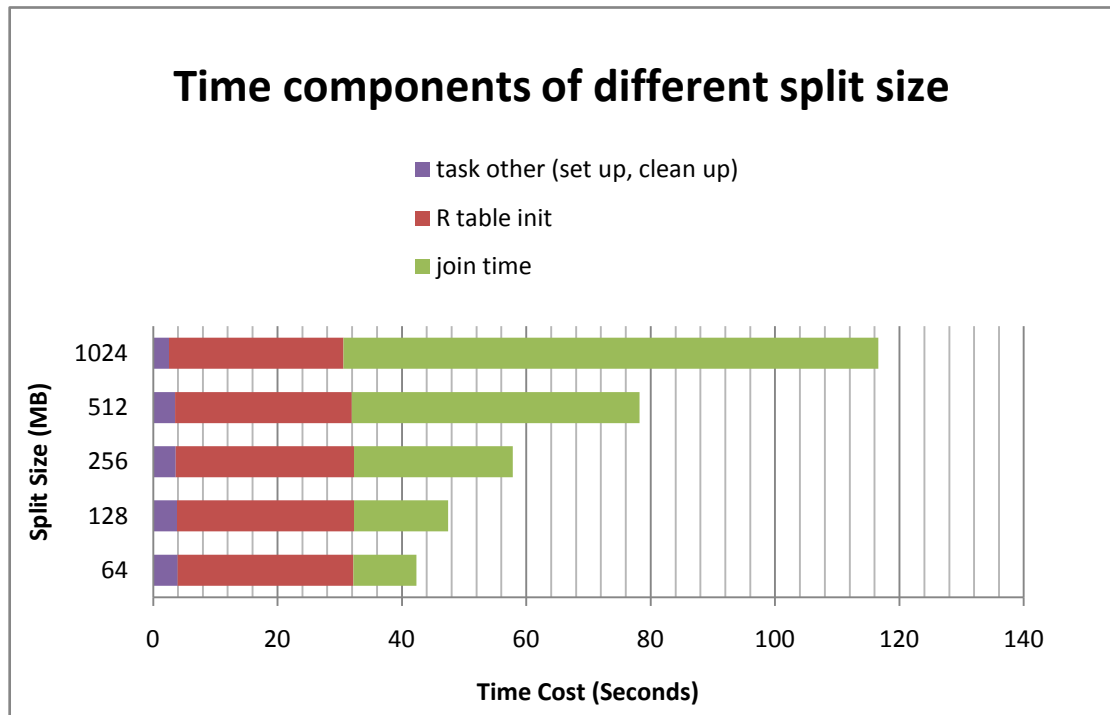


Figure 7: Time components of different split size

Seen from the figure, the proportion of join increased with the split size. And while the split size is 1024 MB has the best capability.

Split size (MB)	Task amount	Task cost(s)	Task other(s)	R table init(s)	Join time(s)
64	95	42.3	3.9	28.3	10.1
128	48	47.4	3.8	28.5	15.1
256	24	57.8	3.6	28.7	25.5
512	12	78.2	3.5	28.4	46.3
1024	6	116.6	2.5	28.1	86
2048	3	not support	not support	not support	not support

Table 5: Time cost of different split size

Assume to process 1 GB S table for join as one of our job. With the different split size, here have a corresponded amount task.

Split size (MB)	Task cost (s)	Task amount	Job cost (s)
64	42.3	16	676.8
128	47.4	8	379.2
256	57.8	4	231.2
512	78.2	2	156.4
1024	116.6	1	116.6

Table 6: Time cost of 1GB S Table

The figure shows below is the result of completed the 1GB join.

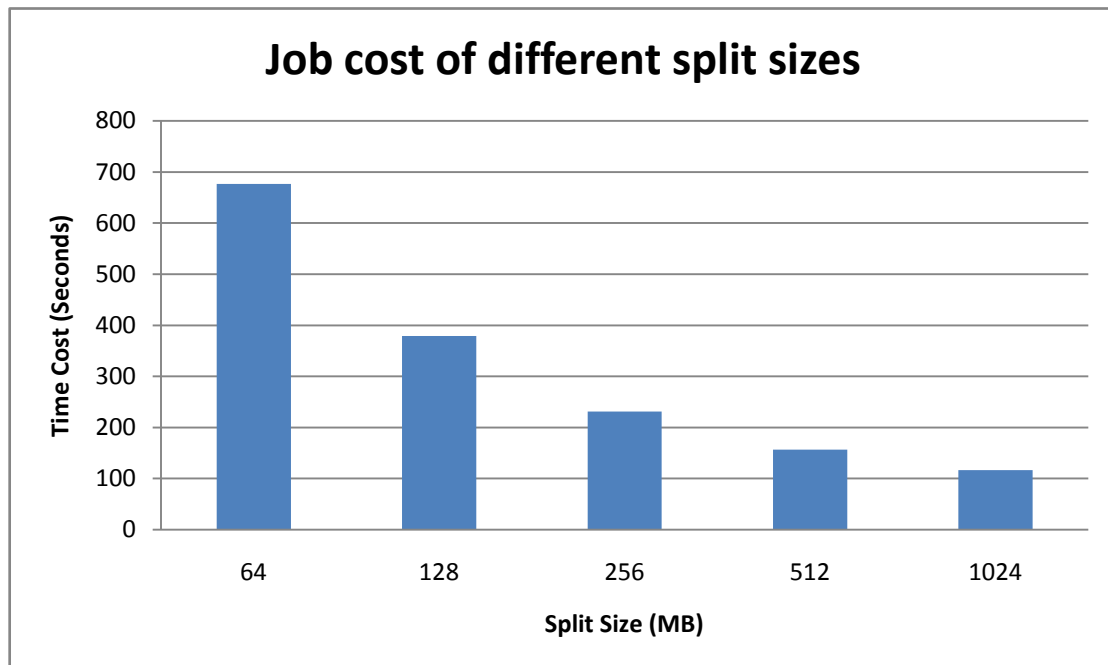


Figure 8: Job cost of different split size

In the figure, we see when the split size is 1024 MB, it has the fastest speed.

The figure below is the join speed (within the join phase) in the different size.

