Efficient Entity Matching over Multiple Data Sources with MapReduce

Demetrio Gomes Mestre, Carlos Eduardo Pires

Universidade Federal de Campina Grande, Brazil
demetriogm@gmail.com, cesp@dsc.ufcg.edu.br

Abstract. The execution of data-intensive tasks such as entity matching on large data sources has become a common demand in the era of Big Data. To face this challenge, cloud computing has proven to be a powerful ally to efficient parallel the execution of such tasks. In this work we investigate how to efficiently perform entity matching over multiple large data sources using the MapReduce programming model. We propose MSBlockSlicer, a MapReduce-based approach that supports blocking techniques to reduce the entity matching search space. The approach utilizes a preprocessing MapReduce job to analyze the data distribution and provides an improved load balancing by applying an efficient block slice strategy as well as a well-known optimization algorithm to assign the generated match tasks. We evaluate our approach against an existing one that addresses the same problem on a real cloud infrastructure. The results show that our approach increases significantly the performance of distributed entity match task by reducing the amount of generated data from the map phase and minimizing the execution time.

Categories and Subject Descriptors: H.2.4 [**Systems**]: Distributed Databases; H.3.4 [**Systems and Software**]: Distributed Systems

Keywords: Entity Matching, LoadBalancing, MapReduce, Multiple Data Sources

1. INTRODUCTION

Distributed computing has received a lot of attention lately to perform high data-intensive tasks. Extensive powerful distributed hardware and service infrastructures capable of processing millions of these tasks are available around the world. Programming models have being created to make efficient use of such cloud environments. In this context, MapReduce (MR) [Dean and Ghemawat 2008], a well-known programming model for parallel processing on cloud infrastructures, emerges as a major alternative for the efficient distributed data-intensive tasks.

Entity Matching (EM) (also known as entity resolution, deduplication, or record linkage) is a data-intensive and performance critical task and demands studies on how it can benefit from cloud computing. EM is applied to determine all entities (duplicates) referring to the same real world object given a set of data sources [Kopcke and Rahm 2010]. The task has critical importance for data cleaning and integration, e.g., to find duplicate product descriptions in databases.

Two common situations can be found when dealing with EM over data sources, the single and the multiple data sources matching. The first one refers to find all the duplicates in a single data source (traditional) and the second one, focus of this work, refers to the special case of find the duplicates between two or more data sources [Kopcke and Rahm 2010]. Both situations share the main problem that makes EM heavy to perform, the need of applying matching techniques on the Cartesian product of all input entities (naive) leading to a quadratic complexity of $O(n^2)$. For large datasets, the application of such approach is very ineffective.

To minimize the workload caused by the Cartesian product execution and to maintain the match quality, techniques like blocking [Baxter et al. 2003] become necessary. Such techniques work by partitioning the input data into blocks of similar entities and restricting EM to entities of the same block. For instance, it is sufficient to compare entities of the same manufacturer when matching product offers.

Nevertheless, even using blocking techniques, EM remains hard to process for large datasets [Kolb et al. 2012a]. Therefore, EM is an ideal problem to be treated with a distributed solution. The execution of blocking-based EM over single and multiple sources can be done in parallel with the MR model by using several map and reduce tasks. More specifically, the map tasks can read the input entities in parallel and redistribute them among the reduce tasks according to the blocking key. Entities sharing the same blocking key are assigned to a common reduce task and several blocks can be matched in parallel.

However, this simple MR implementation, known as **Basic**, has vulnerabilities. Severe load imbalances can occur due to large blocks (skew problem) occupy a node for a long time and leave the other nodes idle. This is not interesting since there is an urgency to complete the EM process as quickly as possible.

The load imbalance problem when dealing with single sources has been addressed by **BlockSplit** [Kolb et al. 2012a], a general load balancing MR-based approach that takes the size of blocks into account. More details about this work will be shown in the Related Work section. However, this solution has load imbalances and excessive entity replication issues that undermine its running time. Consequently, **BlockSplit**'s extension that address the multiple data sources problem, also found in [Kolb et al. 2012a], extends the same issues of the single source solution. To overcome these problems, we make the following contributions:

—We propose **MSBlockSlicer** (Multiple Source BlockSlicer), an extension of our **BlockSlicer** approach [Mestre and Pires 2013] (proposed to address the single source problem) that provides a load balancing improvement by applying an efficient block slice strategy over multiple data sources. The approach takes the size of blocks into account and generates match tasks of entire blocks only if this does not violate the load balancing constraints. Larger blocks are sliced into several match tasks to enable a fewer number of comparisons respecting the Cartesian product. A greedy optimization is used to assign match tasks of entire blocks and sliced ones to the proper reduce tasks aiming to optimize the load balancing parallel matching.

—We evaluate **MSBlockSlicer** against **BlockSplit** extension for multiple data sources and show that our approach provides a better load balancing strategy by reducing the amount of data generated from the map phase and diminishing the overall execution time. The evaluation is performed on a real cloud environment and uses real-world data.

