ABSTRACT

MapReduce is a framework for processing large data sets, where straightforward computations are performed by hundreds of machines on large input data. Data could be stored and retrieved using structured queries. Join queries are used frequently. So it's crucial to find out efficient join processing techniques. In this project we have analyzed theoretically and practically various join processing algorithms in MapReduce. We have proposed some techniques for the improvement of performance of join queries and a join selection strategy is proposed to find out best suitable join processing algorithm for a particular application.

Comparison of various join processing algorithms is done. Join query processing algorithms studied in this thesis are Default Hadoop Join, Broadcast Join, Optimized Broadcast Join, Trojan join and Multijoin algorithm. These algorithms are compared on the basis of number of MapReduce jobs involved, their advantages and disadvantages.

We have proposed optimization techniques such as Dynamic Hash table creation, Compressed Broadcast Join and Hash Broadcast join. Also we have suggested a Join selection strategy which helps to select the join processing algorithm based on various parameters.

Also experiments were conducted to measure the performance of these algorithms. Experiments were conducted on Amazon cloud using Elastic MapReduce,EC2 and S3 technologies provided by Amazon Web Services. Results of the experiments proved that the proposed optimization techniques had improved the performance on join query execution in MapReduce environment.

Organization of the Thesis

This thesis is divided into six chapters followed by Conclusion, Future work and References.

Chapter 1 – provides details about concept of MapReduce, various steps involved in the execution of a MapReduce are also described. Then a brief description about Hadoop and Hadoop distributed file system is provided.

Chapter 2- describes various join query processing algorithms already proposed by various authors. Algorithms described in this section are Repartition Join, Broadcast Join, Semi join, Trojan Join and Replicated join.

Chapter 3- Comparison of the algorithms described in chapter 2 is provided in this section. Comparison is based on number of MapReduce jobs, advantages and issues of the join algorithm.

Chapter 4- Describes the optimization technique proposed to improve the performance of the join query execution. Techniques described in this section are – Dynamic hash table creation, Zip broadcast join and Hash broadcast join.

Chapter 5 – Provides details about the experimental setup used and the Results of various experiments conducted to test the performance of join query processing algorithms are shown in this section. Comparison of time required for execution of join query by different algorithms is done is this section.

Chapter 6 – In this chapter a Join Algorithm selection strategy is described. It is in the form of decision tree, which can be used to select proper join algorithm for execution based on the dataset to be joined.

Publications: A paper describing various join query processing algorithms and the join selection strategy was presented at the conference mentioned below, and another paper is under review –

1. Anwar Shaikh, Rajni Jindal : Join Query Processing in MapReduce Environment. In Third International Conference on Advances in Communication, Network and Computing (CNC), Springer LNICST. Pages: 275-281 (2012).
2. Anwar Shaikh, Rajni Jindal : Improving performance of Join Query Algorithms in MapReduce Environment. In Journal of Parallel and Distributed Computing, ELSEVIER. (Communicated)

CHAPTER 1 Introduction to MapReduce

1.1 General Concept

As data on the web is increasing every day, there is a need of a scalable, efficient and powerful tool to analyze this huge amount of data. MapReduce [1] is a large-scale data analysis framework by Google. It hides many complex tasks such as parallelism, fault tolerance, data distribution and load balancing from the user; thus making it simple to use. User have to write the map and reduce function and the remaining things are handled by the underlying infrastructure of MapReduce. So, now writing code for distributed applications is very simple. Hadoop [2] is open source implementation of MapReduce.

Hadoop [2] can be used to store and retrieve data using structured queries. Join queries are the most frequently used and important. So finding out efficient techniques for processing of join queries in Hadoop is crucial. This project is aimed at study of existing join query processing techniques and improving performance in MapReduce environment.

These large scale databases are used for processing of the archived data and to analyze large volume of data. They can generate results in less time by performing job in the distributed manner.

1.2 MapReduce

MapReduce performs the task in two phases Map and Reduce. Map function takes input in the form of key/value pairs and produces intermediate key/value pairs by applying some transformation. The domain of the input data and intermediate data can be different. Reducer performs merging of Intermediate values with same keys to form smaller set of values as output [1].

Map Phase: map(InputKEY, InputVALUE) → list (IntermediateKEY, intermediateVALUE)

Reduce Phase: reduce(IntermediateKEY, list(intermediateVALUE)) → list(intermediateVALUE)

Map and Reduce function are specified by the user, but the execution of these functions in the distributed environment is transparent to the user.

Example

Consider the problem of counting the number of occurrences of each word in a large collection of documents. The user would write code similar to the following pseudo-code:

```
map(String key, String value):
// key: document name
// value: document contents
for each word w in value:
      EmitIntermediate(w, "1");

reduce(String key, Iterator values):
// key: a word
// values: a list of counts
int result = 0;
for each v in values:
      result += ParseInt(v);
Emit(AsString(result));
```

The map function emits each word plus an associated count of occurrences (just `1' in this simple example). The reduce function performs summation of all counts emitted for a particular word.

Eg. **Map** (**book.txt**," This book is about Data structures. Data structures are basic building blocks….")

Output of map phase –

This-1, book-1, is–1, about–1, Data-1, Structures-1, Data-1, Structures-1 ….

Eg. **Reduce**("**Data**", {1,1}) , **Reduce**("**Structures**", {1,1})

Output: **Data** -2, **Structures** -2, **This** -10…..

In this way the final output gives number of occurrences of word in all given files.

Various steps involved in a MapReduce job are described below-

1. Split input file into N small parts.
2. Start many threads of programs on different nodes.
3. Master node picks idle workers/data nodes to work as mappers and other nodes as reducers.
4. Mapper process – reads the file and gives it to map function and write result in local buffer.
5. Buffer contents are written to local disk partitions based on partition criteria and its information is sent to Master.

6. Master gives information of this data to reducer, then reducer performs RPC to get data from mapper's local disk.

7. The data is then sorted to group data with similar key.

8. Reducer iterates over the input data and executes reduce function.

9. Result is written to HDFS.

Figure 1, depicts the entire MapReduce process and the flow of data between various phases.

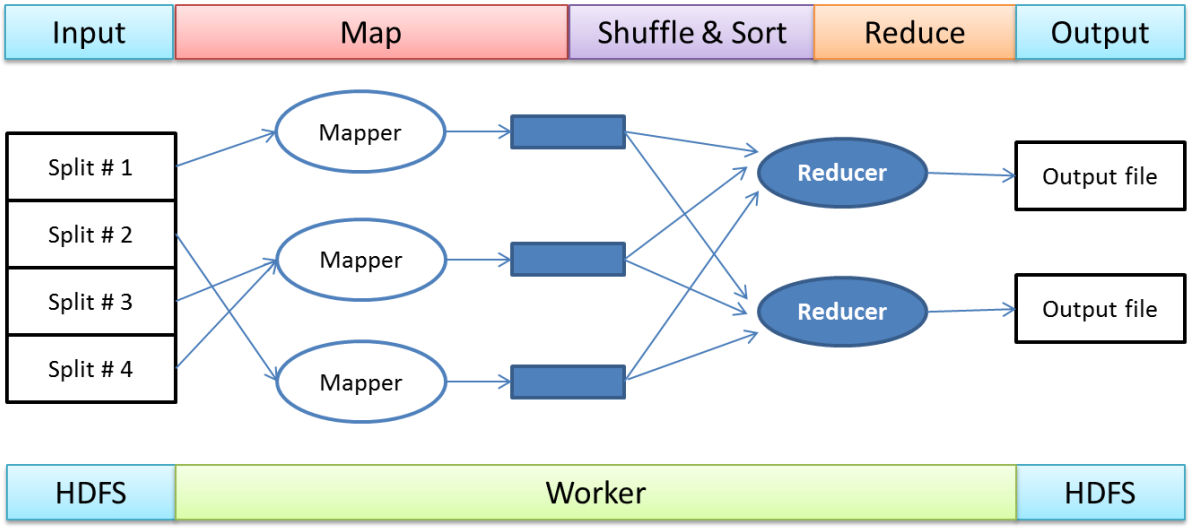

Figure 1. MapReduce framework.

1.3 Hadoop

Hadoop is an open source implementation of MapReduce framework by Apache. It was released in 2009 and was able to sort petabytes of data [2]. Need to process Multi Petabyte Datasets, Hadoop is used mainly when

- Data may not have strict schema
- Expensive to build reliability in each application.
- Nodes fail every day (Failure is expected, rather than exceptional)
- The number of nodes in a cluster is not constant.
- Need common infrastructure

- Efficient, reliable, Open Source.

Hadoop is used by - Amazon, Facebook, Google, IBM, Joost, Yahoo! And many other firms for processing large scale data.

Hadoop is used for

1. Search by Yahoo, Amazon, Zvents.
2. Log processing by Facebook, Yahoo, Joost.
3. Recommendation Systems by Facebook.
4. Data Warehousing by Facebook, AOL.
5. Video and Image Analysis by New York Times.

1.4 Hadoop Distributed File System (HDFS)

MapReduce framework is built on top of a distributed file system. Hadoop uses reliable file sys-tem called as Hadoop Distributed File System (HDFS). HDFS can handle petabytes of data [10]. Data availability is increased by replicating data over multiple nodes. There are some exciting features provided by HDFS like Distributed Cache which would be explored in further sections.

Some characteristics of HDFS are –

1. It is a Very Large Distributed File System – it can handle 10,000 nodes with 100 million files, data size of about 10 to 100 Peta Bytes.
2. Replication of files over multiple nodes is done to handle failures such as hardware failure and allows recovery from it.
3. Map and Reduce computation are performed on the nodes where the input data is already stored.
4. The access model followed is Write-once and read-many.
5. Client accesses data directly from Data Node, Client can find location of blocks.

Name nodes – maps a file to a file-id and list of data nodes. It keeps metadata about entire clusters data nodes. Name node is supported with secondary name node which acts as a backup to the primary name node.

Data nodes – Maps a block id to a physical location on a disk. Data nodes hold the actual data blocks. These data nodes also perform the actual MapReduce job.

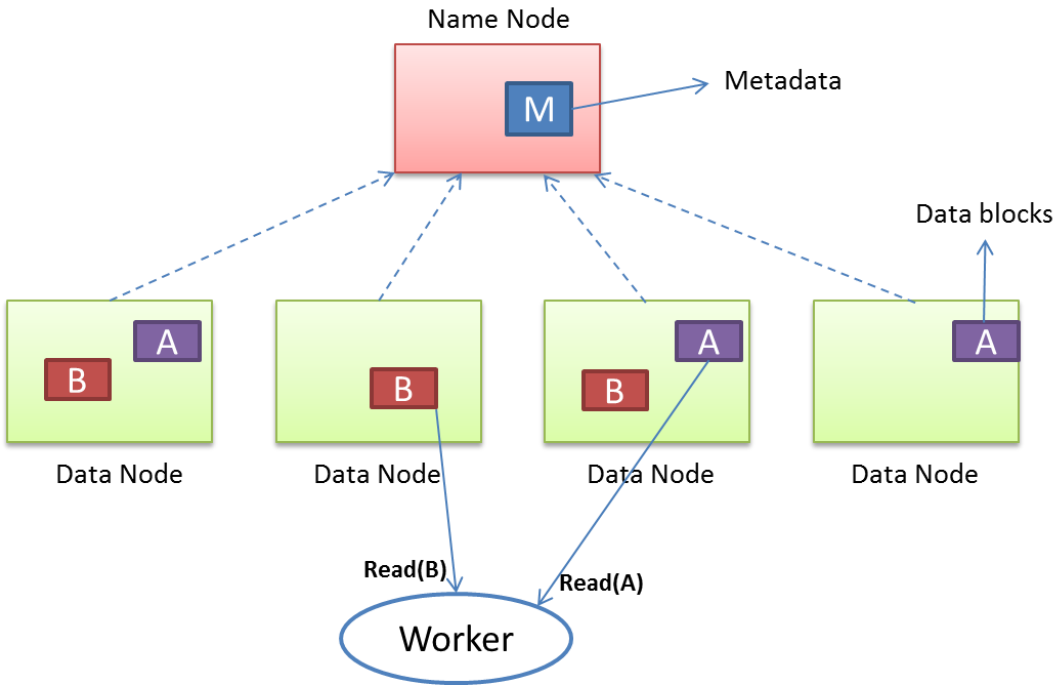

Figure 2. Hadoop Distributed File System Architecture.

Figure 2, depicts the architecture of HDFS. Observe that there is single Name node responsible to hold the Metadata about various blocks/files stored on other data nodes. Each data node may contain data blocks of different files. A single block is replicated on more than one data nodes. By default the replication factor is three. Generally one copy of the data is kept at the different rack so that if a rack fails to provide data, it can be fetched from another rack.

A worker node fetches the required data from the nearest data node. Hadoop tries to assign input block to a task tracker (slave nodes) which holds that input block – to reduce the cost of communication.

CHAPTER 2 Join Query Processing Algorithms in MapReduce

Join algorithms used by Conventional DBMS and MapReduce are different, because join execution in MapReduce uses Map and Reduce functions to get results. This section describes various join processing algorithms for MapReduce environment.

2.1 Repartition Join

Repartition Join [5] is Default Join mechanism in the Hadoop [3]. It is implemented as a single MapReduce job. In Map Phase, each mapper processes a single split (block) of relation involved in the join and outputs Join key (k1), tuple (t1), relation name (R) – {k1, t1, R}. The relation name is used as a tag to identify the relation which a particular tuple belongs.

Output of Map phase is sorted and partitioned based on the join key. Each reducer gets records with same join keys. Reducer process removes the Tag attached to the tuple and separates them to form two different relations. Then cross join is per-formed between these two relations to produce final output.

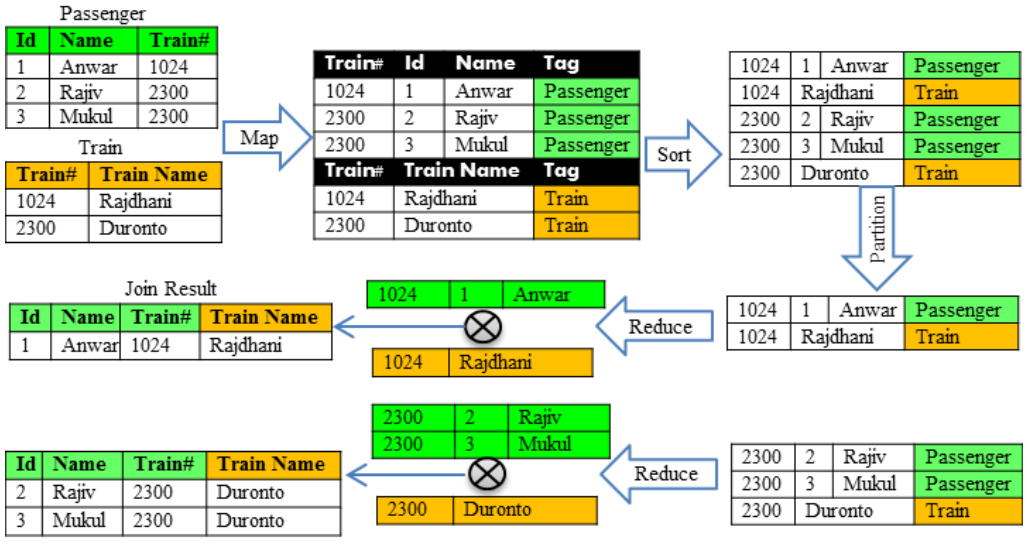

Figure 3. Repartition Join.

Figure 3 depicts natural join between relations Passenger (Id, Name, Train Number) and Train (Train Number, Train Name). In the Map phase both relations are tagged with the Relation name Passenger and Train respectively. Then all tuples are sorted based on the join key-Train Number. After sorting

records with same key are distributed to reach reducer. Tuples from Train Number 1024 are sent to reducer 1 and tuples from Train Number 2300 are sent to reducer 2.

Each Reducer then removes the tag and performs the cross join between tuples from different relations. And the final result is written to the HDFS.

Algorithm : Repartition Join
Input: Relation R1(P,Q) & Relation R2(Q,R).
Output: Natural Join between R1 and R2.

```
Map(Key nameOfRelation, Value tuple)          // key = name of relation, value= single tuple.
{
        Tag = nameOfRelation;
        Join_Attribute = tuple.Q;             // Q is the join attribute
        Other_Attribute = Non join attribute  // either tuple.P or tuple.R
        Write( Join_Key, Tag + Other_Attribute);
}
Reduce(Key Join_Attribute, Value list)        // list contains Tag and other attributes
{
        R1_list = Other_Attributes with Tag R1;
        R2_list = Other_Attributes with Tag R2;

        //perform the cross product between R1_list and R2_list

        For(each X in R1_list){
          For(each Y in R2_list)
                Write(Join_Attribute + X + Y);
        }
}
```

Algorithm 1: Repartition Join.

