Implementing the repartition join for processing big data using Hadoop

Nestor Ivan Escalante Fol, Alberto Portilla Flores[1], Genoveva Vargas-Solar[2],
Marva Angélica Mora Lumbreras[1] and Carolina Rocío Sánchez Pérez[1].
[1]Facultad de Ciencias Básicas, Ingeniería y Tecnología, Universidad Autónoma de Tlaxcala,
Calzada Apizaquito s/n. C.P. 90300 Apizaco, Tlaxcala, México.
[2]French Council of Scientific Research, LIG-LAFMIA,
681 rue de la Passerelle BP 72, 38402 Saint Martin d'Heres, France.
{nestorescalantefol, alberto.portilla, genoveva.vargas, marva.mora, krlinasp}@gmail.com
***Research Area** : Computer Systems*

Abstract
This paper is related to the processing of large volumes of information known as Big Data. We present the implementation of the algorithm "repartition join" which is used to implement the join operation in large datasets. The join algorithm was programmed under the MapReduce programming model. Because of implementing a join in the context of Big Data is complex and costly, we used the Hadoop platform, which provides the necessary tools for managing large volumes of information profits. Next, we analyze the behavior of the algorithm on a cluster consisting of 3 nodes for evaluating its performance.

Keywords: Big Data, MapReduce, Hadoop, Join.

1. Introduction

In recent years it has been generated and stored data at an unprecedented scale. Such information need to be extracted, processed, stored and analyzed in order to take decisions that help organizations. Big Data research aims to address challenges that arise when capturing, storing and maintaining such data, it also worries about the search of information for analysis and visualization of results. The digital universe comprises all kinds of data, however the majority of new data being generated is from unstructured type. This means that data have no type definitions, it is not organized according to any pattern, and there is no concept of variables or attributes, such as is usually in database area. Companies have realized that the only way to gain advantage of such kind of data is to have the capacity to process the information that the company generates in an effective way. However, processing large volumes of information is a complex and costly task.

In this paper we present the implementation of the algorithm "repartition join" which is used to implement the join operation in large datasets. The join algorithm was programmed under the MapReduce programming model under the Hadoop platform, which provides the necessary tools for managing large volumes of information. The rest of the paper is organized as follows, Section 2 introduces related work, Section 3 introduces the MapReduce paradigm, Section 4 presents the Hadoop platform, Section 5 introduces the join algorithm, Section 6 presents the implementation, Section 7 presents experimental results and finally Section 8 concludes this paper.

2. Related works

There are several papers related to join-operator using MapReduce, however we present the papers that we use as basis for our implementation. In [7] authors model the cost of processing a join in MapReduce and argue that the main issue to be considered is to balance input and/or output between reducers. Therefore they present several algorithms and how to use them depending on the job and the statistics available. In [5] different algorithms to perform equi-join are presented. They present an evaluation of the algorithms that shows that depending on the setup, some algorithms do better than others. In [8] the authors propose strategies to implement join algorithms in situations where memory is limited. They proposed to exploit the similarity concept, basically a similarity function between user profiles in relations participating in a join. Besides, there are other research works that address the problem of optimizing joins directly into the platform by modifying the Hadoop core (e.g. [9,10]).

3. MapReduce

MapReduce (MR) is a programming model used to handle large amounts that provides an implementation framework for processing large-scale data sets [3]. The basis of this programming model is to break a problem into smaller independent tasks that are addressed in parallel way by different processes (e.g. on different machines in a cluster). The results of each process are then combined and displayed as a final output. The sets of input data can come from a database or a file and their values can be integers, floats, strings, bytes or complex structures such as lists, tuples and arrays. Key-value pairs are the basic work structure in MapReduce. A MapReduce job is divided into 4 stages:

Initialization: In this step the input data (BD, html files, etc.) are prepared and divided into small tasks or processes.

Map: The Map function takes as parameters a pair (key, value) and returns a list of pairs. The Map function is applied to each element of the input data, so that a list of pairs for each call to the Map function is obtained.

