Exploiting MapReduce-based Similarity Joins

Yasin N. Silva
Arizona State University
4701 W. Thunderbird Road
Glendale, AZ 85306, USA
ysilva@asu.edu

Jason M. Reed
Arizona State University
4701 W. Thunderbird Road
Glendale, AZ 85306, USA
jmreed3@asu.edu

ABSTRACT

Cloud enabled systems have become a crucial component to efficiently process and analyze massive amounts of data. One of the key data processing and analysis operations is the Similarity Join, which retrieves all data pairs whose distances are smaller than a pre-defined threshold ε. Even though multiple algorithms and implementation techniques have been proposed for Similarity Joins, very little work has addressed the study of Similarity Joins for cloud systems. This paper presents *MRSimJoin*, a multi-round MapReduce based algorithm to efficiently solve the Similarity Join problem. MRSimJoin efficiently partitions and distributes the data until the subsets are small enough to be processed in a single node. The proposed algorithm is general enough to be used with data that lies in any metric space. We have implemented MRSimJoin in Hadoop, a highly used open-source cloud system. We show how this operation can be used in multiple real-world data analysis scenarios with multiple data types and distance functions. Particularly, we show the use of MRSimJoin to identify similar images represented as feature vectors, and similar publications in a bibliographic database. We also show how MRSimJoin scales in each scenario when important parameters, e.g., ε, data size and number of cluster nodes, increase. We demonstrate the execution of MRSimJoin queries using an Amazon Elastic Compute Cloud (EC2) cluster.

Categories and Subject Descriptors

H.2.4 [**Database management**]: Systems—*query processing, parallel databases*

Keywords

Similarity Join, MapReduce, Hadoop

1. INTRODUCTION

The analysis of massive amounts of data is a routine activity in many companies and scientific labs. Internet companies, for instance, collect large amounts of data such as content produced by web crawlers, service logs and click streams. Analyzing these datasets may require processing tens or hundreds of terabytes of data. Cloud systems constitute an answer to the requirements of processing massive amounts of data in a highly scalable and distributed fashion. These systems are composed of large clusters of commodity machines and are often dynamically scalable, i.e., cluster nodes can be added or removed based on the workload. The main software framework for distributed processing over cloud systems is MapReduce [4]. This framework processes massive datasets by splitting them into independent chunks that are processed in a highly parallel fashion. The work in [11] extended this framework with a *merge* phase to facilitate the implementation of regular join operations.

One of the most useful data processing and analysis operations is the Similarity Join (SJ), which retrieves all data pairs whose distances are smaller than a pre-defined threshold ε. Similarity Joins have been studied and extensively used in multiple application domains. Several Similarity Join algorithms and implementation techniques have been proposed. They range from approaches for only in-memory or external memory data [5, 8] to techniques that make use of database operators to answer Similarity Joins [3, 9]. Unfortunately, there has not been much work on the study of this operation on cloud systems. To the best of our knowledge, the only work that addressed the problem of Similarity Joins in this context is the one presented in [10]. The work in [10], however, focuses on the study of a different and more specialized type of Similarity Join (Set-Similarity Join) which constrains its applicability to set-based data.

This paper presents MRSimJoin, a MapReduce based algorithm that solves the Similarity Join problem by iteratively partitioning and distributing the data until the subsets are small enough to be processed in a single node. We have implemented MRSimJoin in Hadoop [2], a highly used open-source cloud system. The proposed approach is general enough to be used with any dataset that lies in a metric space. We show how this operation can be used in multiple real-world data analysis scenarios with multiple data types and distance functions. Particularly, we show the use of MRSimJoin to identify: (1) similar images represented as feature vectors, and (2) publications in the DBLP bibliographic database with similar titles.

The remaining part of this paper is organized as follows. Section 2 describes the MRSimJoin algorithm. Section 3 describes the demonstration scenarios. Section 4 presents the conclusions and future research directions.

SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.