Default Hadoop Join mechanism has some drawbacks –

1. Sorting and movement of all tuples from both relations is required between Map and Reduce phases. It increases network traffic and hence affects the performance.
2. If there is a Popular key then all records for that key are sent to single reducer. This would consume more time.
3. Minor overhead of tagging tuples in map phase is involved.

Improved Repartition Join was suggested by authors of [5] where the output of map phase was adjusted such that tuples of smaller relation appeared before tuples of larger relation and generation of join result needed buffering the tuples of smaller relation and streaming the tuples of larger relation.

2.2 Broadcast Join

Broadcast join described by [5] is similar to Naïve Asymmetric join in [3]. Broadcast join is a type of asymmetric join because it treats two relations differently. When relations R1 and R2 are such that, number of tuples from R1 are very less as compared to R2, then the smaller relation R1 is copied mapper nodes before execution of the join. This is possible by using Distributed Cache Mechanism provided by HDFS. It is used for efficient distribution of large and read only files.

When relation R1 is not available at mapper then it is retrieved from the Distributed Cache. Once the smaller relation is available to each mapper node, a Hash table is built such that Join attribute act as Hash key and the tuple of R1 act as Value.

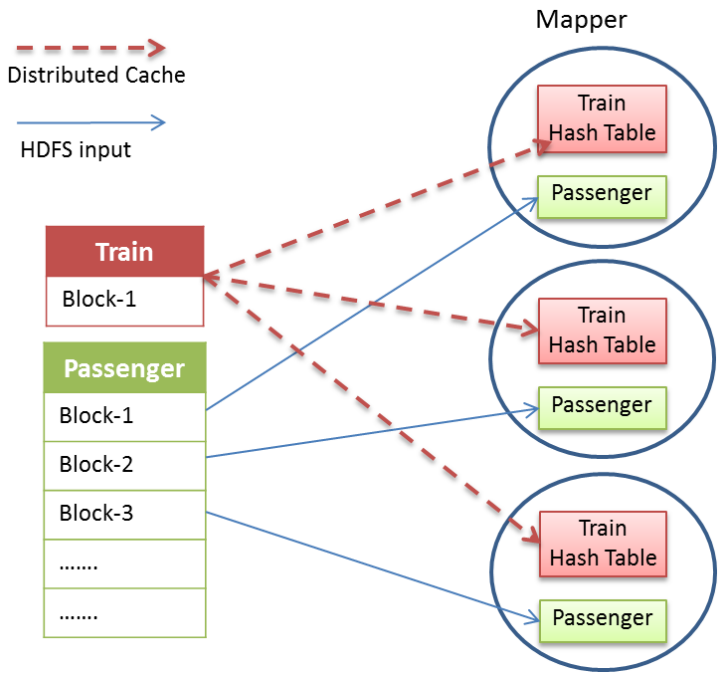

Figure 4. Broadcast Join.

Now, each mapper is assigned a split of a bigger relation R2. All tuples of R2 are scanned sequentially. For each tuple in R2, Hash of the Join Key of R2 is used to retrieve the matching tuple of R1 from the Hash table built earlier; tuple from R2 along with hash table value is added to the join result.

Note that Broadcast join need only Map phase, and network bandwidth requirement is reduced because only smaller relation is transmitted over network.

```
Algorithm : Broadcast Join
Input: Relation R1(P,Q)  & Relation R2(Q,R).
Output:  Natural Join between R1 and R2.

//Method to initialize mapper
Setup()
{
        Load data from Distributed Cache (R1);

        For(each tuples r1 in R1)          //Build Hash table for cached data
        {
          HashTable.put(r1.Q, r1.P);       //add key attribute and non-key attributes
        }

}

//invoked once for each tuple of R2
Map(Key nameOfRelation, Value tuple)       // key = name of relation, value= single R2 tuple.
{
        r1.P = HashTable.get( r2.Q);       //Lookup the matching tuple in Hash table

        Write( r1.Q + r1.P + r2.R);        //Write result to HDFS
}
```

Algorithm 2: Broadcast Join.

Figure 4, Depicts the processing of Broadcast join between relations Passenger and Train, Smaller relation Train is copied to all mapper nodes and hash table is built locally at each node before execution of map phase. And in map phase, a split of Passenger table is received by mapper and join execution is performed independently. Result is written to the HDFS.

2.3 Optimized Broadcast Join (Semijoin Approach)

It is also termed as Optimized Asymmetric Join [3]. While performing join, most of the times it happens that there are many tuples which do not contributed to the join result. So if the relation R1 and R2 are large, then it may be costly to perform Broadcast join. In such case extraction of only those tuples of R1 which contribute to the join can reduce the size of R1 by significant number and can make it candidate for Broadcast join using semi join mechanism.

Semi join between R1 and R2 is performed using two MapReduce jobs. First, projection of unique join attribute values from relation R2 is done. Using these unique value, second MapReduce job is performed to find matching tuples from R1. These are the tuples required for performing actual join. As the size of R1 is reduced by semi join, it is copied to all mapper nodes and further join processing is performed same as Broadcast Join.

The join processing of Optimized broadcast join is depicted in figure 5. The join is processed in three steps. First step the unique train numbers are extracted from the Passenger table using one complete MapReduce job. These unique train numbers are added to the distributed cache.

Second step, filters the required tuples from the Train table using unique train numbers obtained from Distributed cache. As a result we get only those tuples of the Train relation which contributes to the join result. These tuples are then added to distributed cache.

In third step the Broadcast join is performed which produced the final join result. Here we can observe that instead of Broadcasting entire Train relation, only one tuple from Train relation is broadcasted/transmitted over network which reduces the join processing time.

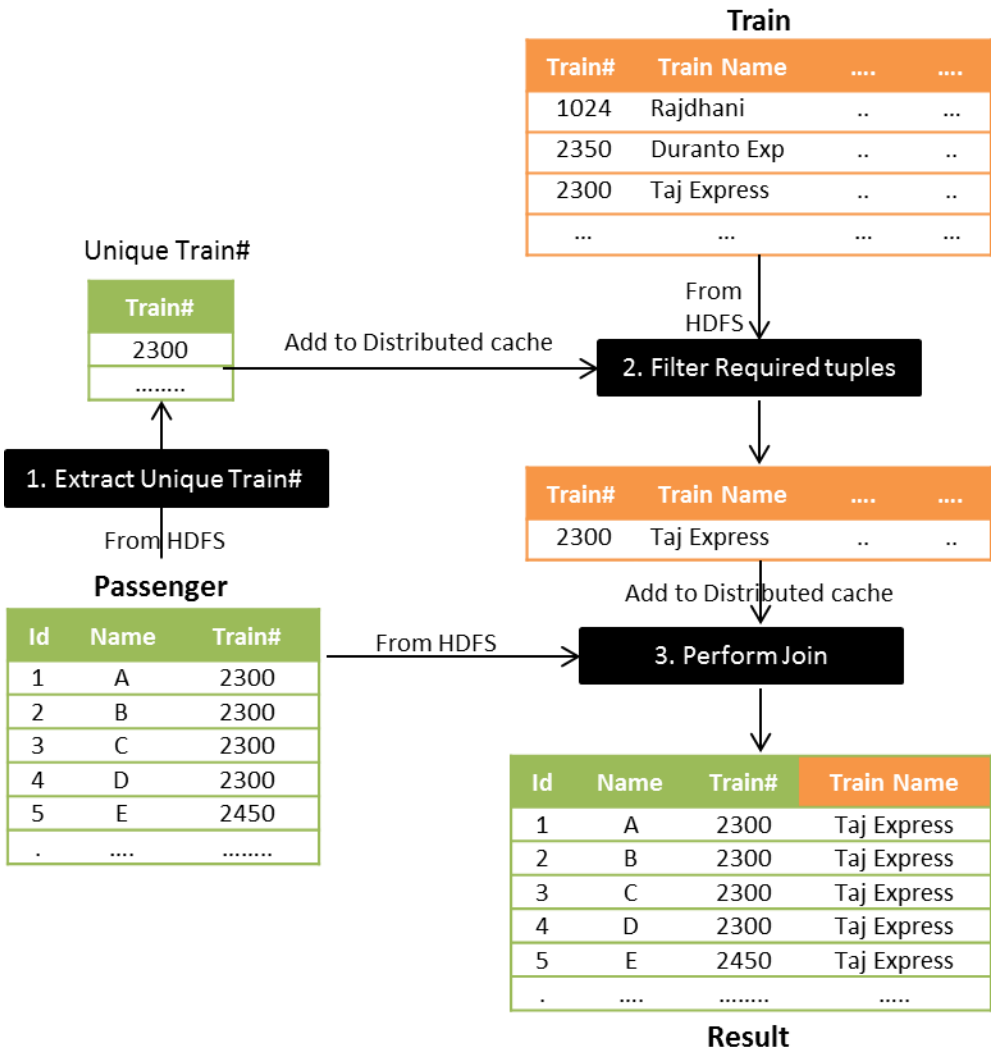

Figure 5. Example of Semijoin.

2.4 Trojan Join

Join strategies discussed above do not take advantage of schema knowledge, which is available earlier. And many cases join conditions do not change; there may be change in the number of tables used in the join query. Trojan join [6] is designed to take advantage of this.
Co-partitioning is the basic idea behind Trojan join. Applying same partitioning function to both relations involved in join is called co-partitioning.

Co-partitioning of the data is done at load time; and co-group pairs from each of two relations having similar join attribute values are kept on the same split. As the data from both relations having the same join key value is available at the same split, joins are executed locally at that mapper node and there is no need of shuffle and reduce phases required, hence reducing the network communication.

Co-grouped data is stored using headers to differentiate between two relations and split. Figure 6 depicts the storage structure of co-grouped data. Two headers are required to indicate starting position of data of relations R1 and R2 and one header to indicate end of logical split.

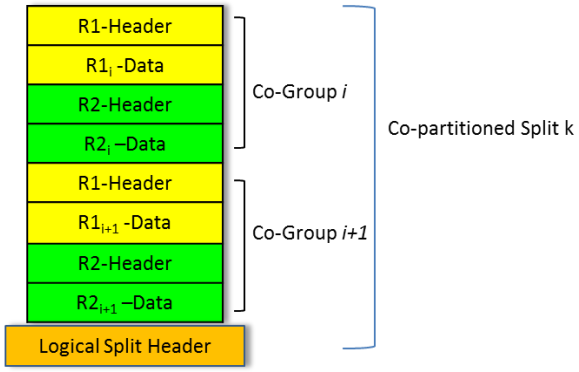

Figure 6. Co-partitioning data of relations R1, R2.

Once data is loaded, map function collects data from a Co-Group and performs cross product between them and adds it to final result. This is possible because each Co-Group contains data with same join key value.

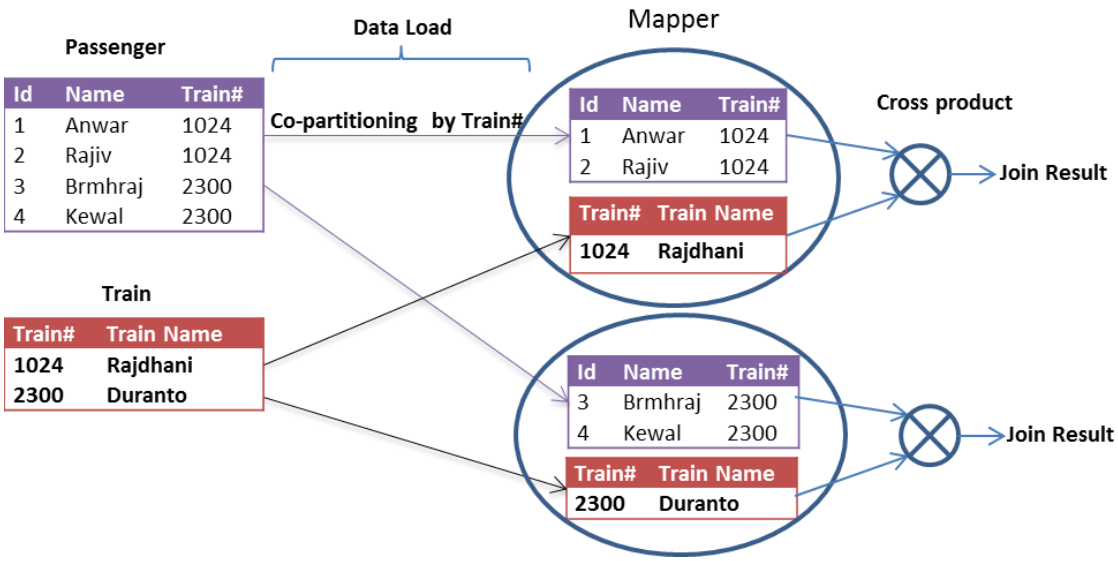

Figure 7. Trojan Join execution

Figure 7 depicts the processing of the Trojan join. The Passenger relation and Train relation are co-partitioned based on the train numbers. Tuples with the same train number are kept in the same co-partition. Also an index is built based on train number. All these steps are done as part of the pre-processing. At the time of join execution, just the cross product is performed between the data of two different relations stored in same co-partition.

2.5 Replicated Join

This is multi-way join query processing algorithm proposed in [4]. It performs the three-way join as a single map-reduce operation. Consider natural join between three relations R1(A,B), R2(B,C), R3(C,D). If we use previous approaches then we need to perform this in two map reduce processes, because we can join only two relations at a time.

In this approach, each tuples from relations R1 and R3 are sent to multiple reducers, this may increase the communication cost, but it is acceptable because the join will be performed in the single MapReduce job. Each tuple from R2 is sent to single reducer only. As you can observe that Replicated join should be used when tuple from one relation is joined with many tuples of other relations.

Relation	Reducer process number
R1(A,B)	[hash(b),*]
R2(B,C)	[hash(b),hash(c)]
R3(C,D)	[*,hash(c)]

Table 1. Distribution of Tuples to Reduce processes, * indicates any value between 1 and m.

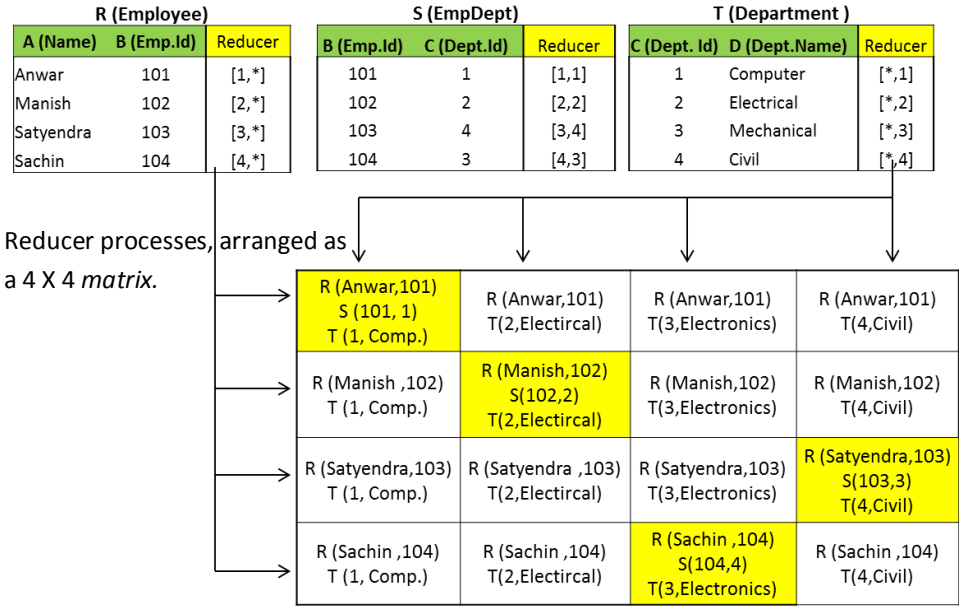

Figure 8. Replicated join.

If we have ‘k’ number of reduce processes where k=m*m, for some value of m, reducers are numbered as [i,j] where values of i and j are 1,2,..m. A hash function selected such that tuples from R1, R2 and R3 can be hashed into ‘m’ buckets. Each reducer is responsible to process one bucket for B and one for C. Tuples are sent to reducer using the hashed values of join attributes B and C. Table 1, shows the distribution of tuples to the reduce processes. After distribution of tuples at each reducer numbered [hash(b), hash(c)] we have tuples from R1, R2, R3 such that they have same value for join attribute. Now the join can be performed locally at the reducer.

Figure 8, depicts the distribution of tuples in Replicated join, where k=16=4*4. Relations R, S, T represent relations Employee, EmpDept and Department respectively. 4X4 matrix represents reducer processes, cells marked in Yellow represents reducers contributing to final results.
An optimization algorithm to find minimum number of replicas/reducers needed for join execution was proposed in [4].