Grouping and Sorting: At this stage different groups are created within each process and these groups are sorted for easy and proper handling of the data.

Reduce: The Reduce function is applied in parallel for each group created by the Map function. The Reduce function is

called once for each unique key output of the Map function. Along with this key, a list of all the values associated with the key is used to make a merger to produce a smaller set of values.

3.1 Example Matrix Multiplication

In this Section we present the MapReduce algorithm to perform the multiplication of two matrices AxB (see Figure 1 and Figure 2).

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \begin{pmatrix} 19 & 24 \\ 50 & 10 \\ 8 & 21 \end{pmatrix} \cdot \begin{pmatrix} 7 & 9 & 60 \\ 59 & 31 & 20 \end{pmatrix} = \begin{pmatrix} 1549 & 915 & 1620 \\ 940 & 760 & 3200 \\ 1295 & 723 & 900 \end{pmatrix}$$

Figure 1. AxB example

$$\mathbf{AB} = \begin{bmatrix} \sum_{j=1}^{n} a_{1j}b_{j1} & \sum_{j=1}^{n} a_{1j}b_{j2} & \cdots & \sum_{j=1}^{n} a_{1j}b_{jp} \\ \sum_{j=1}^{n} a_{2j}b_{j1} & \sum_{j=1}^{n} a_{2j}b_{j2} & \cdots & \sum_{j=1}^{n} a_{2j}b_{jp} \\ \vdots & \vdots & \ddots & \vdots \\ \sum_{j=1}^{n} a_{mj}b_{j1} & \sum_{j=1}^{n} a_{mj}b_{j2} & \cdots & \sum_{j=1}^{n} a_{mj}b_{jp} \end{bmatrix}$$

Figure 2. Algorithm to calculate AxB

The multiplication algorithm for matrices under the MapReduce model is as follows:

i) Map phase:

-For each element (i, j) of A, create a key-value pair where the key is (i, k) and the value A [i, j] for k from 1 to M.

-For each element (j, k) of B, create a key-value pair where the key is (i, k) and the value A [j, k] for i from 1 to P.

At this stage we will obtain as a result, codes which are composed of the position given by the row and column of each element of the array A and B, and the value comprises the value of the array in position. The values of our example in Figure 1 are the following:

Map output

A[1,1] -> ((1,1), 19) - ((1,2), 19) - ((1,3), 19)
A[1,2] -> ((1,1), 24) - ((1,2), 24) - ((1,3), 24)
A[2,1] -> ((2,1), 50) - ((2,2), 50) - ((2,3), 50)
A[2,2] -> ((2,1), 10) - ((2,2), 10) - ((2,3), 10)
A[3,1] -> ((3,1), 08) - ((3,2), 08) - ((3,3), 08)
A[3,2] -> ((3,1), 21) - ((3,2), 21) - ((3,3), 21)
B[1,1] -> ((1,1), 07) - ((2,1), 07) - ((3,1), 07)
B[1,2] -> ((1,2), 09) - ((2,2), 09) - ((3,2), 09)
B[1,3] -> ((1,3), 60) - ((2,3), 60) - ((3,3), 60)
B[2,1] -> ((1,1), 59) - ((2,1), 59) - ((3,1), 59)
B[2,2] -> ((1,2), 31) - ((2,2), 31) - ((3,2), 31)
B[2,3] -> ((1,3), 20) - ((2,3), 20) - ((3,3), 20)

ii) Reduce phase:

We review and compare all pairs generated in the previous phase, where the key is (i, k) and the value is the data to perform the operation A [i, j] * B [j, k], thereby obtaining the elements of the product matrix. For our example in Figure 1 the values obtained are:

((1,1), (19*07, 24*59))
((2,1), (50*07, 10*59))
((3,1), (08*07, 21*59))

((1,2), (19*09, 24*31))
((2,2), (50*09, 10*31))
((3,2), (08*09, 21*31))