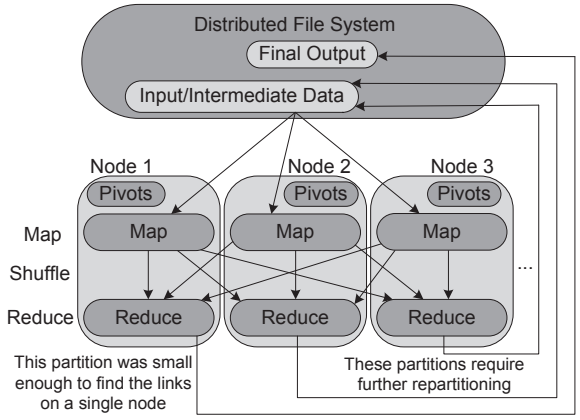

Figure 1: A MRSimJoin round.

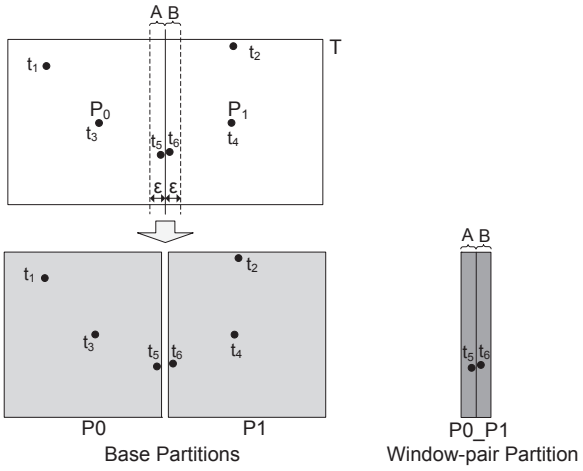

Figure 2: Repartitioning a base partition.

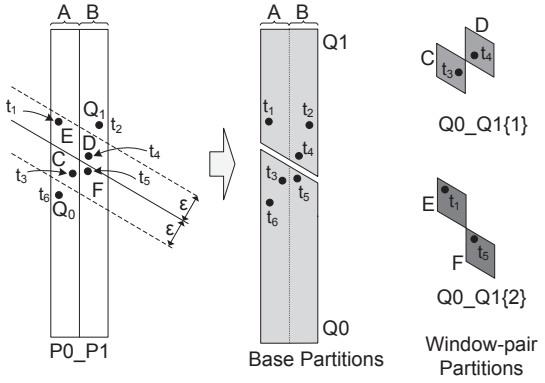

Figure 3: Repartitioning a window-pair partition.

2. THE MRSinJoin ALGORITHM

The Similarity Join (SJ) operation between two datasets R and S is defined as $R \bowtie_{\theta_\varepsilon(r,s)} S = \{\langle r, s\rangle | \theta_\varepsilon(r,s), r \in R, s \in S\}$, where $\theta_\varepsilon(r,s)$ represents the SJ predicate, i.e., $dist(r,s) \leq \varepsilon$. The result pairs $\langle r, s\rangle$ are referenced as *links*.

The input data can be given in one or multiple distributed files and each file can contain records of both R and S. Each record contains the id of the dataset that the record belongs to and the id of the record in the dataset.

MRSimJoin iteratively partitions the input data into smaller partitions until each partition is small enough to be efficiently processed by a single-node SJ routine. The process is divided into a sequence of rounds. The initial round partitions the input data while any subsequent round repartitions a previously generated partition. Each round corresponds to a MapReduce job. The input and output of each job is read from or written to the distributed file system (DFS). The output of a round includes: (1) result links for the small partitions that were processed in a single-node, and (2) intermediate data for partitions that require further partitioning. Fig. 1 represents the execution of a single round and shows that data partitioning and the generation of intermediate and final results are performed in parallel by multiple nodes. The main MRSimJoin routine executes the required rounds until all the input and intermediate data is processed.