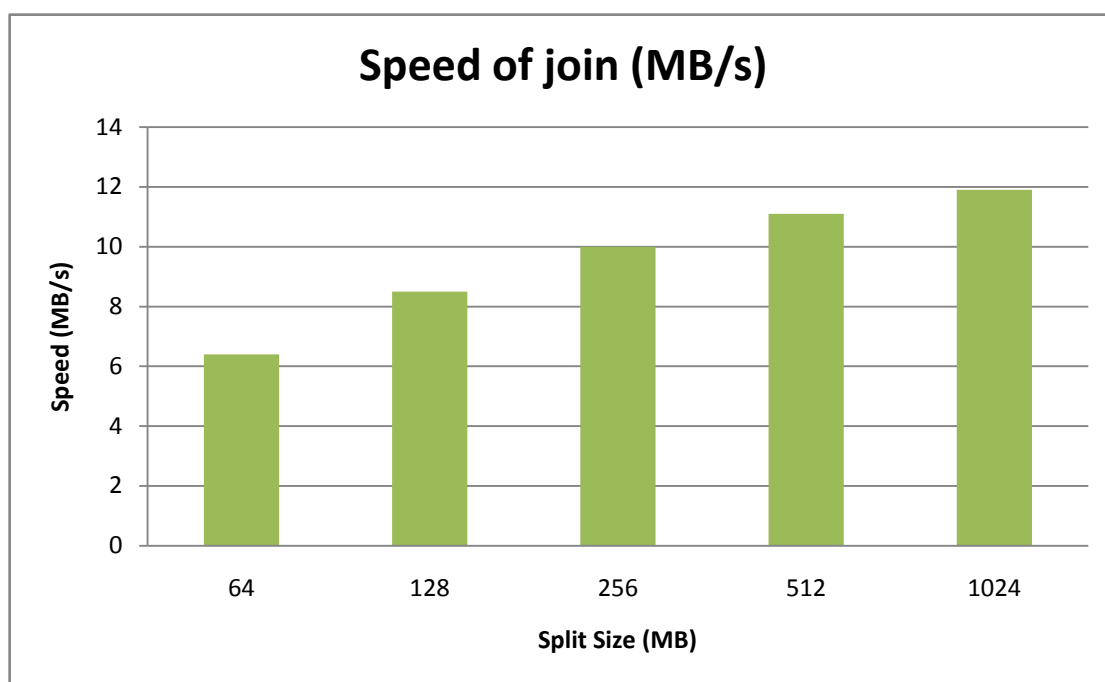


Figure 9: Speed of Join

It can be seen that when the split size is 1024 MB has the best performance.

From the above two aspects we both find that 1024 MB as the split size will have the best efficiency of the job.

Therefore, we could to set *mapred.min.split.size* 1024 MB to achieve our goal.

In here, *dfs.block.size* theoretically can also be set as the same purpose, but it may cause a performance problem, we will explain in the next.

4.4.2 Replication Number Test

The default number of replication is 3 in normal Hadoop cluster.

We found that in some cases, if a file is accessed by many other nodes simultaneously and the number of replication is less (for example, the replication factor is 2 in here), it will occur an additional replication event, and takes extra time to the copy operations.

We also found that in some cases, while the *dfs.block.size* is increasing or the *dfs.replication* is decreasing, it will both have a greater likelihood of occurrence of a sharp decline in performance when read a file in HDFS.

The reason is the file blocks are not very evenly distributed in nodes of a cluster.

From the figure shows below, we can see when the replication factor is 2 and 3, the job cost will have a large difference.

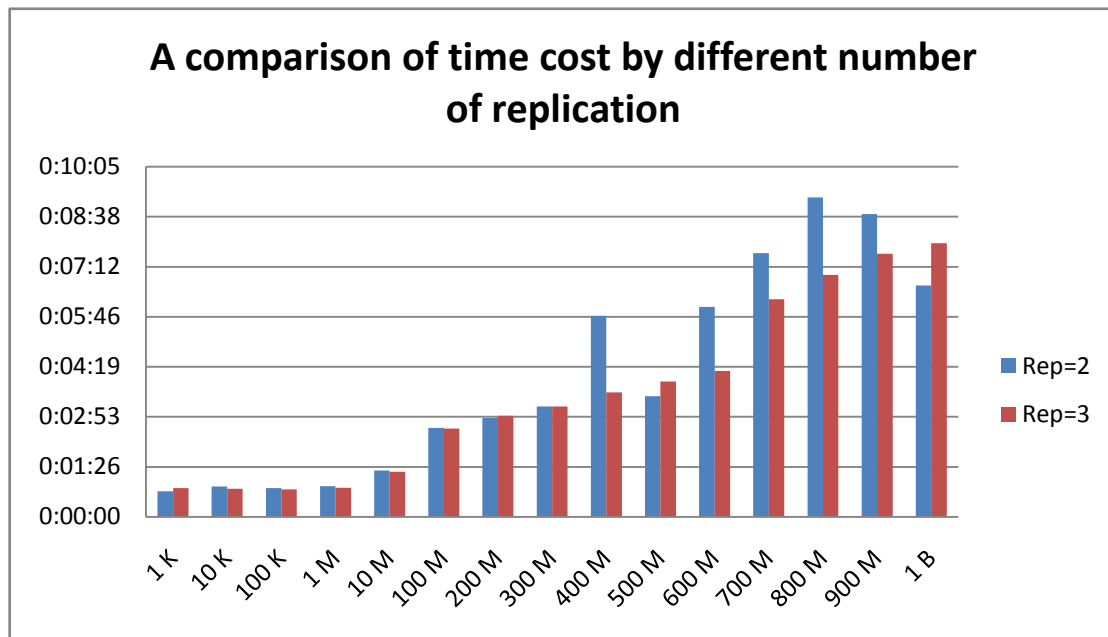


Figure 10: A comparison of time cost by different number of replication

In general, more replications will cause the more time cost in data copy operations and more transmission in the network. As the results in 500M and 1B, it shows clearly the Rep = 2 time cost is faster than Rep = 3.

However, due to the presence of some blocks which belong to one file may too concentrated in one DataNode, it will cause performance degradation while the other DataNode read the file at the same time.

In the figure, the reason of unstable for Rep = 2 in 400M, 600M, 700M, 800M, 900M is because the file blocks heavily concentrated in one node's disk.

An example, the S table file obr.400M.txt (13.28 GB) is stored in the HDFS and has 54 * 2 = 108 blocks as the whole when the replication factor is 2, the block size is 256 MB.

We found that it has 54 blocks stored in the node 152.94.1.122 in totally 108 blocks.

That is one case of the uneven distribution and can cause the bad performance.




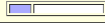
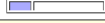
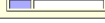
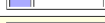

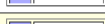

The figure 11 shows the uneven distribution phenomenon in a cluster.

NameNode 'haisen1.ux.uis.no:54310'

Started: Tue Apr 17 23:25:02 CEST 2012
Version: 0.20.203.0, r1099333
Compiled: Wed May 4 07:57:50 PDT 2011 by oom
Upgrades: There are no upgrades in progress.

[Browse the filesystem](#)
[Namenode Logs](#)
[Go back to DFS home](#)

Live Datanodes : 10

Node	Last Contact	Admin State	Configured Capacity (GB)	Used (GB)	Non DFS Used (GB)	Remaining (GB)	Used (%)	Used (%)	Remaining (%)	Blocks
haisen2	2	In Service	888.95	138.68	119.14	631.13	15.6		71	587
haisen3	2	In Service	888.95	67.89	436.85	384.2	7.64		43.22	324
haisen10	2	In Service	888.95	62.54	417.16	409.24	7.04		46.04	296
haisen6	2	In Service	888.95	59.87	394.51	434.57	6.73		48.89	280
haisen4	2	In Service	888.95	59.02	358.92	471.01	6.64		52.98	287
haisen5	2	In Service	888.95	58.22	404.3	426.43	6.55		47.97	277
haisen11	2	In Service	888.95	57.6	103.25	728.1	6.48		81.91	273
haisen9	2	In Service	888.95	55.87	386.19	446.89	6.29		50.27	271
haisen8	2	In Service	888.95	54.96	390.38	443.61	6.18		49.9	272
haisen7	2	In Service	888.95	53.81	443.76	391.37	6.05		44.03	258

This is Apache Hadoop release 0.20.203.0

Figure 11: The cluster block distribution

In order to increase the number of blocks and to avoid performance degradation, we set *dfs.block.size* = 64 MB, and *dfs.replication* = 3 at the same time. Another optional method is to use the load balancing tool to eliminate the problem.