Chapter 3 Comparison of Join Algorithms

Consider that a join is performed between relations R1(a,b) and R2(b,c). Table 1 compares above mentioned join algorithms based on number of MapReduce jobs required for execution, advantages of using a particular method and issues involved.

Join Type	MapReduce jobs	Advantages	Issues
Repartition	1 MapReduce job	Simple implementation of Reduce phase	Sorting and movement of tuples over network.
Broadcast	1 Map phase.	No sorting and movement of tuples.	Useful only if one relation is small.
Optimized Broadcast	2 MapReduce jobs for Semi join and 1 Map phase for Broadcast join.	Applicable when the selectivity of a relation is very low.	Extra MapReduce jobs are required to perform semi join
Trojan	1 Map phase.	Uses schema knowledge.	Useful, if join conditions are known.
Replicated	1 MapReduce job.	Efficient for Star join and Chain join.	For large relations more number of reducers / replicas are required

Table 2. Comparison of Join processing methods

Repartition join requires one complete MapReduce job. It involves sorting of all tuples based on the join key and then the tuples with same join key are sent to single reducer. This movement of tuples degrades the performance of repartition join. Its performance is lowest as compared to other algorithms.

Broadcast join is executed in single Map phase. But, before execution of the map phase the data of the smaller relation is made available to the mapper using the distributed cache. It is useful only when one of the relation involved in the join is very small such that it could be transmitted to all nodes

in less amount of time. It is better than Repartition join as no sorting and movement of the data is involved.

Optimized broadcast join requires three MapReduce jobs. First two jobs are required for the execution of the semijoin, and the third job performs the normal broadcast join. This join is useful only when all tuples from smaller relation do not contribute to the final result. More number of jobs are required as compared to the other join algorithms. Also, it is useful in very few cases.

Trojan join requires prior knowledge of database schema and join conditions. It involves preprocessing steps in which data from both relations is stored such that the tuples with same join key are stored in the same co-partition. Also an index is maintained which reduces the join execution time. Trojan join required only one Map phase, as data is locally available on the same partition. Trojan join is more similar to the traditional databases as it requires schema knowledge.

Performing join between more than two relations may require more than one MapReduce jobs, because a cascade of two way joins is required to be performed. But, in some special cases like the star join and chain joins – mostly used in the data mining applications, a Multiway join could be performed. But, it requires more number of reducers and high amount of replication of data when the relation is large.

Broadcast join and Trojan join are Map side join algorithms and Repartition join and Replicated join are the Reduce side join algorithms.

Chapter 4 Optimizations

4.1 Related Work

- One new framework is designed to improve the join processing, called as Map-Reduce-Merge [8]; it includes one more stage called Merge which is used for joining tuples from multiple relations.

- Join execution can be improved by including indexes; [6] introduces Hadoop++ which describes Trojan join and Trojan index to improve performance.

- Methods described in this report are applicable when the data is organized in Row-wise manner. For Column-wise data store, join optimization algorithms are described in [9].

- Authors in [7] described a design of query optimizer for Hadoop which creates optimized query execution plan before execution of query.

4.2 Proposed Optimization Techniques

We have proposed three optimization techniques – Dynamic Hash Table Creation, Compressed Broadcast Join and Hash Broadcast Join. These techniques are applicable in different scenarios described below.

4.2.1 Dynamic Hash Table Creation

As per the definition of Broadcast join provided by authors in [3] Distributed Cache distributes a copy of smaller relation to each Mapper before performing the actual job. Then a hash table is created for the smaller relation locally.

This solution provides better results when the size of smaller relation (R1) is less than the input split size of larger relation (R2). Because the size of hash table/number of entries in the hash table are less. But, if the size of R1 is greater than the size of input split of R2 then the performance gain can be achieved by constructing the hash table for the input split of R2. Doing this will result in less number of entries in hash table and would be more efficient.

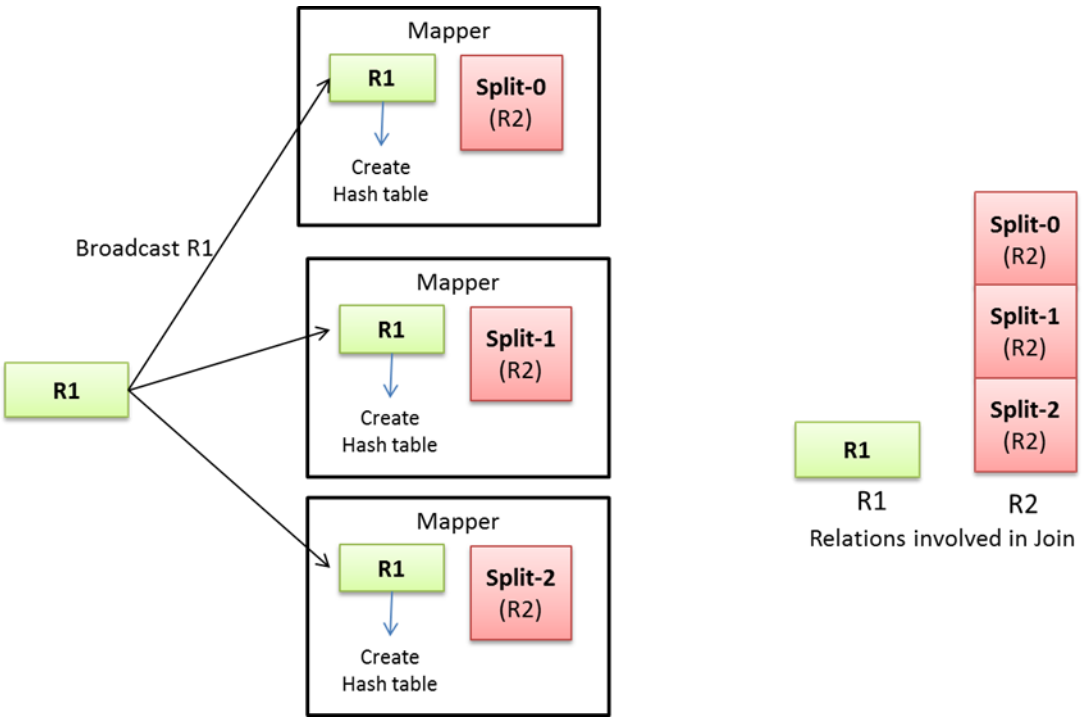

Figure 9. Hash table creation in Normal Broadcast join.

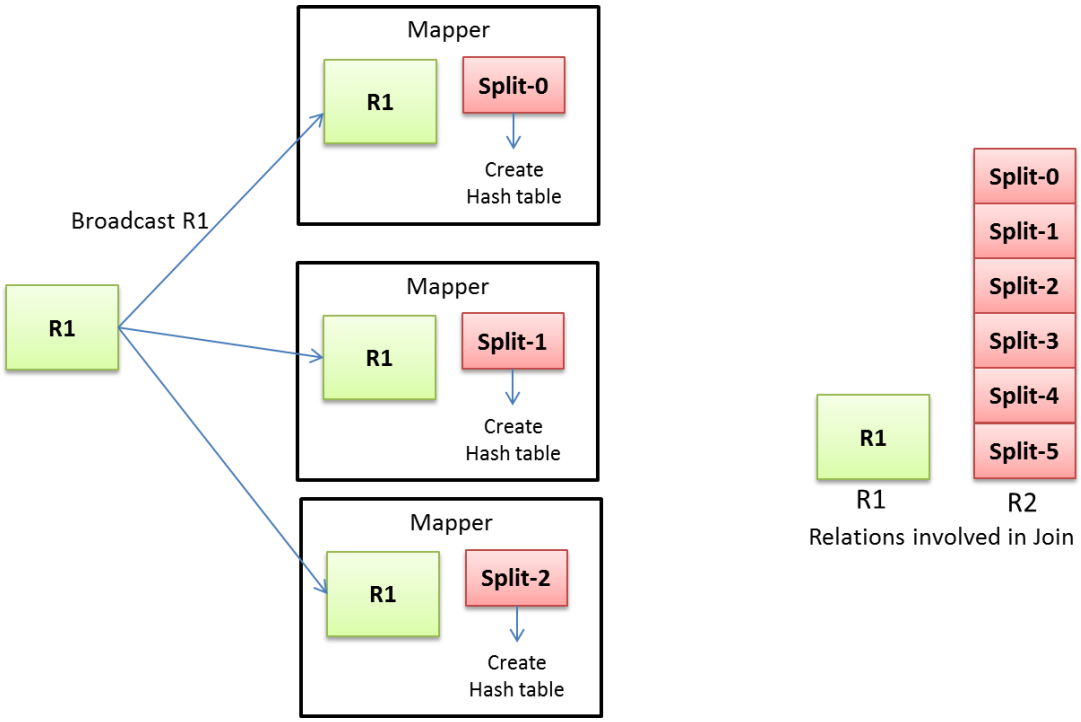

Figure 10. Hash table creation based on the input size.

Following code described the process of Broadcast join with Dynamic Hash table creation –

Input: Relation R1(P,Q) & Relation R2(Q,R).
Output: Natural Join between R1 and R2.

```
Setup()
{       //Method to initialize mapper

        Load data from Distributed Cache (R1);

        R1_Size=Calculate size of cached data;
        R2_Size=Calculate size of Input split of R2;

        If(R1_Size < R2_Split_Size)
        {
                Build Hash table for cached data (R1_HASH_TABLE)
        }
}

Map()
{       //invoked once for each tuple of R2
        If(R1_Size<R2_Split_Size)
        {
                Lookup the matching tuple in Hash table (R1_HASH_TABLE)
                Write result to HDFS
        }
        else
        {
                //build hash table for R2
                Add this tuple of R2 to hash table.( R2_HASH_TABLE)
        }
}

CleanUp()
{
        //function called at the end of Map Phase.

        If(R2_Split_Size > R1_Size)
        {
                //For each tuple in Cached relation R1
                Lookup the matching tuple in Hash table (R2_HASH_TABLE)
                Write result to HDFS.
        }
}
```

Algorithm 3 – Dynamic Hash table Broadcast Join.

The Mapper contains a run() method which calls its setup() method once, its map() method for each input record, and finally its cleanup() method [2].

Setup Method : Called once for each Mapper. In the approach suggested by authors in [3], this method is used to build the hash table for the smaller relation R1 obtained from the distributed cache. But, in our approach this method checks the size of relation R1 and the size of input split which is to be processed by this mapper. If the R1 relation is small then the hash table is built for it called as R1_HASH_TABLE, else no action is taken.

Map Method : This method is called once for each tuple in the input split of relation R2. If R1 is small then R1_HASH_TABLE is looked up to obtain matching tuples. In case match is found, the result of join is written to the HDFS.

Else if split of R2 is small, then the Map function builds the Hash table (R2_HASH_TABLE) for the split of R2- by adding each input tuple of Map function into the Hash table- this Hash table is used in the cleanup method to compute the final join result

Cleanup Method: This function is called at the end of Map function call. If split of R2 is small then the actual join results are produced in this function. For each tuple in the cached relation R1- the hash table (R2_HASH_TABLE) is looked up to find matching tuples. And the result is written to HDFS.

4.2.2 Zip Broadcast Join

As described earlier broadcast join Broadcasts the smaller relation using Distributed. The distributed cache needs to distribute/transfer these files over network to all nodes participating in the Hadoop job, MapReduce job starts only after receiving the files from Distributed Cache. So the time required for the broadcast of relation R1 can be reduced if we perform compression of the data.

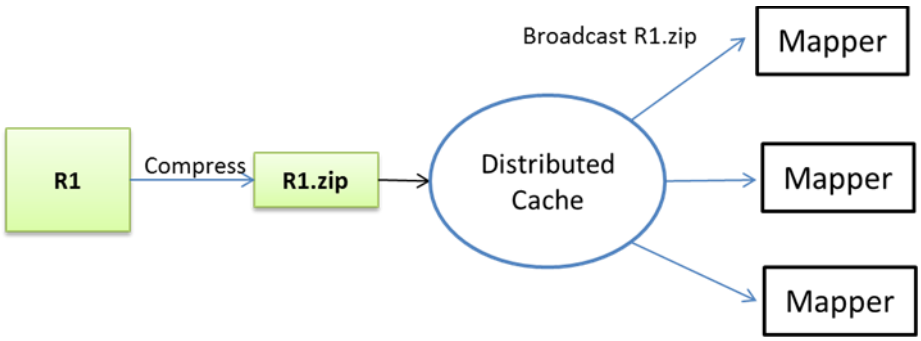

Figure 11. Zip Broadcast Join.

Distributed Cache passes the compressed files to each slave node and decompresses the archive as part of caching. Distributed Cache supports the following compression formats:

- zip – This format is suitable for the archival of a large number of small files. In the zip archive format, the metadata for each entry—the information about each individual entry—is not compressed. [13]
- tar- tar stands for Tape Archive. As the name suggests, this file format is used to bundle various files into a single file so that it can be written to the tape for archival purpose. Files are not compressed in this. [14]
- tgz (tar.gz)-

  gzip compression algorithm is used to compress the entire tape archive. The metadata is also compressed, so the individual files can't be accessed without complete decompression. [14]

- jar - This file format is used to group together various Java class files and associated resources like images, text into single file – to distribute the code over java platform. [15]

As described above, the tar format is not useful because it does not compress the data, also the jar is not suitable because we are not broadcasting java code. So, the remaining formats zip and tgz can be used. But, since zip format is suitable for compressing large number of small files, we prefer using zip format for the compression of smaller relation involved in the join.

The performance gain achieved by this method is because of less amount of data to be transmitted to different number of users. Suppose for a MapReduce job, we have more than 50 slave nodes involved in the Cluster, then Broadcasting a relation means we need to provide same copy of the data to 50 different locations, so if we reduce the size of data to be broadcasted then the network traffic would be reduced by large amount. So, this techniques requires less time for execution.

4.2.3 Hash Broadcast Join

In Memory Hash table:

The Map phase of Broadcast join involves building of hash table for smaller relation. As JAVA is most widely used language for the Hadoop, we have observed some issues with the Hash Table for JAVA. For a given relation we construct a -In Memory Hash table, which uses the JVM heap space for construction of Hash Table.

JVM Heap size limitation:

We have used HashMap class of JAVA for creation of Hashtable. We have tested it for user relation which was used for Broadcast. For each user very few details were considered such as username, age, contact number. Number of tuples in the relation were 10,00,000 – 20,00,000 – 30,00,000.

Heap size for the JVM was set to 640MB. Hash table was successfully created for the first two cases. But, for 30,00,000 entries the JVM ran out of memory and OutOfMemory exception was thrown.

So, we can conclude that it is not always possible that the in memory hash table could be built for smaller relation. Because smaller relation means the size of relation is such that it can be transmitted over the network. Now there are two situations –

1. Size of each tuple is small – We have more number of tuples – Thus, more number of entries in hash table.
2. Size of each tuple is larger- We get less number of tuples – Thus, less number of entries in hash table.

So, we a Hash table mechanism which –

1. Could handle large number of tuples.
2. Does not depend on the JVM heap size.

To handle above requirements we suggest use of **DISC BASED HASH TABLE** – one such Disc based hash table available is JDBM2 [17].

JDBM2-

- It was developed to support data which does not fit into memory.
- It provides fast access.
- Scales well up to 10^9 records.
- Uses java serialization.
- It does not support integrity checks such as foreign keys.
- This library is provides simple usage.

Based on these features we can say that it would be better alternative for the in memory hash table when there are large number of tuples to be broadcasted.

Compressed Hash table file distribution-

In the Broadcast join - Each Map process creates Hash table for the smaller relation. If we have 150 Map processes then 150 times the Hash Table will be created. This would waste significant amount of time. This can be optimized by following the approach mentioned below-

1. Generate Disc based hash table only once.
2. Compress the hash table files.
3. Broadcast the compressed hash table files.
4. Use the JDBM hash table directly.

