Department of Digital Systems

UNIVERSITY OF PIRAEUS

Master of Science in

DIGITAL SYSTEMS & SERVICES

Area of study: Network-Oriented Information Systems

Efficient processing of Top-k joins in MapReduce/Hadoop

Master Thesis

By

Mei Saouk

Supervisor: Christos Doulkeridis

Piraeus, 26/02/2016

Abstract

Top-k joins are widely used in the area of data analytics. One of the most popular frameworks for data analytics is MapReduce, especially its open source implementation in Apache Hadoop. However, due to certain limitations of the model, the processing of top-k joins on Hadoop MapReduce becomes inefficient for very large datasets. In particular, MapReduce processes the whole input even if the best k tuples can be produced by processing only a part of the input datasets. In addition to this, MapReduce does not provide a load balancing technique for the fair load distribution to the reducers. These two weaknesses make top-k join processing on MapReduce inefficient. In this thesis, we propose three algorithms to tackle the problem of early termination and load balancing. Our techniques are based on algorithms that use data synopses such as histograms. Our experimental evaluation proves the efficiency of our proposed algorithms in terms of execution time and resources used, for a number of factors such as the k value, the dataset size, the join selectivity and the data distribution.

Περίληψη

Οι επερωτήσεις σύζευξης με κατάταξη χρησιμοποιούνται ευρέως στην ανάλυση δεδομένων. Ένα από τα πιο γνωστά μοντέλα ανάλυσης δεδομένων είναι το MapReduce και ειδικότερα η ανοιχτού λογισμικού υλοποίησή του, το Apache Hadoop. Εντούτοις, εξαιτίας συγκεκριμένων περιορισμών του μοντέλου, η επεξεργασία των επερωτήσεων σύζευξης με κατάξη στο Hadoop MapReduce, κρίνεται μη αποδοτική για μεγάλους όγκους δεδομένων. Συγκεκριμένα, το μοντέλο MapReduce επεξεργάζεται το σύνολο των δεδομένων που λαμβάνει ως είσοδο, ακόμα και αν είναι εφικτό να γίνει ο υπολογισμός των k καλύτερων αποτελεσμάτων με μέρος μόνο των δεδομένων εισόδου. Επιπροσθέτως, το μοντέλο MapReduce δεν παρέχει τεχνική κατανομής φόρτου για τη δίκαιη κατανομή του φόρτου στους reducers. Αυτές οι δύο αδυναμίες καθιστούν την επεξεργασία των επερωτημάτων σύζευξης με κατάταξη στο MapReduce προβληματική. Στην παρούσα εργασία, προτείνονται τρεις αλγόριθμοι για την αντιμετώπιση των προβλημάτων του έγκαιρου τερματισμού και της κατανομής φόρτου. Οι τεχνικές που προτείνονται, βασίζονται σε αλγόριθμους που χρησιμοποιούν συνόψεις δεδομένων όπως τα ιστογράμματα. Η πειραματική αποτίμηση αποδεικνύει την αποδοτικότητα των προτεινόμενων αλγορίθμων από άποψη χρόνου και εκμεταλλευόμενων πόρων, για ένα πλήθος παραγόντων όπως η τιμή του k, το μέγεθος των αρχείων δεδομένων, την επιλεξιμότητα των δεδομένων και το είδος κατανομής των δεδομένων.

Contents

Figures

Tables

1 Introduction

1.1 Motivation

Efficient processing of rank aware queries has attracted the interest of the scientific community over the last few years. In particular, top-k join, which is a category of rank aware queries, is an essential tool for data analysis. Top-k joins enable selective retrieval of the k best combined results that come from multiple different input datasets. These queries return the k most important join tuples from the potentially huge results of a join among relations, according to a given ranking function.

Efficiency in terms of time and resources is a crucial requirement in many large-scale environments that handle vast amounts of data in a frequent basis. The most common solutions to address large-scale data analytics, evolve architectures of commodity machines to powerful parallel architectures. Some examples of such architectures from the industry are Google's MapReduce [7], Yahoo's PNUTS [3], Microsoft's SCOPE[28], LinkedIn's Kafka [9] and Apache Hadoop which is an open source implementation of MapReduce [12]. Several companies, such as Facebook [1] use and contribute to Apache Hadoop implementation.

Arguably, nowadays, the most popular framework for parallel processing is MapReduce due to its significant features that include scalability, fault tolerance, ease of programming and flexibility. The model was proposed by Dean et al.[7] as a programming model for processing and generating large datasets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key. MapReduce automatically parallelizes and executes the program on a large cluster of commodity machines. Thus, MapReduce has been proposed as a flexible tool for big data analytics [8].

However, the model has also been criticized in terms of performance against parallel databases by Pavlo et al. [21] and by Stonebraker et al. [25]. To improve the performance of the traditional MapReduce model, many solutions have been proposed which address different factors that affect the job execution. A complete list of the factors affecting MapReduce jobs' performance and the corresponding scientific surveys that aim to mitigate these factors are presented by Doulkeridis et al. [5] and by Lee et al. [19].

In the context of top-k joins, there are two factors recognized as weaknesses of the MapReduce model that make the processing of such queries inefficient on a framework which implements the MapReduce model such as Hadoop. The first factor is the lack of early termination of map tasks. A MapReduce job is detached from the input datasets context, therefore it processes the whole datasets even if the top-k results asked by the user can be produced by a small part of the input datasets. This adds delay to the completion of the job and thus the user waits for the redundant processing to complete. Another weakness of the model is the lack of an efficient load balancing technique in the reduce phase. This weakness is stressed out particularly in skewed datasets where some keys aggregate a very large number of values, so consequently the reduce tasks responsible for this key take longer time to complete processing. The result of this behavior is that the most loaded reduce task delays the completion of the job while the other reduce tasks have completed their processing.

1.2 Scope

The scope of this thesis is to propose, implement and test a set of algorithms that tackle the weaknesses of MapReduce model in the context of processing top-k join queries. All proposed techniques are implemented on top of Hadoop MapReduce model without requiring any modifications on its internal mechanism.

The algorithms proposed, aim to address the problems of early termination and load balancing in MapReduce. The efficiency of these algorithms is proved by a number of experiments performed on a Hadoop cluster. Both early termination and load balancing techniques require a preprocessing on the input data. More specifically, data synopses like histograms are used to determine a) the number of the tuples that need to be processed and b) the load produced by these tuples in the reduce phase.

1.3 Thesis structure

This thesis is divided in 8 chapters. Chapter 1 is the introductory section where the motivations and the scope of the thesis are presented. Chapter 2 summarizes the related research and proposals that have been made to address the same problem. Chapter 3 analyzes the technical and algorithmic background of the proposed solution. Chapter 4 desribes the architectural design of the proposed solution. Chapter 5 is more technical and describes the algorithms implemented by providing pseudocode and documentation for each algorithm. Chapter 6 presents the experimental evaluation of the algorithms. The conclusions of this work and some proposals for further research are presented in Chapter 7. Chapter 8 lists references and bibliography. Last but not least, an Appendix is provided with the complete experimental results.

2 Related Work

2.1 Early Termination

Many attempts have been made to address the lack of early termination in MapReduce. Some of those are presented in this section.

In Grover's et al. work [10] , the issue of lack of early termination of map tasks is addressed by using sampling based on predicates. The concept of this solution is based on the fact that Hadoop maintains an abstraction between itself and the job's semantics. Therefore, a new type of job called dynamic job is presented. It runs as a smaller job with the intention of processing only a small subset of the input partitions. Another new concept, called Input Provider, is introduced and its role is to receive periodic statistics of the job and decide dynamically whether more input data should be accessed by the job or not. The decisions that Input Provider may result to are three: a) "end of input" which indicates that the job need not consume additional input partitions, b) "input available" which means that more input is needed and thus Input Provider provides to Hadoop the list of additional partitions to be processed next, c) "no input available" which means that Input Provider has to wait until its next invocation to reassess the job's progress.

Another approach to the early termination matter, is EARL [17] which stands for Early Accurate Results Library. The concept of this approach is the calculation of k results by producing uniform random samples in an iterative way. In each iteration, the accuracy of the result is assessed. The assessment relies on the accuracy error which is estimated via the bootstrapping technique. The iteration termination condition is the user defined threshold for the error. The implementation of EARL in Hadoop MapReduce framework requires the implementation of a pipelining technique so that reduce tasks can process input before the completion of processing in the map tasks. Moreover, a communication channel between map tasks and reduce tasks is implemented for checking the satisfaction of the termination condition.

The lack of early termination in the context of rank aware processing has been recognized by Doulkeridis et al. [6] as well, were various individual techniques are introduced to address this problem. Some of the techniques proposed are the intelligent data placement using advanced partitioning schemes tailored to top-k queries and the use of synopses for the data in HDFS that allow efficient identification of blocks which most probably contain the top-k results.

The following approaches, address the problem of early termination specifically, in rank aware query processing in cloud environments. RanKloud [2] is a framework which uses a data partitioning strategy called uSplit, which partitions data in a utility-sensitive manner. A tuple is considered utility sensitive when it is likely to produce a top-k result. RanKloud aims at enhancing rank aware queries processing in non-uniform data by estimating a threshold which represents the lowest score of the top-k results so as to prevent wasted work. Ntarmos et al. [20], present a set of three algorithms which promise efficient processing of rank aware queries. The first algorithm, involves Pig and Hive approaches to calculate top-k joins. The second algorithm involves MapReduce jobs which use specialized indices stored in a NoSQL database and its goal is to reduce the number of MapReduce jobs needed to estimate the top-k results. The last algorithm uses histograms and Bloom filters which are again stored in a NoSQL database and accessed from MapReduce jobs. Wang et al. [26] introduce a MapReduce-like framework for online analytics to address algorithms such as top-k and k-means. Online analytics refer to the concept that the input is being processed progressively and every time a portion of data is analyzed, an estimated result can be returned. This implies that some level of accuracy needs to be sacrificed for better performance and responsiveness. This solution consists of three key features: 1) an online sampling module to obtain data samples for incremental processing 2) an estimation module that can be defined by the user and is responsible for computing approximate results based on the running statistics and 3) an early termination module that can also be defined by the user and it can enable the execution to be stopped before all data is processed [26].

2.2 Load Balancing

There exist several approaches that use pre-processing and sampling techniques to address the problem of fair work allocation in the MapReduce model.

Kold et al [14] propose two algorithms. The first one, BlockSplit, considers the block sizes and assigns entire blocks to reduce tasks, while respecting load balancing and memory constraints. The larger blocks are divided in smaller chunks to enable their parallel processing. The second algorithm is PairRange and it distributes the entities in a manner that each reduce task computes the same number of entity comparisons. Both algorithms, rely on a preprocessing MapReduce job which produces a matrix containing the number of entities per block.

In Ramakrishnan's et al. [22] work, the proposed algorithm splits the large load that corresponds to certain reduce keys by producing a number of medium load reduce keys with the means of whole tuple hashing or with the use of secondary keys (bin-packing algorithm). The identification of large load keys takes place before the MapReduce job execution and is performed by sampling. The preprocessing result is stored in a file called partition in HDFS.

Another category of approaches to the problem of handling data skew in MapReduce contains solutions that focus on data repartitioning. Guer et al. [11] use a cost estimation method to calculate the load assigned to reduce tasks. In each map task, local statistics are maintained and then combined in a global histogram by the TopCluster algorithm. The information stored in global histogram is used to achieve a fair load distribution in reduce phase. SkewReduce[15] consists of an API for spatial feature extraction algorithms and a static optimizer. The functions of the API are translated into a dataflow that can run in a MapReduce platform. The optimizer uses user defined cost functions that estimate processing time and its role is to partition the data so as to ensure skew-resistant processing. However, the fact that the cost functions are defined by user can be a disadvantage. This is avoided in SkewTune[16]. SkewTune detects straggling tasks without using user defined functions and repartitions the unprocessed input of the most loaded task to the next available.

Sailfish[23] introduces a solution which batches the disk accesses that each reduce task has to make to fetch the data from map tasks. In this solution, the map tasks do not produce intermediate files but instead, they shuffle their outputs to the reduce tasks. Each reduce task aggregates the map outputs and writes them in a file. So, a file is produced per reduce task and the reduce task has to access only this file. Themis[24] performs batching as well in a similar way that Sailfish does, but it also considers moderate clusters, with a number of nodes less than one hundred, where hardware failures are rare to happen. As supported by Rasmussen et al. [24] in moderate clusters, fault tolerance techniques can be eliminated and thus enhance overall execution time, so, the proposed technique in a hardware failure is the re-execution of the job.

3 Background

3.1 Technical Background

3.1.1 Hadoop

The Apache Hadoop framework is an open source implementation of the MapReduce algorithm. This framework allows the distributed processing of large data sets across computer clusters. It is designed to scale up from single servers to thousands of machines and to take advantage of the each node's storage and local computation. A significant goal that it fulfills is the detection and handling of failures, at the application layer, that may have been caused by hardware failures.

Although the best known components of Hadoop are MapReduce and HDFS (its distributed filesystem), the framework also offers a number of other subprojects concerning distributed computing which provide supplementary services to the users while adding higher abstraction levels. Some of the subprojects are mentioned below and their place in the technology stack is shown in figure 1.

- Core
- Avro
- MapReduce
- HDFS
- Pig
- HBase
- ZooKeeper
- Hive
- Chukwa

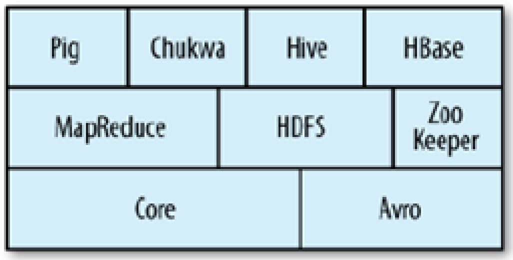


Figure 1 : Hadoop subprojects [27]

3.1.2 HDFS

HDFS stands for Hadoop Distributed File System and it is designed to provide high throughput access to application data. It can store very large files using simple commodity hardware. The idea, upon which, HDFS was designed, is a write-once but read-many times pattern. This implies that HDFS is a very good choice for the storage and processing of very large dataset (gigabytes, terabytes and even petabytes in size) that are not to be altered often.

The implementation of this distributed file system takes into consideration the hardware node failures which are likely to happen when using commonly available hardware. Therefore, it implements certain techniques to ensure data integrity such as checksums, each time a file enters the filesystem or is transferred through the network, and replicas. HDFS cluster consists of two types of nodes. One node is the namenode which maintains and manages the filesystem namespace and there are a number of datanodes (workers) which contain the actual data. There is also a replica node of the namenode which is called secondary namenode.

The HDFS architecture is briefly presented in figure 2. Whenever a client uploads or creates a file in HDFS, the file is internally split into HDFS blocks. HDFS uses the concept of the block similarly to the hard disk drives. The difference is that an HDFS block is a much larger unit than the ones created in a filesystem for a single disk. The default block size in HDFS is 64MB. The namenode decides the replication strategy and keeps information like which node will store each block or replica. Accordingly, each time a client requests to read a file from HDFS, a request is sent to namenode to extract the necessary information for the file location on the datanodes.

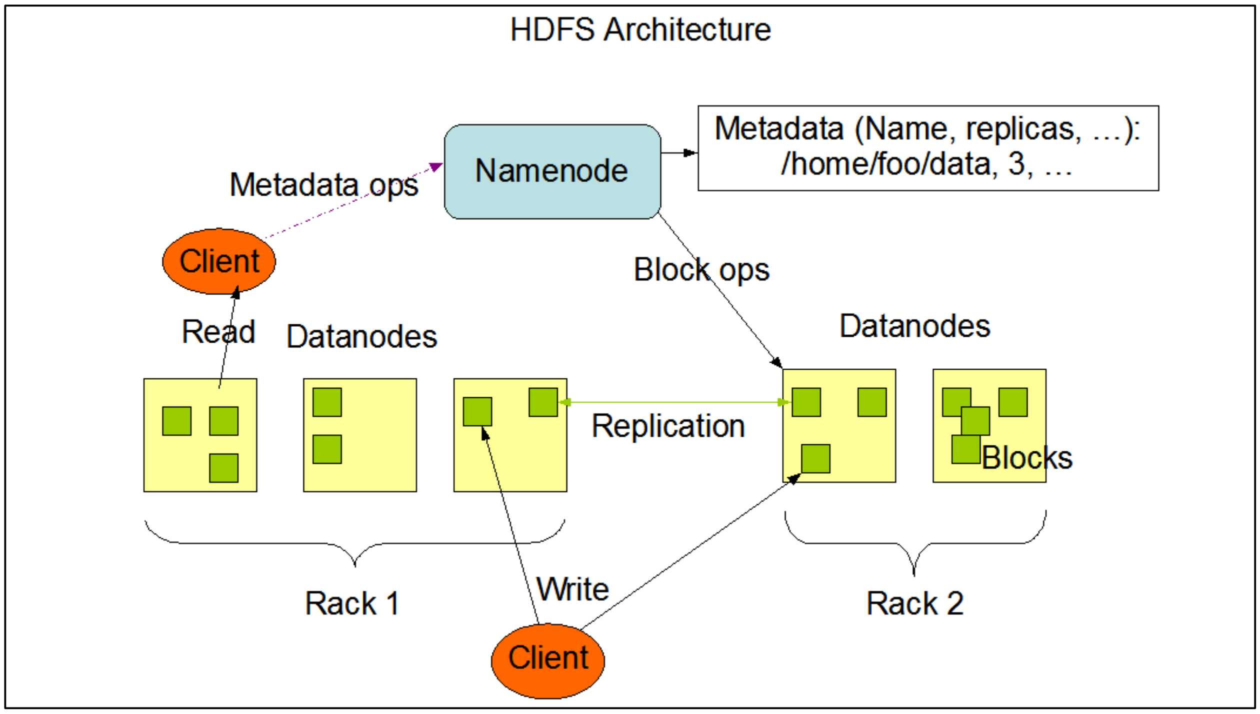


Figure 2 : HDFS Architecture [12]

3.1.3 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs and a reduce function that merges all intermediate values associated with the same intermediate key[7].

Hadoop framework implements the MapReduce model, offering users the opportunity to implement and run their MapReduce jobs without worrying about issues like the parallelization of the computation, the data distribution and the handling of failures.

When a MapReduce job is submitted for execution, one node, the jobtracker, coordinates and assigns the map and reduce tasks to other nodes, the tasktrackers. The job input is divided into logical fixed sized units which are called input splits. Each input split is the input of a map task. The code of the map task is repeated for every record contained in the input split. The output of each map task is saved locally (not on HDFS) on each tasktracker as intermediate result. The intermediate map results are then sorted and partitioned by key on each node. The code that performs the partitioning can be either the default Hadoop Partitioner (which buckets keys using a hash function) or a user defined method. There can be many keys handled by one reducer but all the records for a specific key must be transferred to a single reducer. This is achieved during the shuffle phase. Once the intermediate data, reach the reducer that will process them they are merged and the reducer starts its execution. The output of the reducers is stored on the HDFS. The MapReduce data flow with multiple reduce tasks, is presented in figure 3.

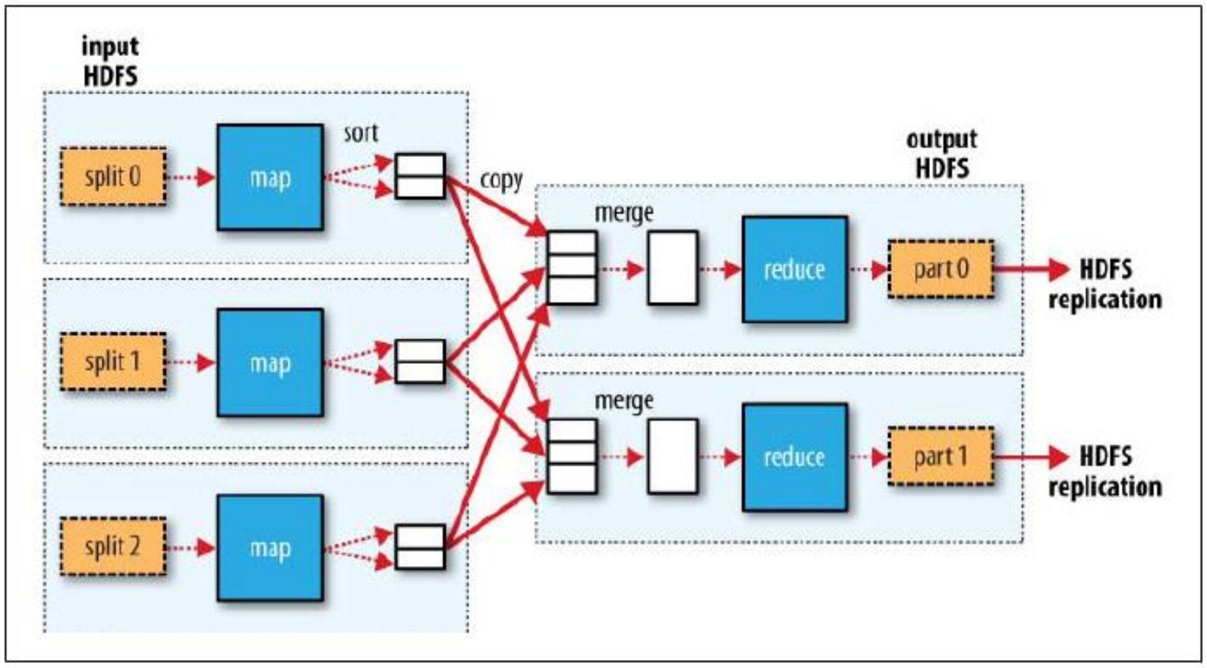


Figure 3 : MapReduce data flow with multiple reduce tasks [27]

The most interesting part of the MapReduce data flow is the one where shuffle and sort take place. Proper handling of these phases can enable the implementation of complex queries like Top-K joins result on significant differences in job's total execution time. The shuffle and sort phase is partly implemented on the Map side and partly on the Reduce side. From the Map side, when the intermediate results are produced they are stored in the memory buffer of the map task. Once the contents of the memory buffer reach a threshold size the contents are spilled to the disk. Before the intermediate data are actually written on the disk, they are divided into partitions corresponding to the reducers that they will ultimately be sent to. Within each local partition an in-memory sort by key is performed.

From the Reduce side, the reduce tasks fetch the map outputs that correspond to their partition and store them initially in the memory buffer, this is also known as the copy phase of the reduce task. As soon as the memory buffer reaches a certain threshold, the fetched map outputs are merged and spilled to the disk. Once spilled on the disk, they are merged again on larger files. As soon as all the map outputs have been copied the reduce task performs the sort phase which merges the map outputs while maintaining their sort ordering. The whole Shuffle and Sort phase is summarized in figure 4.

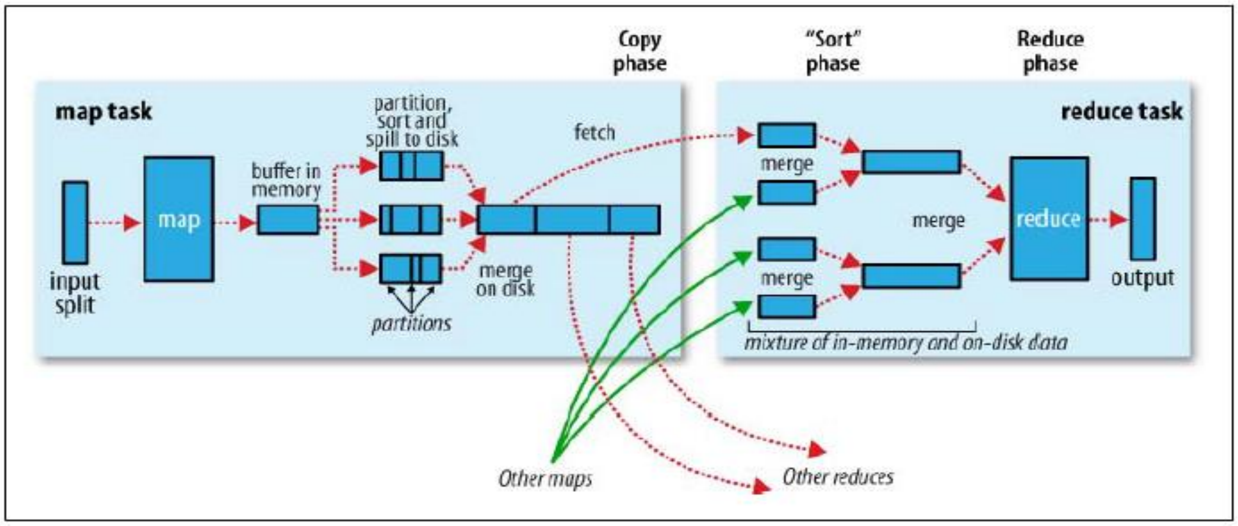


Figure 4 : Shuffle and Sort [27]

It is worth mentioning that values such as the map memory buffer threshold or the memory buffer default size or reduce memory buffer can also be configured by the user. Some indicative properties are mentioned below:

- **io.sort.mb** → map task default memory buffer
- **io.sort.spill.percent** → threshold that indicates when map memory buffer contents should be spilled on the disk
- **mapred.local.dir** → the local directories that contain the map spilled records

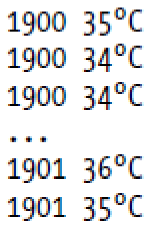
- **mapred.reduce.parallel.copies** → the number of threads that a reduce task uses to copy map output results locally
- **mapred.job.shuffle.merge.percent** → threshold that indicates when reduce memory buffer contents should be spilled on the disk

3.1.4 Secondary Sort in MapReduce

Another way with which the user can intervene in the shuffle – sort phase is the use of secondary sort technique in the implementation of the MapReduce jobs. Even though Hadoop makes sure that all records are sorted by key when they reach the reducers, it makes no sorting on the values of a particular key. In certain cases there is the need for a sorting amongst values as well.

A very simple example mentioned in Tom White's book: *Hadoop : The Definitive Guide* is the weather dataset and the need is to sort the year and the temperatures taken for each year in an ascending order. Each record of the input weather dataset has two fields, the first is the year and the second is a temperature.

So, the output would look like figure 5.



```
1900 35°C
1900 34°C
1900 34°C
...
1901 36°C
1901 35°C
```

Figure 5: Weather MapReduce Job Output [27]

To achieve that, the user has to define a composite key. A composite key would be the combination of the two fields: year-temperature. Once the composite key is defined, the following classes have to be implemented by the user:

- A Partitioner which partitions the map output records by the first part of the key, which is the natural key. In that way we make sure that all records for the same year reach the same reducer.
- A KeyComparator which sorts the composite key (by year and by temperature).
- A GroupingComparator which sorts only by the natural key (the year).