((1,3), (19*60, 24*20))
((2,3), (50*60, 10*20))
((3,3), (08*60, 21*20))

((1,1), (133 + 1416))

((1,2), (171 + 744))

((2,1), (350 + 590))

((2,2), (450 + 310))

((3,1), (56 + 1239))

((3,2), (72 + 651))

((1,3), (1140 + 480))
((2,3), (3000 + 200))
((3,3), (480 + 420))

Reduce output

((1,1), (1549))
((1,2), (915))
((1,3), (1620))

((2,1), (940))
((2,2), (760))
((2,3), (3200))

((3,1), (1295))
((3,2), (723))
((3,3), (900))

4. Hadoop

Hadoop is a platform to develop scalable and reliable software for distributed computation under the MapReduce programming model [4]. It can be run in one or more nodes and in both cases, the operation is based on the implementation of four processes that communicate on the client / server model. The architecture of Hadoop is shown in the Figure 3.

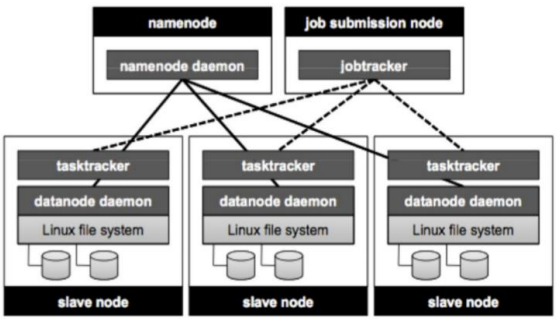

Figure 3. Hadoop Architecture

The elements of architecture in Hadoop are the following [4]:

JobTracker: The MapReduce JobTracker process receives user programs, creates and assigns map and reduce tasks to TaskTracker processes. Subsequently, maintains communication with these processes to monitor the progress of implementation of each map and reduce tasks. If the JobTracker process fails, the MapReduce environment breaks and cannot be run MapReduce programs.

TaskTracker: TaskTracker processes are responsible for implementing the map and reduce tasks that have been assigned by the JobTracker and report their progress in implementing the same. Although only runs a TaskTracker

per client node, each TaskTracker generates multiple Java Virtual Machines to run a map task or reduce task in parallel.

NameNode: The NameNode process maintains the directory tree and file system HDFS files. Now that all nodes split, each file and other related metadata are located. Their information is not persistent; it is constructed using the DataNodes when start the system. Partition splits files (with a default size of 64MB each split) and distributes the splits; DataNodes orders them to the corresponding replication. If a node running a DataNode process fails, the other directs the replica, DataNodes perform the splits that are located on that node to maintain the replication factor. If the NameNode fails HDFS file system is corrupted.

DataNode: DataNodes processes are in charge of the operations of input / output in HDFS system. They maintain communication with the NameNode to report where splits are located and receive information to create, move, read, or delete splits. On the other hand, communicate with each other to perform the data replication. The NameNode JobTracker and processes are server processes while TaskTracker and DataNode processes are client processes. Hadoop allows users to specify the server and client nodes.

5. Join operator

There is a kind of queries executed in multiple tables that are called JOINS combinations or compositions. Such kinds of queries retrieve data from two or more tables using logical relationships among them, called join conditions. A join condition defines the way in which two tables are related in a query by specifying the column from each table to be used for the combination. A specific combination provided a foreign key in a table and its associated key in another table. The join conditions can use logical operators (=, <>, etc.) for comparing the values of the columns.

The simplest way to build a join is, for every tuple in the first relation R (outer loop), entirely cover the second relation S (internal cycle, see the Figure 4). The simple version of the join algorithm is of order n×m, this represents the cardinality of the database. This fact makes it very expensive.

```
foreach tuple r ∈ R do
    foreach tuple s ∈ S do
        if ri==sj then add<r,s> to result
```

Figure 4. Simple Join Algorithm

5.1 Repartition Join

The Repartition Join is a join strategy in the MapReduce framework. In this strategy, 'L' and 'R' are dynamically partitioned on the join key and the corresponding pairs of partitions are joined. This join strategy resembles a partitioned sort-merge join in the parallel RDBMS.