Size of Compressed Hash table files is little more than the size of Compressed relation. Below is one example –

Original File size	Compressed file size	JDBM Hash table file size	Compressed JDBM hash table file size
98 MB	30.3 MB	99 MB	32.1 MB

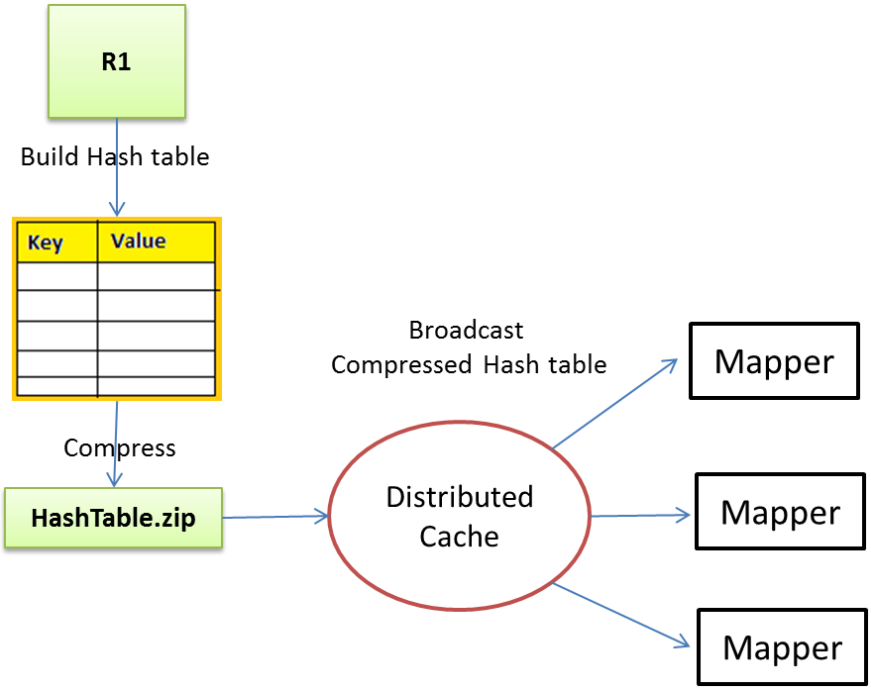

Figure 12. Hash Broadcast Join.

Figure 12, depicts the various stages of Hash broadcast join. For a relation R1, a disc based hash table is created on one node (Master node). Then the hash table is compressed. This compressed hash table is added to the distributed cache. Hadoop distributes this hash table to each node in the cluster. Then Mappers at the slave nodes, can use this hash table directly – no time is wasted in creation of the hash table. Hence performance is improved.

CHAPTER 5 Experiments

Experiments to test the performance of various techniques described above were conducted on Amazon web services (AWS) platform. AWS provides cloud infrastructure to perform distributed computing. We have used EC2 instances for the execution of the MapReduce jobs along with Elastic MapReduce.

5.1 AWS Cloud -

Amazon Web services (AWS) provides a flexible, cost-effective, scalable, and easy-to-use cloud computing platform that is suitable for research, educational use, individual use, and organizations of all sizes. It's easy to access AWS cloud services via the Internet. Because the AWS cloud computing model allows to pay for services on-demand and to use as much as you need

5.1.1 Amazon Elastic Compute Cloud (Amazon EC2)-

Amazon EC2 is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. Amazon EC2's simple web service interface allows us to obtain and configure capacity with minimal friction. It provides us with complete control of our computing resources and lets us run on Amazon's proven computing environment.

- Amazon EC2 provides **resizable compute capacity** in the cloud.
- It is designed to make **web-scale computing easier** for developers.
- In **Minutes** – you can **obtain a new server** and boot it.
- **Quickly scalable** (scale up and scale down).
- Features - Elastic Load Balancing, Auto scaling, Amazon Cloud Watch.

Amazon EC2 reduces the time required to obtain and boot new server instances to minutes, allowing you to quickly scale capacity, both up and down, as your computing requirements change. Amazon EC2 provides various features like Amazon Elastic Load Balancing, Auto scaling, Amazon CloudWatch for monitoring to developers the tools to build failure resilient elastic applications and isolate themselves from common failure scenarios. [18]

5.1.2 Amazon Simple Storage Service (Amazon S3)-

Amazon S3 is storage for the Internet. Amazon S3 provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, fast, inexpensive data storage infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

5.1.3 Amazon Elastic MapReduce-

Amazon Elastic MapReduce is a web service that enables businesses, researchers, data analysts, and developers to easily and cost-effectively process vast amounts of data. It utilizes a hosted Hadoop framework running on the web-scale infrastructure of Amazon Elastic Compute Cloud (Amazon EC2) and Amazon Simple Storage Service (Amazon S3).[18]

5.2 Experimental Setup –

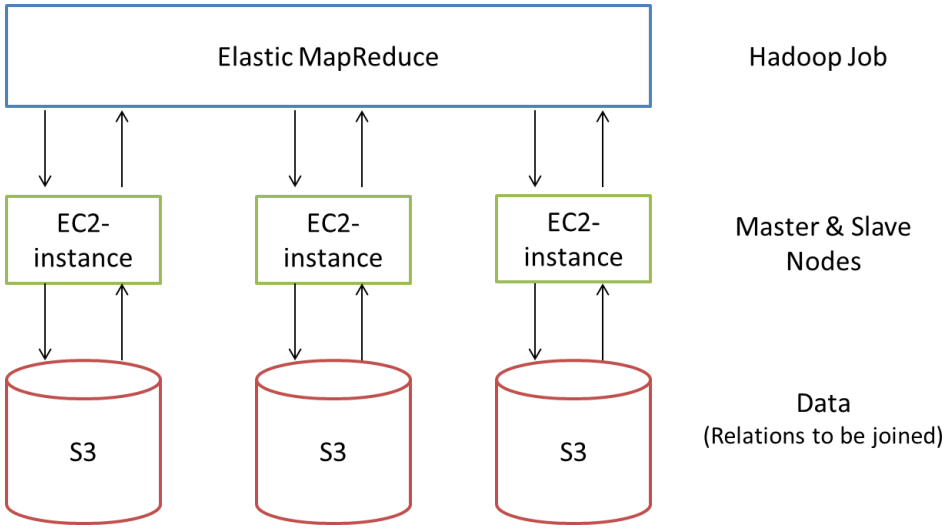

Figure 13. Experimental Setup

Figure 13 depicts the experimental setup. Data for performing the experiment is stored in the AWS S3 buckets, S3 provides storage capacity similar to the hard disc. Hadoop environment is provided by the Elastic MapReduce, jar file of the new job is deployed on the Elastic MapReduce.

To execute each job we need to specify number of EC2 instances to be used, based on the requirement the EC2 instances becomes available before execution of MapReduce job. These EC2 instances get the data from S3 buckets for the processing.

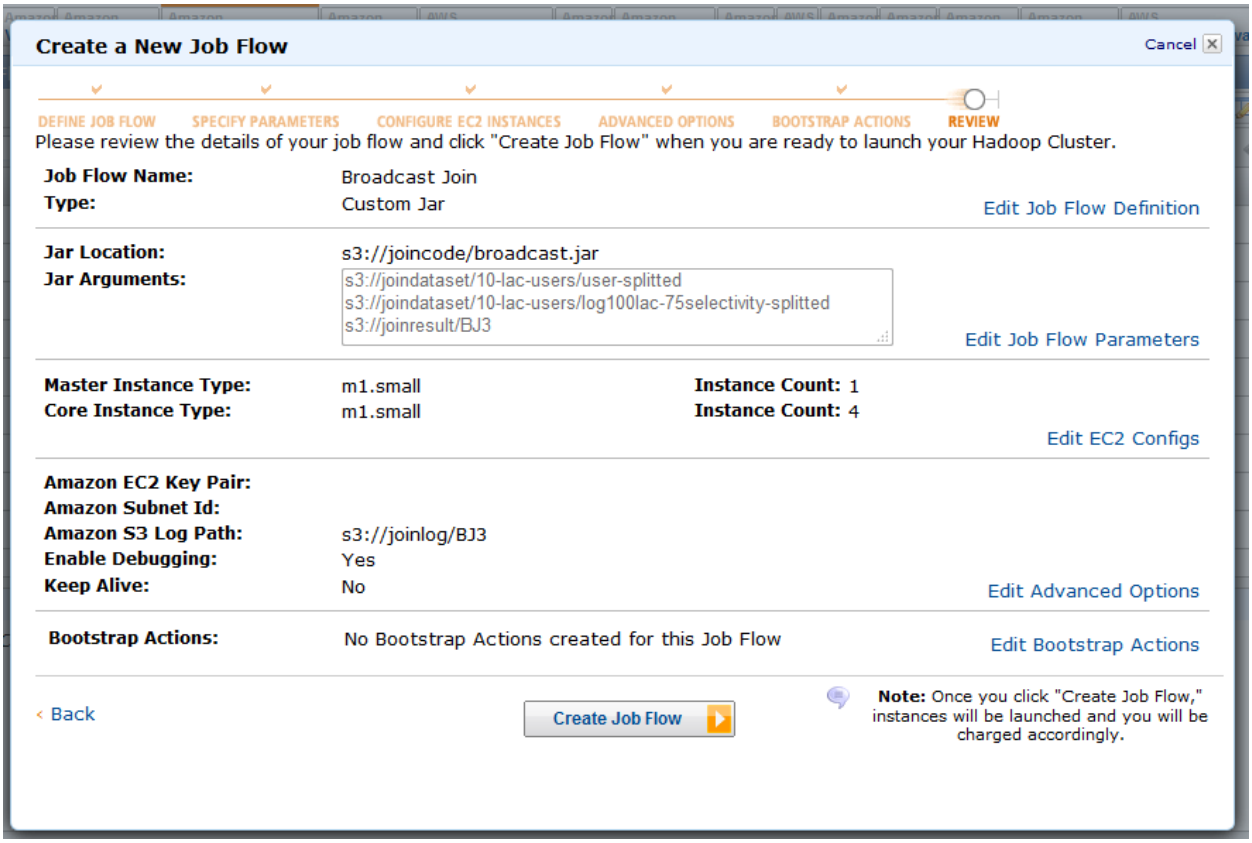

Figure 14. Elastic MapReduce Job

Figure 14, shows a sample job flow created for the execution of Broadcast join with 1 master and 4 slave nodes.

5.3 Dataset –

Two relations user and log are considered for join. User relation is small relation used for broadcast and the log relation contained more tuples.

User

User ID	Age	Email ID	Country

Log

User ID	Click Action	Ip Address	URL

The dataset was generated by using vbscript, various data was randomly assigned to the tuples from a set of values. Dataset was split into smaller files, each file containing 1,00,000 entries. All this data was uploaded to the S3 bucker named s3://joindataset.

For testing performance of semi join, we have generated dataset with different amount of selectivity- 10%, 25%, 75%. 10% selectivity means that only 10% records from the User relation contribute to the final result, 90% tuples are not useful for the final result.

5.4 Results –

5.4.1 Comparison of Default Join and Broadcast Join

Experiment#	Nodes		Tuples		Time Required (in milli-sec)	
	Master	Slave	User	Log	Default Join	Broadcast join
1	1	2	5,00,000	7,00,000	176589	130344
2	1	4	5,00,000	15,00,000	188692	134254
3	1	4	5,00,000	25,00,000	238164	152238
4	1	4	10,00,000	50,00,000	284396	207016
5	1	4	10,00,000	1,00,00,000	406418	266595
6	1	4	10,00,000	1,50,00,000	529810	378064
7	1	10	10,00,000	50,00,000	192933	156184
8	1	10	10,00,000	1,00,00,000	257297	208333
9	1	10	10,00,000	1,50,00,000	391207	254261

Table 3. Result of Comparison of Default Join and Broadcast Join

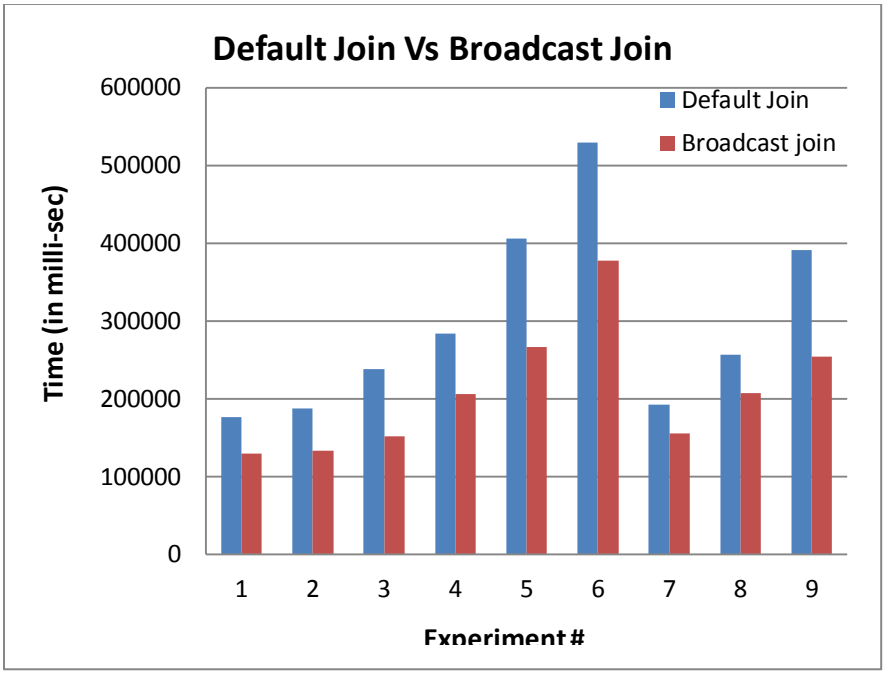

Figure 15. Comparison graph of Default Join and Broadcast Join processing time.

Observation: In all the experiments it is observed that the Broadcast join always took less time as compared to the Default Hadoop join.

5.4.2 Comparison of Broadcast Join and Semijoin (Optimized Broadcast Join)

Expt. No.	Nodes		Selectivity	Tuples		Time Required (in milliseconds)	
	Master	Slave		User	Log	Broadcast join	Semi Join
1	1	10	100%	10,00,000	1,00,00,000	208333	477895
2	1	10	75%	10,00,000	1,00,00,000	199741	447728
3	1	10	25%	10,00,000	1,00,00,000	193040	387249
4	1	10	10%	10,00,000	1,00,00,000	219046	388568

Table 4. Result of Comparison of Broadcast Join and Semi Join.

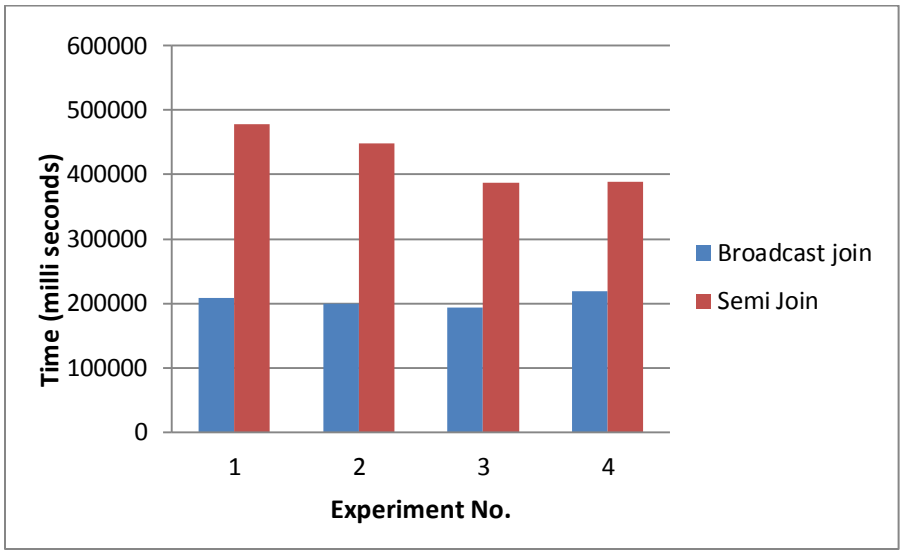

Figure 16. Comparison graph of Semi Join and Broadcast Join processing time.

Observation: Experiments were conducted for different selectivity of user relation tuples. Selectivity 10% means only ten percent of the User tuples were contributing to the join result. The time taken by Semi join was found to be more than the Broadcast join in all the experiments. Actually at the lower value of selectivity the performance should have increased. But due to the more number of MapReduce jobs involved, semi join approach took more time to complete the join execution.

5.4.3 Comparison of Normal Broadcast Join & Dynamic hash table Broadcast

Expt No.	Nodes		Tuples		Time Required (in milliseconds)	
	Master	Slave	R1 size	R2 split size	Broadcast Join	Dynamic Hash Table
1	1	4	1,00,000 (3178 KB)	4,00,000 (8333 KB)	98634	96745
2	1	4	1,00,000 (10857KB)	4,00,000 (8333 KB)	103010	98602
3	1	5	10,00,000 (110516KB)	100,00,000 (8333 KB)	Out of Memory Exception	540045

Table 5. Result of Comparison of Broadcast Join and Dynamic Hash table broadcast.