So, the secondary sort technique assures that with the use of the custom Partitioner all records for one year, from all map tasks will be delivered to the same reducer. The KeyComparator makes sure that the map output records are sorted by year and temperature and finally the GroupingComparator makes sure that all groups of key-value pairs that reached the reducer are sorted by year. In this final step, each group will be already sorted by value from the KeyComparator.

3.1.5 Input Split, InputFormat and RecordReader

As mentioned earlier, an input split is the result of a logical split of a file in HDFS. An input split is also the input of a map task. Hadoop creates input splits with the Java Interface org.apache.hadoop.mapred.InputSplit. Input Splits are created by an instance of Interface org.apache.hadoop.mapred.InputFormat. InputFormat describes the input-specification for a MapReduce job.

One of the commonly used implementing classes of InputFormat interface is org.apache.hadoop.mapred.FileInputFormat. FileInputFormat is the base class for all file-based InputFormats. This provides a generic implementation of getSplits(JobConf, int) [12].

The method getSplits calculates the number of splits and sends their storage locations to the jobtracker. Once a map tasks start its execution it passes the split to method getRecordReader() of InputFormat. A RecordReader is an iterator over the records of the map input split and is used to generate key-values which will be passed to the map function.

A user can create custom InputFormat classes that extend one of the implementing classes of the Interface InputFormat in order to extend the capabilities of a MapReduce program.

To adjust the number of input splits returned by the method getSplits(JobConf,int) the following steps could be performed:

1. *Create a custom InputFormat class which extends FileInputFormat*
2. *Override the method getSplits()*
3. *Implement getSplits() method so as to return the desired number of input splits*

The user can also intervene in the implementation of RecordReader and define the type and content of the key-value pairs that will be passed to the map function. The steps to do that are the following:

1. *Create a custom RecordReader class which implements the Interface org.apache.hadoop.mapred.RecordReader*
2. *Implement the method next() according to the program's specifications. This method is invoked each time a record is fetched from the InputSplit.*
3. *Create a custom InputFormat class which extends the FileInputFormat and override the method getRecordReader() so as to invoke the new custom RecordReader*

3.1.6 Side Data Distribution

There are some cases where a MapReduce job may need access to certain read only data in order to process a large input dataset. This extra piece of data should be available to all map tasks or to all reduce tasks depending on the implementation and is commonly referred to as side data. There are two basic distribution mechanisms offered by Hadoop which can be used by the user to handle side data, the Job Configuration and the Distribute Cache.

3.1.6.1 *Job Configuration*

Each MapReduce job has a specific configuration defined by the Java class: org.apache.hadoop.mapred.JobConf which extends the class org.apache.hadoop.conf.Configuration. All the built-in MapReduce job properties can be read or altered with methods offered by the class JobConf. For example, the property that defines the number of reduce tasks can be read by the method: getNumReduceTasks() and can be set by the method setNumReduceTasks(int n).

Apart from the default MapReduce properties, there are the user defined properties. A user can add his/her own property and pass a primitive type as its value, via JobConf class by using the methods *setInt(String propertyName, int value), setString(String propertyName, String value) etc.* inherited by the class Configuration. The JobConf instance, with all its properties, is visible from all Map and Reduce tasks. So, the user can override the method configure(JobConf job) of the class org.apache.hadoop.mapred.MapReduceBase, within the implementation of the Map and Reduce classes, in order to access the properties that he/she defined on the JobConf instance. For this purpose, JobConf offers getter methods inherited from Configuration such as getInt(String propertyName), getString(String propertyName) etc.

3.1.6.2 Distributed Cache

JobConfiguration is a good choice when the side data size is a few kilobytes maximum because it can put pressure on the memory usage in the Hadoop daemons. Instead, Hadoop exposes distributed cache mechanism as a service that copies and archives files to the tasks nodes. The files are copied once per job and the user can specify which files need to be handled in the distributed cache with the class org.apache.hadoop.filecache.DistributedCache and the method addCacheFile(URI uri, Configuration conf).

Once the side data is added in DistributedCache, it is visible from all Map and Reduce tasks. So, the user can get the paths of the localized cached files, within the Map or Reduce class implementation, by invoking the method getLocalCacheFiles(Configuration conf) from class DistributedCache.

3.1.7 Joins In MapReduce

Joins in MapReduce can be performed programmatically. This means that the user has to implement a MapReduce job that will perform the join between two datasets. There are three options for the implementation approach and the proper choice depends on the size and structure of the datasets to be joined.

If the one dataset is very large while the other is very small, then the simplest way to perform a join using MapReduce, is to use the distributed cache machine. Therefore, the MapReduce job will have as input only the large dataset. For each record fetched from the dataset, the join value will be looked up in the cached file and the corresponding records of the second dataset will be fetched and joined with the current record of the first (the input dataset).

However, if both datasets are very large then the distribute cache mechanism is not an efficient solution. In this case, the join can be performed either on the map side or the reduce side.

Map side joins are an appropriate solution when the input datasets have the structure of the output of a MapReduce job. This means that both datasets should be divided on the same number of partitions and each should be sorted by the same key. Also, all records for a certain key should reside on the same partition. These restrictions are necessary because on a map side join, the join is performed before the data reaches a map function and this is achieved with the use of the class: org.apache.hadoop.mapred.join.CompositeInputFormat<K> that represents an InputFormat capable of performing joins over a set of data sources sorted and partitioned the same way. Due to the above mentioned restrictions, Map side joins can only be used for the join of several outputs of map reduce jobs.

A more general category of MapReduce joins are the Reduce side joins. They are not constrained by the size, partitioning and sorting of the input datasets. A commonly used technique used for the implementation of Reduce side joins is the secondary sort technique described earlier in 3.1.4 in combination with the tagging technique. Tags are used to mark the source dataset from which each record derives. So, the composite key is consisted by a tag, which is usually a number indicating the first or second dataset, and the actual join value.

With the use of the secondary sort and tagging techniques, it is certain that each reducer will receive first all the records for a specific join value from the first dataset and then from the second. Afterwards, the reducer can perform the join.

3.1.8 Job Counters

When a MapReduce job in Hadoop completes its execution, there are five basic metrics provided by the framework: Total Duration, Average Map Time, Average Reduce Time, Average Shuffle Time and Average Merge Time. Here is the definition for each of them:

Total Duration = Time elapsed from the beginning of job execution until the last reduce task completes.

Average Map Time = Total time taken by all Map tasks/ Count of Map Tasks

Average Reduce Time = Total time taken by all Reduce tasks/Count of Reduce tasks

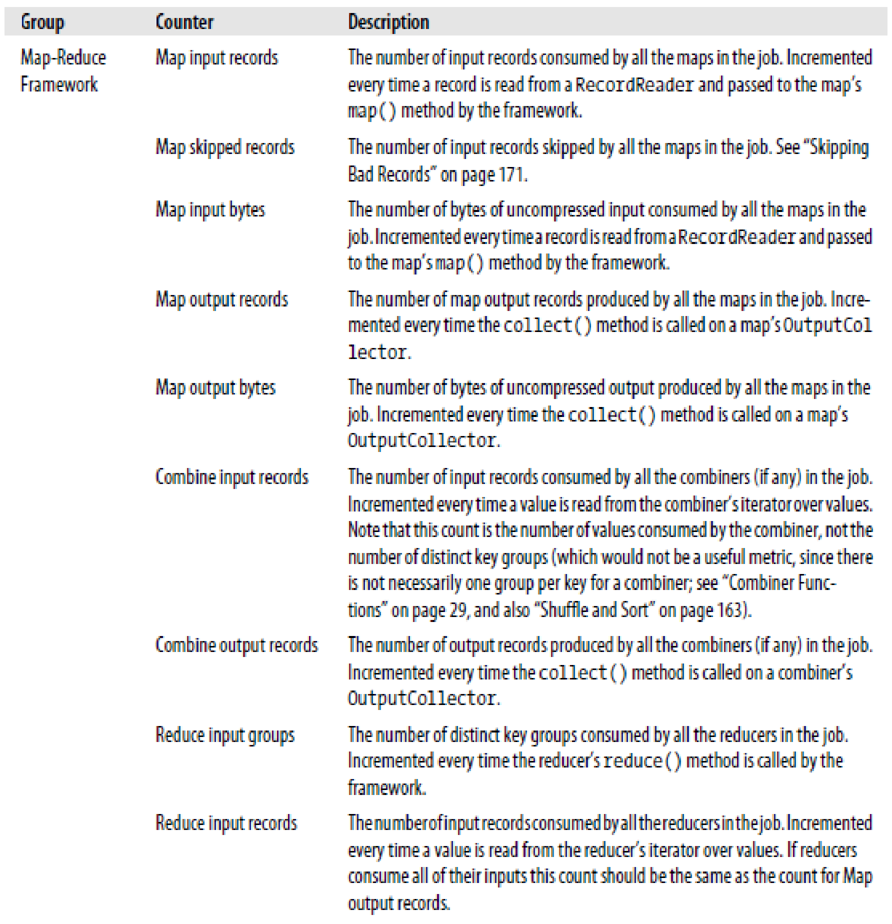
Average Shuffle Time = Total time taken by all Map outputs to be copied / Count of Map outputs

Average Merge time = Average of (sort_FinishTime – shuffle_FinishTime)

The above metrics break down the execution of a MapReduce job in its phases and provide a first image of the job’s efficiency.

Moreover, the framework provides details concerning the job execution through the counters. Counters are metrics of the MapReduce jobs and provide helpful information for the detection of possible bugs in the MapReduce program, for the validity of the input data, for the efficiency of the MapReduce program etc. It is a very helpful tool to evaluate the performance and efficiency of the MapReduce job. Hadoop contains some built in counters but a user can define his/her own user defined counters. Some of the most commonly used built in counters and their purposes are described in Figure 6 : MapReduce Framework built-in counters .

In addition to the counters presented in figure 6, there are two more significant counters: *CPU time spent (ms) on Map* and *CPU time spent (ms) on Reduce*. *CPU time spent on Map* counter represents the total CPU time across all of the nodes in the cluster during map phase. More map tasks result to more cpu time spent. Accordingly, *CPU time spent on Reduce* counter represents the total CPU time across all of the nodes in the cluster during reduce phase. Two more interesting counters of the MapReduce Framework are *Map Spilled Records* and *Reduce Spilled Records. Map Spilled records counter* represents the number of records spilled to disk in all map tasks in the job. Respectively, *Reduce Spilled records counter* represents the number of records spilled to disk by all reduce tasks in the job. The *Shuffled Maps* counter represents the number of map output files transferred to reducers by the shuffle phase.

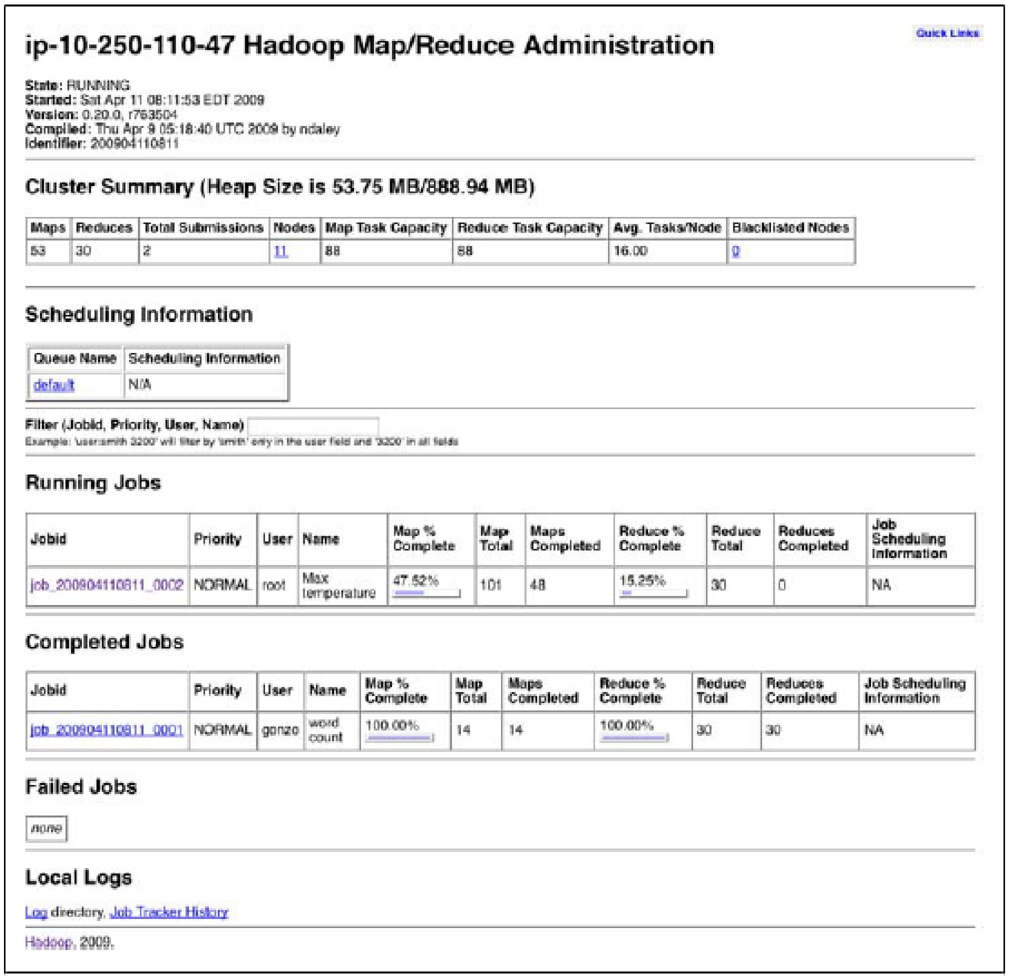


Group	Counter	Description
Map-Reduce Framework	Map input records	The number of input records consumed by all the maps in the job. Incremented every time a record is read from a `RecordReader` and passed to the map's `map()` method by the framework.
	Map skipped records	The number of input records skipped by all the maps in the job. See "Skipping Bad Records" on page 171.
	Map input bytes	The number of bytes of uncompressed input consumed by all the maps in the job. Incremented every time a record is read from a `RecordReader` and passed to the map's `map()` method by the framework.
	Map output records	The number of map output records produced by all the maps in the job. Incremented every time the `collect()` method is called on a map's `OutputCollector`.
	Map output bytes	The number of bytes of uncompressed output produced by all the maps in the job. Incremented every time the `collect()` method is called on a map's `OutputCollector`.
	Combine input records	The number of input records consumed by all the combiners (if any) in the job. Incremented every time a value is read from the combiner's iterator over values. Note that this count is the number of values consumed by the combiner, not the number of distinct key groups (which would not be a useful metric, since there is not necessarily one group per key for a combiner; see "Combiner Functions" on page 29, and also "Shuffle and Sort" on page 163).
	Combine output records	The number of output records produced by all the combiners (if any) in the job. Incremented every time the `collect()` method is called on a combiner's `OutputCollector`.
	Reduce input groups	The number of distinct key groups consumed by all the reducers in the job. Incremented every time the reducer's `reduce()` method is called by the framework.
	Reduce input records	The number of input records consumed by all the reducers in the job. Incremented every time a value is read from the reducer's iterator over values. If reducers consume all of their inputs this count should be the same as the count for Map output records.

Figure 6 : MapReduce Framework built-in counters [27]

3.1.9 Job History

Hadoop provides its users with a web UI for viewing job related information in a graphic environment. The jobtracker page contains information about the currently running jobs on the cluster.



Quick Links

ip-10-250-110-47 Hadoop Map/Reduce Administration

State: RUNNING
Started: Sat Apr 11 08:11:53 EDT 2009
Version: 0.20.0, r763504
Compiled: Thu Apr 9 05:18:40 UTC 2009 by ndaley
Identifier: 200904110811

Cluster Summary (Heap Size is 53.75 MB/888.94 MB)

Maps	Reduces	Total Submissions	Nodes	Map Task Capacity	Reduce Task Capacity	Avg. Tasks/Node	Blacklisted Nodes
53	30	2	11	88	88	16.00	0

Scheduling Information

Queue Name	Scheduling Information
default	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0002	NORMAL	root	Max temperature	47.52%	101	48	15.25%	30	0	NA

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information
job_200904110811_0001	NORMAL	gonzo	word count	100.00%	14	14	100.00%	30	30	NA

Failed Jobs

none

Local Logs

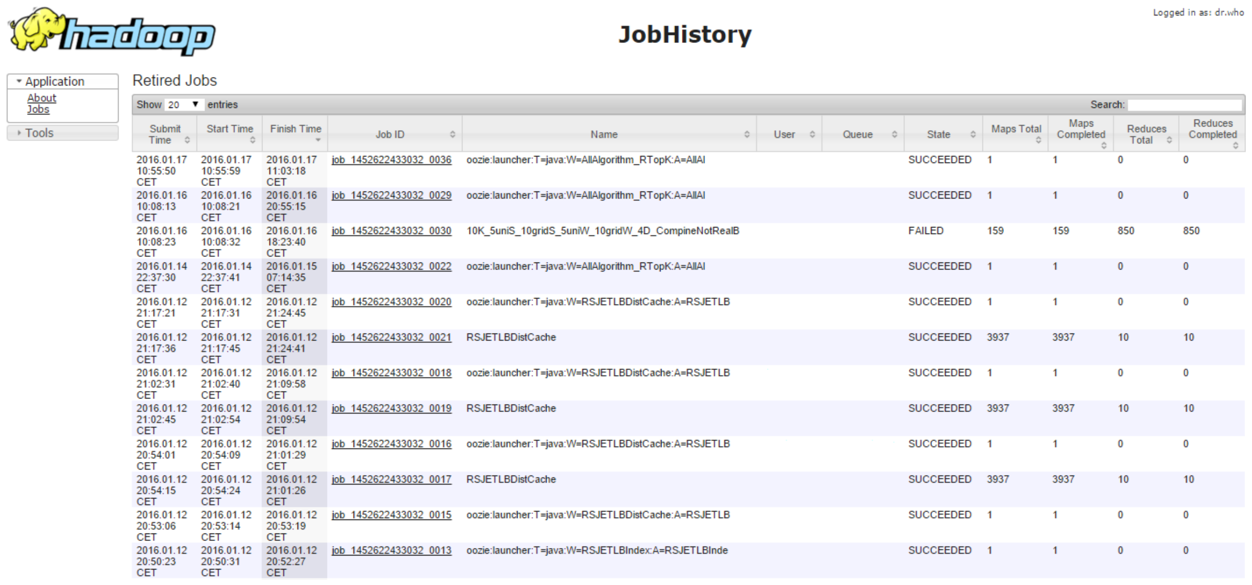
Log directory, Job Tracker History

Hadoop, 2009.

Figure 7 : Screenshot of the jobtracker page [27]

An instance of the jobtracker page is captured in figure 7. Information about the completed jobs is provided through the Job History page. The link to the Job History page is available on the bottom of the jobtracker page.

Job history contains detailed information and links to logs either for successful or failed jobs. An instance of Job History page is captured in figure 8. By clicking on one Job Id, the user is directed to a page where an overview of the job is available as well as links to the counters and logs pages (figure 9). By clicking on the Counters link on the Job overview page, the user is directed to a page where all values for all counters are available, categorized by purpose (figure 10).



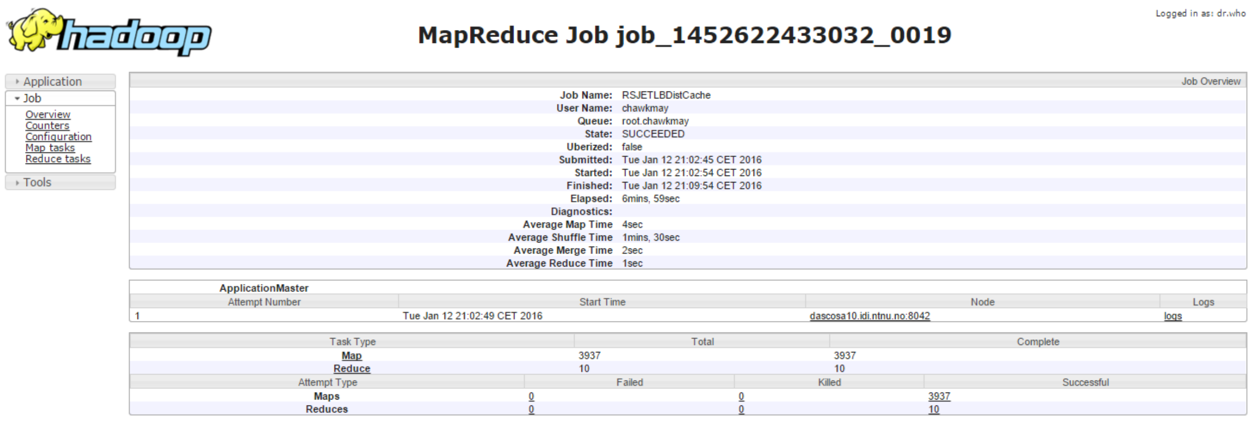
JobHistory

Retired Jobs

Show 20 entries   Search:

Submit Time	Start Time	Finish Time	Job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed
2016.01.17 10:55:50 CET	2016.01.17 10:55:59 CET	2016.01.17 11:03:18 CET	job_1452622433032_0036	oozie:launcher:T=java:W=AllAlgorithm_RTopK:A=AllAl			SUCCEEDED	1	1	0	0
2016.01.16 10:08:13 CET	2016.01.16 10:08:21 CET	2016.01.16 20:55:15 CET	job_1452622433032_0029	oozie:launcher:T=java:W=AllAlgorithm_RTopK:A=AllAl			SUCCEEDED	1	1	0	0
2016.01.16 10:08:23 CET	2016.01.16 10:08:32 CET	2016.01.16 18:23:40 CET	job_1452622433032_0030	10K_5uniS_10gridS_5uniW_10gridW_4D_CompineNotRealB			FAILED	159	159	850	850
2016.01.14 22:37:30 CET	2016.01.14 22:37:41 CET	2016.01.15 07:14:35 CET	job_1452622433032_0022	oozie:launcher:T=java:W=AllAlgorithm_RTopK:A=AllAl			SUCCEEDED	1	1	0	0
2016.01.12 21:17:21 CET	2016.01.12 21:17:31 CET	2016.01.12 21:24:45 CET	job_1452622433032_0020	oozie:launcher:T=java:W=RSJETLBDistCache:A=RSJETLB			SUCCEEDED	1	1	0	0
2016.01.12 21:17:36 CET	2016.01.12 21:17:45 CET	2016.01.12 21:24:41 CET	job_1452622433032_0021	RSJETLBDistCache			SUCCEEDED	3937	3937	10	10
2016.01.12 21:02:31 CET	2016.01.12 21:02:40 CET	2016.01.12 21:09:58 CET	job_1452622433032_0018	oozie:launcher:T=java:W=RSJETLBDistCache:A=RSJETLB			SUCCEEDED	1	1	0	0
2016.01.12 21:02:45 CET	2016.01.12 21:02:54 CET	2016.01.12 21:09:54 CET	job_1452622433032_0019	RSJETLBDistCache			SUCCEEDED	3937	3937	10	10
2016.01.12 20:54:01 CET	2016.01.12 20:54:09 CET	2016.01.12 21:01:29 CET	job_1452622433032_0016	oozie:launcher:T=java:W=RSJETLBDistCache:A=RSJETLB			SUCCEEDED	1	1	0	0
2016.01.12 20:54:15 CET	2016.01.12 20:54:24 CET	2016.01.12 21:01:26 CET	job_1452622433032_0017	RSJETLBDistCache			SUCCEEDED	3937	3937	10	10
2016.01.12 20:53:06 CET	2016.01.12 20:53:14 CET	2016.01.12 20:53:19 CET	job_1452622433032_0015	oozie:launcher:T=java:W=RSJETLBDistCache:A=RSJETLB			SUCCEEDED	1	1	0	0
2016.01.12 20:50:23 CET	2016.01.12 20:50:31 CET	2016.01.12 20:52:27 CET	job_1452622433032_0013	oozie:launcher:T=java:W=RSJETLBIndex:A=RSJETLBInde			SUCCEEDED	1	1	0	0

Figure 8 : JobHistory screenshot



MapReduce Job job_1452622433032_0019

Job Name:	RSJETLBDistCache
User Name:	chawkmay
Queue:	root.chawkmay
State:	SUCCEEDED
Uberized:	false
Submitted:	Tue Jan 12 21:02:45 CET 2016
Started:	Tue Jan 12 21:02:54 CET 2016
Finished:	Tue Jan 12 21:09:54 CET 2016
Elapsed:	6mins, 59sec
Diagnostics:	
Average Map Time	4sec
Average Shuffle Time	1mins, 30sec
Average Merge Time	2sec
Average Reduce Time	1sec

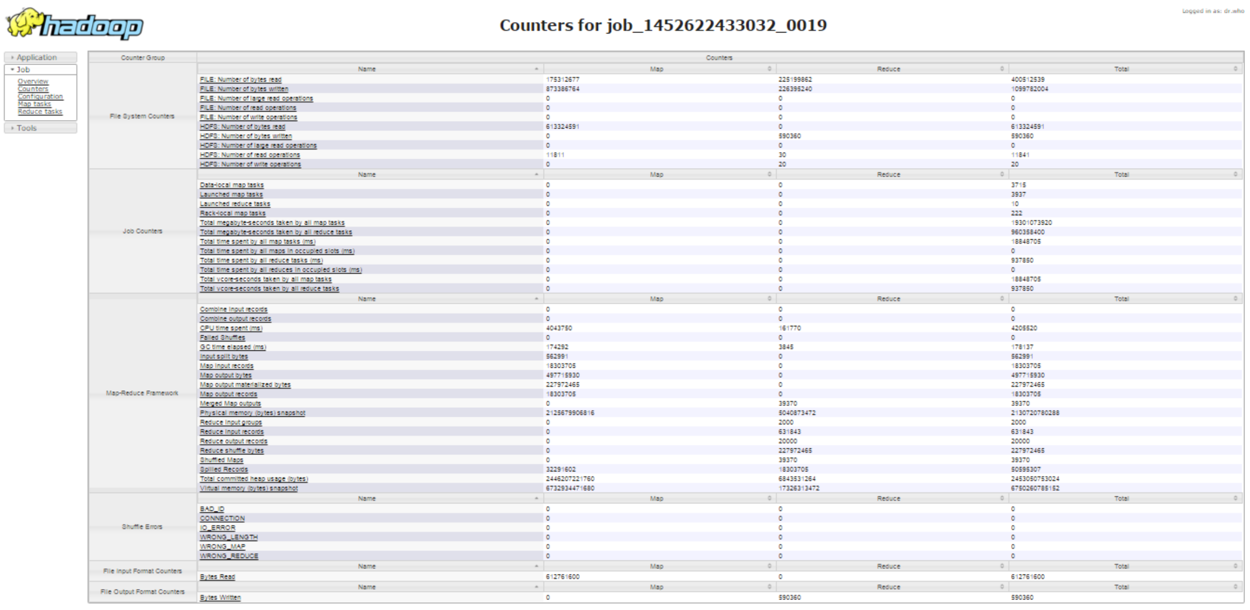
ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Tue Jan 12 21:02:49 CET 2016	dascosa10.idi.ntnu.no:8042	logs