Data partitioning is performed using a set of K pivots, which are a subset of the records to be partitioned. The process generates two types of partitions: *base partitions* and *window-pair partitions*. A base partition contains all the records that are closer to a given pivot than to any other pivot. A window-pair partition contains the records in the boundary between two base partitions. In general, the window-pair records should be a superset of the records whose distance to the hyperplane that separates the base partitions is at most ε. However, this hyperplane does not always explicitly exist in a metric space. Instead, it is implicit and known as a *generalized hyperplane*. Since the distance of a record t to the generalized hyperplane between two partitions with pivots P_0 and P_1 cannot always be computed exactly, a lower bound of the distance is used [7]:

$$gen\_hyperplane\_dist(t, P_0, P_1) = (dist(t, P_0) - dist(t, P_1))/2$$

This distance can be replaced with an exact distance if this can be computed, e.g., in Euclidean spaces.

Processing the window-pair partitions guarantees the identification of the links between records that belong to different base partitions. At the logical level, the data partitioning in MRSimJoin is similar to the one in the Quickjoin algorithm [8]. The core difference, however, is that in MRSimJoin the partitioning of the data, the generation of the result links, and the storage of intermediate results is performed in a fully distributed and parallel manner.

Fig. 2 represents the repartitioning of a base partition using two pivots. In this case, the result of the Similarity Join operation on the dataset T is the union of the links in $P0$ and $P1$, and the links in $P0\_P1$ where one element belongs to window A and the other one to window B. We refer to this last type of links as *window links*. Fig. 3 represents the repartitioning of the window-pair partition $P0\_P1$ of Fig. 2. In this case, the set of window links in $P0\_P1$ is the union of the window links in $Q0$, $Q1$, $Q0\_Q1\{1\}$ and $Q0\_Q1\{2\}$. Note that windows C and F do not form a window-pair partition because their window links are a subset of the window links in $Q0$. Similarly, the window links between E and D are a subset of the window links in $Q1$.

Rounds that aim to identify all the similarity links in the input data are referred to as base rounds. Rounds that identify only the window links, i.e., links between records that correspond to different previous partitions, are referred to as window-pair rounds.

Algorithm 1 *MRSimJoin*($inDir$, $outDir$, $numPiv$, eps, $memT$)

Input: $inDir$ (input directory with the records of datasets R and S), $outDir$ (output directory), $numPiv$ (number of pivots), eps (epsilon), $memT$ (memory threshold)
Output: $outDir$ contains all the results of the Similarity Join operation $R \bowtie_{\theta_\varepsilon(r,s)} S$

```
1.  intermDir ← outDir + "/intermediate"
2.  roundNum ← 0
3.  while true do
4.    if roundNum = 0 then
5.      job_inDir ← inDir
6.    else
7.      job_inDir ← GetUnprocessedDir(intermDir)
8.    end if
9.    if job_inDir = null then
10.     break
11.   end if
12.   pivots ← GeneratePivots(job_inDir, numPiv)
13.   if isBaseRound(job_inDir) then
14.     MR_Job(Map_base, Reduce_base, Partition_base,
        Compare_base, job_inDir, outDir, pivots,
        numPiv, eps, memT, roundNum)
15.   else
16.     MR_Job(Map_windowPair, Reduce_windowPair,
        Partition_windowPair, Compare_windowPair,
        job_inDir, outDir, pivots, numPiv, eps, memT,
        roundNum)
17.   end if
18.   roundNum++
19.   if roundNum > 0 then
20.     RenameFromIntermToProcessed(job_inDir)
21.   end if
22. end while
```

2.1 The Main MRSimJoin Routine

The main routine of MRSimJoin is presented in Algorithm 1. The routine uses an intermediate directory (line 1) to store the partitions that will need further repartitioning. Each iteration of the while loop (lines 3-22) corresponds to one round and executes a MapReduce job. In each round, the initial input data or a previously generated partition is repartitioned. If a generated partition is small enough to be processed in a single node, the SJ links are obtained running a single-node SJ algorithm (we use Quickjoin [8]).

At each round, the main routine sets the values of the job input directory (lines 4-8) and randomly selects $numPivots$ pivots from this directory (line 12). Then the routine executes a base partition MapReduce job (line 14) or a window-pair partition MapReduce job (line 16) based on the type of the job input directory. The MapReduce job uses the provided *map*, *reduce*, *partition* and *compare* functions. The *MR_Job* routine makes sure that the $outDir$, $numPiv$, eps and $memT$ parameters are available at every node that will be used in the MapReduce job and that the *pivots* are available at each node that will execute *map* tasks. Each MapReduce job is executed as follows.