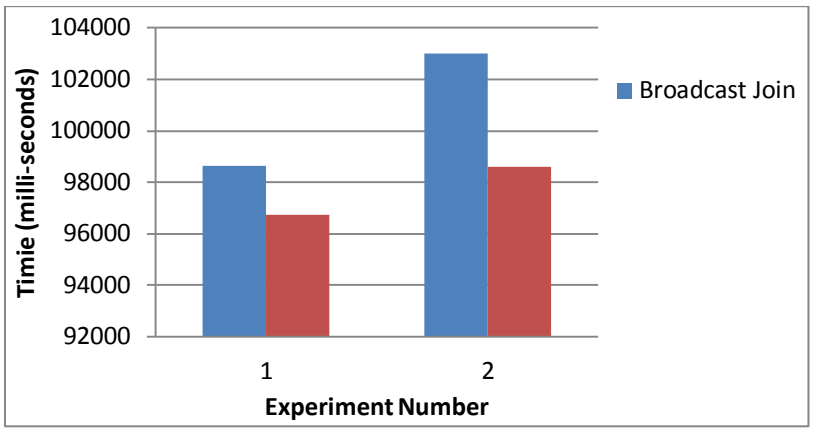

Figure 17. Comparison graph of Broadcast Join & Dynamic Hash table processing time.

Observation: Dynamic hash table Broadcast join, builds the hash table based on the size of the input split and the broadcasted relation. It builds the hash table for the smaller data. Experiments with different R1-size and R2-split size were conducted.

For the first experiment the size of R1 relation which is broadcasted was less than the input split of R2, so the hash table will be built for the R1 relation by both relations, so the time required is near about same.

But, for the second experiment the size of R1 is greater than the size of input split of R2, so the Normal Broadcast join will create hash table for the R1 relation (takes more time for creation and probing of the hash table) and Dynamic Broadcast Join will create the hash table for the split of R2 relation (takes less time for creation and probing of the hash table). And as expected the time for completion is less in case of Dynamic Broadcast Join.

In the third experiment it is observed that the Normal Broadcast join could not work because the hash table for the 10,00,000 tuples could not be built as the size of relation R1 is 107MB. So, JVM throws out of memory exception. So, it is better to use Dynamic hash table creation approach for the broadcast join.

5.4.4 Comparison of Broadcast join and Zip Broadcast Join

Expt #	Nodes		Tuples		Time Required (in milliseconds)		
	Master	Slave	User	Log	Default Join	Broadcast Join	Zip-Broadcast Join
1	1	10	10,00,000	50,00,000	192933	156184	135514
2	1	10	10,00,000	1,00,00,000	257297	208333	195752
3	1	10	10,00,000	1,50,00,000	391207	254261	243593

Table 6. Result of Comparison of Broadcast Join and Zip Broadcast join.

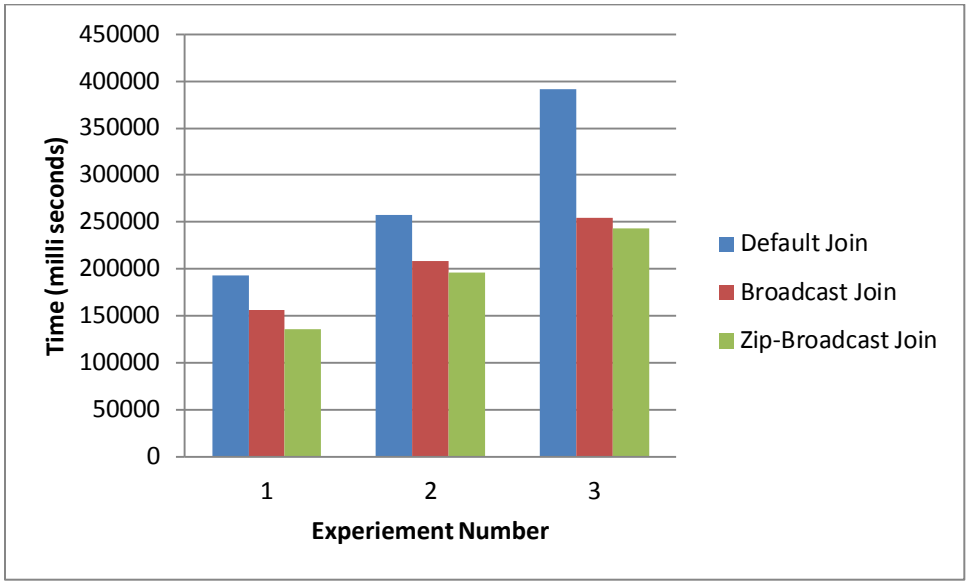

Figure 18. Comparison graph of Broadcast Join and Zip Broadcast join processing time.

Observation: Zip broadcast join transmits the compressed version of the smaller relation to the working nodes. From the above experiments it can be observed that the Zip broadcast join takes less time than the broadcast join and default join, because of less network communication.

5.4.5 Comparison of Default join and Hash Broadcast Join

	Nodes		Tuples		Time (in milliseconds)	
Expt #	Master	Slave	User	Log	Default Join	Hash Broadcast Join
1	1	10	10,00,000	1,00,00,000	296331	291220
2	1	10	10,00,000	1,50,00,000	413040	403476

Table 7. Result of Comparison of Default join and Hash Broadcast join.

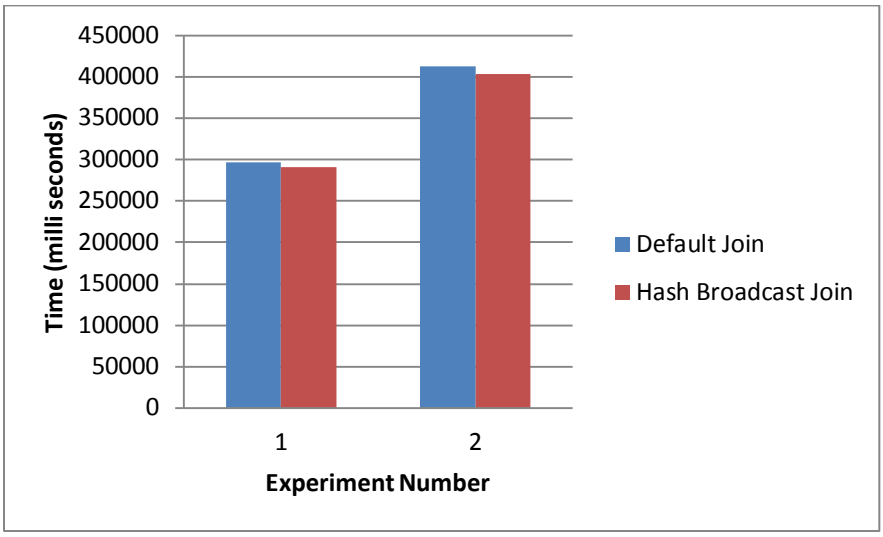

Figure 19. Comparison graph of Broadcast Join and Hash Broadcast join processing time.

Observation: In Hash broadcast join the disc based hash table of the smaller relation is compressed and then transmitted to the working nodes. Here we are comparing it with the Default join because the Hash Broadcast Join is designed for the situations when the normal Broadcast join fails due to lack of the heap space. Results show that the Hash broadcast join performed better than the Default join in both experiments.

CHAPTER 6 Proposed Join Algorithm Selection Strategy

Based on the results of experiments described above, we have proposed a join algorithm selection strategy, depicted as a decision tree in Figure 20. One such strategy was proposed in [5] based on tradeoff between few join algorithms and preprocessing of data, but we have considered more number of join algorithms and optimizations described above.

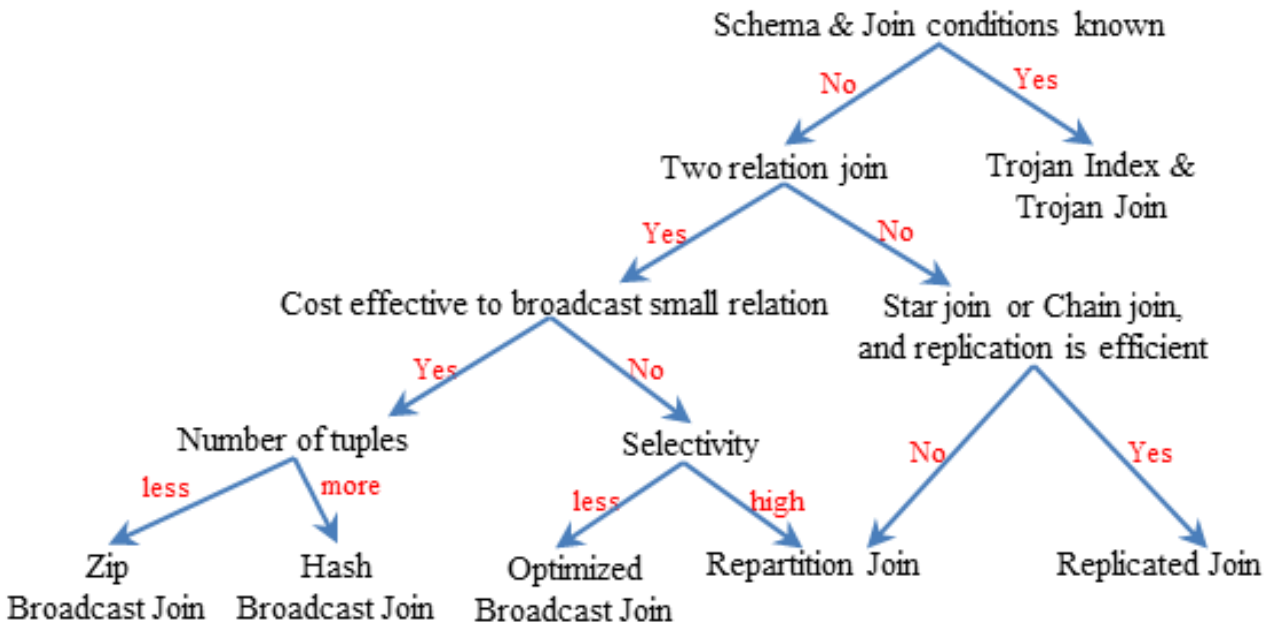

Figure 20. Join Selection Strategy.

The join algorithm selection stratergy is described below in details-

- In case of Prior knowledge of schema and join condition, Trojan index and Trojan join should be used for better performance.
- Multiway join (Replicated join) would be efficient when star or chain join is performed between more than two relations, otherwise performing cascade of two-way join would be better.

- For join between two relations such that one relation is smaller and efficient to transmit over network then Broadcast join would be a good choice. It would reduce the network traffic and time required for the transmission is also less.
- Based on the number of tuples in the smaller relation, we perform either Zip Broadcast join or Hash Broadcast join. If we can construct in memory hash table for the relation then Zip broadcast join is preferred else disc based hash table could be constructed and broadcasted using Hash Broadcast join.
- When selectivity of relation is less (less number of tuples of a relation contributes to join result) then prefer Optimized Broadcast Join or Semi join. One more condition to be checked is the size of tuple in the smaller relation, if the size of tuple is considerably large then performing semi join would be better.

This join selection is useful when we know the nature of the data we are going to receive as the input for the join operation. It would allow us to make proper decision in selecting the most appropriate algorithm for the join operation. In case we don't have any knowledge about the input, then it would be better to perform the Default Hadoop join as it involves single MapReduce job and its implementation is simple. But, when details about input are available then it's better to select good algorithms from various available options.

Conclusion

In this thesis we have described various join query processing algorithms for MapReduce environment. These algorithms are compared on basis of number of jobs required, communication overhead and time required for execution. Based on our observations we have proposed three optimization techniques- Dynamic hash table creation, Compressed Broadcast and Disc Based Hash table Broadcast- to improve the performance of the Broadcast join. Our goal was to reduce the cost of communication and reduce the time of join execution. These optimizations could be utilized when performing join between large data sets. Experiments proved that these optimizations provided better performance than the original Broadcast join. Dataset used for performing experiments contained millions of tuples and experiments were conducted on 5 to 10 nodes. Not a single algorithm described in thesis is ideal in all situations; each algorithm has its own area of application, where it provides better performance. So, based on study of various join processing algorithms a join selection strategy is proposed, which would help in the selection of right algorithm. Overall three things are done in this thesis – Survey of various join query processing algorithms, proposed techniques for optimization of the Broadcast join, proposed a join algorithm selection strategy.

Future Work

Join algorithms described in this thesis focus on the data sets stored in the row wise format. In future this work can be extended for the optimization of join query processing for the column wise data. This work can be utilized for further improvement of join query execution in MapReduce environment. Any of the algorithms above does not provide generic solution which will give optimized results in all cases. These algorithms can be applied dynamically based on the various parameters like size of relation; knowledge of schema etc. So a query optimizer can be designed based on various join processing techniques. Optimizations discussed in this thesis could be applied to Replicated join algorithm, because replicated join involves lots of replication of the data. Trojan join could be optimized further by storing the data in compressed format and metadata in un-compressed format. Further improvement to the semi join could be done by maintaining an index, which would increase the performance of the semi join step. Another variant of MapReduce that is Map-Reduce-Merge could also be utilized for the join execution, so optimization of join could be done for Map-Reduce-Merge environment also.

References

1. Jeffrey, D., Sanjay G.: MapReduce: Simplified Data Processing on Large Clusters. In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation. Pages: 137-149 (2004).
2. Apache Foundation – Hadoop Project, http://hadoop.apache.org
3. Miao, J., Ye, W.: Optimization of Multi-Join Query Processing within MapReduce. In Universal Communication Symposium (IUCS), 2010 4th International. Pages: 77-83 (2010).
4. Foto, N.A., Jeffrey, D.U.: Optimizing Multiway Joins in a Map-Reduce Environment. In IEEE Transactions on Knowledge and Data Engineering, Vol. 23, No. 9. Pages: 1282-1298 (2011).
5. Spyros, B., Jignesh M.P., Vuk, E., Jun, R., Eugene, J., Yuanyuan, T.: A Comparison of Join Algorithms for Log Processing in MapReduce. In SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA., ACM. Pages: 975-986 (2010).
6. Jens, D., Jorge-Arnulfo, Q., Alekh, J., Yagiz, K., Vinay, S., Jorg, S.: Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing). In Proceedings of the VLDB Endowment, Vol. 3, No. 1. Pages: 518-529 (2010).
7. Sai, W., Feng, L., Sharad, M., Beng, C.: Query Optimization for Massively Parallel Data Processing. In Symposium on Cloud Computing (SOCC) 2011 - Cascais, Portugal, ACM. Article No. 12 (2011).
8. Hung-chih, Y., Ali, D., Ruey-Lung, H., D. Scott, P.: Map-Reduce-Merge: Simplified Relational Data Processing on Large Clusters. In SIGMOD'07, June 12–14, 2007, Beijing, China. ACM. Pages: 1029-1040 (2007).
9. Minqi, Z., Rong, Z., Dadan, Z., Weining, Q., Aoying, Z.: Join Optimization in the MapReduce Environment for Column-wise Data Store. In 2010 Sixth International Conference on Semantics, Knowledge and Grids. IEEE. Pages: 97-104 (2010).
10. Konstantin, S., Hairong, K., Sanjay, R., Robert, C.: The Hadoop Distributed File System. MSST '10 Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST). Pages: 1-10 (2010).
11. Mapper process : http://hadoop.apache.org/mapreduce/docs/r0.22.0/mapred_tutorial.html
12. Distributed Cache: http://docs.amazonwebservices.com/ElasticMapReduce/latest/DeveloperGuide/DistributedCache.html
13. Zip file format: http://en.wikipedia.org/wiki/ZIP_(file_format)
14. Tar file format: http://en.wikipedia.org/wiki/Tar_(file_format)
15. Jar file format : http://en.wikipedia.org/wiki/JAR_file
16. Comparison of archive formats : http://en.wikipedia.org/wiki/Comparison_of_archive_formats
17. JDBM library - http://code.google.com/p/jdbm2/
18. Amazon Web Services - http://aws.amazon.com/documentation/