Task Type	Total	Complete
Map	3937	3937
Reduce	10	10

Attempt Type	Failed	Killed	Successful
Maps	0	0	3937
Reduces	0	0	10

Figure 9 : JobHistory - Overview of specific job



Counters for job_1452622433032_0019

Counter Group	Name	Map	Reduce	Total
File System Counters	FILE: Number of bytes read	175312677	225199862	400512539
	FILE: Number of bytes written	873386764	226395240	1099782004
	FILE: Number of large read operations	0	0	0
	FILE: Number of read operations	0	0	0
	FILE: Number of write operations	0	0	0
	HDFS: Number of bytes read	613324591	0	613324591
	HDFS: Number of bytes written	0	590360	590360
	HDFS: Number of large read operations	0	0	0
	HDFS: Number of read operations	11811	30	11841
	HDFS: Number of write operations	0	20	20
Job Counters	Data-local map tasks	0	0	3715
	Launched map tasks	0	0	3937
	Launched reduce tasks	0	0	10
	Rack-local map tasks	0	0	222
	Total megabyte-seconds taken by all map tasks	0	0	19301073920
	Total megabyte-seconds taken by all reduce tasks	0	0	960358400
	Total time spent by all map tasks (ms)	0	0	18848705
	Total time spent by all maps in occupied slots (ms)	0	0	0
	Total time spent by all reduce tasks (ms)	0	0	937850
	Total time spent by all reduces in occupied slots (ms)	0	0	0
	Total vcore-seconds taken by all map tasks	0	0	18848705
	Total vcore-seconds taken by all reduce tasks	0	0	937850
Map-Reduce Framework	Combine input records	0	0	0
	Combine output records	0	0	0
	CPU time spent (ms)	4043750	161770	4205520
	Failed Shuffles	0	0	0
	GC time elapsed (ms)	174292	3845	178137
	Input split bytes	562991	0	562991
	Map input records	18303705	0	18303705
	Map output bytes	497715930	0	497715930
	Map output materialized bytes	227972465	0	227972465
	Map output records	18303705	0	18303705
	Merged Map outputs	0	39370	39370
	Physical memory (bytes) snapshot	2125679906816	5040873472	2130720780288
	Reduce input groups	0	2000	2000
	Reduce input records	0	631843	631843
	Reduce output records	0	20000	20000
	Reduce shuffle bytes	0	227972465	227972465
	Shuffled Maps	0	39370	39370
	Spilled Records	32291602	18303705	50595307
	Total committed heap usage (bytes)	2446207221760	6843531264	2453050753024
	Virtual memory (bytes) snapshot	6732934471680	17326313472	6750260785152
Shuffle Errors	BAD_ID	0	0	0
	CONNECTION	0	0	0
	IO_ERROR	0	0	0
	WRONG_LENGTH	0	0	0
	WRONG_MAP	0	0	0
	WRONG_REDUCE	0	0	0
File Input Format Counters	Bytes Read	612761600	0	612761600
File Output Format Counters	Bytes Written	0	590360	590360

Figure 10 : Counters for specific job

3.2 Algorithmic Background

3.2.1 Top-k Joins

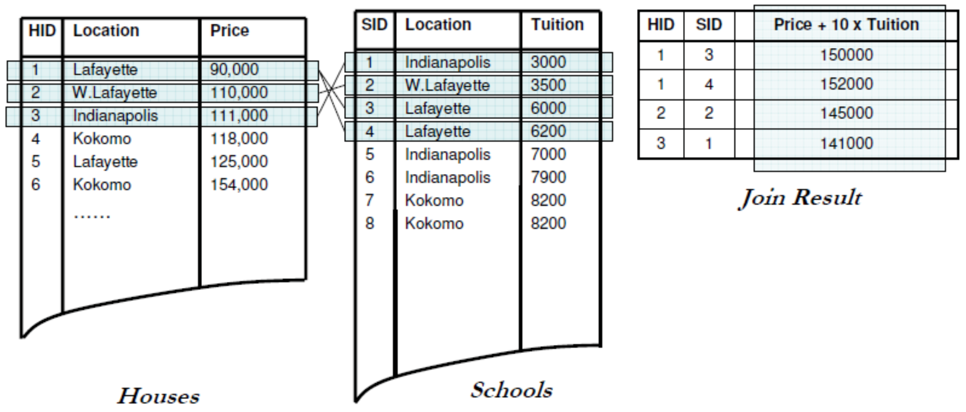
Rank aware queries help users to identify a limited set of the most interesting results of a query answer. Top-k queries belong in the category of rank aware queries and they return the k answers matching better to the user's preferences. An example of a top-k query is the one presented in figure 11. With this query the user wishes to select the two cheapest laptops from a list that contains many products with their respective price. In a Top-k query the classification (ranking) of tuples is based on an aggregated score that occurs when a function f (scoring function) is applied to certain attributes of the table (scoring attributes)

```
SELECT * FROM PRODUCTS
WHERE Category = “Laptop”
ORDER BY Price ASC
LIMIT 2;
```

ID	Category	Model	Price
1	Laptop	Acer Aspire ES1 -111M	255,00
2	Laptop	HP - r100 nv	239,00
3	Tablet	Dell Venue 11 Pro 4G	974,70
4	Laptop	Asus F553MA-SX418H	308,90
5	Smartphone	HTC One M9	611,98
6	Tablet	Crystal Audio TAB-722	39,90
7	Laptop	HP - Stream - 13-c010nv	239,90

Figure 11 : Example of Top-k query

Another category of rank aware queries are Top-k joins. A very good example of a Top-k join is described by Ilyas et al. [13]. Consider a user interested in finding a location (e.g., city) where the combined cost of buying a house and paying school tuition for 10 years at that location is minimum. The user is interested in the five least expensive places. Assume that there are two external sources (databases), Houses and Schools, that can provide information on houses and schools, respectively. The Houses database provides a ranked list of the cheapest houses and their locations. Similarly, the Schools database provides a ranked list of the least expensive schools and their locations. Figure 12 gives an example of the Houses and Schools databases [13].



HID	Location	Price
1	Lafayette	90,000
2	W.Lafayette	110,000
3	Indianapolis	111,000
4	Kokomo	118,000
5	Lafayette	125,000
6	Kokomo	154,000
	……	

SID	Location	Tuition
1	Indianapolis	3000
2	W.Lafayette	3500
3	Lafayette	6000
4	Lafayette	6200
5	Indianapolis	7000
6	Indianapolis	7900
7	Kokomo	8200
8	Kokomo	8200

HID	SID	Price + 10 x Tuition
1	3	150000
1	4	152000
2	2	145000
3	1	141000

Figure 12 : A Top-k Join example [13]

A simple way to calculate the above mentioned Top-k join would be summarized in the following steps:

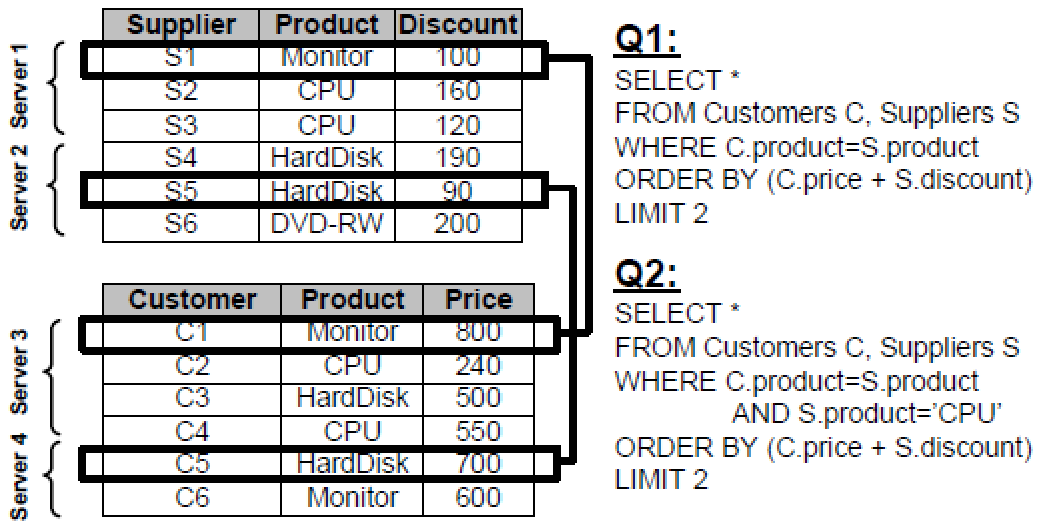
- Retrieve the list of the cheapest houses and the list of cheapest schools
- Perform a join on location for all schools and houses
- Calculate the cost of house and tuitions for each joined tuple
- Sort in ascending order by the cost

Such queries when performed on a traditional relational database, have two main disadvantages. First of all, the Top-k results will be returned to the user only after the join of all tuples is complete, so the larger the tables the longer the processing time. The second disadvantage is that this algorithm does not take into consideration the processing load that corresponds to each join value. For example, if one location has many more schools and houses than other locations, then the processing time for this specific location will be greater than the processing time needed to perform joins for other locations.

On the other hand, when referring to distributed databases, a Top-k join query, joins m relations Ri that may be fragmented into several parts stored at different servers. The resulting tuples are ordered using a scoring function f (order by clause) and the top-k answers based on their scores are returned to the user (limit clause). Rank join queries adhere to the following template, where relations Ri are widely distributed to different servers [4]:

***SELECT** some attributes*
***FROM** R1, R2,. . . ,Rm*
***WHERE** join condition AND selection predicates*
***ORDER BY** f(R1.s1,R2.s2, ...,Rm.sm)*
***LIMIT** k*

Two examples of Top-k join queries in distributed databases are presented by Doulkeridis et al. [4] and depicted in figure 13. In the example, two relations are used. A Suppliers relation (from inventory department) and a Customers relation (from sales department) which are both fragmented in a horizontal manner over the servers which are located in different geographic places. A product is sold to a customer at a certain price, as shown at the Customers table. The company buys these products from suppliers, as depicted in the Suppliers table. A sale maximizes the associated profit, if the amount paid by a customer plus the discount offered by a supplier is maximized .In this scenario, the sales manager is interested in finding the 2 least profitable sales for any product (Query Q1). Similarly, in Query Q2 the manager is only interested in sales of a specific product (CPU in this example) [4].



Supplier	Product	Discount
S1	Monitor	100
S2	CPU	160
S3	CPU	120
S4	HardDisk	190
S5	HardDisk	90
S6	DVD-RW	200

Customer	Product	Price
C1	Monitor	800
C2	CPU	240
C3	HardDisk	500
C4	CPU	550
C5	HardDisk	700
C6	Monitor	600

Q1:

```
SELECT *
FROM Customers C, Suppliers S
WHERE C.product=S.product
ORDER BY (C.price + S.discount)
LIMIT 2
```

Q2:

```
SELECT *
FROM Customers C, Suppliers S
WHERE C.product=S.product
        AND S.product='CPU'
ORDER BY (C.price + S.discount)
LIMIT 2
```

Figure 13 : Example of distributed Top-k join queries [4]

This distributed rank join processing, results in many round trips over the network since the tuples cannot be read iteratively from the table. Another disadvantage is that the tuples that we will be fetched and eventually joined might be significantly more that the k requested by the user.

As mentioned earlier, the scope of this thesis is to improve the phases of MapReduce computational model to increase the efficiency of the model for Top-k join queries on large and distributed data sets, while maintaining scalability. This is achieved with the use of certain techniques discussed in detail in chapter 4.

4 Design Approach

4.1 Architectural Overview

The proposed solution to the problem of efficient processing of Top-k join queries that apply to the following template:

Select *R0.id, R1.id, (R0.score+R1.score) as score from R0, R1*

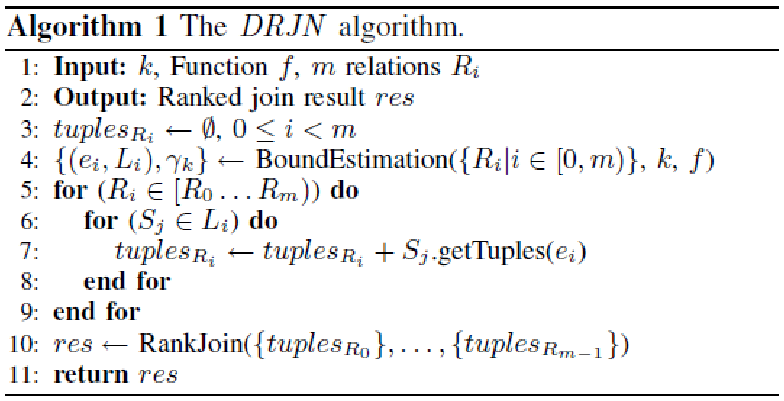
Where *R0.joining_attribute = R1.joining_attribute*

Order by *(R0.score+R1.score) asc*

Limit *k*

involves a number of techniques and is based on the DRJN algorithm proposed by Doulkeridis et al [4].

The DRJN algorithm calculates queries similar to the above in distributed databases. Figure 14 presents the DRJN algorithm in pseudocode for distributed rank join query processing on one server S_Q . S_Q invokes the BoundEstimation function which returns a bound e_i for each relation R_i and a list of servers L_i that store the tuples specified by the bound and the estimated score γ_k of the k-th join result. Then for all servers in L_i, the method getTuples(e_i) is invoked which fetches the records from the other servers with the restriction set from e_i. When all necessary tuples are fetched, S_Q performs a local centralized rank join algorithm and calculates the Top-k join result. The critical part of the DRJN algorithm is the procedure used for estimating the appropriate bounds of the scoring values.



```
Algorithm 1 The DRJN algorithm.
Input: $k$, Function $f$, $m$ relations $R_i$
Output: Ranked join result $res$
$tuples_{R_i} \leftarrow \emptyset,\ 0 \le i < m$
$\{(e_i, L_i), \gamma_k\} \leftarrow$ BoundEstimation($\{R_i | i \in [0, m)\}, k, f$)
for ($R_i \in [R_0 \dots R_m)$) do
    for ($S_j \in L_i$) do
        $tuples_{R_i} \leftarrow tuples_{R_i} + S_j$.getTuples($e_i$)
    end for
end for
$res \leftarrow$ RankJoin($\{tuples_{R_0}\}, \dots, \{tuples_{R_{m-1}}\}$)
return $res$
```

Figure 14 : DRJN Algorithm [4]

The solution proposed in this thesis is the use of MapReduce jobs for the calculation of Top-k joins such as the one presented earlier. The aim is to enhance the MapReduce steps of a simple reduce side join algorithm in order to eliminate the useless records which do not produce top k results.

A simple MapReduce would calculate the above mentioned query in a naïve way. The two relations R0 and R1 would be stored in HDFS. The simple reduce side join would read all contents from both files and perform a sort by joining attribute as the key. Secondary sort technique described earlier in 3.1.4 would be used as well. Then the reducers would perform a join for all records of both datasets, they would calculate the sum of the scoring attributes and finally perform a sort by this sum in all joined tuples.

The above described algorithm can be very expensive in terms of memory and processing time. In addition to this, most of the processing done by map and reduce tasks is useless since, only k joined tuples will be eventually written in the job output.

The idea proposed is to eliminate as much as possible the input to the reduce tasks so that the joins performed are as many needed to produce the Top-k results. This is achieved by estimating a bound for each input dataset that defines how many records will be fetched from each relation, thus making use of the BoundEstimation function of the DRJN algorithm. The BoundEstimation algorithm is described in detail in 4.3.

Once the bounds are estimated, some extra logic must be added so that the job uses these bounds in order to stop reading records from the input relations, when the bounds are reached. This algorithm is described in 4.4.

Both Score Bounds Estimation and Early Termination techniques are injected in the Map phase of the job and make sure that less records reach the Reduce phase, so less joins will be performed.

However, a problem still remains. If one value of the joining attribute appears in a very large number of records in both files, then the joins that correspond to this value will be many more comparing to the joins that will result from other values. Since, the key of the MapReduce job is the joining attribute value, the reducer that will be responsible for that specific key will be unequally overloaded comparing to others. So, in this case the whole job might be delayed by a single reducer. In fact, this is situation is common and can be observed in many kinds of real datasets. An example would be a dataset which follows a Zipfian distribution with high skewness in the joining attribute.

A solution to the problem described above would be a load balancing algorithm that would predict the load of the reduce tasks and make a proper distribution even before the Reduce phase, so as to avoid the overloading of a single reduce task. The load balancing algorithm proposed is described in detail in 4.5. In contrast to the Early Termination technique, Load Balancing is an enhancement in the Shuffle phase of the MapReduce job.

The combination of Early Termination and Load Balancing techniques should already increase the efficiency of the MapReduce job. However, there is a need for one more enhancement. The relations are stored in HDFS where they are divided in HDFS blocks. Once the job starts its execution, the Input Splits are calculated as mentioned in the earlier section 3.1.5, and passed as inputs to the map tasks. Even though the early termination technique sets boundaries to the map tasks, the whole datasets will still be fed to the map tasks. The last technique proposes a way to eliminate the input splits to as many needed to extract the valuable records that will finally lead to the Top-k joins. The estimation of the necessary input splits is described in further detail in 4.6.

All techniques proposed require a pre-process of the datasets to be joined. The results of this preprocessing are stored in data synopses like histograms and uploaded in HDFS so that they can be accessible from the MapReduce job. Further details for the data synopses are provided in 4.3 .

The overall architecture of the framework proposed is presented in figure 15, where each technique is visually associated with the corresponding step of the MapReduce job that it aims to enhance.

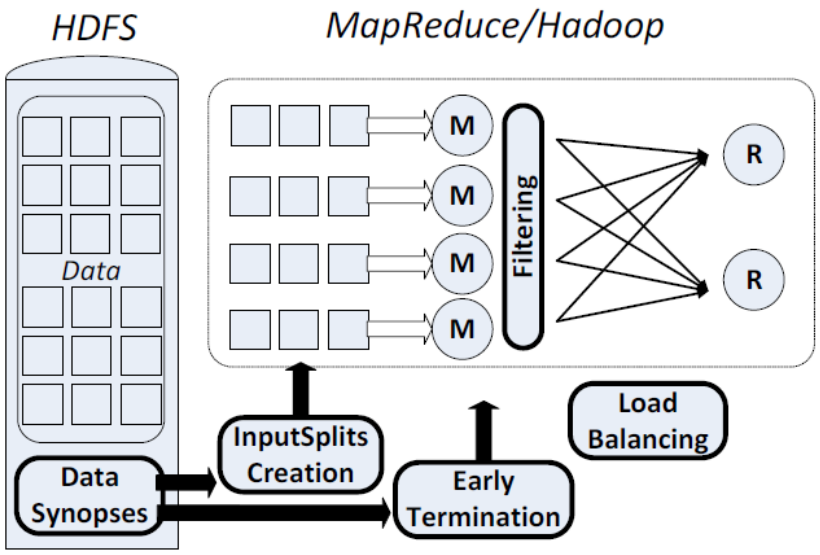
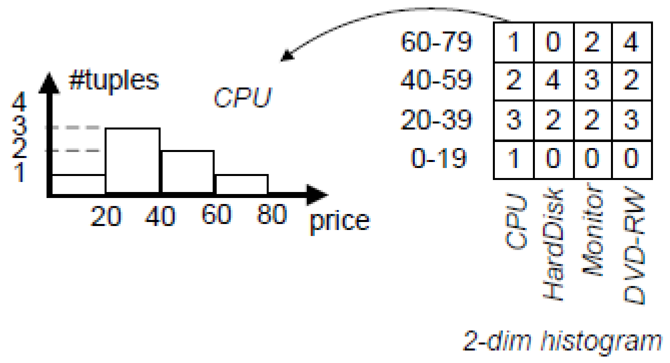


Figure 15 : Architecture overview [4]

4.2 Data Synopses – Histograms

The estimation of the bounds which restrict the number of records fetched from each relation, is based on data synopses and more accurately on histograms. For each relation Ri, a histogram is maintained. This is a two dimensional histogram that records the number of tuples in Ri that correspond to each distinct value of an attribute, that fall in a range of values (defined by the bin's low and high value). As far as the join attributes are concerned, each histogram bin represents only one join value. For each distinct value of the join attribute, the set of bins of the corresponding histogram can be viewed as a one-dimensional histogram that approximates the distribution of scoring values for this join value. In the example presented earlier in figure 13, a histogram of relation *Suppliers* and join attribute *product* equal to *'CPU'* captures the number of products of type CPU for different ranges of *price* (scoring attribute), as depicted in figure 16.



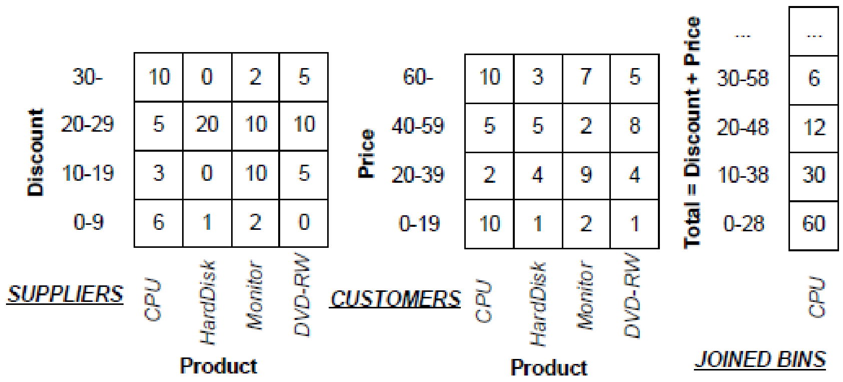
60-79	1	0	2	4
40-59	2	4	3	2
20-39	3	2	2	3
0-19	1	0	0	0
	CPU	HardDisk	Monitor	DVD-RW

Figure 16 : Histogram of relation Suppliers [4]

The DRJN algorithm performs a rank-join on histogram bins rather than on the actual tuples. Histogram bins are accessed sorted in ascending order of their score range. Moreover, histograms bins of different relations (also mentioned as *individual bins*) are joined and produce join combinations of bins. A valid join combination of bins is produced by a set of bins with the same value of the join attribute. This new bin is referred to as *joined bin*. For each joined bin, the number of tuples is computed by multiplying the number of tuples in the individual bins. Furthermore, the score of a joined bin is estimated by applying the scoring function f on the higher value of score range of each individual bin. Thus, the score of the joined bin is an upper bound of the score of any join tuple produced by the individual bins [4].

An example of joining histograms is presented in figure 17. The histograms appearing, depict the data distribution of the relations presented in figure 13. Each bin of the histogram for the R0 relation (Suppliers) represents the number of suppliers that handle a particular product for the range of prices specified by the boundaries of the bin. Accordingly, each bin of the histogram of the relation R1 (Customers) represents the number of customers that request for a particular product, e.g. CPU. On the right, some joined bins are depicted for product 'CPU' when the scoring function f is the sum of discount and price.

For instance, the combination of the first two bins of each histogram for 'CPU', which contain 6 and 10 tuples respectively, produce a joined bin of 60 (=6 × 10) tuples with score in the range 0-28. The score of the tuples in this joined bin is set to 28, i.e., the high value of the range. Recall that the aim is to retrieve the top-k results with minimum scores [4].



SUPPLIERS

Discount	CPU	HardDisk	Monitor	DVD-RW
30-	10	0	2	5
20-29	5	20	10	10
10-19	3	0	10	5
0-9	6	1	2	0

Product

CUSTOMERS

Price	CPU	HardDisk	Monitor	DVD-RW
60-	10	3	7	5
40-59	5	5	2	8
20-39	2	4	9	4
0-19	10	1	2	1

Product

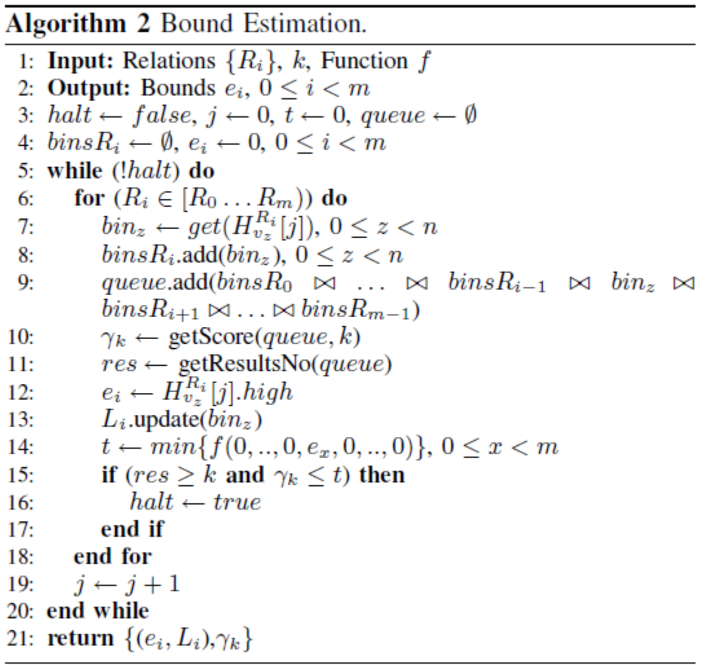
JOINED BINS

Total = Discount + Price	CPU
...	...
30-58	6
20-48	12
10-38	30
0-28	60

Figure 17 : Example of joining histograms [4]

4.3 Score Bounds Estimation

The objectives of the BoundEstimation algorithm are to identify the histogram bins that produce at least k join tuples with the smallest scores and to ensure that no other combination of histogram bins can produce join tuples with smaller score values [4]. For this purpose, the algorithm joins the histogram bins until the joined tuples exceed the k tuple requested and the score of any join tuple produced by any unseen histogram bin is not smaller than the score (γ k) of the current k-th join tuple. The above described logic is depicted in pseudocode as presented by Doulkeridis et al. [4] in figure 18.



```
Algorithm 2 Bound Estimation.
Input: Relations $\{R_i\}$, $k$, Function $f$
Output: Bounds $e_i$, $0 \leq i < m$
$halt \leftarrow false$, $j \leftarrow 0$, $t \leftarrow 0$, $queue \leftarrow \emptyset$
$binsR_i \leftarrow \emptyset$, $e_i \leftarrow 0$, $0 \leq i < m$
while (!$halt$) do
    for ($R_i \in [R_0 \ldots R_m]$) do
        $bin_z \leftarrow get(H^{R_i}_{v_z}[j])$, $0 \leq z < n$
        $binsR_i$.add($bin_z$), $0 \leq z < n$
        $queue$.add($binsR_0 \bowtie \ldots \bowtie binsR_{i-1} \bowtie bin_z \bowtie binsR_{i+1} \bowtie \ldots \bowtie binsR_{m-1}$)
        $\gamma_k \leftarrow$ getScore($queue, k$)
        $res \leftarrow$ getResultsNo($queue$)
        $e_i \leftarrow H^{R_i}_{v_z}[j].high$
        $L_i$.update($bin_z$)
        $t \leftarrow min\{f(0,..,0,e_x,0,..,0)\}$, $0 \leq x < m$
        if ($res \geq k$ and $\gamma_k \leq t$) then
            $halt \leftarrow true$
        end if
    end for
    $j \leftarrow j + 1$
end while
return $\{(e_i, L_i), \gamma_k\}$
```

Figure 18 : Bound Estimation algorithm [4]

4.4 Early Termination

The term early termination in the context of this thesis refers to a map reduce job that performs a rank aware query, e.g. a top-k join, without accessing the whole dataset but only parts of the input datasets which contain the tuples that will certainly produce the top-k joined tuples. The bound estimation algorithm presented earlier in 4.3 provides a way to restrict the accessed tuples of each relation by calculating the bounds of the scoring attributes. This means that if each relation R0, R1 was sorted by the scoring attribute, then in order to perform the join, the MapReduce job would fetch all records with a scoring attribute lower or equal to the bound estimated for each relation. An example of the use of bounds to achieve early termination is presented in figure 19. The score bound for relation Suppliers (R0) is estimated via the BoundEstimation algorithm to 120 and to 500 for the relation Customers (R1).