Map. The MapReduce framework divides the job input data into chunks and creates *map* tasks in multiple nodes to process them. The corresponding map function is called once for each input record. The function identifies the partition of the record and generates one intermediate record for each base or window-pair partition the record belongs to.

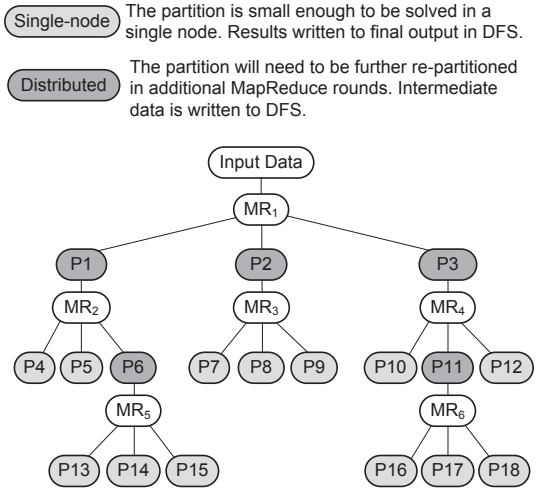

Figure 4: Example of the MapReduce rounds and partitions generated by MRSimJoin.

Partitioning. The MapReduce framework partitions the intermediate data generated by *map* tasks. This partitioning is performed calling the *partition* function. One partition is generated for each base or window-pair partition. After identifying the partition numbers of intermediate records, the *shuffle* phase of the MapReduce job sends the intermediate records to their corresponding reduce nodes.

Sorting and Grouping. The intermediate records received at each reduce node are sorted and grouped using the corresponding *compare* function. This function groups the intermediate records that belong to the same partition.

Reduce. After generating the groups in a reduce node, the MapReduce framework calls the corresponding *reduce* function once for each group. The function receives as input the list of all the records of the group. If the list size is small enough to be processed in a single node, the algorithm calls the single-node Similarity Join routine to get the links in the current partition. Otherwise, all the group records are stored in an intermediate directory for further partitioning. Intermediate data is generated such that the new base partitions generated in a base round are processed in future base rounds, and all other partitions in window-pair rounds.

If a round is processing a previously generated partition, after the MapReduce job finishes, the main routine renames the job input directory to be located under the processed directories (line 20).

Fig. 4 shows an example of the multiple rounds that are executed by the main routine. Each node in the tree with name MR_N represents a MapReduce job. This figure also shows the partitions generated by each job. Light gray partitions are small partitions that are processed running the single-node Similarity Join routine. Dark gray partitions are partitions that require additional repartitioning. A sample sequence of rounds can be: MR_1, MR_2, MR_3, MR_4, MR_5 and MR_6. The original input data is always processed in the first round. Since the links of any partition can be obtained independently, the routine will generate a correct result independently of the order of rounds.

2.2 Implementing MRSimJoin in Hadoop

The MRSimJoin algorithm is generic enough to be implemented in any MapReduce framework. This section presents additional guidelines of our implementation in Hadoop [2].

Distribution of atomic parameters. *MR_Job* sends the atomic parameters, i.e., $outDir$, $numPiv$, eps and $memT$, to every node that will be used in the MapReduce job. This is done using Hadoop's job configuration $jobConf$ object.

Distribution of pivots. *MR_Job* sends the pivots to every node that executes *map* tasks. This is done using *DistributedCache*, a facility that allows the efficient distribution of application-specific, large, read-only files.

Renaming directories. The main MRSimJoin routine renames a directory to flag it as already processed. This is done using the *rename* method of Hadoop's *FileSystem* class. The method will change the directory path in Hadoop's distributed file system without physically moving its data.