The algorithm 'Repartition Join' uses two datasets 'L' and 'R' with a key field in common and comprises two phases [5]. The pseudocode is shown in Figure 5:

Phase Map:

- Each task in each split map works both on R and L
- Each map task tag records according to the original table
- The outputs resulting from the union are labeled as (key, value).
- The results are then partitioned, sorted and grouped.

Phase Reducer:

- All records of each join are grouped and are eventually reduced.
- For each join, reducer function separates the input records into two sets according to the label on your source table.
- Make a cross-product between the records of the previous sets.

The problem with this version of the algorithm is that all records generated, key field from both L and R have to be stored. Therefore, it can cause buffer overflow.

```
Map (K: null, V: a record from a split of either R or L)
    join_key ← extract the join column from V
    tagged_record ← add a tag of either R or L to V
    emit (join_key, tagged_record)

Reduce ( K': a join key,
    LIST_V': records from R and L with join key K' )
  create buffers BR and BL for R and L, respectively
  for each record t in LIST_V' do
    append t to one of the buffers according to its tag
  for each pair of records (r,l) in BR X BL do
    emit ( null, new_record (r,l))
```

Figure 5. Map and Reduce Phase algorithm Repartition Join [5].

6. Implementation

Our Hadoop cluster includes three iMac machines with the following characteristics:

- Operating System Mac OS X Lion version 10.7.5
- Processor Intel Core 2 Duo, 2.16 GHz
- 4GB DDR2 RAM 667 MHz
- Hadoop version 1.0.4
- Oracle Java SRE 1.6

The data set we used in our experiments was obtained from stackoverflow site (archive.org/details/stackexchange). It includes tables of data users and comments made by registered users on a hypothetical website. This data set was chosen because it is reasonable in size, yet not so big that you can't use it on a single node. The files are organized in XML format as shown in the Figures 6 and 7.

```
<users>
  <row
        Id="2492"
        Reputation="101"
        CreationDate="2012-02-08T19:45:13.447"
        DisplayName="Geoff Dalgas"
        LastAccessDate="2013-06-04T21:38:25.000"
        WebsiteUrl="http://stackoverflow.com"
        Location="Corvallis, OR"
        AboutMe="Developer on the StackOverflow team.
        Find me on http://www.twitter.com/SuperDalga"
        Views="15" UpVotes="0" DownVotes="0"
        Age="37" AccountId="2"
  />
```

Figure 6. Users table fields.

```
<comments>
  <row  Id="1" PostId="1" Score="2"
        Text="As far as I know, each site
              calculates WAR a bit differently."
        CreationDate="2012-02-08T20:07:37.227" UserId="2492"
  />
```

Figure 7. Comments table fields.

6.1 Code of Map and Reduce phases

The Figure 8 presents the user mapper implementation. This mapper parses each input line of user data XML. We parse the data set with a helper function "DatosXML". This function takes in a line of StackOverflow data and returns a HashMap. This HashMap stores the labels as the keys and the actual data as the value. It grabs the user ID associated with each record and outputs it along with the entire input value. It prepends the letter A in front of the entire value. This allows the reducer to know which values came from what data set.

```
public static class UserMapper extends Mapper<Object, Text, Text, Text> {

        private Text clave = new Text();
        private Text valor = new Text();
        public void map (Object claves, Text valores, Context contexto) throws
        IOException, InterruptedException {
        Map<String, String> tabla = DatosXML.Xml2Map(valores.toString());
        String IdUsuario = tabla.get("Id");
        if (IdUsuario == null) {
                return;
        }
        clave.set(IdUsuario);
        valor.set("A" + valores.toString());
        contexto.write(clave, valor);
        }
}
```

Figure 8. Phase Map on users.xml file

The reducer code iterates through all the values of each group looking by relationships and then puts the record in one of two lists. After all values are combined in either list, the actual join logic is executed using the two lists. The join logic differs slightly based on the type of join, but always involves iterating through both lists and writing to the Context object.