Therefore, in a MapReduce job that uses early termination, the input datasets must be sorted by the scoring attribute in an ascending order so that it can make use of the bounds calculated by the BoundEstimation algorithm.

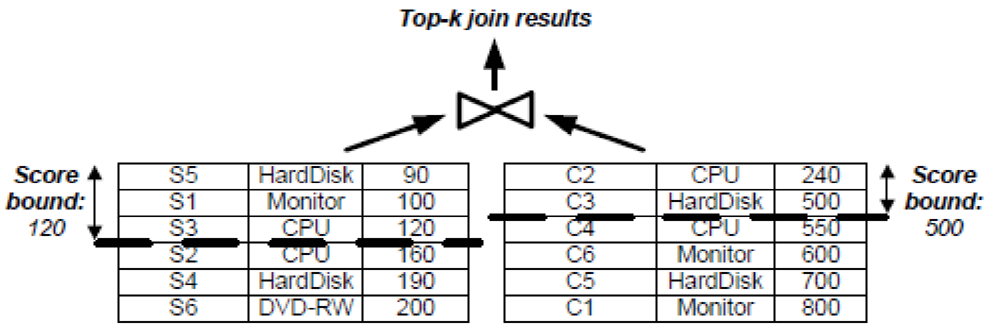


Figure 19 : Example of Score Bounds [4]

4.5 Load Balancing

As described earlier in the previous sections, the early termination technique which uses the BoundEstimation algorithm, provides a way to eliminate the input of the MapReduce job and therefore to speed up the execution time of the top-k join.

However, the problem of fair distribution to the reducers that will actually perform the joins, still remains. A solution to this problem is an algorithm that can estimate the load that corresponds to each reducer and thus distribute the keys as fair as possible so that all reducers have approximately the same execution time.

In 4.3, the main component of the BoundEstimation algorithm was the join performed among the histogram bins. These joins are performed on the joining attributes while at the same time the scoring function is applied. The result of each join between two bins, represents the number of joined tuples that can be produced with scoring values of a certain range. Therefore, the information: *how many joins correspond to each joining attribute value*, is given by the BoundEstimation algorithm along with the bounds.

Since, the correspondence *joining attribute value* → *number of joins* is known, the only thing missing to complete the load balancing solution, is an algorithm that distributes the keys properly to the reducers.

The algorithm used for the load distribution is an existing heuristic algorithm, the Longest Processing Time (LPT) algorithm, which has been widely used in systems that perform parallel processing [18]. The input to LPT is a number of n jobs with processing time {p1,p2,…,pn} and the output is the assignment of jobs to the machines. The assignment is performed as follows:

1. *Order the jobs in descending order according to their processing times*
2. *In this order, assign each job to the machine that currently has the least work assigned to it*

In the solution presented, there is a slight variation of LPT, as the inputs to the LPT are not the processing times of the reduce tasks but the number of joins that correspond to each joining attribute, thus the load that each reduce task is expected to have. Therefore, the output would be the assignment of each key (joining attribute value) to a certain reducer.

4.6 Input Splits Estimation

Even if the records processed by the map tasks are eliminated with the use of the early termination technique, the number of input splits remain the same. The whole datasets are loaded as inputs to the map tasks even if not all of them contain useful records.

A solution to this problem is an algorithm for the calculation of the number of input splits that need to be passed to the map tasks. Input splits are logical divisions of the datasets that exist in the HDFS. However, their actual contents derive from the HDFS blocks. So, the idea is to implement an algorithm that fetches from the HDFS the number of bytes needed to calculate the top-k joins. To achieve that, we need a correlation among score values and byte position. This means that for each relation we have to produce a file that contains the information of how many bytes we have to read from a certain relation in order to reach a certain score value. For example let's consider that the following lines are extracted from the file which contains the correlation *bytes – score value*. The first column represents the number of bytes and the second column represents the score value that can be found when the bytes in the first column are read. If the estimated bound for relation R0 is 500 then according to the file we have to access *268560884* bytes from relation R0 to be sure that we will fetch the records that will produce top-k joins.

134301780 454

268560884 892

402724529 1304

……………………

5638355373 9917

5773621539 9952

5911655836 9978

Accordingly, there will be a file for relation R1 as well which will contain similar information. It is important to notice that these files are based on the sorted by score value form of R0, R1 relations (referred to as R0sorted, R1sorted respectively).

5 Implementation

All four algorithms are implemented in MapReduce jobs (in Java) and perform reduce side Top-k joins between two datasets. The joins performed by these algorithms could be described by the following SQL query:

Select *R0.id, R1.id, (R0.score+R1.score) as score from R0, R1*

Where *R0.joining_attribute = R1.joining_attribute*

Order by *(R0.score+R1.score) asc*

Limit *k*

As the versions augment from 1 to 4, the techniques used, serve the goal of adding efficiency in terms of execution time and resources used. The first algorithm is a simple implementation of reduce side join in MapReduce and was implemented only to be used for experimental purposes, so as to measure the efficiency of the other three algorithms against the simple algorithm. The other three algorithms involve the following techniques: Early Termination, Load Balancing and Custom Input Split creation. Each algorithm contains the techniques of the previous one, so the fourth algorithm consists of all three. Therefore, the fourth algorithm is expected to be the most efficient.

The table below is a short presentation of the algorithms and the techniques used in each of them:

Name	Name used in code	Techniques
Algorithm 1	RSJSimple	-
Algorithm 2	RSJET	Early Termination
Algorithm 3	RSJETLBDistCache	Early Termination Load Balancing
Algorithm 4	RSJETLBIndex	Early Termination Load Balancing Custom Input Splits

Table 1: Algorithms and techniques

5.1 Algorithm 1: RSJSimple

The first algorithm or RSJSimple, as referred to in the implementation, is the simple implementation of the top-K join described in the introduction section. No special technique is used in this algorithm and the purpose of this implementation is only for experimental reasons.

5.1.1 RSJSimple - Pseudocode

In this section the components of RSJSimple algorithm described in 5.1.3, are presented in pseudocode.

map method (class: Map.java):

```
Get input file name from current split info
if input file name starts with R0
   set variable tag = 0
else
   set variable tag = 1
Split input line to words
Set variable joining_attribute = first word of line without the first letter
Set variable scoring_attribute = second word of line
Set composite_key.joiningAttribute = joining_attribute
Set composite_key.tag = tag
Set composite_key.score = scoring_attribute
Set variable taggedLine = tag+line
Return composite_key, taggedLine
```

getPartition method (class: RSJPartitioner.java)

```
Set variable joiningAttribute = composite_key.joiningAttribute
Calculate partition as (joiningAttribute.hashCode & max integer value) MOD numberOfReduceTasks
Return partition
```

compare method (class: SortingComparator.java)

```
Read fields composite_key1.joiningAttribute, composite_key2.joiningAttribute
[if composite_key1.joiningAttribute > composite_key2.joiningAttribute
    Return true
[else  if composite_key1.joiningAttribute = composite_key2.joiningAttribute
        [if composite_key1.score > composite_key2.score
            Return true
       [else
           Return false
 [else Return false
```

compare method (class: GroupingComparator.java)

```
Read fields composite_key1.joiningAttribute, composite_key2.joiningAttribute
[if composite_key1.joiningAttribute > compositeKey2.joiningAttribute
    Return true
[else Return false
```

reduce method (class: RSJReducer.java)

```
Do until k values have been read from R0
     Read and store value for joining attribute from R0
Do until k values have been read from R1
    Read and store value for joining attribute from R1
Do until k joined tuples have been produced
  Join value from first input file with value from second input file
Return k joined tuples
```

5.1.2 Inputs

The inputs of this map-reduce job are four:

1. *The input folder path:* The input folder path should contain two files: R0.txt, R1.txt which represent the datasets to be joined. Each dataset contains 3 columns: id, joining attribute, score.

2. *The output folder path:* This argument indicates to the job the folder in which the join results will be produced.

3. *The number of k*

4. *The number of reducers:* The number of reducers is passed as an argument so that we can run experiments using different number of reducers.

5.1.3 Components

- *DriverRSJ.java*
  The class which contains the job configuration and starts the Map-Reduce job.

- *CompositeKey.java*
  This class contains three variables representing: joining attribute, score and tag. In each instance of this class, the first two variables, joiningAttribute and score carry the actual values of one record of one of the datasets. The field tag indicates the dataset from which the record came from e.g. the value 0 represents the dataset R0.txt.

- *Map.java*
  This class implements the map method. The map inputs are the outcome of the FileInputFormat. They are pairs of LongWritable keys and Text values. So, each map task receives a pair which consists of a key (a number) and a value which is one record from one of the datasets. The map task creates a CompositeKey instance in which it sets the joiningAttribute and score values as derived from the record that it received. Afterwards, the code of the map task checks the filename from which the record came. This information is carried on the InputSplit that feeds each map task with the input pairs. So, if the filename is R0.txt then the map task sets the CompositeKey.tag to 0. If the filename is R1.txt then the map task sets the CompositeKey.tag to 1. Finally, the output key - value pair of every map task contains:
    - the CompositeKey instance as the key
    - a Text instance which consists of tag+record

- *SortingComparator.java*
  Compares by the CompositeKey instances, which means that it sorts the Map output keys by comparing joiningAttribute and score variables.

- GroupingComparator.java
  Compares by the natural key which is the joiningAttribute.

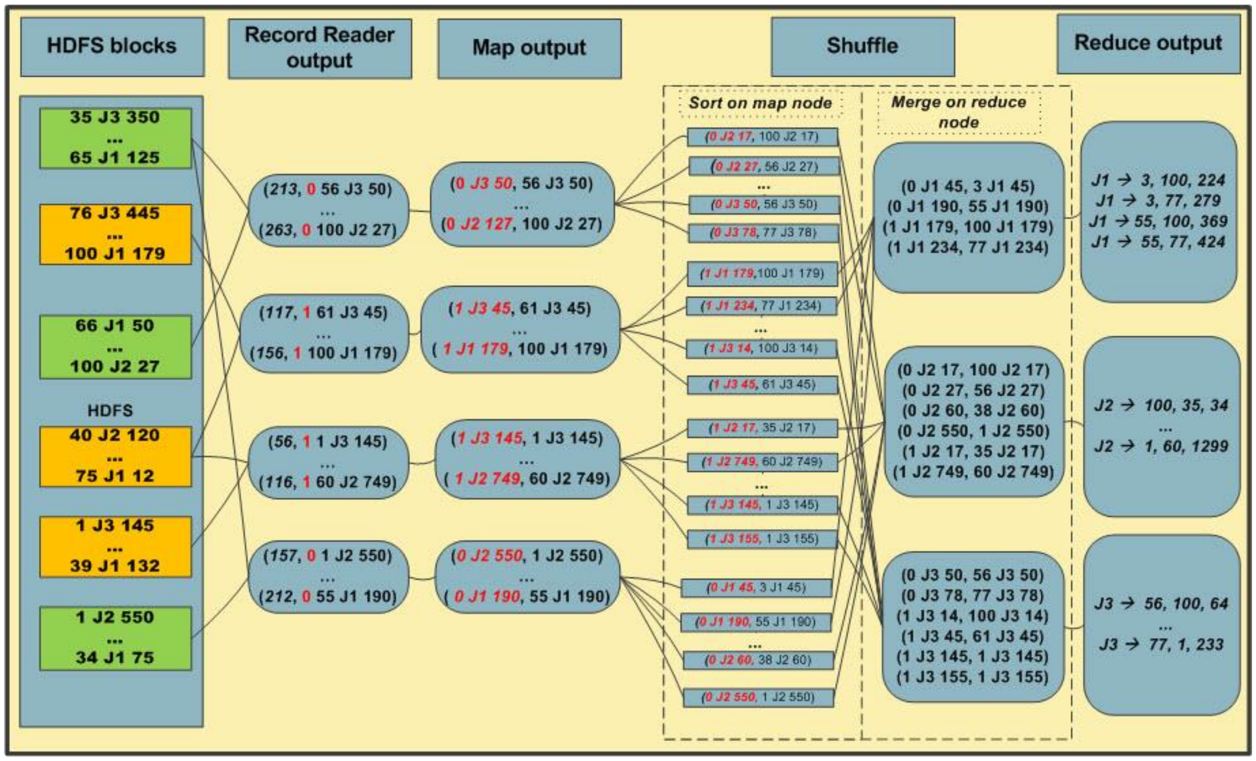
- *RSJPartitioner.java*
  The role of RSJPartitioner is to make sure that all joiningAttributes with the same value are delivered to the same reducer. For this purpose, it uses a hash method based on the joining attribute. SortingComparator along with GroupingComparator and RSJPartitioner follow the SecondarySort implementation as explained in *Hadoop The Definitive Guider by Tom White*[27] (chapter 8). The implementation of these classes makes sure that the following happen:
    - all records for same joining attribute reach the same reducer
    - all score values reach the reducer in sorted ascending order

- *ScoredTuple.java*
  An instance of ScoredTuple contains a variable named score and a variable named value. The method *compareTo(ScoredTuple obj)* implemented in this class is invoked whenever an instance of ScoredTuple is inserted in a PriorityQueue (see RSJReducer.java) to compare the score inserted against the first element of the PriorityQueue.

- *RSJReducer.java*
  This class implements the reduce method and performs the join between the two datasets. The input of each reducer is a CompositeKey and a list of tagged records (map's output). It collects the k first records of the first dataset and the k first records of the second dataset. Once k records have been collected from each relation, it is guaranteed (from the secondary sort step) that the reducer has the k smallest scores for each joining attribute from both datasets. Therefore, it performs a join among those records while performing the function R0.score+R1.score and the outcome of each join is stored in a ScoredTuple. The ScoredTuple is then stored in a PriorityQueue. Finally, the k top elements of the PriorityQueue are sent to collector as the output of the job.

5.1.4 RSJSimple Dataflow

Figure 20 is a visual representation of the data flow in RSJSimple. In the beginning, the two relations R0, R1 are stored in HDFS, divided into HDFS blocks. The default RecordReader processes the input splits and produces key-value pairs. Each pair consists of a random number as the key and a whole line from one relation tagged with its source, as the value. The map tasks receive these pairs and perform a transformation on them. Thus, the outcome of the map tasks is a number of key-value pairs, each in the form of a composite key as the key and the whole tagged line as the value. In the shuffle phase, a sorting is performed on map nodes and all key-value pairs are sorted by joiningAttribute and score and grouped by joining attribute. After sorted and grouped, the key-value pairs are sent to the corresponding reduce nodes. All partitions (groups) that come from map nodes are merged on the reduce nodes. Once the merge phase finishes, the reduce tasks start their processing. The reduce phase output is a number of key-value pairs where a key is a joiningAttribute and the corresponding value is a joined tuple in the form: *R0.id R1.id R0.score+R1.score.*



HDFS blocks
Record Reader output
Map output
Shuffle
Reduce output
Sort on map node
Merge on reduce node
35 J3 350 ... 65 J1 125
76 J3 445 ... 100 J1 179
66 J1 50 ... 100 J2 27
HDFS
40 J2 120 ... 75 J1 12
1 J3 145 ... 39 J1 132
1 J2 550 ... 34 J1 75
(213, 0 56 J3 50) ... (263, 0 100 J2 27)
(117, 1 61 J3 45) ... (156, 1 100 J1 179)
(56, 1 1 J3 145) ... (116, 1 60 J2 749)
(157, 0 1 J2 550) ... (212, 0 55 J1 190)
(0 J3 50, 56 J3 50) ... (0 J2 127, 100 J2 27)
(1 J3 45, 61 J3 45) ... (1 J1 179, 100 J1 179)
(1 J3 145, 1 J3 145) ... (1 J2 749, 60 J2 749)
(0 J2 550, 1 J2 550) ... (0 J1 190, 55 J1 190)
(0 J2 17, 100 J2 17)
(0 J2 27, 56 J2 27)
(0 J3 50, 56 J3 50)
(0 J3 78, 77 J3 78)
(1 J1 179,100 J1 179)
(1 J1 234, 77 J1 234)
(1 J3 14, 100 J3 14)
(1 J3 45, 61 J3 45)
(1 J2 17, 35 J2 17)
(1 J2 749, 60 J2 749)
(1 J3 145, 1 J3 145)
(1 J3 155, 1 J3 155)
(0 J1 45, 3 J1 45)
(0 J1 190, 55 J1 190)
(0 J2 60, 38 J2 60)
(0 J2 550, 1 J2 550)
(0 J1 45, 3 J1 45) (0 J1 190, 55 J1 190) (1 J1 179, 100 J1 179) (1 J1 234, 77 J1 234)
(0 J2 17, 100 J2 17) (0 J2 27, 56 J2 27) (0 J2 60, 38 J2 60) (0 J2 550, 1 J2 550) (1 J2 17, 35 J2 17) (1 J2 749, 60 J2 749)
(0 J3 50, 56 J3 50) (0 J3 78, 77 J3 78) (1 J3 14, 100 J3 14) (1 J3 45, 61 J3 45) (1 J3 145, 1 J3 145) (1 J3 155, 1 J3 155)
J1 → 3, 100, 224
J1 → 3, 77, 279
J1 → 55, 100, 369
J1 → 55, 77, 424
J2 → 100, 35, 34 ... J2 → 1, 60, 1299
J3 → 56, 100, 64 ... J3 → 77, 1, 233

Figure 20: RSJSimple Dataflow

5.2 Algorithm 2: RSJET

The second algorithm or RSJET, as referred to in the implementation, has the same logic with Algorithm 1 concerning the reduce side join implementation. However, the technique of Early Termination is applied so as to achieve faster execution time. Therefore, many components are exactly the same with the ones of Algorithm 1 in terms of implementation but some new are also added.

So the following components remain the same:

- *CompositeKey.java*
- *SortingComparator.java*
- *GroupingComparator.java*
- *RSJPartitioner.java*
- *ScoredTuple.java*

- *RSJReducer.java*

Whereas the following are **added**/altered:

- *DriverRSJ.java*
- ***SelectiveInputFormat.java***
- ***SelectiveRecordReader.java***
- *Map.java*

5.2.1 RSJET – Pseudocode

In this section, the new and the altered components introduced in 5.2.3 are presented in pseudocode. The parts of the algorithm that have been changed or added (comparing to RSJSimple) are highlighted.

computeBounds method (class: DriverRSJ.java)

```
Call BoundEstimation algorithm
Return bounds for R0sorted and R1sorted
```

next method (class: SelectiveRecordReader)

```
Read input file split path name
[if input file split path name contains R0
    Set variable tag=0
[Else
    Set variable tag = 1
Read bounds for R0sorted, R1sorted from job Configuration
Read the next line from input split
Split the line in words
Set variable score = the second word
[if tag=0
     [if score>bound for R0sorted
         //Do not create key
         Return
    [else  create key for Map
```

```
[else
            [if score> bound for R1sorted
              //Do not create key-value pair
              Return
          [else create key-value pair for Map
```

map method (class: Map.java):

```
Get input file name from current split info
if input file name starts with R0
    Set variable tag = 0
else
    Set variable tag = 1
Split input line to words
Set variable joining_attribute = first word of line without the first letter
Set variable scoring_attribute = second word of line
Read bounds for R0sorted, R1sorted from Configuration
if (tag=0 and scoring_attribute < bound for R0sorted) or (tag = 1 and scoring_attribute< R1sorted)
        Set composite_key.joiningAttribute = joining_attribute
        Set composite_key.tag = tag
        Set composite_key.score = scoring_attribute
        Set variable taggedLine = tag+line
        Return composite_key, taggedLine
else
       Do not create key-value pair
```

5.2.2 Inputs

The inputs of this map-reduce job are seven:

1. *The input path for R0sorted.txt*
2. *The input path for R1sorted.txt*
3. *The output folder path*
4. *The input path to the folder that contains hist0.txt and hist1.txt*
5. *The number of k.*
6. *The number of reduce tasks*
7. *The number of different values that a joining attribute can have (number of histogram bins)*

5.2.3 Components

- *DriverRSJ.java*
  The class contains the job configuration and starts the Map-Reduce job. Before the Map-Reduce job starts, the method computeBounds is invoked. This method returns the bounds for both datasets respectively which are then stored as properties on the job configuration so that they can be visible from the rest of the components.

  Method computeBounds:
  This method is implemented in the DriverRSJ class and it creates an instance of BoundEstimatorHist (package: boundEstimation) so as to invoke the method BoundEstimatorHist.estimate, which calculates the bounds of each dataset for the given k, based on the histograms. The package boundEstimation contains the full implementation of the BoundEstimation algorithm described in 4.3. One of the steps of this algorithm is the join between the two histograms to estimate the join results that correspond to each joining attribute value.

- ***SelectiveInputFormat.java***
  This class represents a custom InputFormat and extends FileInputFormat. Its role is to invoke the custom record reader: SelectiveRecordReader.java.

- ***SelectiveRecordReader.java***
  This class implements RecordReader<LongWritable, Text> and is the class that feeds the map tasks with the key-value pairs it expects. The purpose of this custom RecordReader is to read records from each sorted by score dataset until it reaches a record which contains score values smaller or equal to the dataset bound. This technique eliminates the execution time of the RecordReader (because it does not have to process the whole dataset), however it does not have

any effect on the size of input split that is finally loaded as the input of a map task (which means that map tasks will finally receive all the records from both datasets).

- *Map.java*
  This class implements the map method. The map input key-value pairs are the outcome of the SelectiveInputFormat which are pairs of LongWritable keys and Text values. So each map task receives a pair which consists of a key (a number) and a value which is one record from one of the datasets.
  Each map task reads the bounds from the configuration properties and performs a check on the score of the map record that it receives. So, the record is sent to the reduce phase only if the score contained in the record was smaller or equal than the bound.
  If the above condition is met, then the map task creates an instance of CompositeKey in which it sets the joiningAttribute and score values as derived from the record that it received. Afterwards, the code of the map task checks the filename from which the record came. This information is carried on the InputSplit that feeds each map tak with the input pair. So, if the filename is R0sorted.txt the map task sets the CompositeKey.tag to 0. If the filename is R1sorted.txt then the map task sets the CompositeKey.tag to 1.
  Finally, the output pair key - value pair of every map task contains:
    - the CompositeKey instance as the key
    - a Text instance which consists of tag+record

5.2.4 RSJET Dataflow

The dataflow in RSJET is very similar to the dataflow of RSJSimple. The most significant difference is that only a certain number of map tasks produce output and this is because of the early termination algorithm. Therefore, in RSJET, less key-value pairs reach the reduce phase but it is guaranteed that these key-value pairs can produce the top-k joins. The dataflow of RSJET is depicted in figure 21

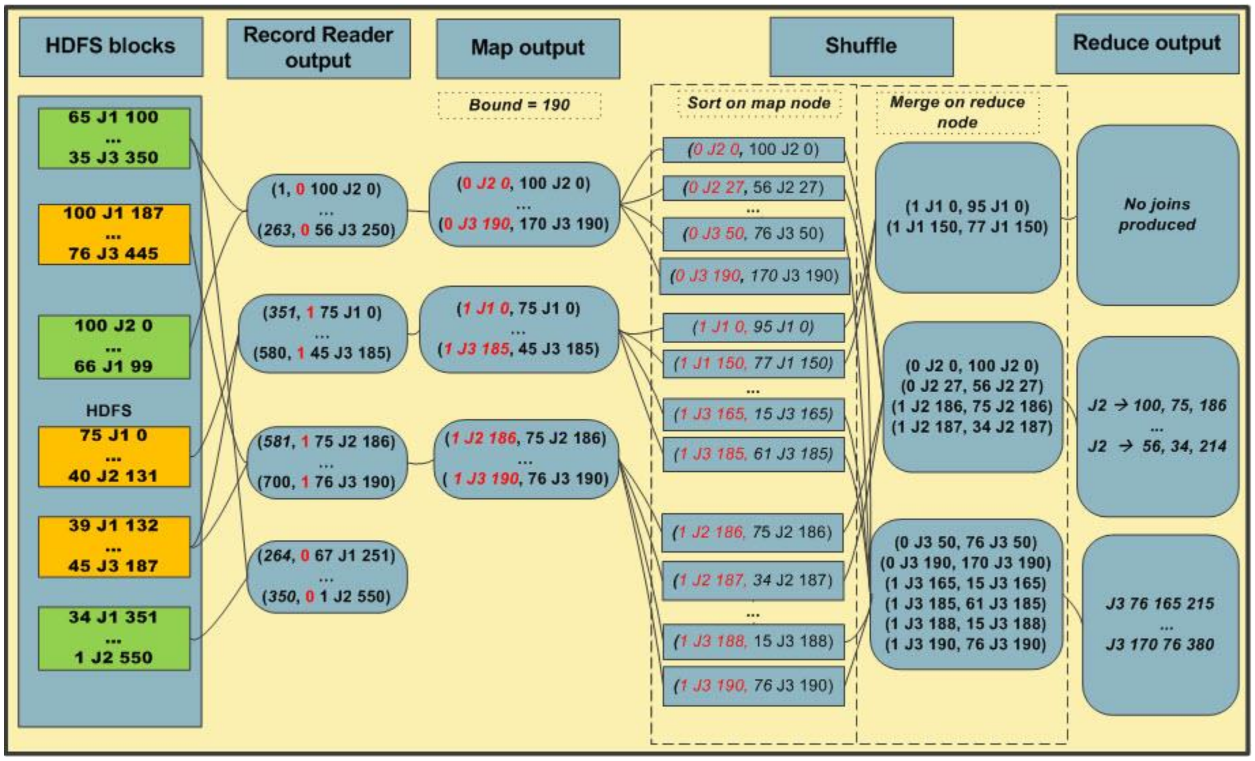

HDFS blocks
Record Reader output
Map output
Shuffle
Reduce output
Bound = 190
Sort on map node
Merge on reduce node
HDFS
65 J1 100 ... 35 J3 350
100 J1 187 ... 76 J3 445
100 J2 0 ... 66 J1 99
75 J1 0 ... 40 J2 131
39 J1 132 ... 45 J3 187
34 J1 351 ... 1 J2 550
(1, 0 100 J2 0) ... (263, 0 56 J3 250)
(351, 1 75 J1 0) ... (580, 1 45 J3 185)
(581, 1 75 J2 186) ... (700, 1 76 J3 190)
(264, 0 67 J1 251) ... (350, 0 1 J2 550)
(0 J2 0, 100 J2 0) ... (0 J3 190, 170 J3 190)
(1 J1 0, 75 J1 0) ... (1 J3 185, 45 J3 185)
(1 J2 186, 75 J2 186) ... (1 J3 190, 76 J3 190)
(0 J2 0, 100 J2 0)
(0 J2 27, 56 J2 27)
...
(0 J3 50, 76 J3 50)
(0 J3 190, 170 J3 190)
(1 J1 0, 95 J1 0)
(1 J1 150, 77 J1 150)
...
(1 J3 165, 15 J3 165)
(1 J3 185, 61 J3 185)
(1 J2 186, 75 J2 186)
(1 J2 187, 34 J2 187)
...
(1 J3 188, 15 J3 188)
(1 J3 190, 76 J3 190)
(1 J1 0, 95 J1 0) (1 J1 150, 77 J1 150)
(0 J2 0, 100 J2 0) (0 J2 27, 56 J2 27) (1 J2 186, 75 J2 186) (1 J2 187, 34 J2 187)
(0 J3 50, 76 J3 50) (0 J3 190, 170 J3 190) (1 J3 165, 15 J3 165) (1 J3 185, 61 J3 185) (1 J3 188, 15 J3 188) (1 J3 190, 76 J3 190)
No joins produced
J2 → 100, 75, 186 ... J2 → 56, 34, 214
J3 76 165 215 ... J3 170 76 380

Figure 21: RSJET Dataflow

5.3 Algorithm 3: RSETLBDistCache

The third algorithm or RSJETLBDistCache, as referred to in the implementation, has the same logic with Algorithm 2 concerning the reduce side join and the Early Termination technique. In this version, the technique of Load Balancing is also applied so as to achieve a better distribution of load to the reducers. Therefore many components are exactly the same with the ones of Algorithm 2 in terms of implementation but some new are also added.