To avoid the additional replication event we can set the R table replication number is 10.

Configuration method is in the Appendix.

4.4.3 Relation Table Size Test

It is the time cost for load relation file. Include read file from HDFS and build as a HashMap in the memory for the split files. (It is the initialization time previously mentioned)

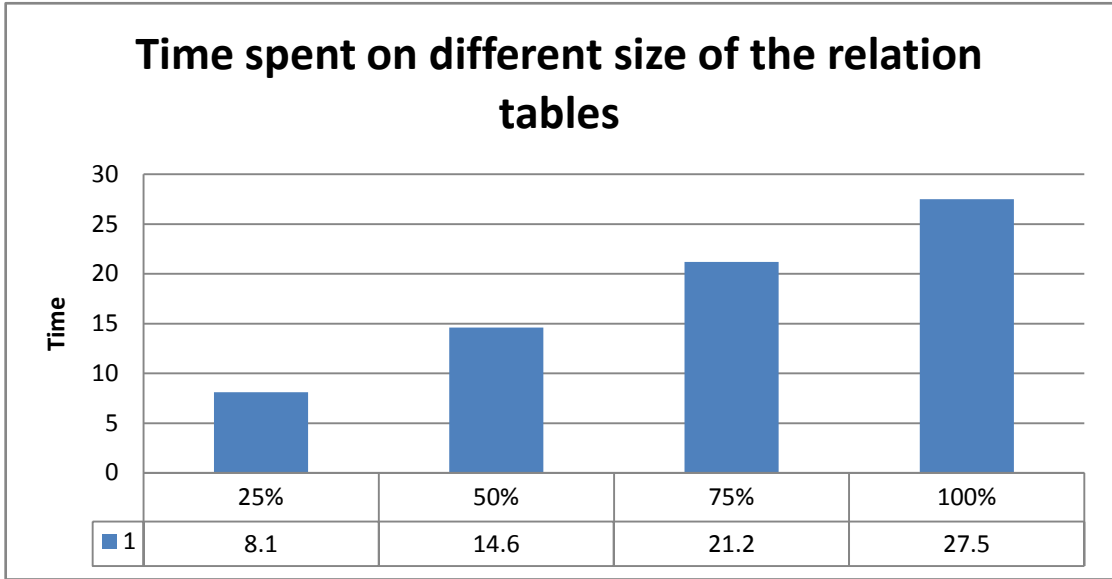


Figure 12: Time cost on different size of the relation table

We just have one relation table and the NCBO should have different size of relation table, so we test different size of relation files. The test results show that it almost linearly proportional to table size.

In another word, with different size of relation table, it will cost proportionate initialization time.

4.4.4 Split Table Size Test

Exponential distribution is from 1k to 1b.

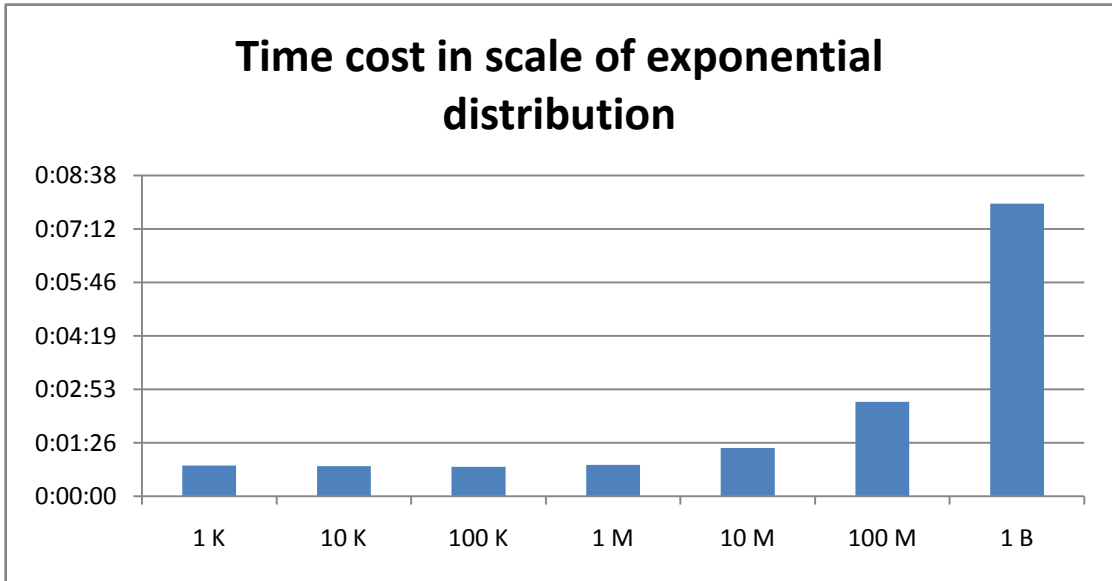


Figure 13: Time cost in scale of exponential distribution

As we set 1GB in *mapred.min.split.size*, the number of the task is the same 1 from 1K to 10 M. With a long initialization time, time cost seems nearly from 1K to 1M.

Linear distribution is from 100m to 1b.

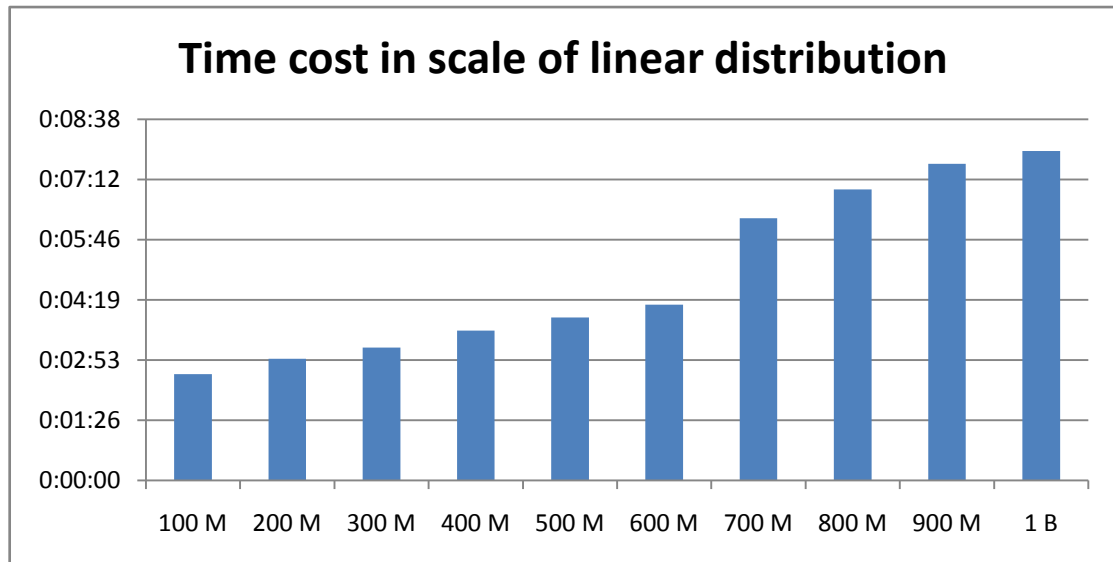


Figure 14: Time cost in scale of linear distribution

From the following table, the task number is 20 when the number of tuples is 600M, and the task number is 24 when the number of tuples is 700M.