3. DEMONSTRATION SCENARIOS

The demonstration of MRSimJoin will be performed using our implementation in Hadoop 0.20.2. We will demonstrate the execution of multiple MRSimJoin queries using a Hadoop cluster running on the Amazon Elastic Compute Cloud (EC2). We will show how this operation can be used in multiple real-world data analysis scenarios using multiple data types and distance functions. In each demonstration scenario, we will also show the way MRSimJoin scales when important parameters, e.g., ε, data size, number of cluster nodes, and number of dimensions increase.

3.1 Identifying Similar Images

In this scenario, we will use MRSimJoin to identify similar images in the Corel image collection [6] as shown in Fig. 5. We use two different datasets: *ColorMoments* and *CoOccurrenceTexture*. The scale factor 1 datasets have 5 million records. Each record of *ColorMoments* is a 9D feature vector with components in the range [-4.8 - 4.4]. Each vector contains the following values: the mean, standard deviation, and skewness for each of H, S and V in the HSV color space. Each record of *CoOccurrenceTexture* is a 16D feature vector. *CoOccurrenceTexture* was generated converting the images to 16 gray-scale images. Each vector contains the following values: the Second Angular Moment, Contrast, Inverse Difference Moment, and Entropy in 4 directions (horizontal, vertical, and two diagonal directions). We use the Euclidean distance function to measure the similarity between images.

3.2 Identifying Similar Publications

In this scenario, we will run multiple MRSimJoin queries to identify publications with similar titles in the DBLP bibliographic dataset [1]. The scale factor 1 dataset has 10,000 records. The title of each publication record is a string of 7 to 342 characters. We use the Levenshtein distance function to measure the similarity between publication titles.

4. CONCLUSIONS AND FUTURE WORK

Cloud-based systems have become a crucial component to analyze large amounts of data. The Similarity Join is recognized as one of the most useful data analysis operations and has been used in many application scenarios. While multiple Similarity Join implementation techniques have been proposed, very little work has addressed the study of Similarity Joins for cloud systems. This paper presents MRSimJoin, a multi-round MapReduce based algorithm to efficiently solve the Similarity Join problem. MRSimJoin can be used with any dataset that lies in a metric space. We demonstrate the use of MRSimJoin using real-world datasets and multiple data types and distance functions. The demonstration will be performed using a Hadoop cluster running on Amazon EC2. Particularly, MRSimJoin queries will be used to identify similar images and similar publications. We will also show the way MRSimJoin scales when various parameters like ε, data size, number of nodes and number of dimensions increase. Future work paths include the study of: indexing techniques to improve the efficiency of similarity operations, and cloud queries with multiple similarity operators.

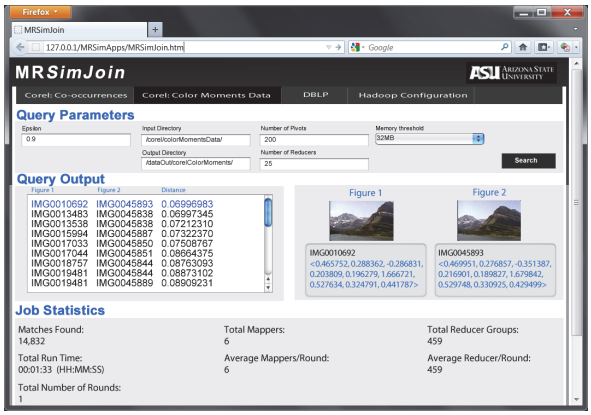

Figure 5: Finding similar images in *ColorMoments*.

5. REFERENCES

[1] Dblp bibliography. `http://www.informatik.uni-trier.de/~ley/db/`.

[2] Apache. Hadoop. `http://hadoop.apache.org/`.

[3] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[5] V. Dohnal, C. Gennaro, and P. Zezula. Similarity join in metric spaces using ed-index. In *Database and Expert Systems Applications*, volume 2736 of *Lecture Notes in Computer Science*, pages 484–493. 2003.

[6] A. Frank and A. Asuncion. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2010.

[7] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28:517–580, December 2003.

[8] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33:7:1–7:38, June 2008.

[9] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *ICDE*, 2010.

[10] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *ACM SIGMOD*, 2010.

[11] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *ACM SIGMOD*, 2007.