Appendix - A

VB Script – to generate User and Log data.

```
' Function to generate random user tuples
' List of countries & cities is provided, one of them is selected randomly
' Age, email id and IP address are generated for each user

Sub generateUserNames()

  numberOfUsers = 1000000

  Dim country(0 To 16) As String
  country(0) = "india (asia)"
  country(1) = "nepal (asia)"
  country(2) = "us (usa)"
  country(3) = "uk (europe)"
  country(4) = "pak (asia)"
  country(5) = "bangla (asia)"
  country(6) = "korea (continent-1)"
  country(7) = "africa (african)"
  country(8) = "germny (asia)"
  country(9) = "lanka (asia)"
  country(10) = "saudi (asia)"
  country(11) = "dubai (asia)"
  country(12) = "rusia (rusia)"
  country(13) = "china (asia)"
  country(14) = "france (french)"
  country(15) = "japan (asia)"
  country(16) = "bhutan (asia)"

  Dim city(0 To 16) As String
  city(0) = "delhi"
  city(1) = "kathmandu"
  city(2) = "new york"
  city(3) = "london"
  city(4) = "karachi"
  city(5) = "dhaka"
  city(6) = "koreacity"
  city(7) = "johanceberg"
  city(8) = "hitler"
  city(9) = "srilanka"
  city(10) = "baghdad"
  city(11) = "dubaicity"
  city(12) = "rusiacity"
  city(13) = "bejing"
  city(14) = "francecity"
  city(15) = "tokyo"
  city(16) = "bhutancity"


    ' format
```

```
  ' username  age email country
  For i = 1 To numberOfUsers
     Range("A" & i).Value = "usr" & i
     age = CInt(Int((80 - 18 + 1) * Rnd() + 18))
     Range("B" & i).Value = age
     Range("C" & i).Value = "email" & i & "@domain.com"
     Index = CInt(Int((16 + 1) * Rnd()))
     Range("D" & i).Value = country(Index)
     Range("E" & i).Value = city(Index)
     Range("F" & i).Value = Round((9999999999# - 9010200000# + 1) * Rnd() +
                            9010200000#)
     Range("G" & i).Value = CInt(Int((255 - 10 + 1) * Rnd() + 10)) & "." &
                            CInt(Int((255 - 0 + 1) * Rnd() + 0)) & "." &
                            CInt(Int((255 - 0 + 1) * Rnd() + 0)) & "." &
                            CInt(Int((255 - 0 + 1) * Rnd() + 0))

  Next i

End Sub

' Function to generate log entries
' 10,00,000 entries are created in each excel sheet
' Various possible click actions are provided along with their urls
' Dynamically a user is associated with a click action and url along with ip
address


Sub generateLogEntries()

  numberoflog = 3000000
  perSheet = 1000000
  numberOfUsers = 500000

  Dim action(0 To 16) As String
  action(0) = "login"
  action(1) = "share photo"
  action(2) = "uploade photo"
  action(3) = "edit profile info"
  action(4) = "change profile pic"
  action(5) = "comment"
  action(6) = "like"
  action(7) = "unlike"
  action(8) = "create album"
  action(9) = "accept friend request"
  action(10) = "remove friend"
  action(11) = "send friend request"
  action(12) = "deactivate account"
  action(13) = "logout"
  action(14) = "reset password"
  action(15) = "add phone number"
  action(16) = "change status"

  Dim url(0 To 16) As String
  url(0) = "login.html"
    url(1) = "share_photo.html"
```

```
    url(2) = "uploade_photo.html"
    url(3) = "edit_profile_info.html"
    url(4) = "change_profile_pic.html"
    url(5) = "comment.html"
    url(6) = "like.html"
    url(7) = "unlike.html"
    url(8) = "create_album.html"
    url(9) = "accept_friend_request.html"
    url(10) = "remove_friend.html"
    url(11) = "send_friend_request.html"
    url(12) = "deactivate_account.html"
    url(13) = "logout.html"
    url(14) = "reset_password.html"
    url(15) = "add_phone_number.html"
    url(16) = "change_status.html"

  Dim i As Double


  For counter = 0 To numberoflog - 1

     If counter Mod perSheet = 0 Then
         sheetNum = (counter / perSheet) + 1
         Sheets(sheetNum).Select
         i = 0
     End If

     UserId = Round((numberOfUsers + 1) * Rnd())
     Range("A" & i + 1).Value = "usr" & UserId

     Range("B" & i + 1).Value = action(CInt(Int((16 - 0 + 1) * Rnd() + 0)))

     ipaddress = CInt(Int((255 - 10 + 1) * Rnd() + 10)) & "." &
                 CInt(Int((255 - 0 + 1) * Rnd() + 0)) & "." &
                 CInt(Int((255 - 0 + 1) * Rnd() + 0)) & "." &
                 CInt(Int((255 - 0 + 1) * Rnd() + 0))

     Range("C" & i + 1).Value = ipaddress

     Range("D" & i + 1).Value = "http://www.mywebsite.domain/" &
                                url(CInt(Int((16 - 0 + 1) * Rnd() + 0)))

     i = i + 1
  Next counter

End Sub
```

Appendix - B

Code

1. Broadcast Join

JOBSETUP.java

```
/*
Package – broadCastJoin
Class – Main
Purpose – Main class to set the job parameters
*/

// smallFilePath - path of the smaller relations files
// inputPath - path of the larger relation files
// outputPath - path of the output directory
// start - start time of the job
// end - end time of the job

package broadCastNORMAL;

public class JOBSETUP {
  public static void main(String... args) throws Exception {

    // set path
    Path smallFilePath=new Path(args[0]);
    Path inputPath=new Path(args[1]);
    Path outputPath=new Path(args[2]);

    //get start time
    long start = new Date().getTime();

    //create new job configuration
    Configuration conf = new Configuration();

    FileSystem fs = smallFilePath.getFileSystem(conf);

    FileStatus smallFilePathStatus = fs.getFileStatus(smallFilePath);

    // add all files from the directory to the distributed cache
    if(smallFilePathStatus.isDir()) {
      for(FileStatus f: fs.listStatus(smallFilePath)) {
            System.out.println("cached file="+f.getPath().getName());

           DistributedCache.addCacheFile(f.getPath().toUri(), conf);
```

```
    }
  } else {
        DistributedCache.addCacheFile(smallFilePath.toUri(), conf);
  }

  Job job = new Job(conf);

  job.setJarByClass(JOBSETUP.class);
  job.setMapperClass(BroadcastJoin_Normal.class);
  job.setInputFormatClass(KeyValueTextInputFormat.class);

  //since it is Map only job, set the number of Reducers to zero
  job.setNumReduceTasks(0);

  outputPath.getFileSystem(conf).delete(outputPath, true);

  //set input and output path
  FileInputFormat.setInputPaths(job, inputPath);
  FileOutputFormat.setOutputPath(job, outputPath);

  //wait till job completes
  job.waitForCompletion(true);

  //mark the end time
  long end = new Date().getTime();

  //display the time taken by job
        System.out.println("Job took "+(end-start) + " milliseconds");

 }

}
```

BroadcastJoin_Normal.java

```
/*
Package – broadCastJoin
Class – BroadcastJoin_Normal
Purpose –  Adds smaller relation to cache and then perform broadcast join.
*/

package broadCastNORMAL;

public class BroadcastJoin_Normal  extends Mapper<Object, Object, Object, Object> {

 private Map<Object, List<Pair>> cachedRecords = new HashMap<Object, List<Pair>>();

 private Path[] cachedFiles;

 //Setup Method - called before Map method
 // Purpose - to cache the tuples of the smaller relation
 @Override
 protected void setup(Context context)throws IOException, InterruptedException
 {
  cachedFiles = DistributedCache.getLocalCacheFiles(context.getConfiguration());

  DistributedCacheFileReader reader = getDistributedCacheReader();

  //read each file
  for (Path distFile : cachedFiles) {

   File distributedCacheFile = new File(distFile.toString());

   reader.init(distributedCacheFile);

   //add each pair to cache
   for (Pair p : (Iterable<Pair>) reader)
         cahceThePair(p);

   reader.close();
  }
}

 //Cache the key value pair
 private void cahceThePair(Pair pair) {

  List<Pair> values = cachedRecords.get(pair.getKey());
  if (values == null) {
   values = new ArrayList<Pair>();
   cachedRecords.put(pair.getKey(), values);
  }
```

```
  values.add(pair);
 }

 // Map method - Get a tuple from the larger relation, and perform join for that tuple
 @Override
 protected void map(Object key, Object value, Context context)
    throws IOException, InterruptedException {

            Pair pair = readFromInputFormat(key, value);
    joinThisPair(pair, context);

 }

 // For a input tuple from large relation, find a matching tuple from the hash table.
 // If match found output the joined tuple
 public void joinThisPair(Pair inputPair, Context context)
    throws IOException, InterruptedException {

  List<Pair> cached = cachedRecords.get(p.getKey());
  if (cached != null)
    for (Pair cachedPair : cached) {

            StringBuilder sb = new StringBuilder();
            if (inputPair.getData() != null) {
                    sb.append(inputPair.getData());
            }
            sb.append("\t");
            if (cachedPair.getData() != null) {
                    sb.append(cachedPair.getData());
            }
            Pair result=new Pair(new Text(inputPair.getKey().toString()),new Text(sb.toString()));

       if (result != null)
            context.write(result.getKey(), result.getData());
    }

 }

}
```

2. Dynamic Hash Table Creation

broadcastJoin_Dynamic.java

```
/*
Package – broadCastJoin
Class – broadcastJoin_Dynamic
Purpose –  Create hash table dynamically based on the size of the input split and the cached file size.
*/

public class broadcastJoin_Dynamic extends Mapper<Object, Object, Object, Object> {

  private boolean inputSplitLarger;
  private Path[] cachedFiles;
  private Map<Object, List<Pair>> cachedRecords = new HashMap<Object, List<Pair>>();

  //Setup Method - called before Map method
  // Purpose - to cache the tuples of the either the smaller relation or the input split of larger relation -
one with the smaller size
  @Override
  protected void setup(Context context) throws IOException, InterruptedException {

    cachedFiles = DistributedCache.getLocalCacheFiles(context.getConfiguration());

    int cacheSize = 0;

    //calculate size of smaller relation files
    for (Path distFile : cachedFiles) {
      File distributedCacheFile = new File(distFile.toString());
      cacheSize += distributedCacheFile.length();
    }

    //get the input split size
    if(context.getInputSplit() instanceof FileSplit) {

      FileSplit split = (FileSplit) context.getInputSplit();

      long splitSize = split.getLength();

      inputSplitLarger = (cacheSize < splitSize);

    } else {
      inputSplitLarger = true;
    }

    // if smaller relation file is smaller then cache its tuples
    if (inputSplitLarger) {
```

```
    for (Path distFile : cachedFiles) {
      File distributedCacheFile = new File(distFile.toString());
      reader.init(distributedCacheFile);

      for (Pair p : (Iterable<Pair>) reader) {
        addToCache(p);
      }

      reader.close();
    }
  }
}

// Map - called once for each tuple in the input split of larger relation
@Override
protected void map(Object key, Object value, Context context)
    throws IOException, InterruptedException {

  Pair pair = readFromInputFormat(key, value);

  if (inputSplitLarger) {
    //if input split is large then perform join for each tuples in larger relation
    joinThisPair(pair, context);
  } else {
    //if input split is small then add the pair to cache
    cacheThePair(pair);
  }
}

// cache the pair for future use
private void cacheThePair(Pair pair) {
  List<Pair> values = cachedRecords.get(pair.getKey());
  if (values == null) {
    values = new ArrayList<Pair>();
    cachedRecords.put(pair.getKey(), values);
  }
  values.add(pair);
}

//find the matching tuple from the hash table
public void joinThisPair(Pair p, Context context)
    throws IOException, InterruptedException {
  List<Pair> cached = cachedRecords.get(p.getKey());
  if (cached != null) {
    for (Pair cp : cached) {
      Pair result;
      if (inputSplitLarger) {
```

```
        result = writeResult(p, cp);
       } else {
        result = writeResult(cp, p);
       }
       if (result != null) {
        context.write(result.getKey(), result.getData());
       }
      }
     }
    }

    //prepare the result of join
    public Pair writeResult(Pair inputPair, Pair cachePair) {
                StringBuilder sb = new StringBuilder();
                if (inputPair.getData() != null) {
                  sb.append(inputPair.getData());
                }
                sb.append("\t");
                if (cachePair.getData() != null) {
                  sb.append(cachePair.getData());
                }

                Pair result=new Pair<Text, Text>(new Text(inputPair.getKey().toString()),new
Text(sb.toString()));
                return result;
              }

    // if the input split is smaller, then the join is performed in the cleanup function
    // this function is called when the Map process is complete
    @Override
    protected void cleanup(Context context) throws IOException, InterruptedException {
     if (!inputSplitLarger) {

      for (Path distFile : cachedFiles) {
       File distributedCacheFile = new File(distFile.toString());
       DistributedCacheFileReader reader =getDistributedCacheReader();
       reader.init(distributedCacheFile);

       //For each pair in the cached file, perform the join
       for (Pair p : (Iterable<Pair>) reader) {
        joinThisPair(p, context);
       }
       reader.close();
      }
     }
    }
}
```

3. Zip Broadcast Join

Changes to be done in the existing code

1. In Main class – add code the cache zip file
2. In BroadcastJoin class – no change as the compressed files (zip files) are unzipped as part of the Distributed cache mechanism.

JOBSETUP.java

```
public class JOBSETUP {
  public static void main(String... args) throws Exception {

    // set path
    Path smallFilePath=new Path(args[0]);
    Path inputPath=new Path(args[1]);
    Path outputPath=new Path(args[2]);

    //create new job configuration
    Configuration conf = new Configuration();

    FileSystem fs = smallFilePath.getFileSystem(conf);

    FileStatus smallFilePathStatus = fs.getFileStatus(smallFilePath);

    // add compressed file to the distributed cache
    DistributedCache.addCacheArchive(smallFilePath.toUri(), conf);

    Job job = new Job(conf);

    job.setJarByClass(JOBSETUP.class);
    job.setMapperClass(BroadcastJoin_Normal.class);
    job.setInputFormatClass(KeyValueTextInputFormat.class);

    //since it is Map only job, set the number of Reducers to zero
    job.setNumReduceTasks(0);

    outputPath.getFileSystem(conf).delete(outputPath, true);

    //set input and output path
    FileInputFormat.setInputPaths(job, inputPath);
    FileOutputFormat.setOutputPath(job, outputPath);

    //wait till job completes
    job.waitForCompletion(true);

  }
}
```

4. Disk Based Hash Table Broadcast

Compressed Hash table is broadcasted to all nodes so that the time required for creation of hash table and the limitation of the heap size are eliminated.

Changes required-

1. Creation of Disc based hash table
2. Broadcast compressed hash table
3. Read tuples from disc based hash table

GenerateHashTableJDBM.java

```
package broadcastHASHjdbm;

import jdbm.PrimaryTreeMap;
import jdbm.RecordManager;
import jdbm.RecordManagerFactory;

public class GenerateHashTableJDBM {

    static PrimaryTreeMap<String,String> userTreeMap;

    public static void main(String args[]) throws Exception
    {
            File file=new File(args[0]);
            String hashFileName = args[1] + "HashTable";
            RecordManager recman =
                    RecordManagerFactory.createRecordManager(hashFileName);

            userTreeMap = recman.treeMap("userTreeMap");

            long start=new Date().getTime();
            processThisFile(file);
            long end=new Date().getTime();

            System.out.println("Time for creating Hash table="+(end-start));

            /** Map changes are not persisted yet, commit them (save to disk) */
            recman.commit();

            /** close record manager */
            recman.close();
    }
```

```
        static void processThisFile(File file) throws IOException
        {
                //create a reader
                DistributedCacheFileReader reader =  new TextDistributedCacheFileReader();

                //process this file
                reader.init(file);

                for (Pair p : (Iterable<Pair>) reader)
        {
                        userTreeMap.put(p.getKey().toString(), p.getData().toString());

                        //System.out.println(p.getKey()+" "+p.getData());
        }

                reader.close();
        }
    }
```

BroadcastJoin_HashJDBM.java

```
package broadcastHASHjdbm;

import jdbm.PrimaryTreeMap;
import jdbm.RecordManager;
import jdbm.RecordManagerFactory;

public class BroadcastJoin_HashJDBM  extends Mapper<Object, Object, Object, Object> {

        static PrimaryTreeMap<String,String> userTreeMap;

        private Path[] cachedFiles;
        File hashTableFile=null;

        static PrimaryTreeMap<String,String> userTreeMap;


 // Initilize the hash table to become usable.
 @Override
 protected void setup(Context context) throws IOException, InterruptedException
 {
        long start = new Date().getTime();
        cachedFiles = DistributedCache.getLocalCacheFiles(context.getConfiguration());

   Path hashPath=cachedFiles[0];

   String jdbmHashFile=null;
```

```
  if(hashPath.toString().contains(".zip"))
  {
        //unzip the file
        hashTableFile=new File("/mnt/var/lib/hadoop/mapred/taskTracker/hadoop/jobcache/unzip");

        if(!hashTableFile.exists())
        {
              context.write("File does not exist", hashTableFile.toURI());
              File zipFile=new File(hashPath.toString());
              FileUtil.unZip(zipFile, hashTableFile);
              context.write("Unzip done", "");
        }

        jdbmHashFile=hashTableFile.listFiles()[0].getAbsolutePath()+"/HashTable" ;
  }
  else
  {
        jdbmHashFile=hashPath.getParent().toString()+"/HashTable";
  }

  context.write(jdbmHashFile, "-path");

  RecordManager recman = RecordManagerFactory.createRecordManager(jdbmHashFile);

        userTreeMap = recman.treeMap("userTreeMap");

        long end=new Date().getTime();
        context.write("TIme required for unzip=",(end-start)+" millisecond");

        start=new Date().getTime();
        userTreeMap.get("usr1245");
        end=new Date().getTime();
        context.write("Time to access the Hash table = ",(end-start)+" millisecond");

  }


  @Override
  protected void map(Object key, Object value, Context context)
     throws IOException, InterruptedException {

        //find the matching tuple for each input tuple and write the result
   String cachedValue = userTreeMap.get(key.toString());
        context.write(key, cachedValue+"\t"+value);
  }

}
```