Because we have 20 slots in the cluster (the cluster can lunch 20 map task at the same time. Based on 10 nodes multiply 2 map task per node).

When the number of task is 24, it will launch 20 map tasks at the first time, and other 4 tasks will be waited in the queue. It is due to the job scheduling mechanism of Hadoop. After one previously task finished and then launch the new task. It will have a long initialization time, so it means extra time gap in the figure.

Number of tuples	File size	Task amount	Time cost (s)
1 K	31.3 KB	1	49
10 K	328.58 KB	1	49
100 K	3.28 MB	1	47
1 M	33.99 MB	1	51
10 M	339.95 MB	1	78
100 M	3.32 GB	4	152
200 M	6.64 GB	7	175

300 M	9.96 GB	10	190
400 M	13.28 GB	14	215
500 M	16.6 GB	17	234
600 M	19.92 GB	20	22
700 M	23.24 GB	24	376
800 M	26.56 GB	27	418
900 M	29.88 GB	30	454
1 B	33.2 GB	34	473

Table 7: Time cost by different size of split table

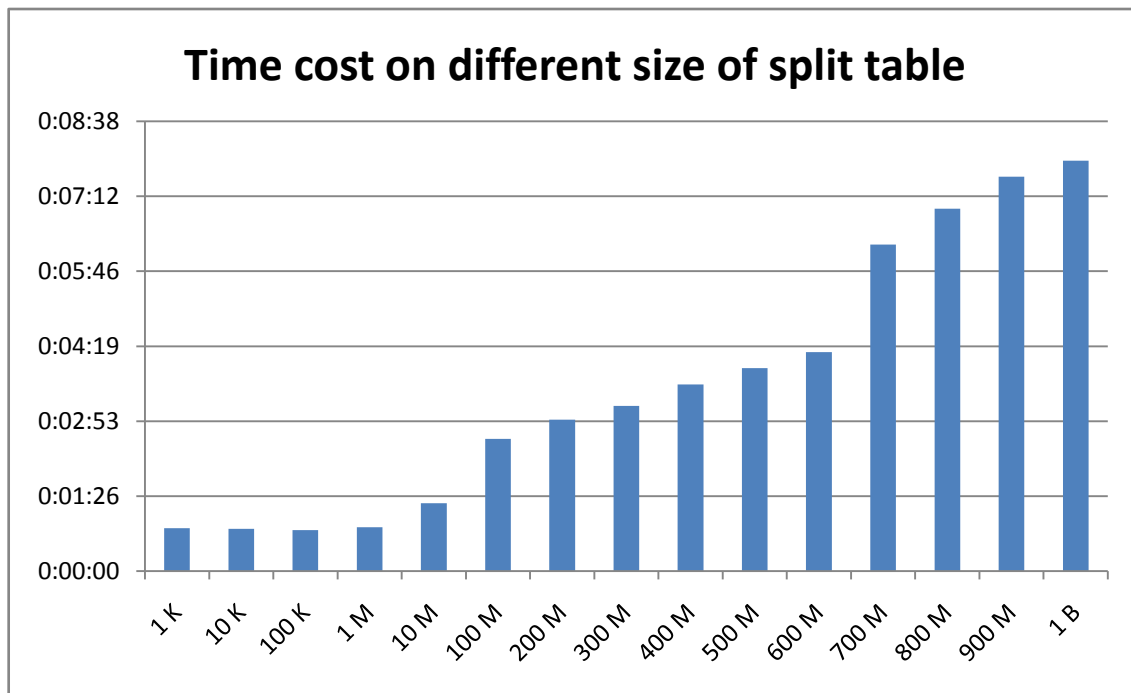


Figure 15: Time cost by different size of split table

4.4.5 DistributedCache Test

Whether or not to use the DistributedCache has been a place of concern in this paper. Because of the advantage of DistributedCache is referred by a lot of references. Therefore, in our experiments also tested it.

Usage is in the Appendix.