So the following components remain the same:

- *SelectiveInputFormat.java*
- *SelectiveRecordReader.java*
- *SortingComparator.java*
- *GroupingComparator.java*
- *ScoredTuple.java*
- *RSJReducer.java*

Whereas the following are altered:

- *DriverRSJ.java*
- *Map.java*
- *CompositeKey.java*
- *RSJPartitioner.java*

5.3.1 RSJETLBDistCache – Pseudocode

In this section, the new and the altered components introduced in 5.3.4 are presented in pseudocode. The parts of the algorithm that have been changed or added (comparing to RSJET) are highlighted.

computeBounds method (class: DriverRSJ.java)

```
Call BoundEstimation algorithm
if bounds are calculated successfully
        Call assignDataToReducerHasMap to fill dataToRecucers
Return bounds for R0sorted and R1sorted
```

assignDataToReducerHasMap method (class: LPT.java)

```
Sort joinResultsPerJoinValue from most busy to least busy
Set counter = 0
Do for all joining attributes
    [if counter<= number of reducers
        assign  current joining attribute values to reducer:  $counter
        counter = counter+1
    [else
        Find reducer with minimum number of joinResults
        Assign current joining attribute to reducer with minimum number of joinResults
Return dataToReducersMap
```

map method (class: Map.java):

```
Get input file name from current split info
if input file name starts with R0
    set variable tag = 0
else
    set variable tag = 1
Split input line to words
Set variable joining_attribute = first word of line without the first letter
Set variable scoring_attribute = second word of line
Read bounds for first/second input file from Configuration
if (tag=0 and scoring_attribute < bound for first input file) or (tag = 1 and scoring_attribute< bound for
second input file)
        Set composite_key.joiningAttribute = joining_attribute
        Set composite_key.tag = tag
        Set composite_key.score = scoring_attribute
        Set variable taggedLine = tag+line
        Read assigned reducer for joining attribute from Distributed Cache
        Set variable composite_key.partition = assigned reducer
        Return composite_key, taggedLine
else
    // Do not create key-value pair
```

getPartition method (class: RSJPartitioner.java)

```
Set variable partition = composite_key.partition
Return partition
```

5.3.2 Inputs

The inputs of this map-reduce job are seven:

1. The input path for R0sorted.txt
2. The input path for R1sorted.txt
3. The output folder path
4. The input path to the folder that contains hist0.txt and hist1.txt
5. The number of k.
6. The number of reduce tasks
7. The number of different values that a joining attribute can have (number of histogram bins)

5.3.3 Preparation for Execution

In this algorithm, an expanded version of the method computeBounds is invoked before the job starts. This expanded version, apart from the BoundEstimatorHist.estimate, also invokes the method assignDataToReducerHasMap which is implemented in the class LPT (package: lpt). This method returns a HashMap which assigns each joining attribute to the appropriate reducer according to the load balancing algorithm. The information carried on this HashMap is stored in the dataToReducers.txt file. This file is then uploaded to the hdfs folder: hist and then added to the distributed cache so that it can be visible from the Map-Reduce job.

5.3.4 Components

- *DriverRSJ.java*
  The class which contains the job configuration and starts the Map-Reduce job. The Early Termination technique is implemented in this class in the same way as it was implemented in Algorithm 2. Moreover, the file dataToReducers.txt is placed in the DistributedCache so that it can be visible from all map tasks.

- *CompositeKey.java*
  A new field: *partition* is added in this class. So now, the class contains the fields: tag, joiningAttribute, score and partition.

- *Map.java*
  The code of Algorithm 2 is reused for the Map.java with an addition. Apart from the configuration properties which contain the bounds, each map task looks up the file: dataToReducers.txt to retrieve the corresponding reducer of the current received joining attributed. The reducer number is then stored in the variable *partition* of the CompositeKey instance.

- *RSJPartitioner.java*
  The code of this class is altered in this version. The method getPartition does not use a hash method anymore, but instead it returns the CompositeKey.partition value.

5.3.5 RSJETLBDistCache Dataflow

The main difference of the RSJETLBDistCache dataflow from the RSJET dataflow, is that all key-value pairs that derive from the map phase, carry with them the information of the responsible reducer. In this way, a load balancing is achieved, since the joining attribute with the greater number of joins will be assigned to the first reducer and this reducer will receive no more load. For example, in figure 22, joining attribute *J3* produces a greater number of joins comparing to J1 and J2. Therefore, it will be sent to Reducer 1 and Reducer 1 will handle only this joining attribute.

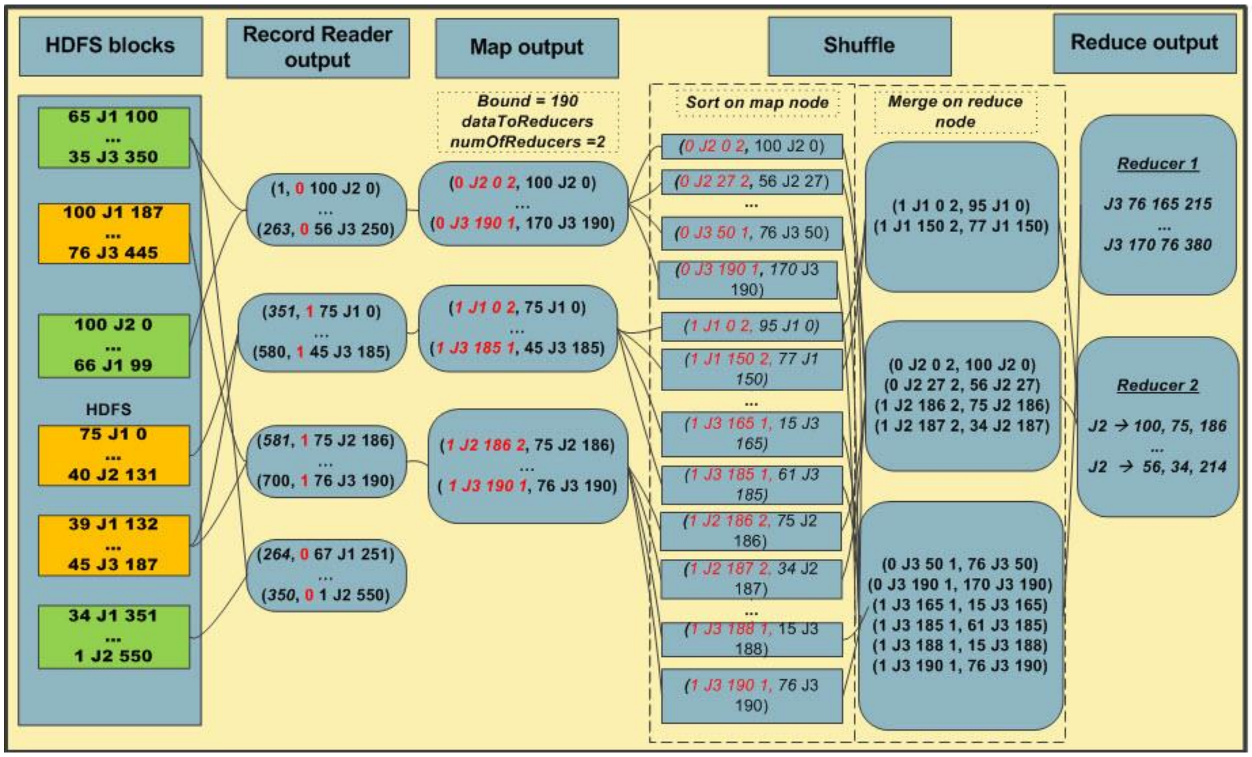


Figure 22: RSJETLBDistCache Dataflow

5.4 Algorithm 4: RSJETLBIndex

The fourth algorithm or RSJETIndex, as referred to in the implementation, embodies both Early Termination and Load Balancing techniques. In this version, the goal is to load on the map tasks only the necessary number of input splits for the production of the k joined records from each dataset and not the whole dataset.

So the following components remain the same:

- *SelectiveRecordReader.java*
- *SortingComparator.java*
- *GroupingComparator.java*
- *ScoredTuple.java*
- *RSJReducer.java*
- *CompositeKey.java*
- *RSJPartitioner.java*
- *Map.java*

Whereas the following are altered:

- *DriverRSJ.java*
- *SelectiveInputFormat.java*

5.4.1 RSJETIndex - Pseudocode

In this section, the new components introduced in 5.4.4 are presented in pseudocode.

RIndex algorithm

```
Read HDFS default block size from Configuration
Read file length from HDFS
Calculate number of output records = input file length/block size
Set variable number of lines already read = 1
[Do until number of lines already read = number of output records
            Calculate current position in input file = blockSize * number of lines already read
            Read file from current position
            [Do while lines exist in block
                  Read line
                  Set current position = current position + line length
                  Split line in words
                  Set record = current position, second word (score)
            Add record to output file
            number of lines read = number of lines read + 1
Return output file
```

findBytesToRead method (class: DriverRSJ.java)

```
Read Index file for input file
Do while Index files has lines
     Read line
     Split line in words
     If second word (score) > bound for input file
          Set variable bytesToRead = first word (bytes)
```

```
            Break loop
    Store bytesToRead in Configuration
    Return bytesToRead
```

getSplits method (class: SelectiveInputFormatV2.java) → override

```
    Read bytesToRead for input file from Configuration
    Set file length = bytesToRead
    Call FileInputFormat.getSplits with new file length
```

5.4.2 Inputs

The inputs of this map-reduce job are seven:

1. *The input path for R0sorted.txt*
2. *The input path for R1sorted.txt*
3. *The output folder path*
4. *The input path to the folder that contains hist0.txt and hist1.txt*
5. *The number of k.*
6. *The number of reduce tasks*
7. *The number of different values that a joining attribute can have (number of histogram bins)*

5.4.3 Preparation for Execution

There is a need for a correlation among the actual score values and the number of bytes that need to be processed by the map reduce job so as to reach a certain score value. This information is necessary for the creation of the custom input splits.

So, as a pre-execution step, the algorithm implemented in RIndex.java needs to be executed for each dataset.

The goal of the RIndex algorithm is to produce a file with the name e.g. R0Index.txt that will contain a number of records that results from the following function: ***number of records = (file length (bytes) / block size (bytes)) - 1***

The function presented above calculates the number of HDFS blocks in which the file is divided on the HDFS minus 1 (that is because there is no need to store any information about the last HDFS block).

For each HDFS block, the algorithm calculates the last score value that appears in the block and the number of bytes that a job must read to reach this value. So, for each block it creates a record in the output file containing the last score value of the HDFS block and the number of bytes accessed. This loop terminates when the number of records printed in the output file reaches the number calculated by the above mentioned function.

The two index files are then uploaded in the HDFS folder hist, so that they can be visible from the Map-Reduce job.

5.4.4 Components

- *DriverRSJ.java*
  After the calculation of bounds, from the method computeBounds, method *findBytesToRead* is invoked for each dataset. This new method looks up the appropriate index file with the bound value and retrieves the number of bytes that need to be accessed from the sorted input dataset so that this score bound is reached. This information is then stored in a configuration property for each dataset.

- *SelectiveInputFormat.java*
  This class is altered so that it overrides the method getSplits. This method looks up the configuration property for each dataset and sets the fileLength variable to be equal to the number of bytes that need to be accessed and the actual length of the file. There is no difference between the rest of the method's implementation concerning the creation of input splits and the default implementation of FileInputFormat. So, since the file length is considered to be smaller the number of the input splits loaded on map tasks will also be smaller.

5.4.5 RSJETLBIndex Dataflow

As shown in figure 23, the map tasks launched are less that the ones launched in previous algorithms. This is because the input splits calculated derive only from the useful HDFS blocks (highlighted in yellow).

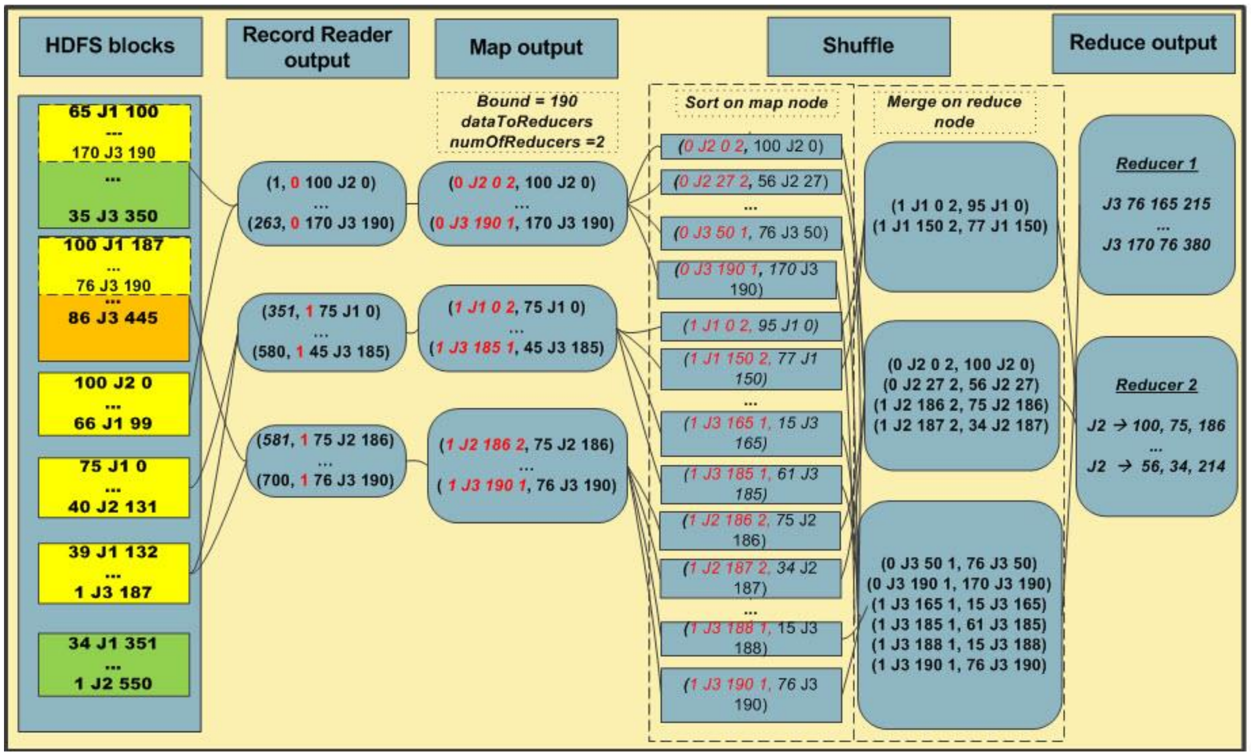


Figure 23: RSJETLIndex Dataflow

5.5 Top-k algorithm

All four algorithms presented earlier produce an output which contains the k top joins per joining attribute. This result could be the output of the following query:

***Select** R0.id, R1.id, (R0.score+R1.score) as score from R0, R1*

***Where** R0.joining_attribute = R1.joining_attribute*

***Group by** joining_attribute*

***Order by** (R0.score+R1.score) asc*

***Limit** k*

This is because each reducer processes records from both files for a specific joining attribute. So, the reducer performs the join for one specific value of the joining attribute and then returns the top k joined records for this joining attribute based on the scoring sum of the joined records. Consequently, the combination of outputs of all reducers give us the top k joined tuples for each joining attribute value. However, this means that the user will receive *number of reducers * k* joined tuples as a result instead of k that he/she asked.

To perform the last step of the query, which is about the filtering and sorting of all the reducers output to select only the top k joined tuples independently of their joining attribute value (without the group by statement), an extra method must be implemented. This method is called topKCalculator and is placed in the DriverRSJ.java and is invoked right after the completion of the MapReduce job. Its role is to fetch the output of each reducer and perform a loop, in order to find the k results with the lowest score sum. This results are then printed in a file and stored in HDFS.

When the input datasets are too large and the number of k is very big, the topKCalculator can be replaced by a second map reduce job. This MapReduce job implements the same Top-k algorithm that topKCalculator implements.

The second MapReduce job must be executed right after the execution of the Reduce Side Join and its goal is to receive as input the output of the first job and feed it to a single reducer which will pick and return only the top k joined tuples to the user.

5.5.1 Top-K algorithm – Pseudocode

map method (class: Map.java)

```
Read key value pair (from all Reduce Side Join output partitions)
Return key value pair of input
```

reduce method (class: Reduce.java)

```
Do while all input key value pairs are processed
    Read value – joined tuple
    Split tuple in words
    Set variable scoring sum = third word
    Store it in a Priority Queue
Return the k elements of the Priority Queue
```

5.5.2 Components

- *DriverRSJ.java*
  This is the main class of the MapReduce job. It contains no complex logic but only the configuration of the classes used for map and reduce phase. This is the class where we set the number of reduce tasks to be executed to 1.

- *Map.java*
  This is the class that implements the map method. Its input is the output of the earlier Reduce Side Join MapReduce job, whether the algorithm used for the join was RSJSimple, RSJET, RSJETLBDistCache or RSJETIndex. So, since all four algorithms that perform the join produce the same outputs, the map implementation of the Top-k algorithm applies to the outputs of all four algorithms without alterations. Its implementation is pass through. It receives a key-value pair where the key is just a number produced by the default RecordReader and the value is the whole joined tuple. The output key-value pair is the same as the input.

- *Reduce.java*
  This is the class where the reduce method is implemented. Since, the number of reduce tasks is set to 1, all joining attribute values will be sent to this single reduce task. The reduce method will loop all the joined tuples received and return only the k with the smallest scoring sum.

5.6 Supplementary Implementation

5.6.1 DataToReducersET

The algorithms explained in the previous sections represent the implementation of the proposed architecture. However, there is a need for an extra implementation to support the experimental part. In order to prove that the load balancing technique implemented in RSJETLBDistCache, actually distributes the load in an efficient manner and not randomly as happens in RSJET, we need two more algorithms that will evaluate the outputs of the RSJET algorithm.

RSJETLBDistCache produces the dataToReducers.txt as mentioned earlier. This file contains the information: *which reducer receives which joining attribute.* This information is not produced in a straight forward way from the RSJET algorithm. It can only be implied from the analysis of the algorithm's output. Therefore, the first supplementary algorithm performs this analysis and is called DataToReducersET. The inputs to this algorithm are all the partitions were the RSJET reducer's output is stored. The algorithm goes through the output records of each partition and stores all the joining attributes that have been processed by a certain partition. The output of this algorithm is a file similar to dataToReducers.txt produced by the RSJETLBDistCache. So, in this way we are able to compare the load distribution logic of the two algorithms.

DataToReducersET – Pseudocode

```
[Do while there are output partitions of RSJET job
    Read partition number from name of current partition
    Set variable reducer = partition number
        [Do while there are lines in partition
            Read line
            Split it in words
            Set variable current_joining_attribute  = first word
            [If current_joining_attribute is not already written in dataToReducers.txt
                Write in dataToReducers.txt : current_joining_attribute, reducer
Return dataToReducersET.txt
```

5.6.2 JoinsPerReducer

The BoundEstimation algorithm which is used in RSJETLBDistCache encapsulates the calculation of the joined values that can be produced from the join of two relations for each joining attribute value. If we combine this information with the dataToReducers information we can calculate the exact number of joins that each are performed by each reducer. This is implemented in the JoinsPerReducer algorithm.

The input to this algorithm is the joinValues HashMap, that is produced by the BoundEstimation algorithm, and the dataToReducers.txt. The algorithm loops the lines of dataToReducers.txt and for each joining attribute it looks up the number of joins that correspond to it. Then, it adds this number to total number of joins performed by reducer responsible for this joining attribute. The output of this algorithm is the file joinsPerReducer.txt which contains the information: *how many joins does each reducer perform*.

Pseudocode

```
Do while dataToReducers.txt has lines
    Read line
    Split line in words
    Set variable reducer_for_joining_attribute = second word
    Set variable current_joining_attribute = first word
    Set variable number_of_join_values = joinValues.get(current_joining_attributes)
        If joinsPerReducer.txt does not contain reducer_for_joining_attribute
                Set variable totalJoinsForReducer = number_of_join_values
                Write in joinPerReducer.txt: reducer_for_joining_attribute, totalJoinsForReducer
        Else
            Set variable totalJoinsForReducerSoFar = joinsPerReducer.get(reducer_for_joining_attribute)
            Set variable totalJoinsForReducer =   totalJoinsForReducerSoFar + number_of_join_values
            Update joinsPerReducer.txt: reducer_for_joining_attribute, totalJoinsForReducer
Return joinsPerReducer.txt
```

6 Experimental Evaluation

6.1 Environment setup

All the experiments for the needs of this thesis were performed on a Hadoop cluster. The characteristics of this cluster are presented in this section.

The cluster consists of 12 nodes. There is one node with the role of namenode and one node with the role of secondary namenode. All nodes are used as tasktrackers and datanodes. The specifications of the physical machines are presented in the following table:

Number of Machines	Disk space per machine	Memory per machine
8	6.3TB	31.4GB
3	14.4TB	125.9GB
1	14.4TB	110.2GB

Table 2: Cluster configuration

So, the total disk space used by the cluster is 108TB and the total memory from all machines is 739.1GB.

6.2 Experimental scenarios

In order to prove that the proposed algorithms for top-k joins add efficiency and decrease the execution time, a number of experimental scenarios were conducted to test a number of factors that affect the efficiency of top-k join query calculation.

The first factor tested is the dataset size. In order to prove that the algorithms proposed are proper for the processing of top-k joins for large datasets, we have to prove that they scale well. For this reason, the first set of datasets used includes datasets with sizes that range from 5GB to 200GB. Another factor is the number of k desired joined results. We prove that the proposed techniques are equally efficient for a wide range of k results requested by the user. More specifically, the four algorithms are executed with the datasets presented in table 4 for k=10, k=100 and k=500. Algorithms RSJET, RSJETLB and RSJETIndex were executed 10 times each, for every dataset and every k value, whereas RSJSimple was executed 5 times for each dataset and for every k value. The results for k=10 are presented in 6.4. Accordingly, the results for k=100 and for k=500 are presented in Appendix. For all scenarios tested, several counters of the map reduce jobs are presented, which prove the efficiency and correctness of the three latter algorithms comparing to the RSJSimple algorithm.

Another factor tested with the datasets of table 5 is join selectivity. In database systems, join selectivity of a table's column is defined as: *the number of distinct values that the column can have / number of* *table's tuples*. We will simplify the meaning of join selectivity by replacing the number of table's tuples with the dataset size. Therefore, the analogy that calculates the join selectivity of the datasets will be: *number of joining attribute values/dataset size (GB).* For example, a dataset of 10GB which has 100 distinct values of joining attributes has join selectivity 100/10 = 10 whereas a dataset of same size with 1000 distinct values has join selectivity 1000/10 = 100. The lower the join selectivity the higher the number of joins that can be produced and vice versa. By testing our proposed algorithms against this factor, we prove that they perform well with low and high join selectivity comparing to RSJSimple.

The fourth factor tested is data distribution on score attribute. All algorithms are tested against datasets of the same size but with different distribution on the scoring attribute. In this way we prove that the proposed algorithms are equally efficient even in different data distributions. As a matter of fact the datasets used for this purpose are the ones presented in table 6.

Last but not least, in order to compare the behavior of RSJET against RSJETLB we alter the data distribution on the joining attribute. If the data distribution on the joining attribute is uniform, then the number of joined tuples is approximately the same for all joining attributes, therefore the load distribution in the reducers will be fair, regardless the data distribution on scoring attribute. However, if the joining attribute has a zipfian distribution then the number of joins that correspond to each joining attribute differs. To test this (Scenario 5) we use the datasets presented in table 7.