Appendix - C

Steps to create a MapReduce job using Amazon Cloud Computing services

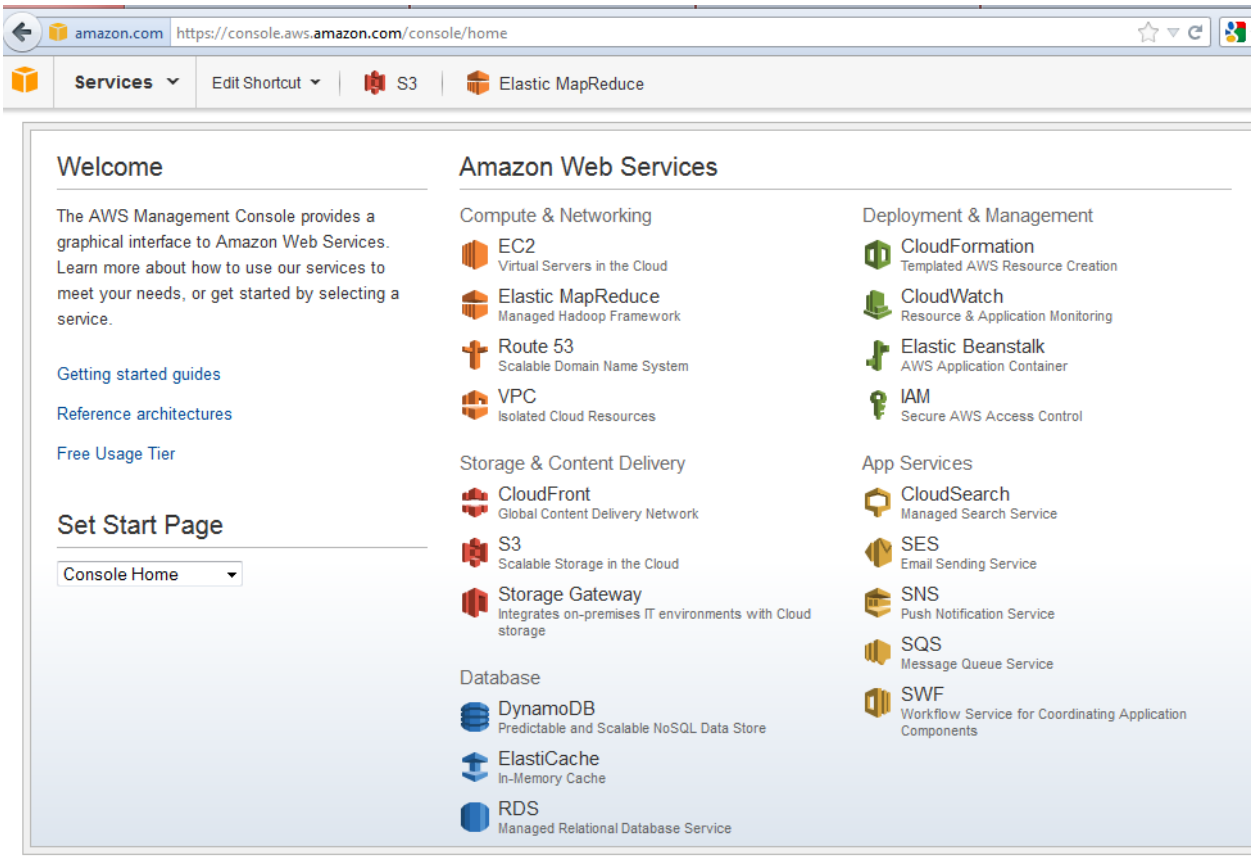

Figure 1. AWS console.

Figure 1, shows the complete view of the Amazon web services console, from where we can control all parameters of a job. Various services of our interest are Simple storage service (S3), Elastic Compute (EC2) and Elastic MapReduce.

In this section various steps involved in the complete execution of the MapReduce job in the Amazon Web service are discussed. First we discuss uploading on the dataset to AWS and then execution of the job over Elastic MapReduce.

1. Upload data and code to S3 :

Figure 2, shows the view of Amazon S3. You can see five buckets at the left hand side. Terminologies used by AWS for S3 related tasks are Buckets, Objects, Regions, Key.

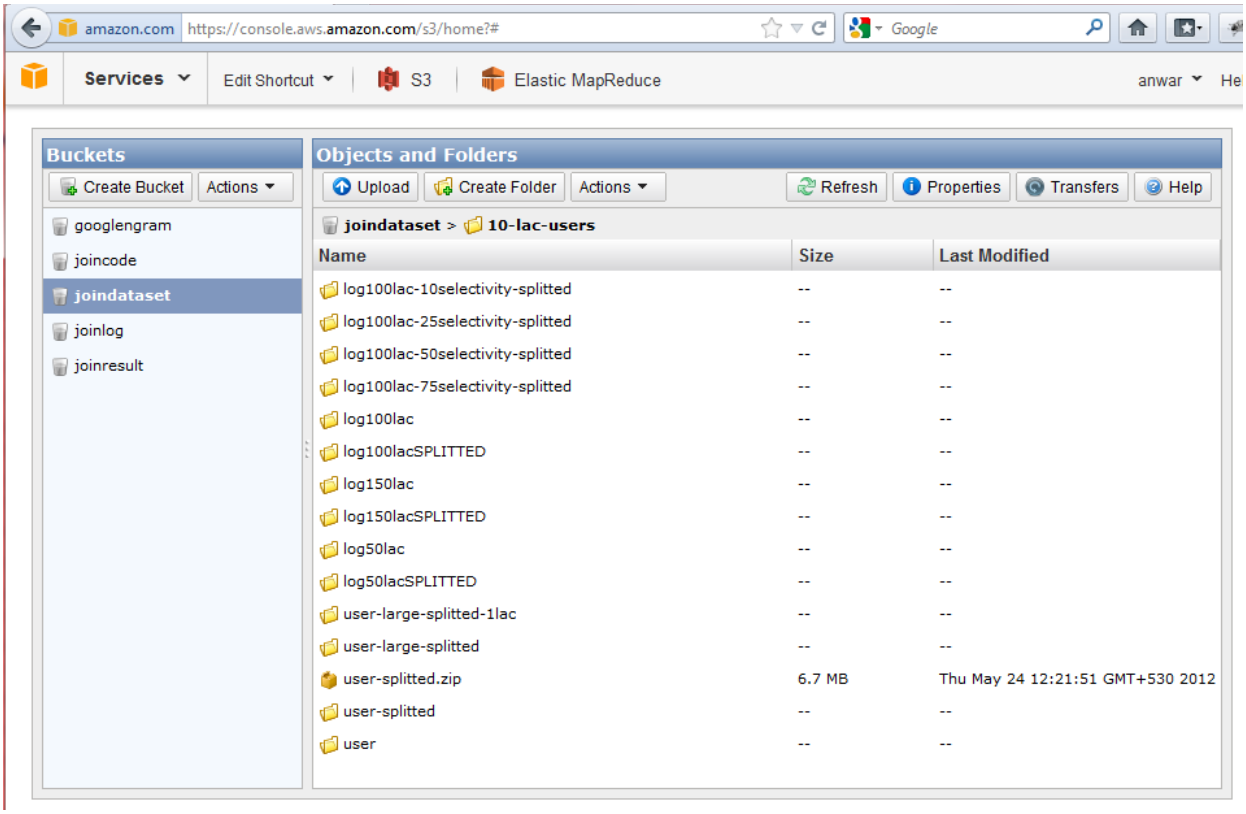

Figure 2. S3 buckets.

- Bucket – It is a container that holds objects of Amazon S3.
  - Each bucket name must be unique across all AWS users.
  - E.g. bucket named "joindataset " can be accessed using URL http:// joindataset.s3.amazonaws.com
  - Buckets belongs to a particular region.
  - Each bucker can contain many files and folders.
  - We have create various buckets
    - Joincode – to store the code for join execution.
    - Joindataset - to store dataset to be joined.
    - Joinlog – stores the log information of each MapReduce job.

- JoinResult – holds the result of join execution.

- Objects – these are the basic entities stored in Amazon S3. In a bucket the Object is uniquely identified using key.
- Key – Unique identifier of the object.
- Regions – These are the geographical location where Amazon stores data, various regions are
    - US Standard
    - US West (Oregon) Region
    - US West (Northern California) Region
    - EU (Ireland) Region
    - Asia Pacific (Singapore) Region
    - Asia Pacific (Tokyo) Region
    - South America (Sao Paulo) Region

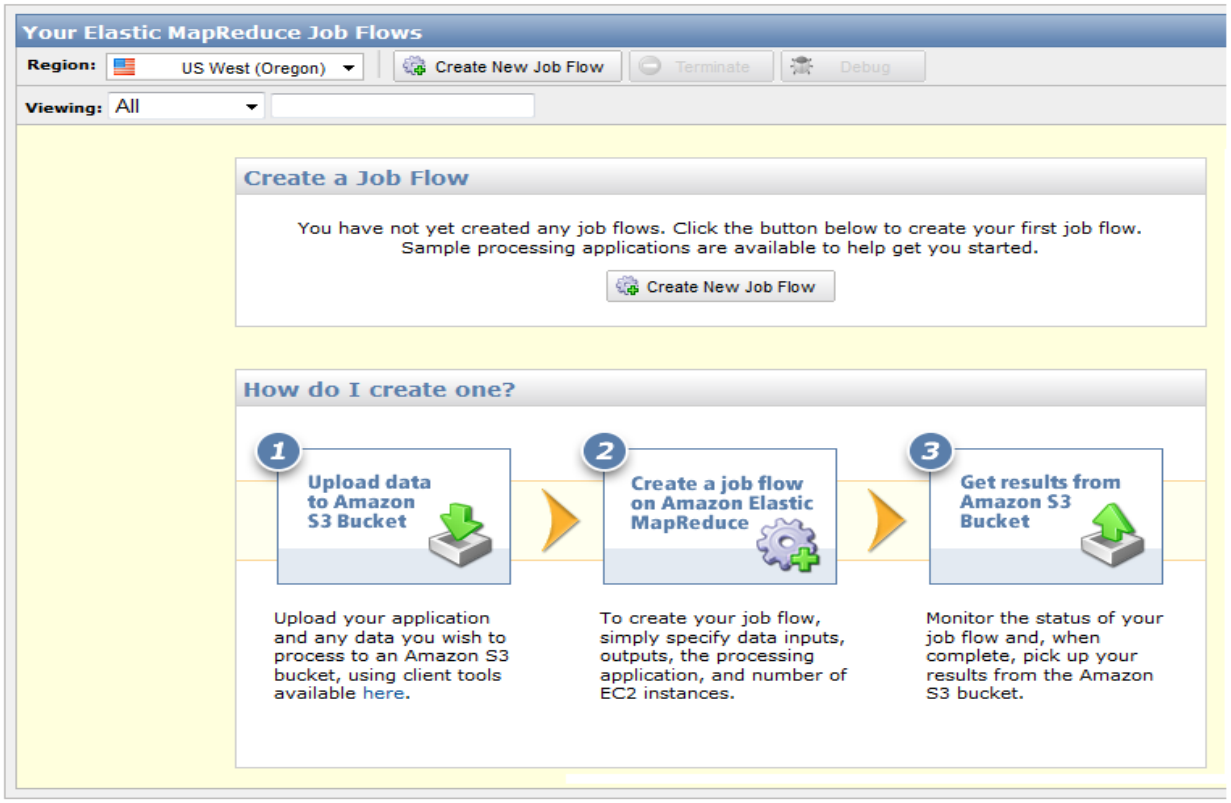

Figure 3. Elastic MapReduce job creation process.

2. **Elastic MapReduce** – Figure 3 shows various steps necessary to run a MapReduce job.

3. **Create a New Job Flow –** provide a name to job flow, it can be descriptive. Also we need to provide which Hadoop version we are going to use, currently Hadoop 0.20.205 is supported. Then select "Run your own application" to execute our code. Figure 4. Shows the corresponding screen.

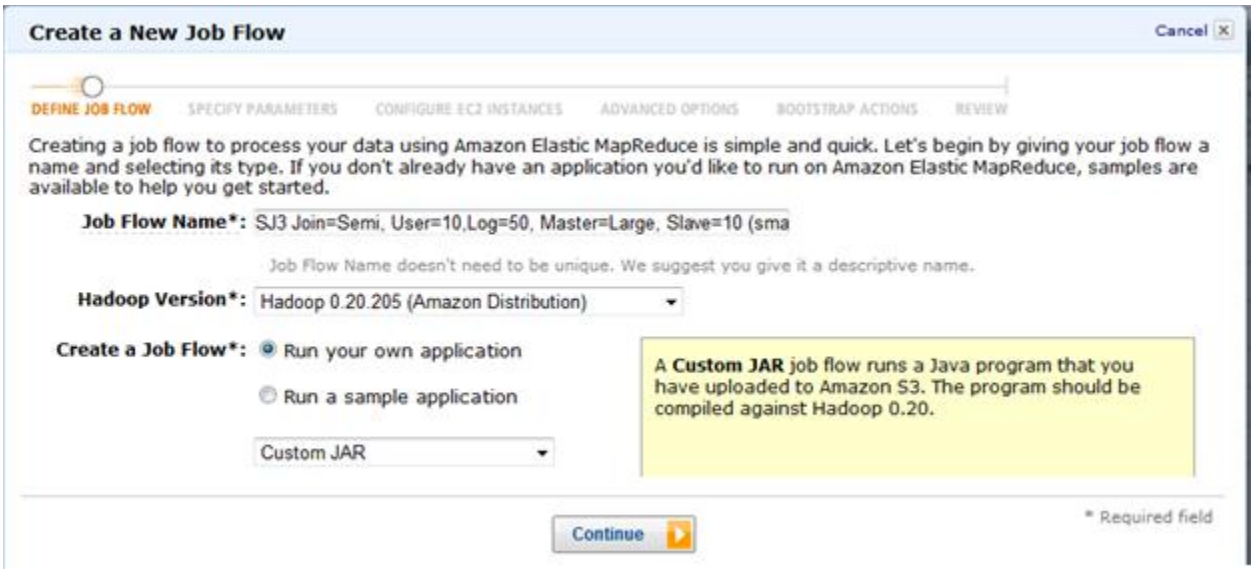

Figure 4. Elastic MapReduce step-1.

4. **Specify Job Parameters -** Here we specify the location of the code, generally we provide a S3 URL – because our code is located in the S3 bucket. Also if there are any arguments that are required for execution then we must provide it in "JAR Arguments" section. Refer figure-5.

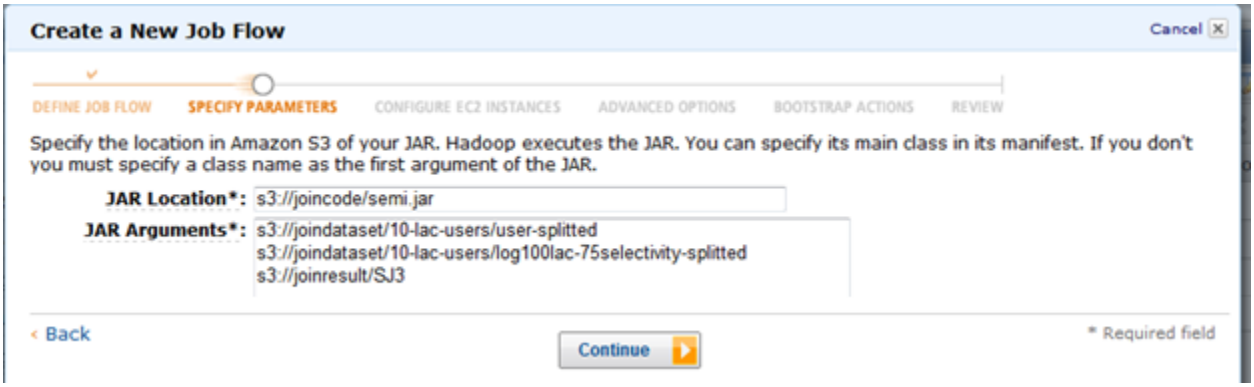

Figure 5. Elastic MapReduce step-2.

5. **Configure EC2 instances –** MapReduce jobs can run on more than one computer. So in this step we specify number of computers and their configuration. We can specify the Instance type of the Master and Slave nodes. Also we need to provide number of Slave nodes that we want to use. Various instance types are – Small, Medium, Large, Extra Large.