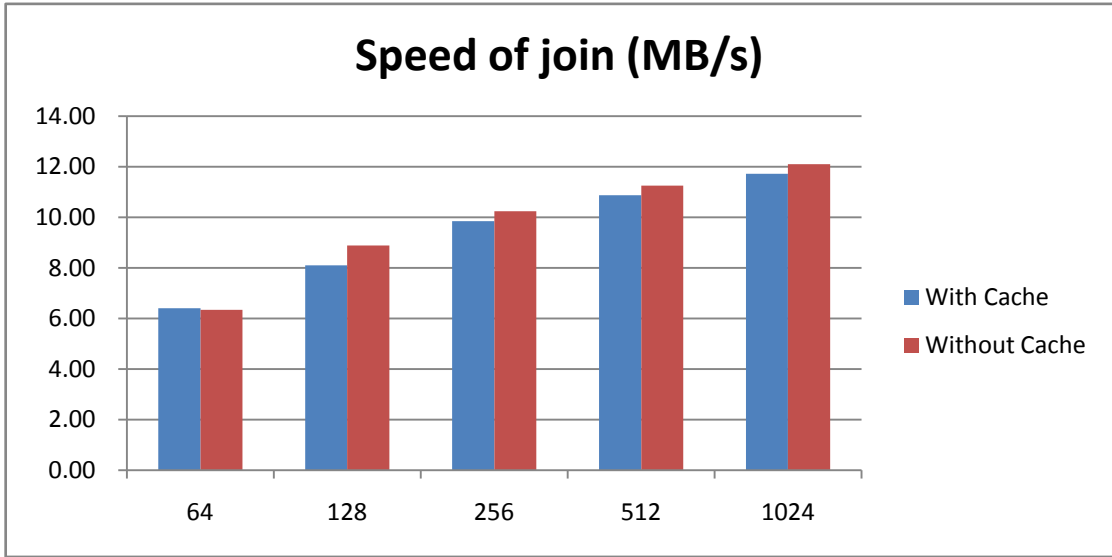


Figure 16: Speed of Join

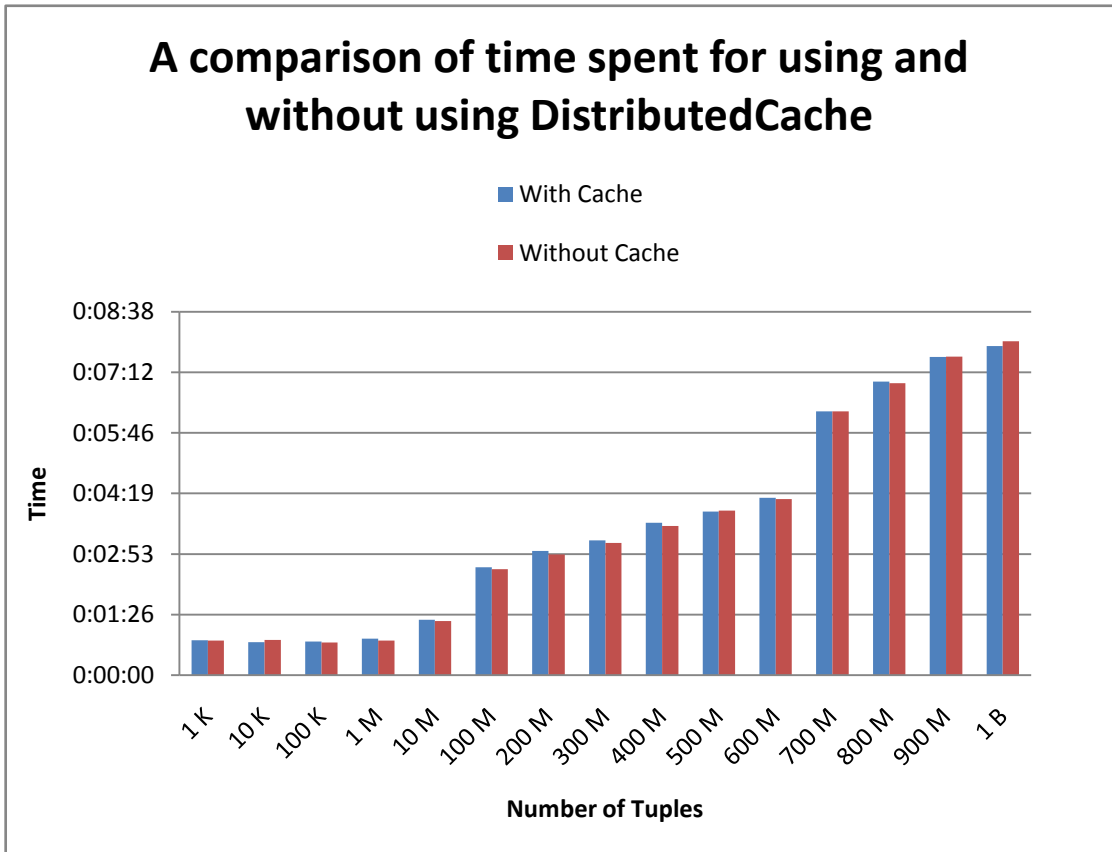


Figure 17: A Comparison of time cost by using and without using DistributedCache

According to the experiment results, DistributedCache did not prove its value.

The reason is, in our Map side join, it will not have the data transmitted during the data processing, so the distributed cache will not be reflected in its advantage. However, the use of DistributedCache is still to be recommended.

4.4.6 A Model Assumption

We have a model assumption for the Map side join in an ideal world.

If we ignore the setup and cleanup time of job, just focus on the job itself:

$$\text{task_cost}_{\text{time}} = \text{task_other}_{\text{time}} + R_{\text{table_init}}_{\text{time}} + \text{join}_{\text{time}}$$

and

$$\text{job_cost}_{\text{time}} = \text{task_cost}_{\text{time}} \times [\text{task}_{\text{num}} \div \text{slot}_{\text{num}}]$$

There is the result with the expected and actual time cost:

Split size (MB)	Task amount	Task cost(s)	Speed of join(MB/s)	Expected cost(s)	Actual cost(s)
64	95	42.3	6.4	1352	1253.1
128	48	47.4	8.5	757.6	726.7
256	24	57.8	10	462	455.4
512	12	78.2	11.1	312.8	321.8
1024	6	116.6	11.9	233.1	248.8

Table 8: Time contrast of expected and actual for different split size

Here is the time contrast of expected and actual for different split size.

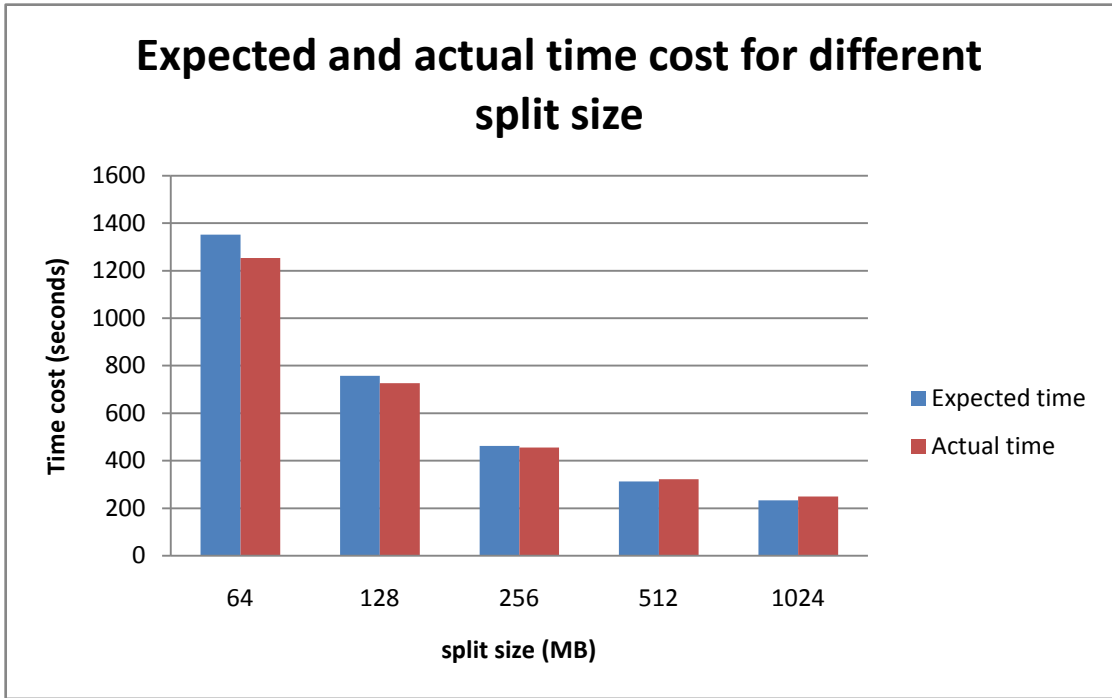


Figure 18: Expected and actual time cost for different split size

From the test result, we think our model assumption is valid. We can estimate the time spent on a job by using this assumption.

Chapter 5. Discussion on the Results

5.1 Strengths and Weaknesses of Map Side Join

One of the advantages is computation locally. In the experiments, the Map side join works well and got a better result in previous works.

The Hadoop environment of Map side join is:

```
dfs.replication = 3
dfs.block.size = 67108864
mapred.min.split.size = 1073741824
mapred.tasktracker.map.tasks.maximum = 2
mapred.map.tasks = 20
mapred.child.java.opts = -Xmx10240m -XX:+UseGCOverheadLimit
```