The following table summarizes the experimental scenarios and the factors that each of them alters to examine the behavior of the tested algorithms:

Scenario	Dataset size	k	Join selectivity	Data distribution on score attribute	Data distribution on joining attribute
1	different	same	same	same	same
2	same	different	same	same	same
3	same	same	different	same	same
4	same	same	same	different	same
5	same	same	same	same	different

Table 3: Scenarios and factors

6.3 Datasets

All datasets used in the experimental scenarios were produced by a Data Generator program which receives as input the number of tuples to be produced (which define the size of the dataset), the desired number of joining attribute values, the type of distribution and the skewness of either the scoring attribute or the joining attribute. The outputs of Data Generator are the following six text files:

- R0.txt → First relation to be joined. Used as input in RSJSimple.
- R1.txt → Second relation to be joined. Used as input in RSJSimple.

- R0sorted.txt → The contents of this file are exactly the same with R0.txt. The only difference is that column score is sorted in ascending order. It is used as input of RSJET, RSJETLBDistCache, RSJETIndex.
- R1.sorted.txt → The contents of this file are exactly the same with R1.txt. The only difference is that column score is sorted in ascending order. It is used as input of RSJET, RSJETLBDistCache, RSJETIndex.
- hist0.txt → Histogram for first relation (R0).
- hist1.txt → Histogram for second relation (R1).

Both relations R0, R1 have the same columns: id, joining attribute, score attribute.

The datasets used as input to Scenario 1 and Scenario 3 are presented in the following table:

Dataset	R0sorted size	R1sorted size	Number of joining attribute values	Distribution	Skewness
DS1	5.7GB	5.8GB	500	Zipfian on scoring attribute	0.5
DS2	11GB	11.3GB	1000	Zipfian on scoring attribute	0.5
DS3	23GB	24GB	2000	Zipfian on scoring attribute	0.5
DS4	115.8GB	118.8GB	1000	Zipfian on scoring attribute	0.5
DS5	240.9GB	251.2GB	2000	Zipfian on scoring attribute	0.5

Table 4: First set of datasets

The datasets used as input to Scenario 2 are presented in the following table. DS1 dataset with 1000 joining attributes has higher join selectivity than DS2 with 100 joining attribute values. Accordingly, DS1 with 2000 joining attribute values has the same joining selectivity with the DS1 dataset of 1000 different joining attributes and a higher join selectivity than DS2 with 200 joining attributes.

Dataset	R0sorted size	R1sorted size	Number of joining attribute values	Distribution	Skewness
DS1	11GB	11.3GB	1000	Zipfian on scoring attribute	0.5
DS1	23GB	24GB	2000	Zipfian on scoring attribute	0.5
DS2	10.5GB	10.8GB	100	Zipfian on scoring attribute	0.5
DS2	21.8GB	22.9GB	200	Zipfian on scoring attribute	0.5

Table 5: Second set of datasets

The datasets used as input to Scenario 4 of experimental scenarios are presented in the following table:

Dataset	R0sorted size	R1sorted size	Number of joining attribute values	Distribution	Skewness
DS1	22.9GB	23.9GB	2000	Uniform on scoring attribute	0
DS2	23GB	24GB	2000	Zipfian on scoring attribute	0.5
DS3	23GB	24GB	2000	Zipfian on scoring attribute	1

Table 6: Third set of datasets

The datasets used as input to the third set of experimental scenarios are presented in the following table:

Dataset	R0sorted size	R1sorted size	Number of joining attribute values	Distribution	Skewness
DS1	1.0GB	1.1GB	100	Zipfian on joining attribute	0.5
DS2	1.0GB	1.1GB	100	Zipfian on joining attribute	0.2

Table 7: Fourth set of datasets

6.4 Scenario 1 for k=10

Scenario 1 is executed with the datasets of table 4. So, the complete configuration for this scenario is the following:

Scenario	Dataset size	k	Join selectivity	Data distribution on score attribute	Data distribution on joining attribute
1	ranges from 5GB to 200GB	10	100 for DS1, DS2, DS3 10 for DS4, DS5	Zipfian with skewness 0.5	uniform

Table 8: Scenario 1 for k=10

6.4.1 Total Duration

The diagram below depicts the total duration of all four algorithms for each dataset of the first set. It is clear that the Early Termination technique decreases significantly the execution time for the calculation of queries such as the one discussed in 5. RSJET and RSJETLBDistCache are equivalent in terms of execution time since the load uniform amongst the reducers due to the uniform data distribution of the joining attribute. It is worth mentioning that RJSETIndex scales better that RSJET and RSJETLBDistCache when the dataset size increases. For example for 100GB and 200GB datasets, RSJETIndex has the smallest total duration.

However, in small datasets, RSJETIndex is slightly slower than RSJET and RSJETLBDistCache. RJSETIndex calculates the Map input splits so that they derive only from the useful HDFS blocks and not from the whole input files. This leads to diminished map tasks in comparison with RSJET and RSJETLBDistCache. However, in datasets such as 5GB this technique does not add significant value since the map tasks loaded from RSJET and RSJETLBDistCache are few and finish very quickly because the input datasets size is small.

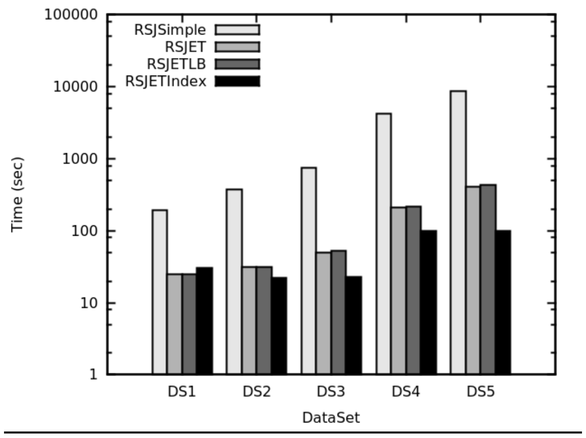


Figure 24: Scenario 1 for k=10 - Total Duration

6.4.2 Average Map Time

Average Map Time is another metric that proves the efficiency of Early Termination technique. RSJET and RJSETLBDistCache have the lowest average map time even for large datasets. Even though the number of map tasks launched by RSJET and RSJETLBDistCache is the same with RSJSimple, not all map tasks produce key value pairs for the reducers. Therefore, the majority of map tasks in RSJET and RSJETLBDistCache complete their execution very fast and most of them without processing any record of the input split.

On the contrary RSJETIndex appears to have a high Average Map Time but this is not a sign of inefficiency. As explained in 3.1.8 **Average Map Time** = Total time taken by all Map tasks/ Count of Map Tasks. In RSJETIndex the number of Map tasks can be very small. For example the number of map tasks launched for 200GB datasets by RSJETIndex is only 4 whereas the number of map tasks launched by all other three algorithms for the same dataset is 3937. The majority of map tasks in RJSET and RSJETLBDistCache complete extremely fast. In fact the number of map tasks that actually produce key value pairs for reducers in RSJET and RSJETLBDistCache is the same with RSJETIndex and these are the only map tasks who need more time to terminate.

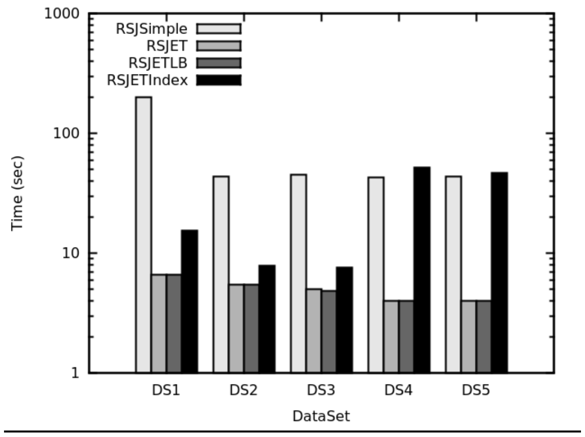


Figure 25: Scenario 1 for k=10 - Average Map Time

6.4.3 Average Shuffle Time

The following diagram proves that all proposed techniques enhance significantly the Shuffle phase. The results depicted in this diagram can be combined with the Average Map Time and Map Output Records diagrams. As explained earlier, in this phase the map outputs are copied in the task trackers that will execute the reduce tasks. In RSET and RSJETLBDistCache the map output records are significantly less comparing to RSJSimple. Therefore, the average time needed for the map outputs to be copied is less than RSJSimple for these two algorithms. RSJETIndex also produces less map output records than RSJSimple, however as explain earlier the number of map tasks raised by this algorithm is very small and this metric depicts average times, so, this is why for small datasets like 5GB RSJETIndex has higher average shuffle time.

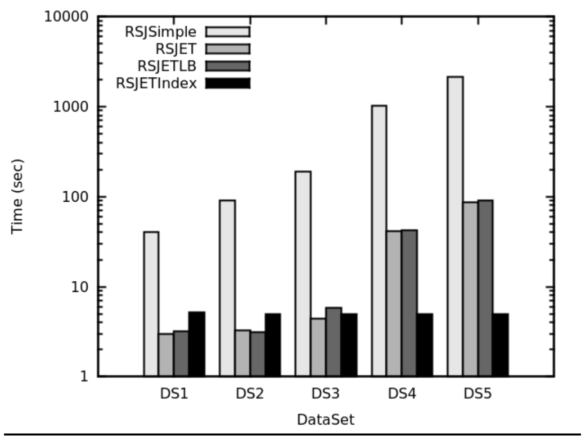


Figure 26: Scenario 1 for k=10 - Average Shuffle Time

6.4.4 Average Merge Time

This diagram proves that Merge phase is also significantly enhanced by all three proposed algorithms. It is worth pointing out that for smaller datasets the Average Merge Time needed for this phase is close to zero seconds. This is because the number of tuples that reached the task trackers were the reducer will be executed is much smaller comparing to RSJSimple.

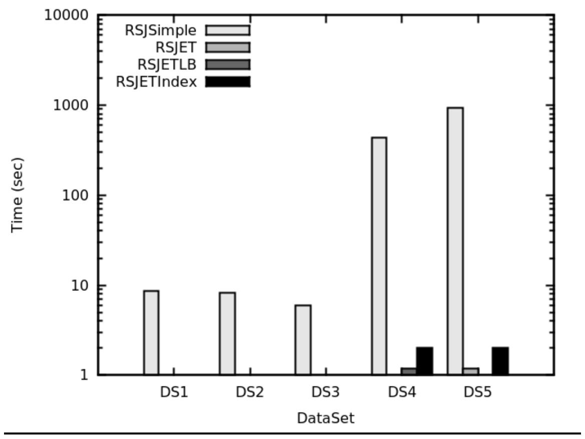


Figure 27: Scenario 1 for k=10 - Average Merge Time

6.4.5 Average Reduce Time

The Reduce phase is the most significant phase of a Reduce Side Join because it is the phase were the two input datasets will be joined. As depicted in the diagram below, this phase is the most enhanced by the Early Termination technique since the average time needed for RSJET, RSJETLBDistCache and RSJETIndex is close to zero, whereas the time needed for RSJSimple is much higher. This is because the load of the reduce tasks is significantly diminished with the use of Early Termination technique.

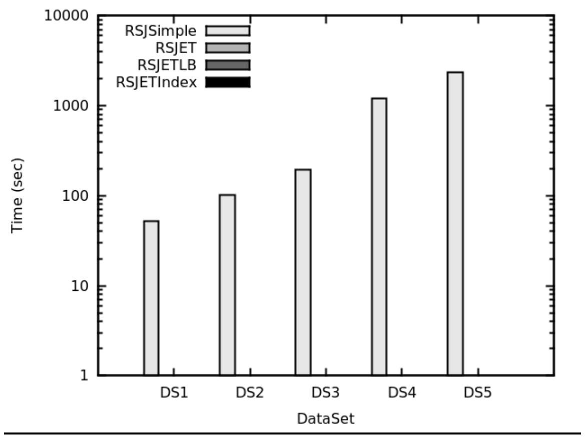


Figure 28: Scenario 1 for k=10 - Average Reduce Time

6.4.6 Map Input Records

This counter shows the number of records read by the RecordReader. In RSJET, RSJETLBDistCache and RSJETIndex, the number of records read by the record reader depends on the result of the BoundEstimation algorithm. In the following table we present the bounds returned for each relation and each dataset:

Dataset	Bound for R0	Bound for R1
DS1	60	60
DS2	30	30
DS3	15	15
DS4	30	30
DS5	15	15

Table 9: Scenario 1 for k=10 - Bounds per dataset

In RSJET, RSJETLBDistCache and RSJETIndex, RecordReader processes all records that have as score attribute values that are lower than the bounds estimated. Since the input datasets to these algorithms are sorted by score column, to reach bound 60, RecordReader must read more records than it would if bound was 30. So, that explains why map input records are more for DS1 than for DS2. DS4 and DS5 have same bounds with DS2 and DS3 accordingly, therefore one would expect that the map input records should be the same. However, DS4 and DS5 have lower join selectivity, which means that more records have to be accessed in order to reach the score bound.

On the other hand, RSJSimple processes all records from both input datasets, so the number of map input records for this algorithm, is proportional to the dataset size regardless the join selectivity.

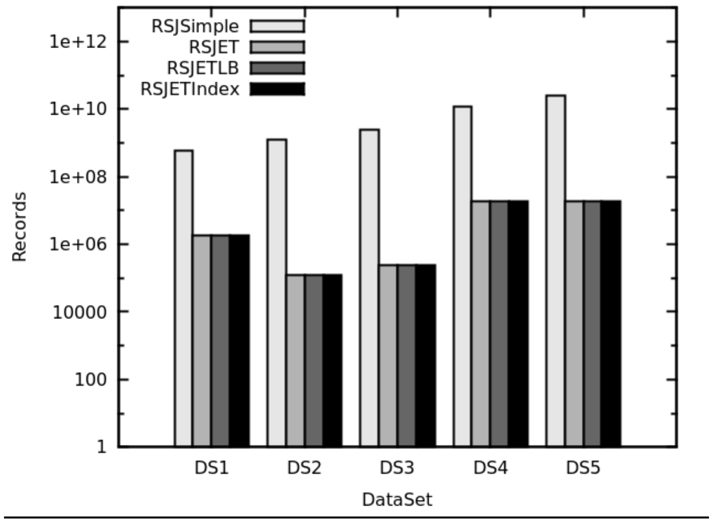


Figure 29: Scenario 1 for k=10 - Map Input Records

6.4.7 Map Output Records

The number of Map Output Records is equal to the number of Map Input Records for all four algorithms. However, due to Early Termination technique, this number is smaller for RSJET, RSJETLBDistCache and RSJETIndex comparing to RSJSimple.

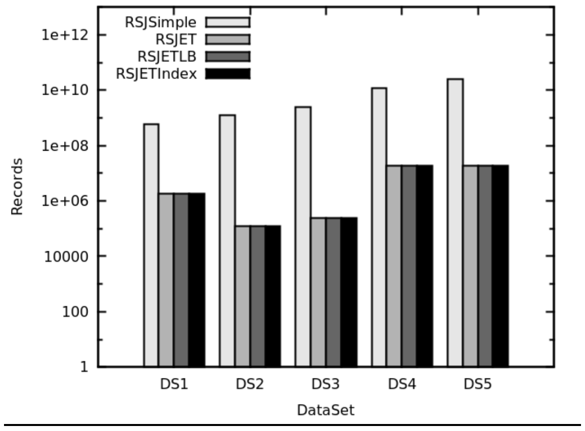


Figure 30: Scenario 1 for k=10 - Map Output Records

6.4.8 Reduce Output Records

This metric represents the total number of records that were produced by the reducers. As we saw earlier, the reduce method in all four algorithms, returns k joined records for each joining attribute. Since, the output of all jobs is a list with the top-k records per joining attribute, then this metric is proportional to the number of joining attributes. For example, for DS1 the number of joining attributes is 500 so for k=10 the number of output records is 5000. Accordingly for DS5 the number of joining attributes is 2000 so for k=10 the number of output records is 20000.

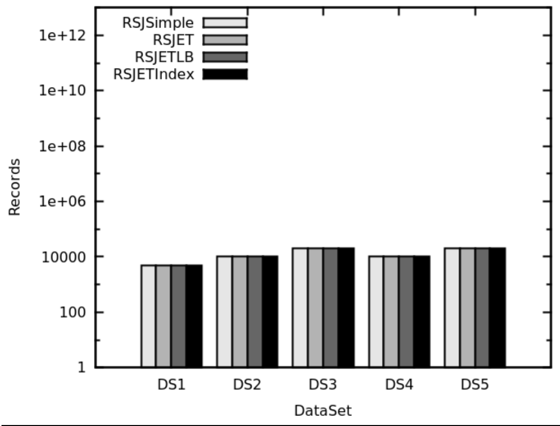


Figure 31: Scenario 1 for k=10 - Recuce Output Records

6.4.9 Shuffled Maps

The following diagram depicts the efficiency added by RSJETIndex in the Shuffle phase. RSJETIndex launches the lowest number of map tasks therefore during the maps copied during the shuffling phase are significantly less comparing to all other three algorithms. As explained earlier, the map tasks launched by RSJSimple, RSJET and RSJETLBDistCache are equal because the whole datasets are divided in input splits and passed to the mappers.

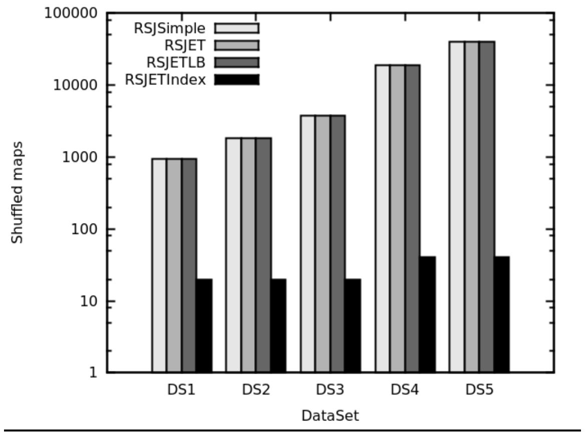


Figure 32: Scenario 1 for k=10 - Shuffled Maps

6.4.10 Map Spilled Records

This metric represents the number of records written in the disk during the map phase. This diagram can be examined in parallel with the Map output records diagram,

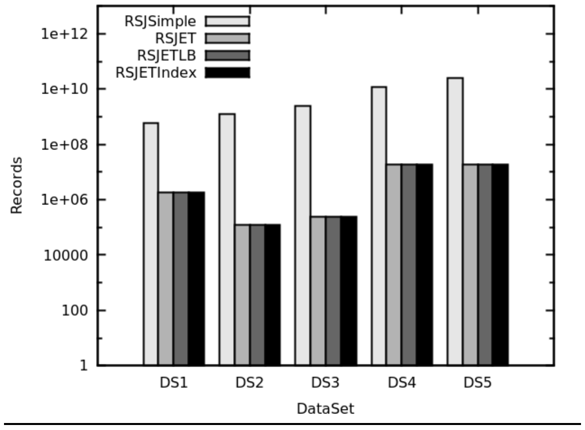


Figure 55: Scenario 1 for k=100 - Map Output Records

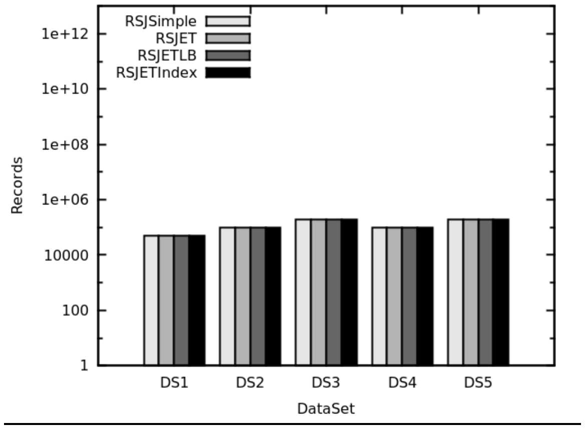


Figure 56: Scenario 1 for k=100 - Reduce Output Records

since the number of map output records is the number of records that are finally spilled on the disk. Again, it is worth mentioning that all three proposed algorithms produce less map output records due to the Early Termination technique therefore they spill less records comparing to RSJSimple which spills all input records to the disk.

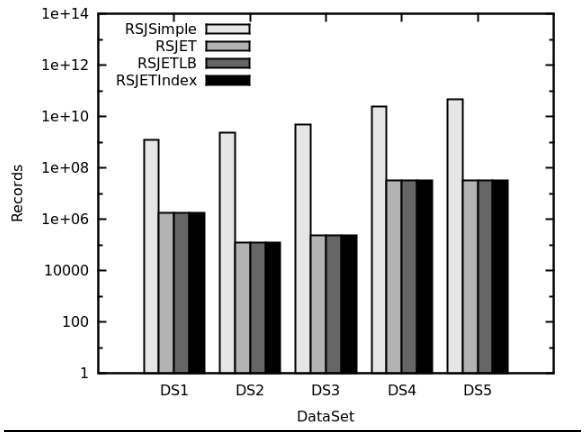


Figure 33: Scenario 1 for k=10 - Map Spilled Records

6.4.11 Input Split Bytes

This metric depicts the total number of bytes passed as input splits to the map tasks. The following diagram proves that RSJETIndex eliminates the number of bytes read from each dataset whereas all other three algorithms load the whole datasets to the map tasks.

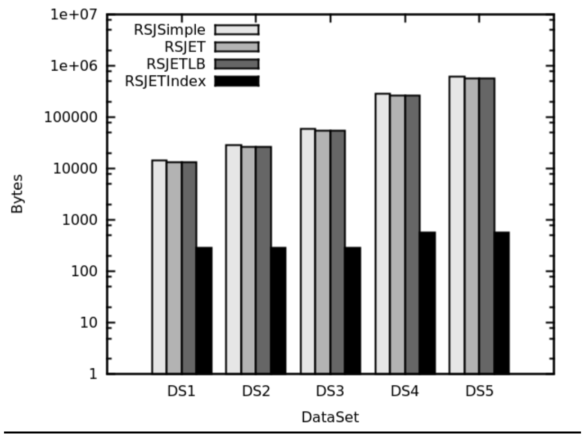


Figure 34: Scenario 1 for k=10 - Input Split Bytes

6.4.12 CPU Map Time

Since, Early Termination technique eliminates the number of records processed by the Map phase, the total processing time is also diminished comparing to RSJSimple. Especially in RSJETIndex, the total processing time in Map phase is even lower since the launched map tasks are fewer.

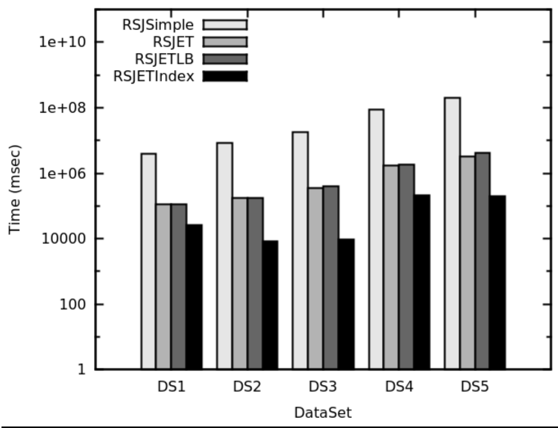


Figure 35: Scenario 1 for k=10 - CPU Map Time

6.4.13 CPU Reduce Time

Accordingly, the CPU Reduce Time for RSJET, RSJETLBDistCache and RSJETIndex is lower since the records passed to the reduce phase are less than the number of records passed in the reduce phase from RSJSimple.

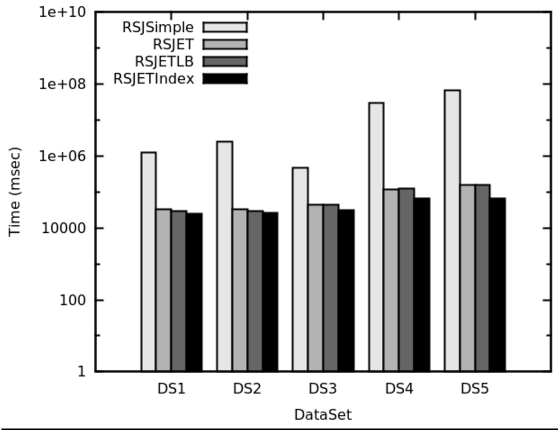


Figure 36: Scenario 1 for k=10 - CPU Recuce Time

6.5 Scenario 1 for k=100

Scenario 1 was re-executed with the datasets of table 4 but this time for k=100. So, the complete configuration for this scenario is the following:

Scenario	**Dataset size**	**k**	**Join selectivity**	**Data distribution on score attribute**	**Data distribution on joining attribute**
1	ranges from 5GB to 200GB	100	100/1 for DS1, DS2, DS3 10/1 for DS4, DS5	Zipfian with skewness 0.5	uniform

Table 10: Scenario 1 for k=100

The behavior of all four algorithms is similar to the behavior they had for k=10. Therefore, the explanation given for all the metrics in 6.4 applies here as well. The diagrams for all metrics are presented in Appendix and can be compared to the relevant diagrams for k=10.

6.6 Scenario 1 for k=500

Scenario 1 was re-executed with the datasets of table 4 but this time for k=100. So, the complete configuration for this scenario is the following:

Scenario	Dataset size	k	Join selectivity	Data distribution on score attribute	Data distribution on joining attribute
1	ranges from 5GB to 200GB	500	100/1 for DS1, DS2, DS3 10/1 for DS4, DS5	Zipfian with skewness 0.5	uniform

Table 11: Scenario 1 for k=500

The behavior of all four algorithms is similar to the behavior they had for k=10 and k=100. Therefore, the explanation given for all the metrics in 6.4 applies here as well. The diagrams for all metrics are presented in this chapter and can be compared to the relevant diagrams for k=10 and k=100.

The fact that all three proposed algorithms depict the same behavior for different values of k proves the scalability of the algorithms in terms of, not only dataset size, but k value as well. All diagrams for this scenario are available on Appendix.

6.7 Scenario 2 for all first set datasets

The results for Scenario 2 are already provided from the experiments executed for Scenario 1 and for the different values of k. In this section the results retrieved from Scenario 1 are combined in such a way so that it is easier to compare the performance of each algorithm for different k values in the same dataset.

The conclusion drawn from the comparison that is presented in the following diagrams, is that the total execution time of all algorithms remains the same regardless the k value.

The following table presents the configuration of the scenarios examined in this section.