7. Results and Testing

The Table 1 shows the results obtained in our experiments. The variables we take into account are the file size and the processing time.

Runtime	File "Users.xml"	File "Comments.xml"
34 sec	1.1 Mb	1.2 Mb
1 min 5 sec	1.1 Mb	2.9 Mb
3 min 6 sec	98.8 Mb	104.9 Mb
13 min 24 sec	809.7 Mb	956.6 Mb
26 min 37 sec	809.7 Mb	4403.2 Mb

Table 1. Results obtained

The Figure 9 shows our experimental results. It can be seen that there is no big difference in time for the first three experiments. In the last two cases the execution time increases due to the size of the files users and comments. Therefore, if we take the first value as reference, in the last case the execution time grows from 34 seconds to 1597 seconds (46.9 times) and the files sizes grow from 1.2Mb to 4403 Mb (3699.1 times) for users file and from 1.1Mb to 809.7Mb (736 times) for comments files. This data shows that the algorithm scales well if we compare the execution time with the size of the files.

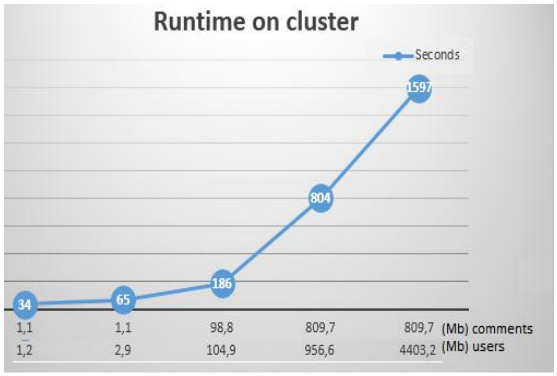

Figure 9. Experimental results

8. Conclusions and future work

MapReduce is a programming model that has been widely accepted to process large amounts of data. In this work we present the implementation of an algorithm reported in research works published in main conferences and journals of DB area. Through this work it has been demonstrated the effective processing of Big Data in Hadoop and how this kind of technology can be adopted in an easy way. As future work we want to evaluate other types of files and to add more nodes to the cluster. We are also searching real situations where can be applied the knowledge gained in this implementation. In addition we propose to study in detail the implementation of the algorithms and the Hadoop platform to propose improvements. Such kind of research will be conducted as part of the MSc studies of the first author.

9. References

1. International Data Corporation and Digital Universe Study. The Digital Universe in 2020: Big Data. Diciembre 2012, http://www.emc.com/leadership/digitaluniverse/iview/index.htm
2. Apache Hadoop http://hadoop.apache.org.
3. J. Lin and C. Dyer. Data-Intensive Text Processing, Manuscript prepared April 11, 2010, University of Maryland, College Park.
4. Chuck Lam. Hadoop in Action, Manning Publications, 180 Broad St. Suite 1323 Stanford, first edition.
5. S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In SIGMOD, pages 975-986, 2010
6. Ch. D. Manning, P. Raghavan, H. Schütze, An Introduction to Information Retrieval, Cambridge University Press, Cambridge, England; 2009, 544 pp.
7. A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. Proceedings of the 2011 ACM SIGMOD, pages 949-960, 2011.
8. R. Vernica, M.J. Carey, and C. Li. Eficient parallel set-similarity joins using mapreduce. In SIGMOD, pages 495-506. 2010.
9. J. Dittrich, J.A. Quiane-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. Proceedings of the VLDB Endowment, 5(11):1591-1602, 2012.
10. H. Yang, A. Dasdan, R.L. Hsiao, and D.S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In SIGMOD, pages 1029-1040. ACM, 2007.