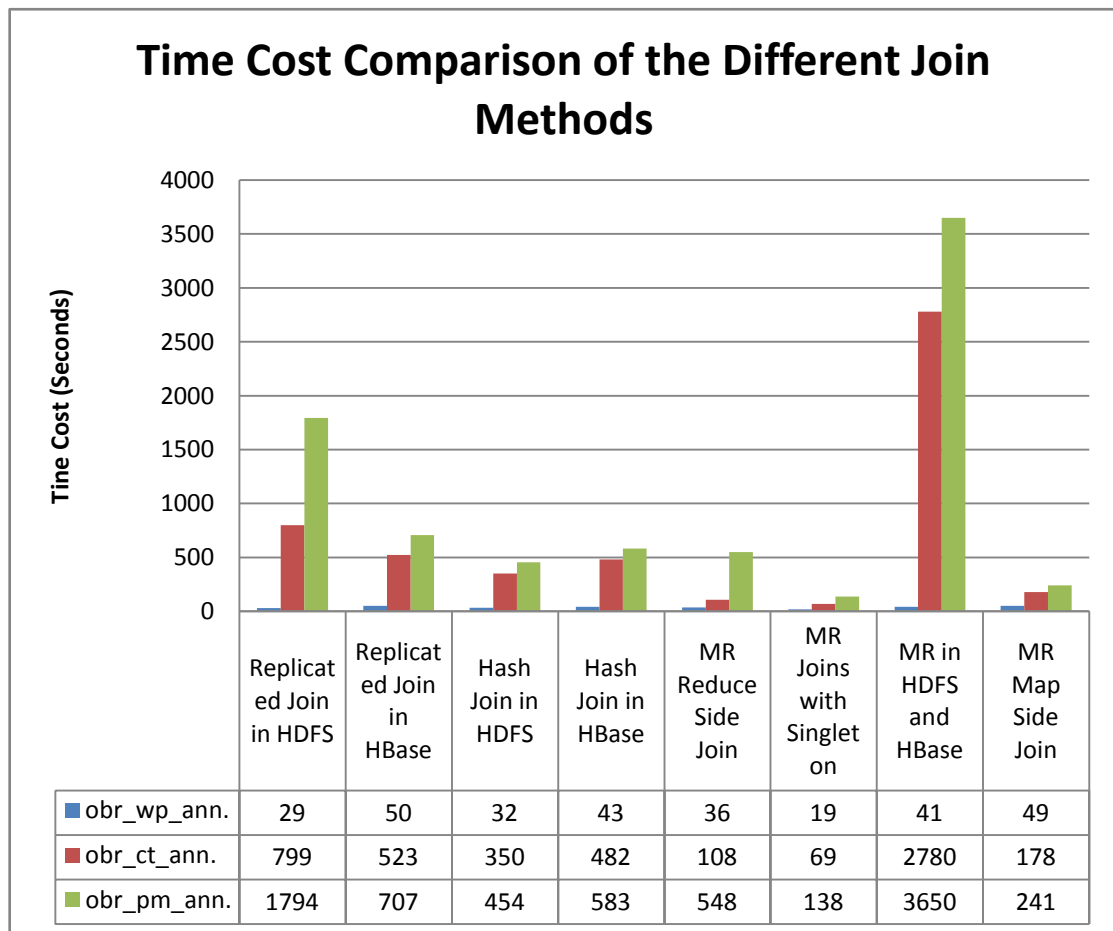


Figure 19: Time cost comparison of the different Join methods

The DataNode is also the computing node. Based on the framework characteristics, MapReduce always trying to ensure that calculating in the same node with the data stored. In this way, to effectively reduce the transmission of data in the network.

However, it is also has very obvious disadvantages.

One disadvantage is the output of the Map Side Join is the final result, rather than the result of intermediate data. Hence, the data should be stored in HDFS. It caused the loss of performance due to the mechanism of HDFS backup.

Another disadvantage is that the Map Side Join is limited by the size of the R table. According to the algorithm, R table should completely place into the memory. If R table is huge, or the node's memory is small, it will existence such a risk that failure of load the R table.

5.2 The Usefulness of DistributedCache

DistributedCache lowers the total amount of data transmission demand to each task by specified that only one data copy will be sent to each node and all the tasks running on that node share this copy. However, this benefit comes at the cost of the overhead introduced by converting data from HDFS to local disk. [4]

Its function is used to save bandwidth, rather than save time. As the result in Figure 17, we can see not much difference between the use and without use the DistributedCache. The DistributedCache did not show its advantages in our experiments is because the Map side join eliminated the data transmission from intermediate data to the Reduce side. Hence, the network bandwidth is not the bottleneck in our scenario any more.

5.3 Load Balancing in HDFS

The load balancing problem is a most important issue in many systems. In HDFS, the NameNode supervised the load balance of the whole cluster, according to the heartbeat mechanism and the location of blocks stored. It could allocate the new block stored in the DataNode which has low load situation and high write performance.

In addition, a tool of load balancing is existed in Hadoop which can be launched by the administrator, to balance the blocks in HDFS. The main function of the tool is to calculate the load state and migrate blocks in the cluster.

We use the tool while the new DataNode adding in the cluster, or we found the cluster is not had a good load balance situation. An instance is the figure 11.

The tool should not be used when the load of the cluster is high. Because if used the tool, it will cause network congestion and high delay for the client.

The usage is in the Appendix.

Chapter 6. Conclusions and Future Work

6.1 NCBO Resource Index Solution

In our master project, we studied the classic join methods on Hadoop, and the features of HDFS, include HDFS accessing, block characteristics, load balancing, replication strategy, also the DistributedCache function.

We explored the Hadoop job scheduling, and the MapReduce behaviors in the cluster, also includes the split size, replication and number of Mapper control.

For various aspects of experiments, we implemented an algorithm of Map side join, generated sufficient number of data which based on the NCBO Resource Index and we contrasted the results also made distinctions of them. At the end, we proposed a hypothetical model of the job time cost calculating.

Based on the result of the experiments, we adjusted the various parameters and we got the solution of NCBO Resource Index. In the final experiment, the Map side join works well and got a better result in previous works.

6.2 Future Work

In our Map side join, we found that the network transfer will not be the bottleneck, but the hard disk drivers.

To improve the throughput of the hard disk, one assumption is to add more hard disks and use RAID 0 (Redundant Array of Independent Disk) method to assemble them as an array.

We think the RAID 0 mode not only can greatly improve hard disk speed in a node, but also without sacrificing the hard disk capacity. And because we use HDFS to store data, we do not need to consider the problem of data loss.

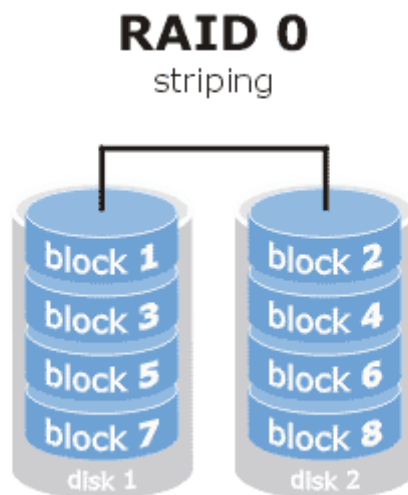


Figure 20: RAID 0 mode

Although we think that the Map side join should be one of the good solutions for the NCBO Resource Index at present. However, things are always changing. With the development and renewal of data structures, someday the method we are using may be failure. New and more easy method will be used.

For example, the Pig and Hive that tools to manipulate data on Hadoop may have greater developments in the future. Actually, Hadoop 2.0.0-alpha is released at this moment, many bugs fixed and more functions are supported.

We believe that more and more, better and better methods are waiting for us to discovery. New technologies are constantly emerging, accompanied by the emergence of new problems.

"No the best but only better" is what we are pursuing.