Scenario	Dataset size	k	Join selectivity	Data distribution on score attribute	Data distribution on joining attribute
2	5GB	k=10, k=100, k=500	100 /1	Zipfian with skewness 0.5	uniform
2	10GB	k=10, k=100,	100 /1	Zipfian with skewness 0.5	uniform

		k=500			
2	20GB	k=10, k=100, k=500	100/1	Zipfian with skewness 0.5	uniform
2	100GB	k=10, k=100, k=500	10/1	Zipfian with skewness 0.5	uniform
2	200GB	k=10, k=100, k=500	10/1	Zipfian with skewness 0.5	uniform

Table 12: Scenario 2 - Configuration

The total execution times for all above datasets and for all k values, are reflected visually in the following diagrams.

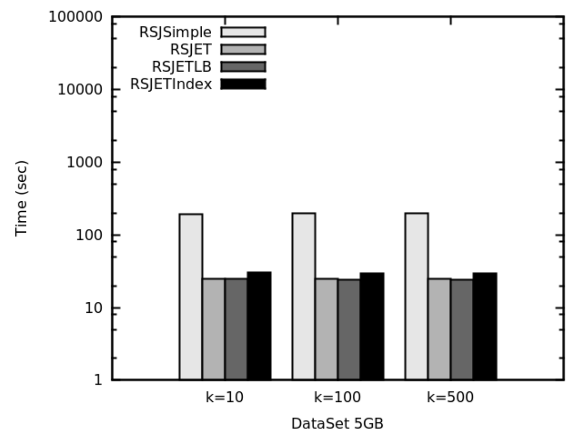


Figure 37: Scenario 2 - 5 GB

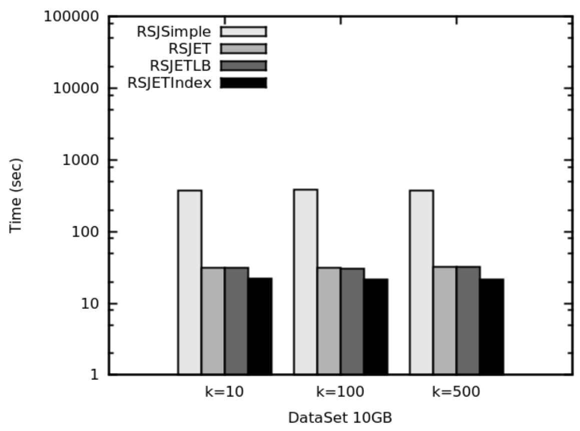


Figure 38: Scenario 2 - 10 GB

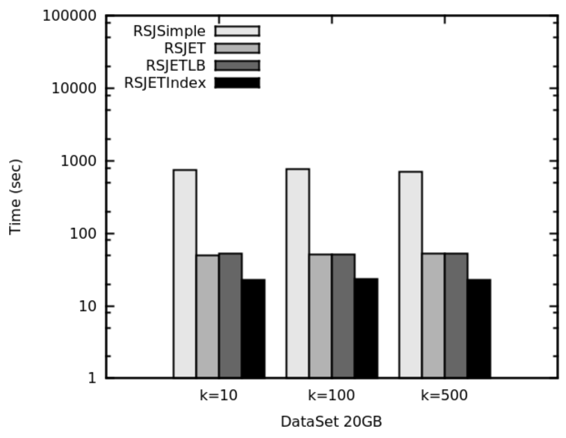


Figure 40: Scenario 2 - 20 GB

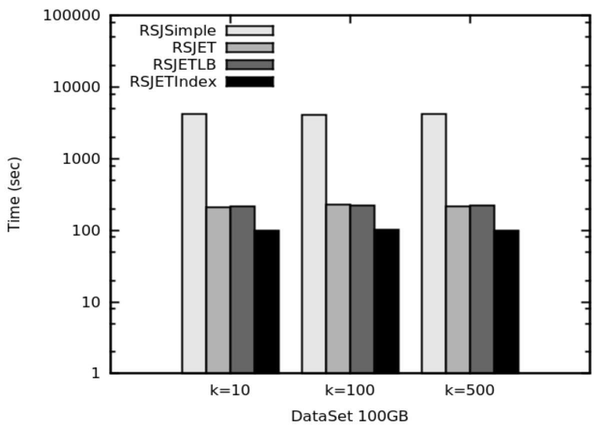


Figure 39: Scenario 2 - 100 GB

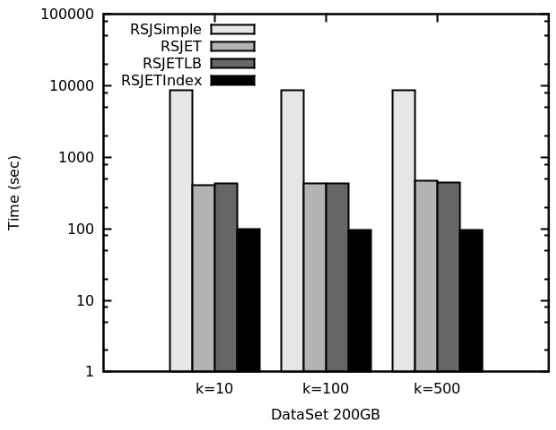


Figure 41: Scenario 2 - 200 GB

6.8 Scenario 3 for DS = 10GB

The datasets used for this scenario are the ones presented in table 5. So, the configuration for this scenario is the following:

Scenario	Dataset size	k	Join selectivity	Data distribution on score attribute
3	10GB	k=10, k=100, k=500	100/1 for DS1 10/1 for DS2	Zipfian with skewness 0.5

Table 13: Scenario 3 10GB

6.8.1 Total Duration for DS=10GB

The following diagram proves that the total duration of the proposed algorithms RSJET, RSJETLBDistCache and RSJETIndex increases slightly for lower join selectivity in the same dataset size. This is because the records that need to be processed in order to reach the estimated bounds for the score attribute, are more in datasets with low join selectivity, since the number of joins that are produced by low selectivity datasets is higher. This is proved by the results of BoundEstimation for the two datasets:

Dataset	Bound for R0	Bound for R1
DS1	30	30
DS2	300	300

Table 14: Scenario 3 10GB - bounds

Since the number of joins that can be achieved in DS2 are more than those in DS1 the total duration time of all three algorithms increases for DS2. However, the total duration remains lower than the total duration of RSJSimple regardless the k value.

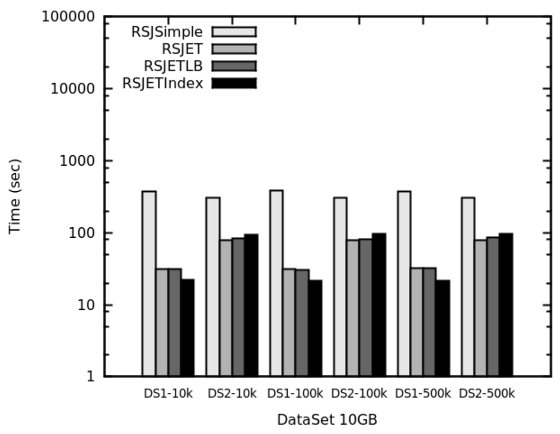


Figure 42: Scenario 3 10GB - Total Duration

6.9 Scenario 3 DS = 20GB

The datasets used for this scenario are the ones presented in table 5. So, the configuration for this scenario is the following:

Scenario	Dataset size	K	Join selectivity	Data distribution on score attribute
3	20GB	k=10, k=100, k=500	100/1 for DS1 10/1 for DS2	Zipfian with skewness 0.5

Table 15: Scenario 3 20GB

6.9.1 Total Duration for DS = 20GB

The BoundEstimation algorithm returned the following bounds for the two datasets:

Dataset	Bound for R0	Bound for R1
DS1	30	30
DS2	300	300

Table 16: Scenario 3 20GB - bounds

Therefore, the explanation given for Total Duration for DS=10GB applies in this scenario too.

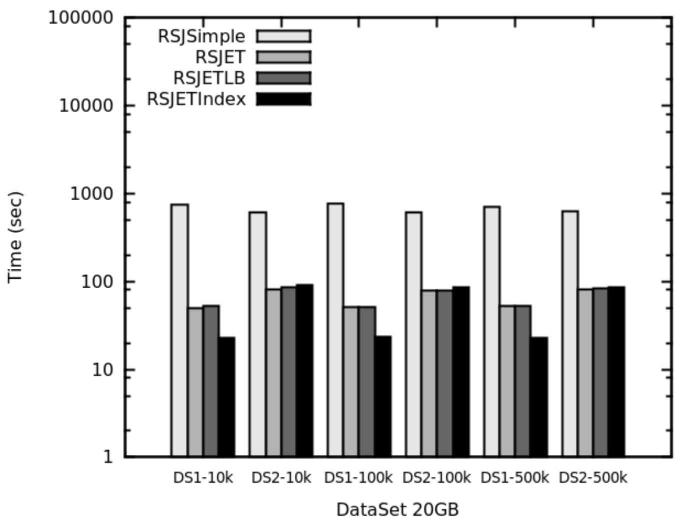


Table 17: Scenario 3 20GB - Total Duration

6.10 Scenario 4

The datasets used for this scenario are the ones presented in table 6. The configuration for these experiments is the following:

Scenario	Dataset size	k	Join selectivity	Data distribution on score attribute
4	20GB	k=10	100/1	Uniform for DS1 Zipfian with skewness 0.5 for DS2 Zipfian with skewness 1.0 for DS3

Table 18: Scenario 4 20GB

As the diagrams show, the behavior of all algorithms, even of RSJSimple is exactly the same for all three datasets regardless the distribution of the scoring attribute. For this reason, only the total duration diagram is presented in the next section and the rest of the diagrams for this scenario are available in Appendix.

Still, the algorithms RSJET, RSJETLBDistCache and RSJETIndex remain faster in execution time than RSJSimple.

6.10.1 Total Duration

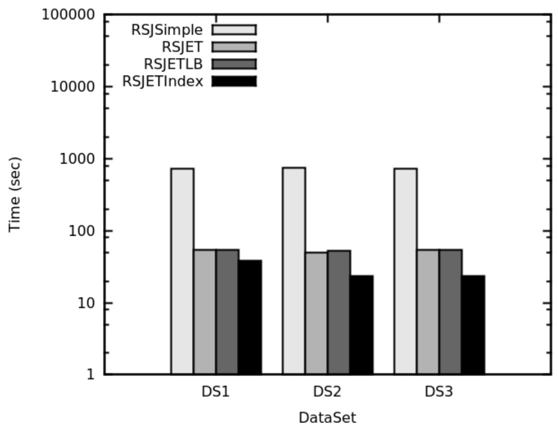


Figure 43: Scenario 4 20GB - Total Duration

6.11 Scenario 5

In all Scenarios examined so far, the behavior of RSJET and RSJETLBIndex was very similar. The total duration was approximately the same for the two algorithms and that was expected since the distribution of the joining attribute of the datasets tested was uniform. Consequently, the value of Load Balancing technique was not revealed, since the load would either way be distributed in a uniform manner to the reducers. In this Scenario we use the datasets presented in table 7 which have the same dataset size but differ in the distribution of the joining attribute. So, the configuration for this scenario is the following:

Scenario	Dataset size	k	Join selectivity	Data distribution on joining attribute
5	1GB	k=10	100/1	Zipfian with skewness 0.5 For DS1 Zipfian with skewness 0.2 For DS2

Table 19: Scenario 5

6.11.1 Total Duration

The expected result would be that RSJETLBDistCache would have a lower total duration due to the Load Balancing technique. In fact, this is not happening as shown in the diagram below. Both RSJET and RSJETLBDistCache take the same time to complete.

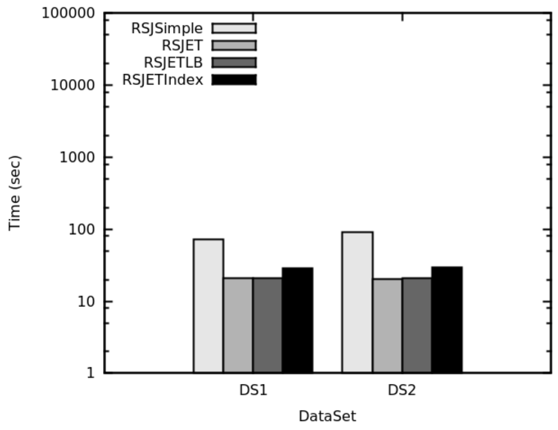


Figure 44: Scenario 5 - Total Duration

To examine in further detail this behavior, we use the implementation explained in Supplementary Implementation. With DataToReducersET and JoinsPerReducer algorithms we retrieve the information: *how many join values does each reducer process* in the RSJET algorithm. We do this for the dataset with 0.5 skewness in joining attribute.

This information is already known for the RSJETLBDistCache, since the calculation of dataToReducers is part of the algorithm's steps.

So, having calculated the joins that each reducer handles for both RSJET and RSJETLBDistCache we can depict them in the following diagrams.

<u>Load Per Reducer in RSJETLBDistCache – 1GB dataset with 0.5 skewness on joining attribute</u>

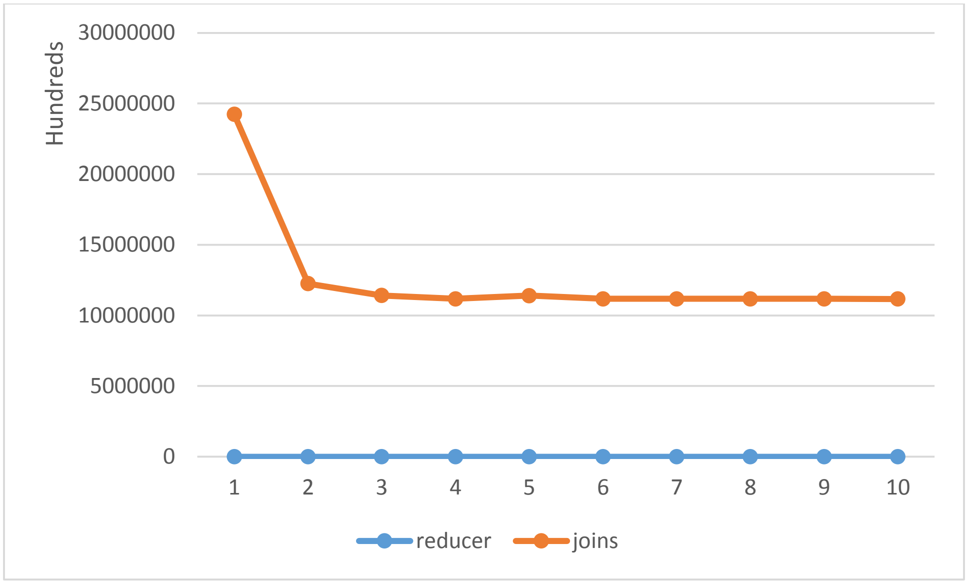


Figure 45: Scenario 5 – RSJETLBDistCache joins per reducer on zipf 0.5

As we can see, RSJETLBDistCache, which uses the LPT algorithm for the load distribution, does the following:

1. Sorts the joining attributes by the total joins that correspond to each one in a descending order.
2. Assigns the first 10 joining attributes from the sorted list, each one in one reducer (considering that the number of reduce tasks is set to 10).
3. For the rest of joining attributes
   - finds the reducer with the least joins to process
   - assigns the current joining attribute to the least busy reducer

The dataset has zipfian distribution on the scoring attribute with 0.5 skewness. According to the above logic, the first joining attribute will be assigned to reducer 1 and no other joining attribute will be sent to this reducer because the first joining attribute carries the greatest number of joins. The second joining attribute will be assigned to reducer 2 and so on until the tenth joining attribute will be assigned to reducer 10. Since we have a zipfian distribution, reducer 10 so far has the least joins to handle therefore, the eleventh joining attribute will be assigned to reducer 10, the next to reducer 9 and so on. In the end, reducer 1 will be still the busiest reducer as depicted in the diagram for RSJETLBDistCache even though it all handles the first joining attribute. So, RSJETLBDistCache distributes the load to the least busy reducer however this does not assure that the load will be equal across the reducers for a much skewed dataset.

On the other hand it is clear that the RSJET algorithm performs a random load assignment which results to the overload of reducer 9 (figure 46). In another run of RJSET, another reducer might be overloaded.

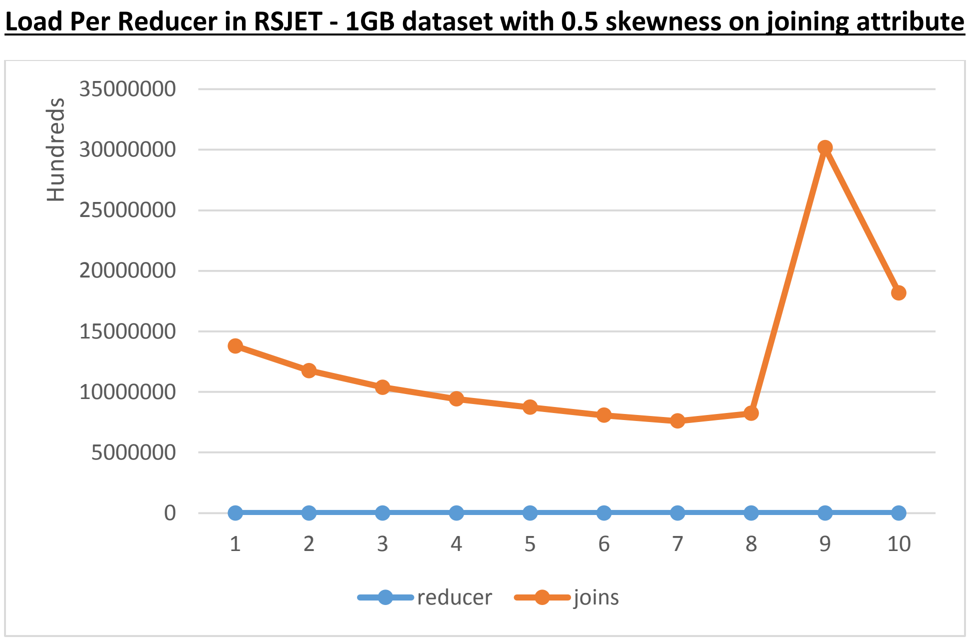


Figure 46: Scenario 5 - RSJET joins per reducer on zipf 0.5

As shown in figure 47, RSJETLBDistCache performs a more uniform load distribution for datasets with low skewness on joining attribute. RSJET still distributes randomly the load, thus resulting again to one very busy reducer (figure 48).

Load Per Reducer in RSJETLBDistCache – 1GB dataset with 0.2 skewness on joining attribute

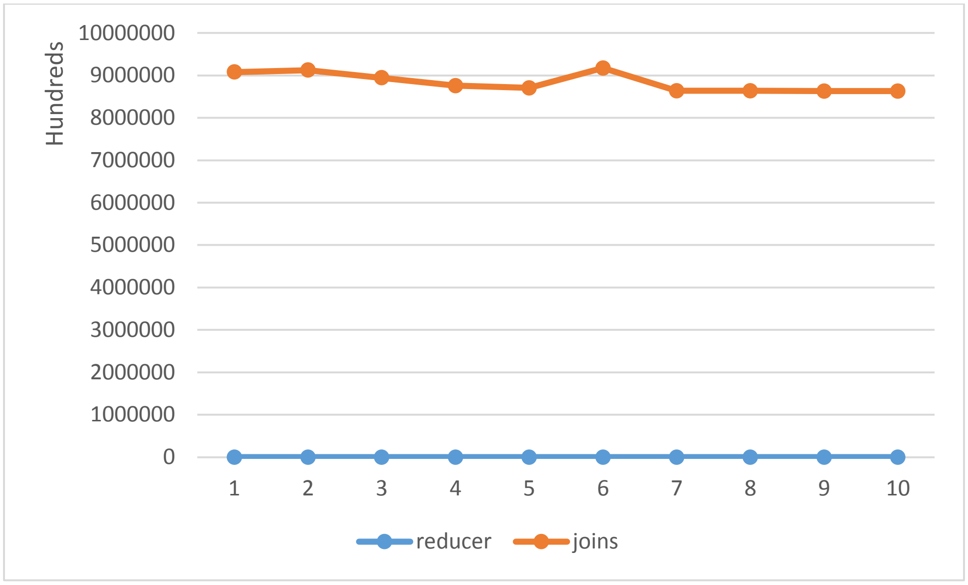


Figure 47: RSJETLBDistCache joins per reducer on zipf 0.2

Load Per Reducer in RSJET - 1GB dataset with 0.2 skewness on joining attribute

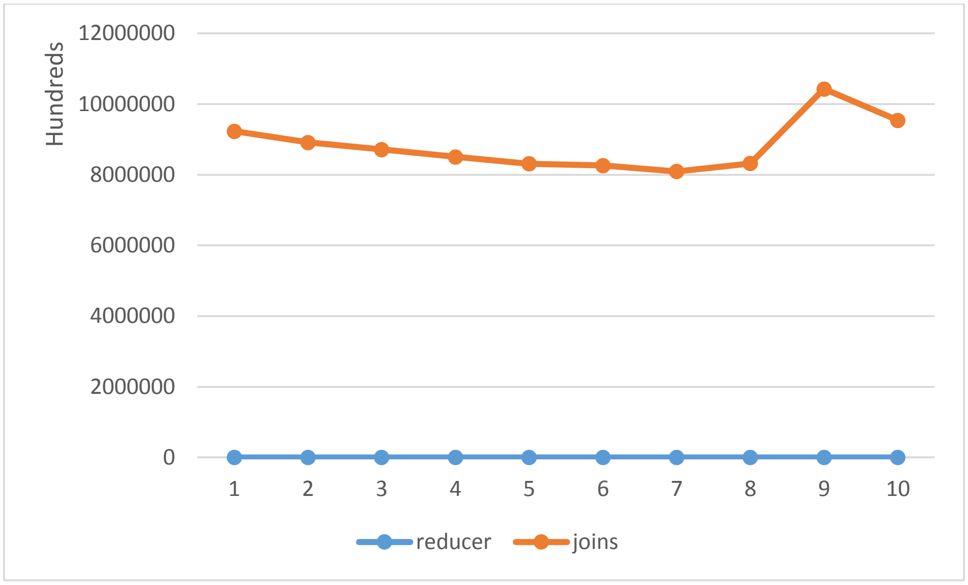


Figure 48: RSJET joins per reducer on zipf 0.2

All four algorithms implement the same mechanism in their reducers. They read only k values from each input relation to perform the join. It is guaranteed that the first k values that will reach a reducer from one relation are sorted by score (due to secondary sort). Therefore, the reducers terminate after k*2 loops when they certainly have calculated the top-k joined tuples for a specific joining attribute. Consequently, even if there is a reducer that should take the longest time to complete due to the number of joins assigned to it, in fact it completes its processing in similar time with the others due to this logic.

This explains the reason why, although we observe a gain in RJSETLBDistCache in terms of load distribution, the execution time of RSJETLBDistCache is still the same with RSJET regardless the skewness of the datasets.

7 Conclusions

In this thesis two techniques were presented for early termination and one technique for load balancing to address the problem of efficient processing of top-k joins in MapReduce. All techniques were implemented on top of Hadoop MapReduce without affecting its internal mechanism.

The experiments presented in Chapter 6 prove that all three algorithms are more efficient than the simple implementation of top-k joins in the traditional MapReduce model, in terms of performance and resources used. This conclusion has proven to be valid for a number of factors tested such as dataset size, different k values, join selectivity and data distribution on score and joining attributes.

The third algorithm RSJETIndex proves to be faster for very large datasets such as 200GB in all experimental scenarios, whereas RSJET and RSETLBDistCache work equally well for smaller datasets.

Even though RJET and RSJETLBDistCache depicted the same overall execution time in all scenarios, the experiments in section 6.11 proved that RSJETLBDistCache makes more uniform load distribution than RSJET, in datasets with skewness on joining attribute.

A further enhancement of the proposed architecture and implementation would be its extension so as to support more complex rank aware queries for multiple relations. This is easily achieved by altering slightly the implementation of the map and reduce methods to handle multiple input datasets. The core logic, though, remains the same. Another set of experiments could be performed to prove the scalability and speed of the implementation for a series of complex rank aware queries.

Last but not least, it would be very interesting to test the proposed algorithms against algorithms proposed in related work, such as RanKloud [2] and compare the efficiency of the solutions in terms of total execution time and resources used.

8 References

1. A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya. Storage infrastructure behind Facebook Messages: using HBase at scale. IEEE Data Engineering Bulletin, 35(2):4-13, 2012.

2. K.S. Candan, K. Selcuk, et al. "RanKloud: scalable multimedia data processing in server clusters." MultiMedia, IEEE 18.1 (2011): 64-77.

3. B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment (PVLDB), 1(2):1277{1288, 2008.

4. C. Doulkeridis, A. Vlachou, K. Nørvåg, Y. Kotidis and N. Polyzotis, "Processing of rank joins in highly distributed systems." Data Engineering (ICDE), 2012 IEEE 28th International Conference on. IEEE, 2012.

5. C. Doulkeridis and K. Nørvåg. A survey of large-scale analytical query processing in MapReduce. The VLDB Journal. DOI 10.1007/s00778-013-0319-9, 2013.

6. C. Doulkeridis and K. Nørvåg. On saying "enough already!" in MapReduce. In Proceedings of International Workshop on Cloud Intelligence (Cloud-I), pages 7:1-7:4, 2012.

7. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2004.

8. J. Dean and S. Ghemawat. MapReduce: a flexible data processing tool. Communications of the ACM, 53(1):72-77, 2010.

9. K. Goodhope, J. Koshy, J. Kreps, N. Narkhede, R. Park, J. Rao, and V. Y. Ye. Building LinkedIn's real-time activity data pipeline. IEEE Data Engineering Bulletin, 35(2):33-45, 2012.

10. R. Grover and M. J. Carey. Extending map-reduce for efficient predicate-based sampling. In Proceedings of International Conference on Data Engineering (ICDE), pages 486{497, 2012.

11. B. Guer, N. Augsten, A. Reiser, and A. Kemper. Load balancing in MapReduce based on scalable cardinality estimates. In Proceedings of International Conference on Data Engineering (ICDE), pages 522-533, 2012.

12. https://hadoop.apache.org/

13. I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," ACM Computing Surveys, vol. 40, no. 4, 2008.3.T. White. Hadoop - The Definitive Guide. O'Reilly, 2012.

14. L. Kolb, A. Thor, and E. Rahm. Load balancing for MapReduce-based entity resolution. In Proceedings of International Conference on Data Engineering (ICDE), pages 618-629, 2012.

15. Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In ACM Symposium on Cloud Computing (SoCC), pages 75-86, 2010.

16. Y. Kwon, M. Balazinska, B. Howe, and J. A. Rolia. SkewTune: mitigating skew in MapReduce applications. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 25-36, 2012.

17. N. Laptev, K. Zeng, and C. Zaniolo. Early accurate results for advanced analytics on MapReduce. Proceedings of the VLDB Endowment (PVLDB), 5(10):1028-1039, 2012.