   Small instance – 1.7 GB memory , 1 EC2 computer unit, 160 GB storage.

   Medium instance – 3.75 GB memory , 2 EC2 computer unit, 410 GB storage.

   Large instance – 7.5 GB memory , 4 EC2 computer unit, 850 GB storage.

   Extra-large instance – 15 GB memory , 8 EC2 computer unit, 1,690 GB storage.

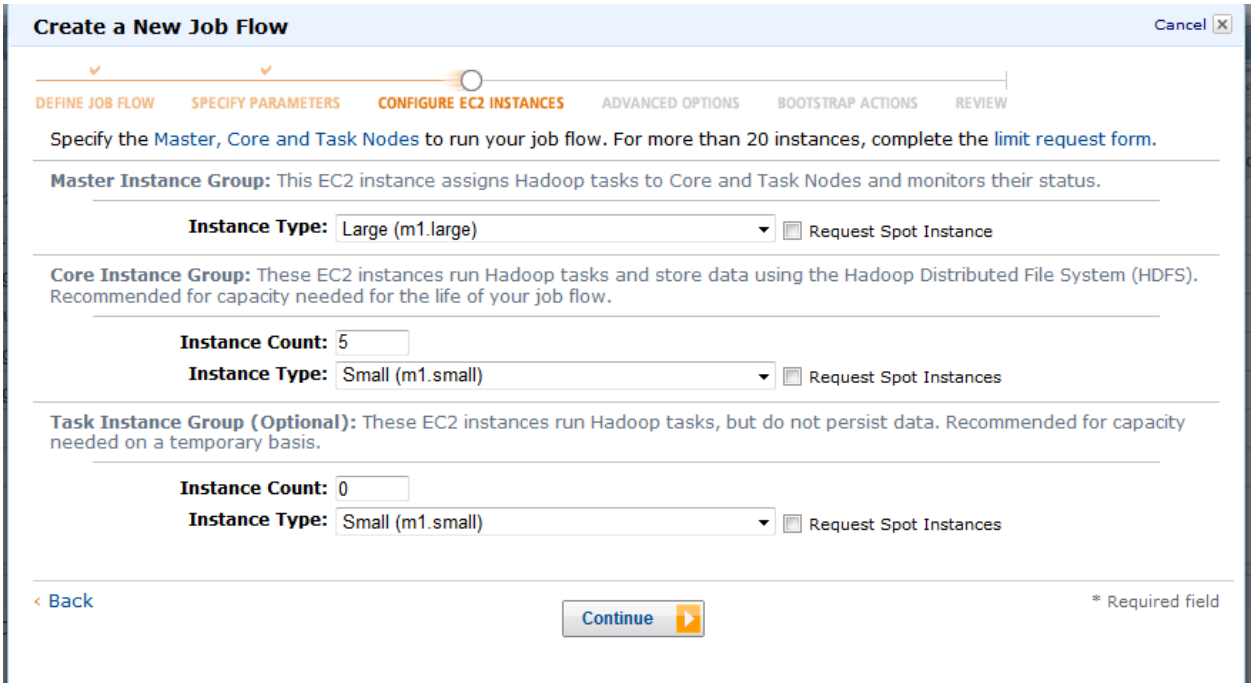

Figure 6. Elastic MapReduce step-3.

6. **Set Advanced Options –** We can specify key pairs , to improve the security, person with the key pair can only access the instance. Also we can mention the S3 location where the log of the MapReduce job would be stored. Debugging can be enabled, which allows us to check the status of each Map and reduce process and number of execution attempts. Keep alive option allows us to make the instance accessible even after completion of the job. Refer figure 7.

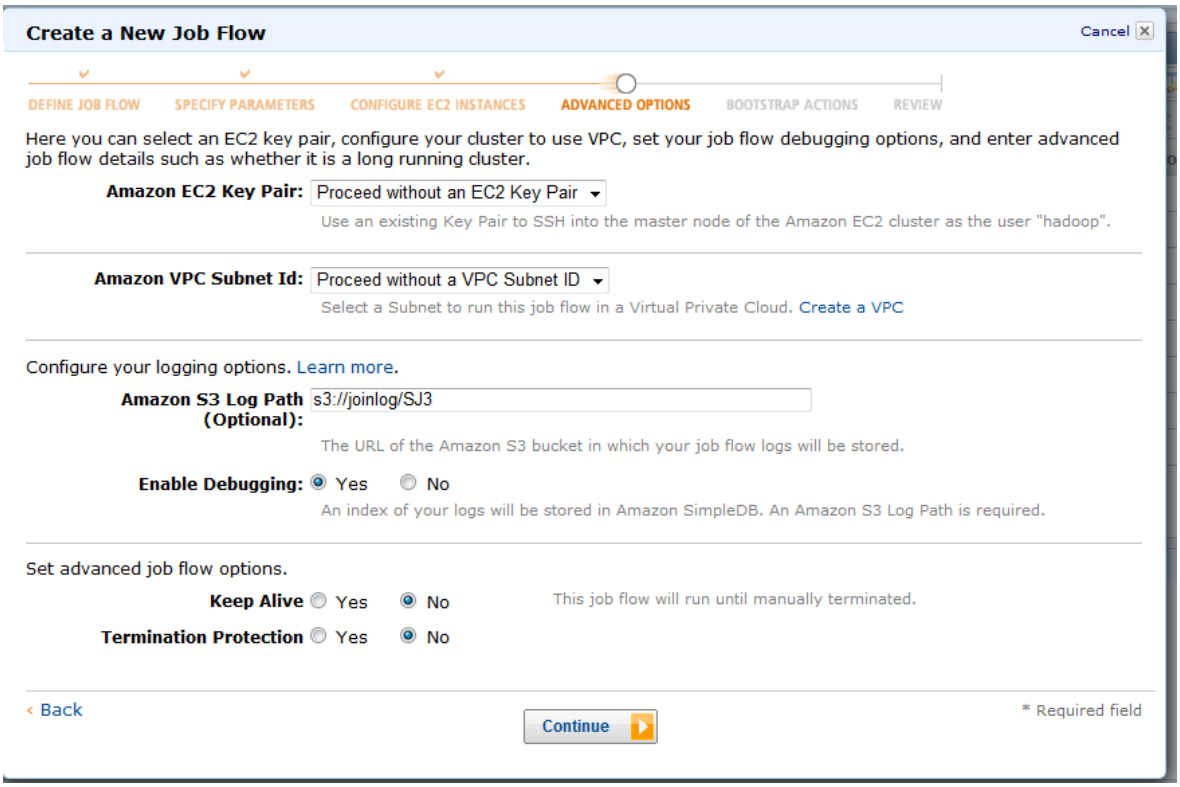

Figure 7. Elastic MapReduce step-4.

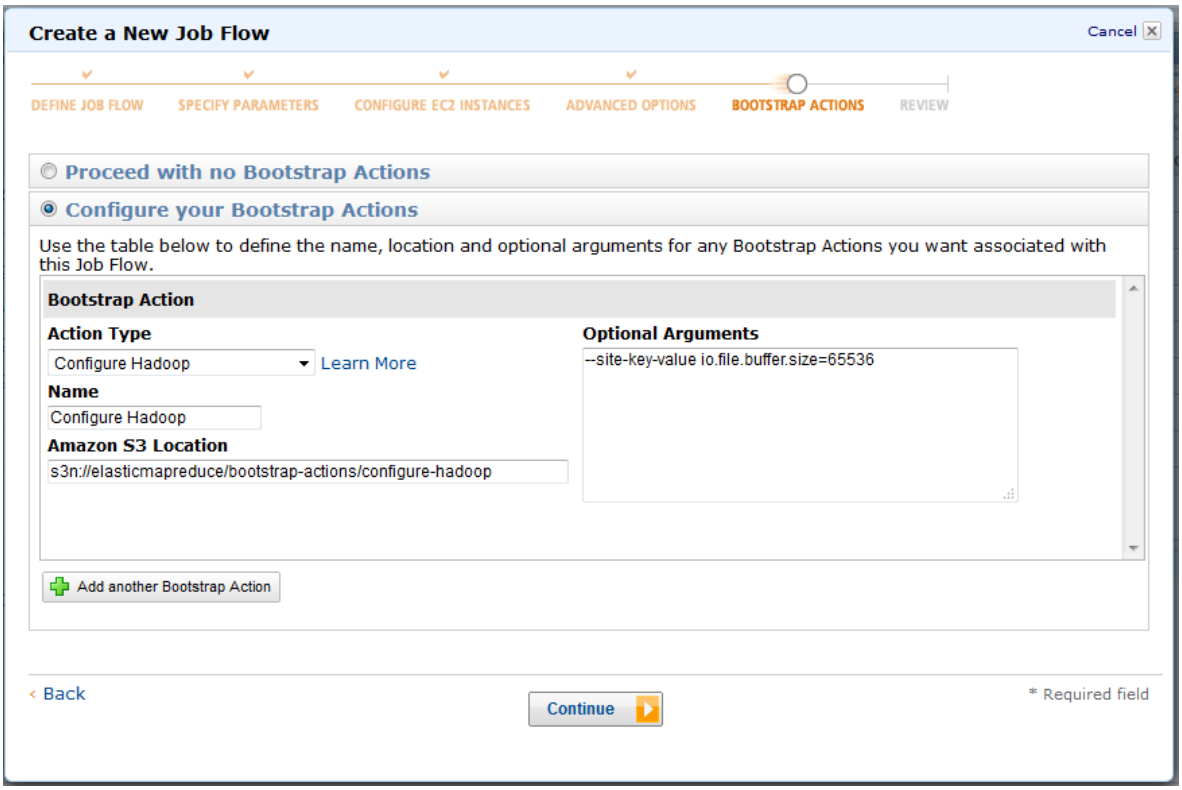

Figure 8. Elastic MapReduce step-5.

7. **Set Bootstrap Actions -** Sometimes we need to configure the Hadoop cluster with some different configuration than the default one. So, we can specify the bootstrap action. These actions can be configuring the Hadoop parameters such as cache size, number of mapper and reducers, HDFS block size and many more. Also we can configure some external libraries on the Hadoop cluster. Refer figure 8.

8. **Review and Execution –** All the job configuration parameters are displayed in this step, user can choose to edit these parameters or can go ahead with this configuration. Figure 9 and 10 depicts this step.

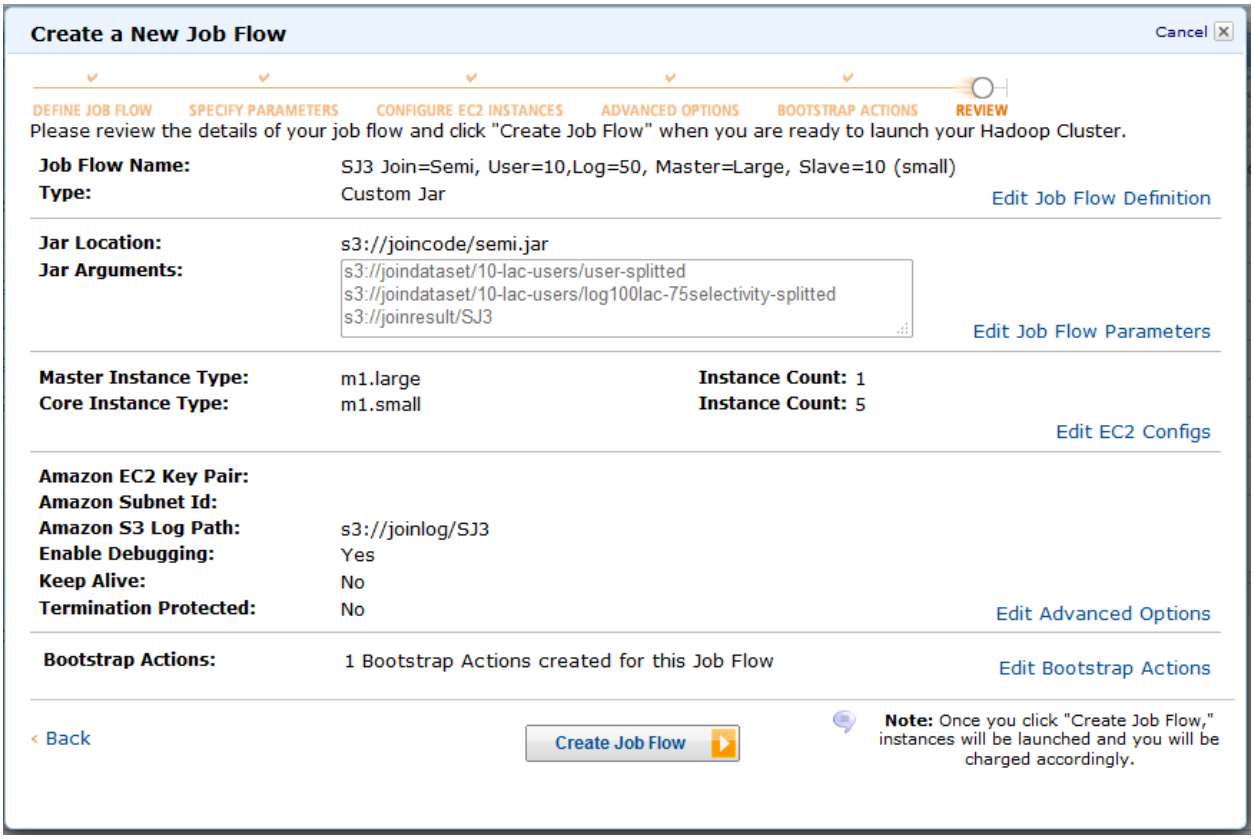

Figure 9. Elastic MapReduce step-6.

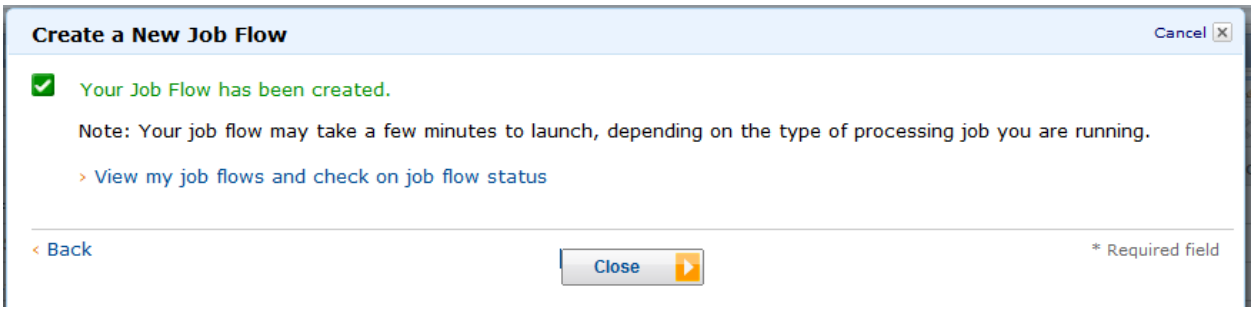

Figure 10. Elastic MapReduce step-7.

9. **Debugging a job flow –** once the job is created, we can check the status of each Map and Reduce process using debug option. Various logs such as system log, error log can be seen. Also we can see individual logs and attempts made by each Mapper and Reducer. Figure 11 shows the debugging screen.

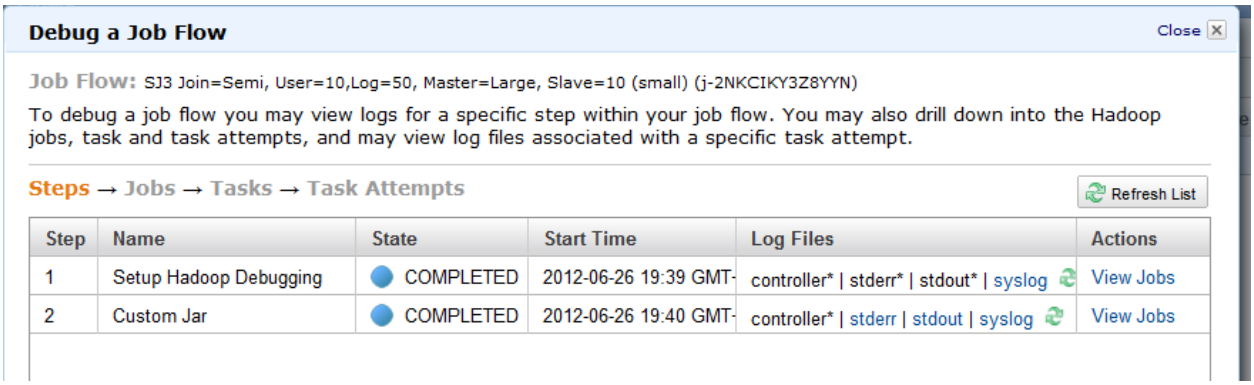

Figure 11. Elastic MapReduce step-8.