Reference

- [1] NCBO Project <http://bmir.stanford.edu/projects/view.php/ncbo>
- [2] B. Byambajav, "Methods for Large-scale Semantic Expansion on Hadoop Architecture.", UiS, Stavanger, Norway
- [3] T. W. Wlodarczyk, P. LePendu, N. Shah, C. Rong, "Scaling-out the NCBO Resource Index Processing and Maintenance", UiS, Stavanger, Norway
- [4] G. Luo, L. Dong, "Adaptive Join Plan Generation in Hadoop", Duke University, Durham NC, USA
- [5] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment", Lausanne, Switzerland
- [6] Hadoop API document, <http://Hadoop.apache.org/common/docs/r0.20.203.0/api/>
- [7] Java API document, <http://docs.oracle.com/javase/6/docs/api/>
- [8] Yahoo! Hadoop Tutorial <http://developer.yahoo.com/Hadoop/tutorial/>
- [9] Liu Peng, "Actual Hadoop - Open a shortcut leading to cloud computing", Electronics Industry Pub. Date: (2011), ISBN-13: 978-7121144752
- [10] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Google Inc.
- [11] HDFS <http://wiki.huihoo.com/wiki/HDFS>
- [12] B. Byambajav, T. W. Wlodarczyk, C. Rong, "Performance of Left Outer Join on Hadoop with Right Side within Single Node Memory Size", UiS, Stavanger, Norway
- [13] 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", Indianapolis, Indiana, USA
- [14] Hadoop Technology Forum <http://www.Hadoopor.com>
- [15] Hadoop.apache.org lists http://mail-archives.apache.org/mod_mbox/Hadoop-general/
- [16] Hadoop Source and Process Analysis <https://www.google.com/search?ie=UTF-8&q=Hadoop%E6%BA%90%E7%A0%81%E4%BB%A5%E5%8F%8A%E6%B5%81%E7%A8%8B%E8%A7%A3%E6%9E%90>

[17] HDFS Architecture Guide

http://Hadoop.apache.org/common/docs/r0.20.203.0/hdfs_design.html

[18] Technical Instructions for Configuring OBR Workflow

http://www.bioontology.org/wiki/images/5/5f/Population_And_Maintenance_Of_The_OBR_Index.doc

[19] Semijoin-based distributed database query optimization research

<http://wenku.baidu.com/view/ae7442db7f1922791688e877.html>

[20] JDBM project <http://jdbm.sourceforge.net/>

[21] Populating OBS database

http://www.bioontology.org/wiki/index.php/Populating_OBS_database

Appendix

I. HDFS Common Commands

List the HDFS files

```
$ bin/Hadoop fs -ls
```

List files in one folder

```
$ bin/Hadoop fs -ls <folder directory>
```

Upload files to the HDFS

```
$ bin/Hadoop fs -put <source file> <destination file>
```

Download files from the HDFS

```
$ bin/Hadoop fs -get <source file> <destination file>
```

Delete files in the HDFS

```
$ bin/Hadoop fs -rmr <target file(s)>
```

View the contents of the file in the HDFS

```
$ bin/Hadoop fs -cat <target file(s)>
```

Report basic information of the HDFS

```
$ bin/Hadoop dfsadmin -report
```

Exit the safe mode

```
$ bin/Hadoop dfsadmin -safemode leave
```

Enter the safe mode

```
$ bin/Hadoop dfsadmin -safemode enter
```

Balanced the load in the cluster

```
$ bin/start-balancer.sh
```

Change the size of the block of a file

```
$ bin/Hadoop distcp -D dfs.block.size=<block size> <source file>  
<destination file>
```

Change the replication of a file

```
$ bin/Hadoop distcp -D dfs.replication=<replication amount> <source file>  
<destination file>
```

II. Usage of the Programs

Left Out Join With DistributedCache

```
$ bin/Hadoop jar JoinWithCache.jar <input> <output> <relation set>
```

Left Out Join Without DistributedCache

```
$ bin/Hadoop jar JoinwithoutCache.jar <input> <output>
```

III. Generation Methods of NCBO Resource Index

Generate the files

```
$ java FillData <source file> <destination file> <number of line>
```

Count the lines of a file

```
$ java CountLine <target file>
```

IV. Hadoop Configuration in Our Experiment

File: core-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<!-- In: conf/core-site.xml -->
<property>
  <name>Hadoop.tmp.dir</name>
  <value>/local/ming/dfs/tmp</value>
  <description>A base for other temporary directories.</description>
</property>

<property>
  <name>fs.default.name</name>
  <value>hdfs://haisen1:54310</value>
  <description>The name of the default file system. A URI whose
  scheme and authority determine the FileSystem implementation.
</description>
</property>

<!-- more generic optimizations -->
</configuration>
```

File: hdfs-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<!-- In: conf/hdfs-site.xml -->
<property>
  <name>dfs.replication</name>
```

```

<value>3</value>
<description>Default block replication.
The actual number of replications can be specified when the file is
created.
</description>
</property>

<property>
<name>dfs.name.dir</name>
<value>/local/ming/dfs/name</value>
<description> </description>
</property>

<property>
<name>dfs.data.dir</name>
<value>/local/ming/dfs/data</value>
<description> </description>
</property>

<property>
<name>dfs.block.size</name>
<value>67108864</value>
<description>HDFS blocksize of 64MB for large file-systems. Default is
64M </description>
</property>

</configuration>

```

File: mapred-site.xml

```

<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>

<!-- Put site-specific property overrides in this file. -->

<configuration>

<!-- In: conf/mapred-site.xml -->
<property>
<name>mapred.job.tracker</name>

```



```
<value>haisen1:54311</value>
<description>The host and port that the MapReduce job tracker runs at.
If "local", then jobs are run in-process as a single map and reduce task.
</description>
</property>

<property>
<name>mapred.tasktracker.map.tasks.maximum</name>
<value>2</value>
<description>Should be the number of processors - 1.
</description>
</property>

<property>
<name>mapred.map.tasks</name>
<value>20</value>
<description>Should be 10x the number of slaves or more.
</description>
</property>

<property>
<name>mapred.min.split.size</name>
<value>1073741824</value>
<description>1GB</description>
</property>

<property>
<name>mapred.child.java.opts</name>
<value>-Xmx10240m -XX:+UseGCOverheadLimit</value>
</property>

</configuration>
```

File: masters

haisen1

File: slaves

haisen2
haisen3
haisen4
haisen5
haisen6
haisen7
haisen8
haisen9
haisen10
haisen11

V. Source Codes

File: JoinWithCache.java

```
// Usage : JoinWithCache <input> <output> <cacheFile>
import java.util.Calendar; //for test
import java.util.Date;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.apache.Hadoop.conf.*;
import org.apache.Hadoop.filecache.DistributedCache;
import org.apache.Hadoop.fs.Path;
import org.apache.Hadoop.io.*;
import org.apache.Hadoop.mapred.*;
import org.apache.Hadoop.util.Tool;
import org.apache.Hadoop.util.ToolRunner;

public class JoinWithCache extends Configured implements Tool {

    public static class JoinMapper extends MapReduceBase implements
Mapper<LongWritable, Text, Text, Text> {

        private Map<Integer,Integer> obs_relation = new
HashMap<Integer,Integer>(25000000, 0.99f); //build hashmap for
obs_relation, stored in memory
        private Text annotations = new Text();
        private Text others = new Text();

        public void configure(JobConf job) {
            Path[] cacheFiles = new Path[0];
            try {
                cacheFiles = DistributedCache.getLocalCacheFiles(job);
                for (Path cacheFile : cacheFiles){
                    //if
(cacheFile.getName().indexOf("obs_relation.txt") != -1 ){
                        BufferedReader fis = new BufferedReader(new
FileReader(cacheFile.toString()));
                        String line = null;
```

```

        printTime("Start to read the relation file @ ");
        while ((line = fis.readLine()) != null) { //read
obs_relation file
            String[] field = line.split(",", 4);
            if ( field[1] != null ){

                obs_relation.put(Integer.valueOf(field[1]),
Integer.valueOf(field[2])); //"59133273", "4865045,4866938,1"

                //System.out.println(Integer.valueOf(field[1]));
            }
        }
        printTime("Finish to read the relation file @ ");
    //}
    }
} catch (Exception e){
    System.err.println(e.toString());
}
}

    public void map(LongWritable key, Text value,
OutputCollector<Text, Text> output, Reporter reporter) throws
IOException {
        String line = value.toString(); //read
obr_**_annotation_cut file
        String[] record = line.split(",", 3);
        if ( record.length == 3 ){
            String concept_id = record[1];
            String other1 = record[0];
            String other2 = record[2];
            int index_id = Integer.parseInt(concept_id);
            if ( obs_relation.get(index_id) != null ){

                annotations.set(obs_relation.get(Integer.parseInt(concept_id)).to
String()); //
                    others.set(other1+other2);
                    output.collect(annotations, others); //collect(k
key, v value) Adds a key/value pair to the output.
            }
        }
    }
}

    static void printTime(String note) {