*18. C.-Y. Lee. Parallel machines scheduling with non-simultaneous machine available time**.** Discrete Appl. Math., 30 (1991), pp. 53–61*

19. K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with MapReduce: A survey. SIGMOD Record, 40(4):11-20, 2011.

20. N. Ntarmos, I. Patlakas, and P. Triantafillou. "Rank join queries in NoSQL databases." Proceedings of the VLDB Endowment 7.7 (2014): 493-504.87.

21. A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. De-Witt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), pages 165-178, 2009.

22. S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in MapReduce workloads using progressive sampling. In ACM Symposium on Cloud Computing (SoCC), pages 16:1-16:13, 2012.

23. S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: a framework for large scale data processing. In ACM Symposium on Cloud Computing (SoCC), pages 4:1-4:13, 2012.

24. A. Rasmussen, V. T. Lam, M. Conley, G. Porter,R. Kapoor, and A. Vahdat. Themis: an I/O e_cient MapReduce. In ACM Symposium on Cloud Computing (SoCC), pages 13:1-13:14, 2012.

25. M. Stonebraker, D. J. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin. MapReduce and parallel DBMSs: friends or foes? Communications of the ACM, 53(1):64-71, 2010.

26. Y. Wang, L. Chen and G. Agrawal. Supporting Online Analytics with User-Defined Estimation and Early Termination in a MapReduce-Like Framework. Proceedings of the 2015 International Workshop on Data-Intensive Scalable Computing Systems.

27. T. White. Hadoop - The Definitive Guide. O'Reilly, 2012.

28. J. Zhou, N. Bruno, M.-C.Wu, P.-A. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet Map-Reduce. VLDB Journal, 21(5):611-636, 2012.

Appendix

Scenario 1 for k = 100

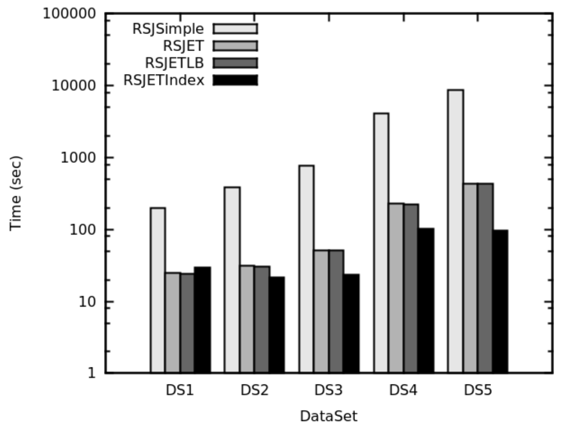


Figure 49: Scenario 1 for k=100 - Total Duration

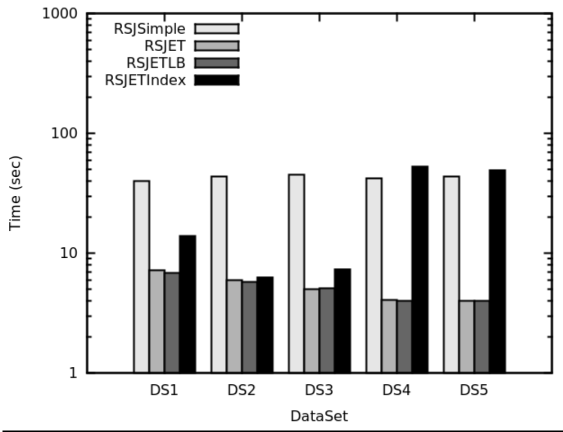


Figure 50: Scenario 1 for k=100 – Average Map Time

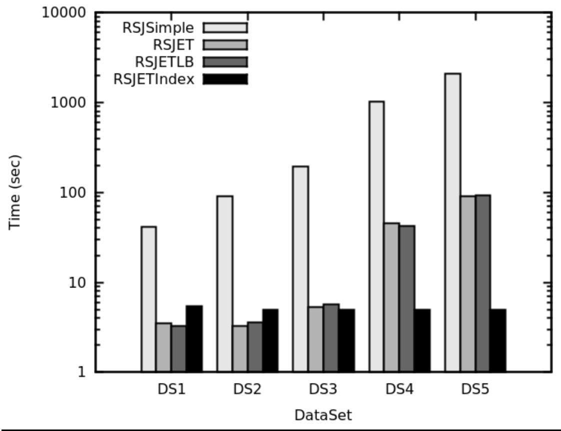


Figure 51: Scenario 1 for k=100 - Average Shuffle Time

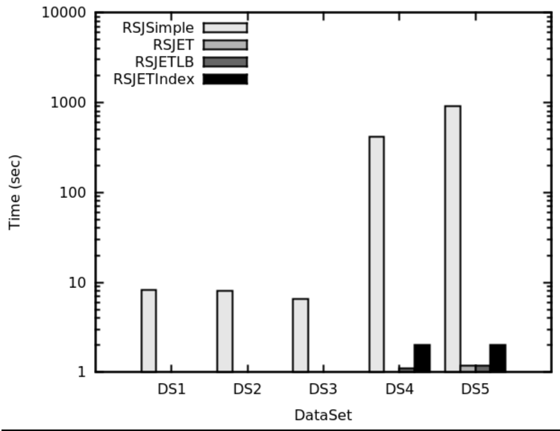


Figure 52: Scenario 1 for k=100 - Average Merge Time

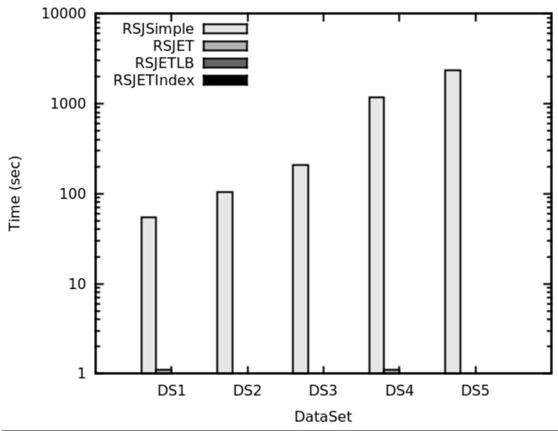


Figure 53: Scenario 1 for k=100 - Average Reduce Time

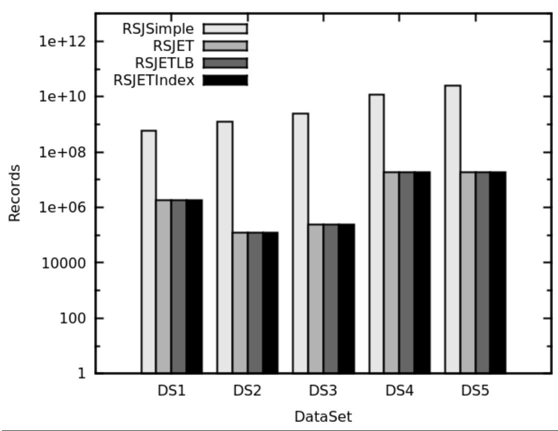


Figure 54: Scenario 1 for k =100 - Map Input Records

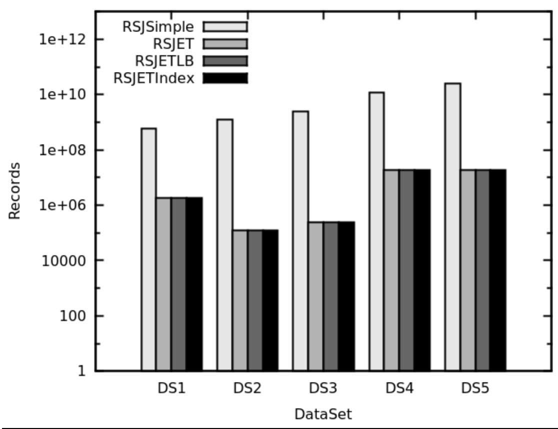


Figure 55: Scenario 1 for k=100 - Map Output Records

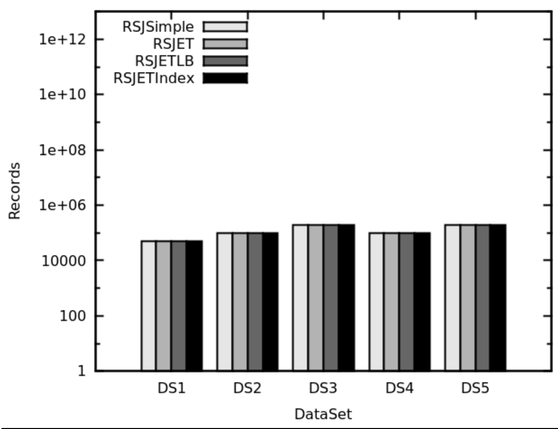


Figure 56: Scenario 1 for k=100 - Reduce Output Records

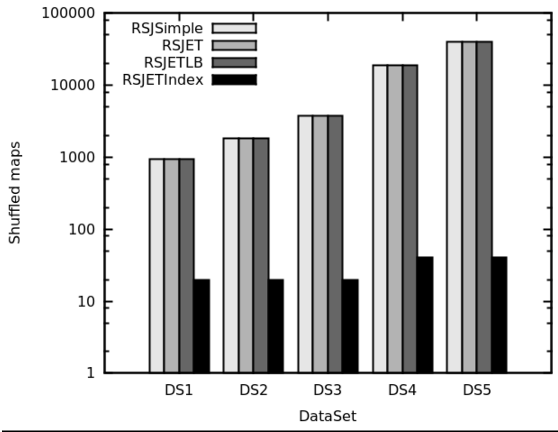


Figure 57: Scenario 1 for k = 100 - Shuffled Maps

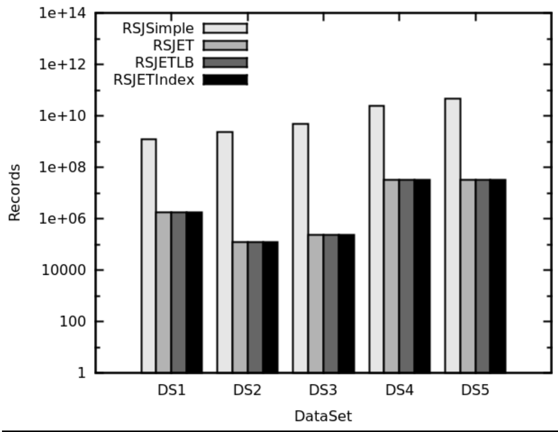


Figure 58: Scenario 1 for k=100 - Map Spilled Records

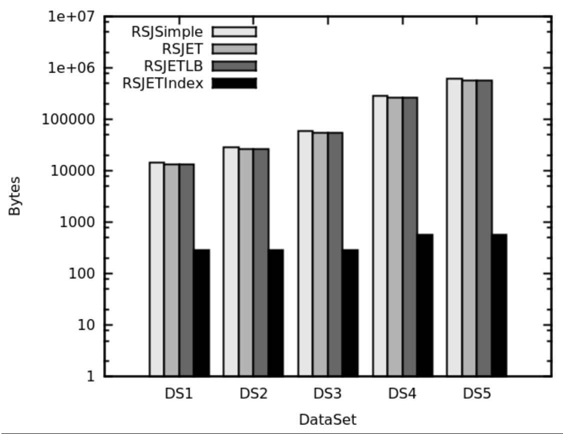


Figure 59: Scenario 1 for k=100 - Input Split Bytes

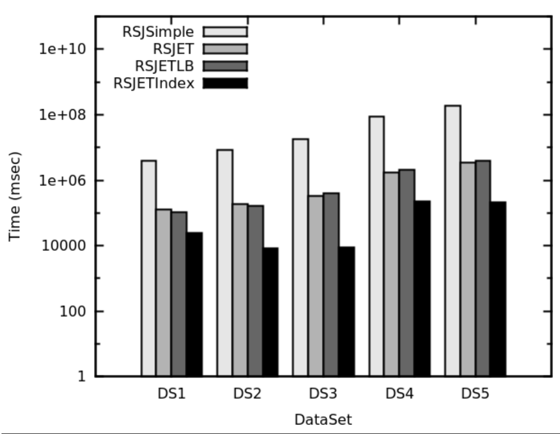


Figure 60: Scenario 1 for k=100 - CPU Map Time

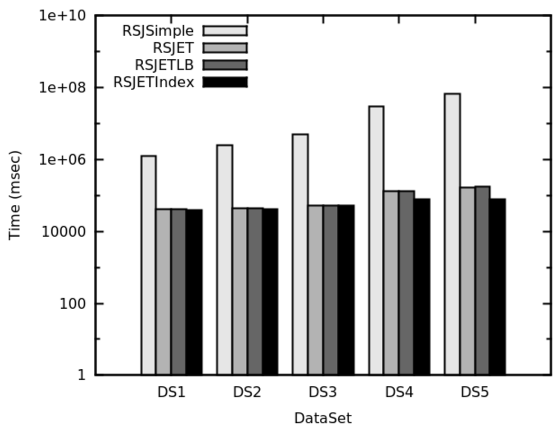


Figure 61: Scenario 1 for k=100 - CPU Reduce Time

Scenario 1 for k = 500

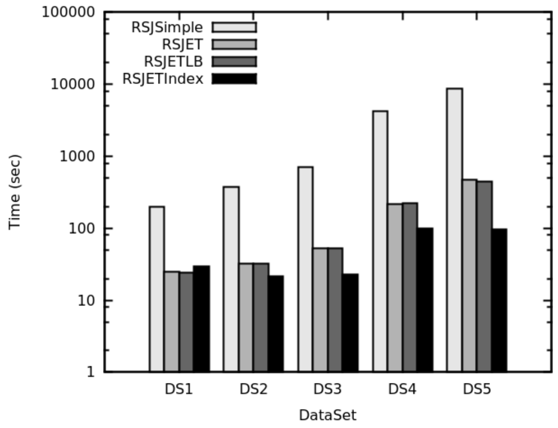


Figure 62: Scenario 1 for k=500 - Total Duration

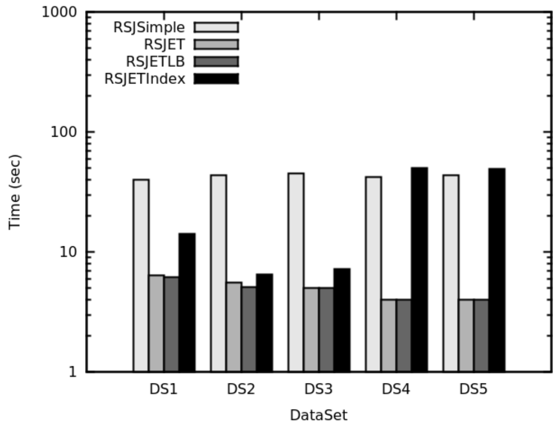


Figure 63: Scenario 1 for k=500 - Average Map Time

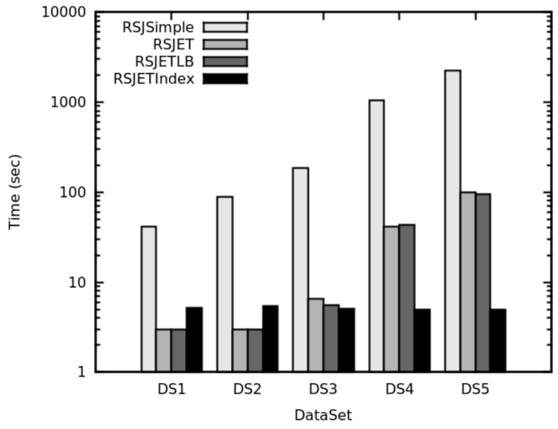


Figure 64: Scenario 1 for k=500 - Average Shuffle Time

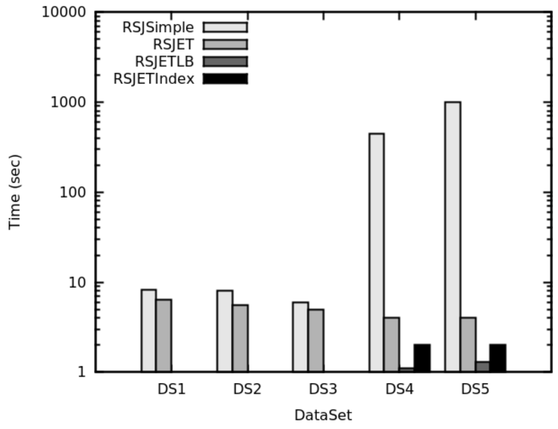


Figure 65: Scenario 1 for k=500 - Average Merge Time

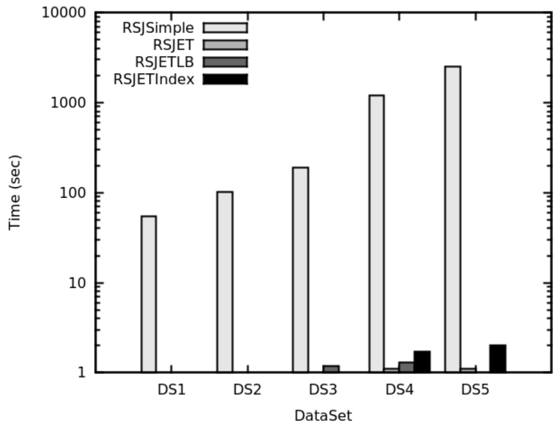


Figure 66: Scenario 1 for k=500 - Average Reduce Time

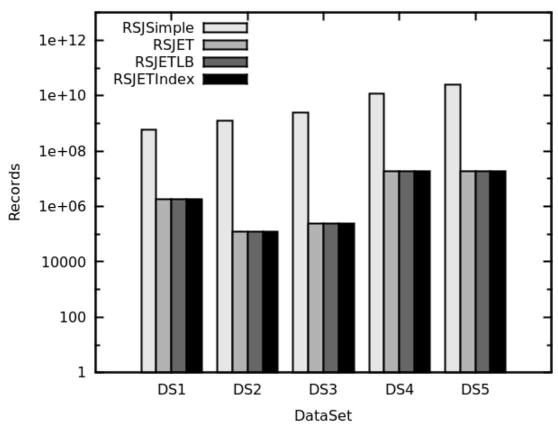


Figure 67: Scenario 1 for k=500 - Map Input Records

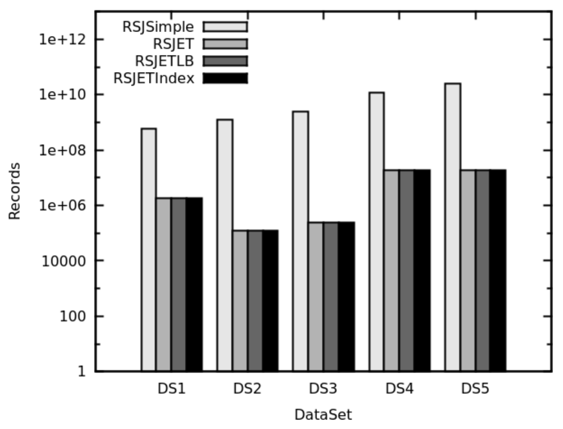


Figure 68: Scenario 1 for k=500 - Map Output Records

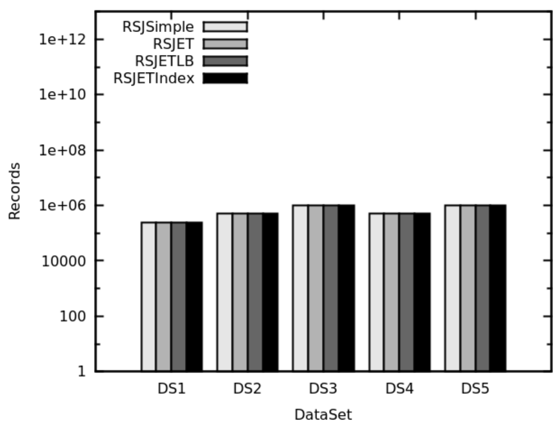


Figure 69: Scenario 1 for k=500 - Reduce Output Records

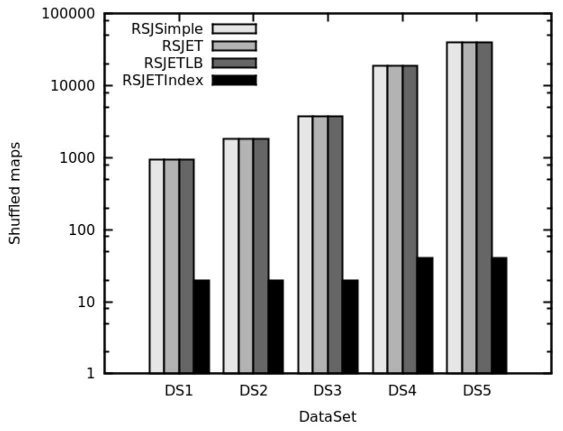


Figure 70: Scenario 1 for k=500 - Shuffled Maps

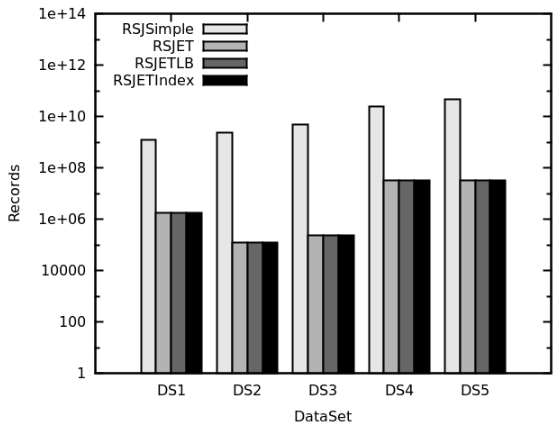


Figure 71: Scenario 1 for k=500 - Map Spilled Records

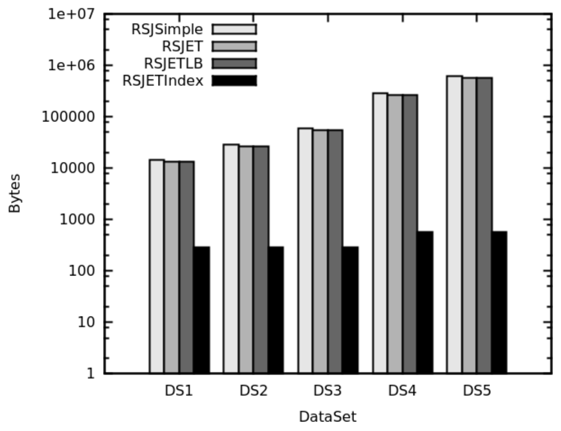


Figure 72: Scenario 1 for k=500 - Input Split Bytes

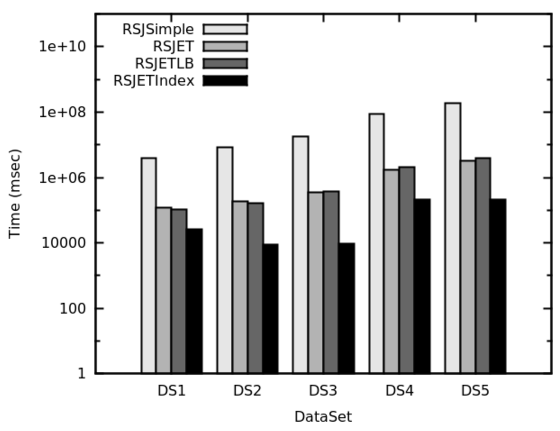


Figure 73: Scenario 1 for k=500 - CPU Map Time

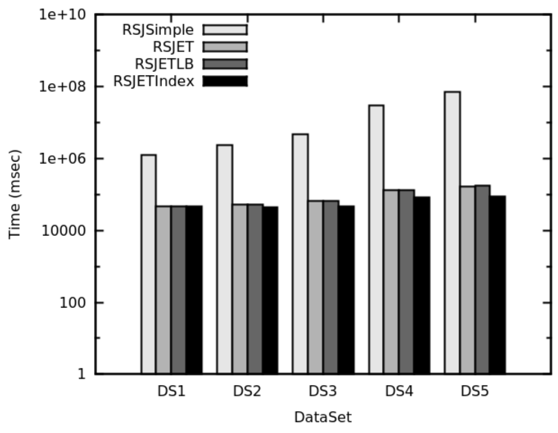


Figure 74: Scenario 1 for k=500 - CPU Reduce Time

Scenario 4

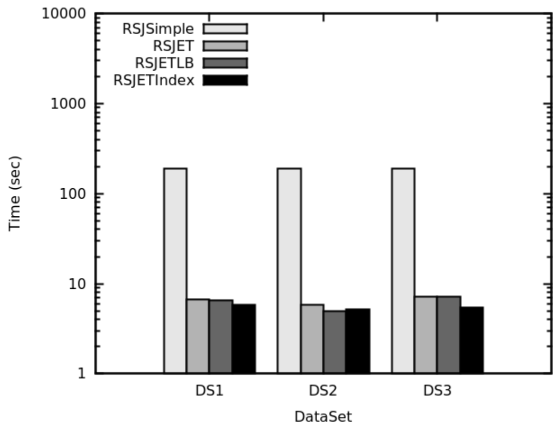


Figure 75: Scenario 4 - Average Shuffle Time

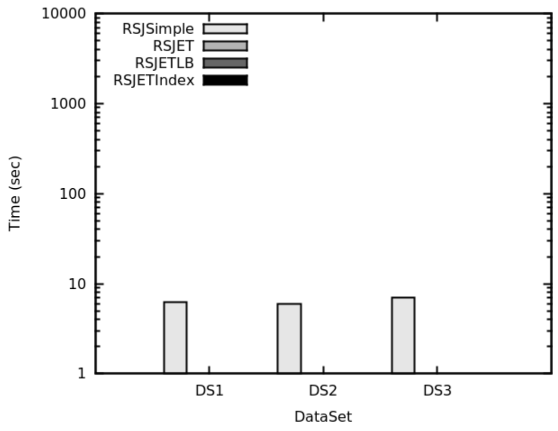


Figure 76: Scenario 4 - Average Merge Time

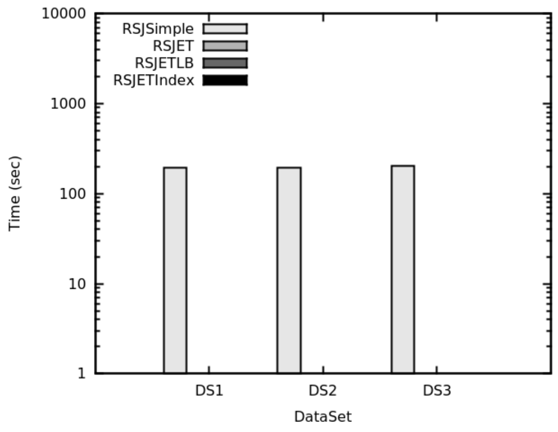


Figure 77: Scenario 4 - Average Reduce Time