```

```

        Calendar rightNow = Calendar.getInstance();
        Date time = rightNow.getTime();
        System.out.println(note + time.toString());
    }

    static int printUsage() {
        System.out.println("JoinWithCache <input> <output>
<cachefile>");
        ToolRunner.printGenericCommandUsage(System.out);
        return -1;
    }

    @Override
    public int run(String[] arg0) throws Exception {
        // TODO Auto-generated method stub
        JobConf jobConf = new JobConf(getConf(), JoinWithCache.class);
        jobConf.setJobName("join with distribute");

        jobConf.setMapperClass(JoinMapper.class);
        jobConf.setInputFormat(TextInputFormat.class);
        jobConf.setOutputFormat(TextOutputFormat.class);
        jobConf.setMapOutputKeyClass(Text.class);
        jobConf.setMapOutputValueClass(Text.class);
        jobConf.setNumReduceTasks(0);

        // Make sure there are exactly 3 parameters left.
        if (arg0.length != 3) {
            System.out.println("ERROR: wrong number of parameters: " +
arg0.length + " instead of 3.");
            return printUsage();
        }
        DistributedCache.addCacheFile(new Path(arg0[2]).toUri(),
jobConf);
        FileInputFormat.setInputPaths(jobConf, new Path(arg0[0]));
        FileOutputFormat.setOutputPath(jobConf, new Path(arg0[1]));
        JobClient.runJob(jobConf);
        return 0;
    }

    /**
     * @param args
     * @throws Exception
     */

```

```

    public static void main(String[] args) throws Exception {
        // TODO Auto-generated method stub
        int res = ToolRunner.run(new Configuration(), new
JoinWithCache(), args);
        System.exit(res);
    }
}

```

File: JoinWithoutCache.java

```

// Usage : JoinWithoutCache <input> <output>
import java.util.Calendar; //for test
import java.util.Date;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

import org.apache.Hadoop.conf.*;
import org.apache.Hadoop.fs.*;
import org.apache.Hadoop.io.*;
import org.apache.Hadoop.mapred.*;
import org.apache.Hadoop.util.Tool;
import org.apache.Hadoop.util.ToolRunner;

public class JoinWithoutCache extends Configured implements Tool {

    public static class MapClass extends MapReduceBase implements
Mapper<LongWritable, Text, Text, Text> {

        Map<Integer,Integer> obs_relation = new
HashMap<Integer,Integer>(25000000, 0.99f);    //build hashmap for
obs_relation, stored in memory
        Text annotations = new Text();
        Text others = new Text();

        public void configure(JobConf job) {

            Path filePath = new
Path("hdfs://haisen1:54310/ming/res/obs_relation.txt");
            try {

```



```

static void printTime(String note) {
    Calendar rightNow = Calendar.getInstance();
    Date time = rightNow.getTime();
    System.out.println(note + time.toString());
}

static int printUsage() {
    System.out.println("JoinWithoutCache <input> <output>");
    ToolRunner.printGenericCommandUsage(System.out);
    return -1;
}

@Override
public int run(String[] arg) throws Exception {
    // TODO Auto-generated method stub
    JobConf jobConf = new JobConf(getConf(), JoinWithoutCache.class);
    jobConf.setJobName("Join without distribute");

    jobConf.setMapperClass(MapClass.class);
    jobConf.setInputFormat(TextInputFormat.class);
    jobConf.setOutputFormat(TextOutputFormat.class);
    jobConf.setMapOutputKeyClass(Text.class);
    jobConf.setMapOutputValueClass(Text.class);
    jobConf.setNumReduceTasks(0);

    // Make sure there are exactly 2 parameters.
    if (arg.length != 2) {
        System.out.println("ERROR: Wrong number of parameters: " +
arg.length + " instead of 2.");
        return printUsage();
    }
    FileInputFormat.setInputPaths(jobConf, new Path(arg[0]));
    FileOutputFormat.setOutputPath(jobConf, new Path(arg[1]));
    JobClient.runJob(jobConf);
    return 0;
}

/**
 * @param args
 * @throws Exception
 */
public static void main(String[] args) throws Exception {
    // TODO Auto-generated method stub

```



```

        int res = ToolRunner.run(new Configuration(), new
JoinwithoutCache(), args);
        System.exit(res);
    }
}

```

File: FillData.java

```

import java.io.*;

public class FillData{

    public static void rwFileByLines(String inputFile, String outputFile,
long lineTowrite, boolean isRepeatRead){
        long line = 0;
        try {
            File in = new File(inputFile);
            BufferedReader    reader    =    new    BufferedReader(new
FileReader(inputFile));
            BufferedWriter    writer    =    new    BufferedWriter(new
FileWriter(outputFile));
            String tempLine = null;
            reader.mark((int) (in.length() + 1));
            while (line < lineTowrite) {
                tempLine = reader.readLine();
                if(tempLine == null){
                    if(line == 0){
                        System.out.println("Error: Input file is null.");
                        break;
                    }else{
                        reader.reset();
                        tempLine = reader.readLine();
                    }
                }
                //write file data
                writer.write(tempLine);
                writer.newLine();
                if( (line % 1000000)==0){
                    writer.flush();
                }
                //    System.out.print("writing: " + (line / 1000000) + "
million lines. \r");
            }
            line++;
        }
    }
}

```

```

        }
        writer.close();
        reader.close();
        System.out.println("lines are written: " + line);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    if (args.length != 3) {
        System.out.println("ERROR: Wrong number of parameters: " +
args.length + " instead of 3.");
        System.out.println("FillData      <input>      <output>
<lineTowrite>");
        System.exit(-1);
    }

    String inputFile = args[0];
    String outputFile = args[1];
    long lineTowrite = Long.parseLong(args[2]);
    rwFileByLines(inputFile, outputFile, lineTowrite, true);
}
}
}

```

File: CountLine.java

```

import java.io.*;
public class CountLine{

    public static void readByLines(String inputFile){
        long line = 0;
        try {
            BufferedReader reader = new BufferedReader(new
FileReader(inputFile));
            while ( reader.readLine() != null ){
                line++;
                if( (line % 1000000)==0){
                    System.out.print("Reading: " + (line / 1000000) + "
million lines. \r");
                }
            }
        }
    }
}

```

```
        reader.close();
        System.out.println("Finish. Lines are read: " + line);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println("ERROR: wrong number of parameters: " +
args.length + " instead of 1.");
        System.out.println("CountLine <input>");
        System.exit(-1);
    }

    String inputFile = args[0];
    readByLines(inputFile);
}
}
```