2. RELATED WORK

Entity Matching is a very studied research topic. Many approaches have been proposed and evaluated as described in the recent survey [Kopcke and Rahm 2010]. However there are only a few approaches that consider parallel entity matching. The first steps in order to evaluate the parallel Cartesian product of two sources is described in [Kim and Lee 2007]. [Kirsten et al. 2010] proposes a generic model for parallel entity matching based on general partitioning strategies that take memory and load balancing requirements into account.

Few MR-based approaches address the load balancing and skew handling problem. [Okcan and Riedewald 2011] applied a static load balancing mechanism, but it is not suitable due to arbitrary join assumptions. Similarly to our work, the authors employ a previous analysis phase to determine the datasets' characteristics (using sampling) and thereafter avoid the evaluation of the Cartesian product. This approach focus on data skew handle in the map process output, which leads to an overhead in the map phase and large amount of map output.

MapReduce has already been employed for EM (e.g., [Wang et al. 2010]) but load balancing was not the main focus and only one mechanism of near duplicate detection by the PPjoin paradigm adapted to the MapReduce framework can be found. [Kolb et al. 2012b] studies load balancing for Sorted Neighborhood (SN). However, SN follows a different blocking approach (fixed window size) that is by design less vulnerable to skewed data. [Vernica et al. 2010] shows another approach for parallel processing entity matching on a cloud infrastructure and an extension that address the multiple data source problem. This study explains how a single token-based string similarity function performs with MR. However, this approach suffers from load imbalances because some reduce tasks process more comparisons than the others.

As mentioned in the introduction, [Kolb et al. 2012a] addresses our problem. The authors present two approaches, **BlockSplit** and **PairRange**. **BlockSplit** is an algorithm that processes small blocks within single match tasks. The larger blocks are split according to the m input partitions into m sub-blocks obeying an appropriate scheme of entity replication (based on m input partitions). Each sub-block is processed by a single match task. **PairRange** on the other hand implements a virtual enumeration of all entities and the relevant comparisons (pairs) based on the data distribution provided by the so-called BDM, Block Distribution Matrix, aiming to send entities to all reduce tasks according to the relevant comparisons belonging to ranges previously established based on the average workload. **PairRange** neither consider input partitions nor blocks, but instead ranges of comparisons. Despite **PairRange** provide a little more uniform distribution than **BlockSplit**, it generates too much entity replication (for load balancing purposes) which leads to an extra overhead. According to [Kolb et al. 2012a], this overhead leads **PairRange** to perform equals or worse than **BlockSplit**. In addition, such overhead can lead to lack of memory problems in the cloud infrastructure when executing **PairRange** for huge datasets. For this reason, we have only implemented **BlockSplit** and compare it with our work in an experimental evaluation.

3. GENERAL MR-BASED ENTITY MATCHING WORKFLOW FOR LOAD BALANCING

To perform our load balancing optimization for EM processing over multiple data sources, we need two MR jobs.

3.1 First Job: Block Distribution Matrix

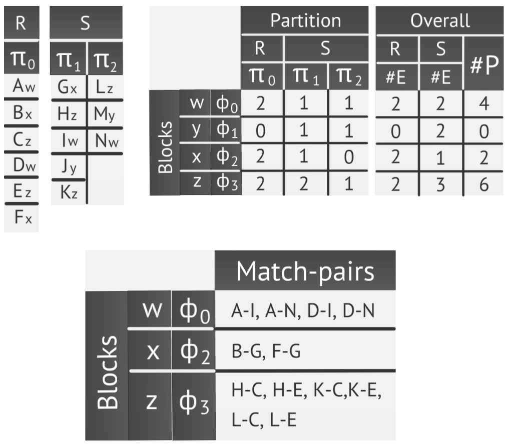
The Block Distribution Matrix (BDM) is a simple preprocessing step to determine some datasets' characteristics. As described in [Kolb et al. 2012a], the BDM consists in a b x m matrix that specifies the number of entities of b blocks across m input partitions.

3.2 Second Job: Improving Block-based Load Balancing

The second job, denoted by **MSBlockSlicer**, performs our load balancing optimization approach for EM over multiple data sources. Since the EM of multiple data sources problem can be simplified to the pairwise comparisons of all envolved data sources, this section describes an extension of **BlockSlicer** [Mestre and Pires 2013] for matching two sources R and S.

As shown in the example data of Figure 1, we utilize the entities A-N and the blocking keys w-z. Each entity belongs to one of the two sources R and S. Source R is stored in the partition Π_0 and source S is stored in the partitions Π_1 and Π_2. The sources are automatic partitioned according to the number of available map tasks.

The BDM computation is the same but adds a source tag to the map output key aiming to identify



R	S	
π_0	π_1	π_2
Aw	Gx	Lz
Bx	Hz	My
Cz	Iw	Nw
Dw	Jy	
Ez	Kz	
Fx		

			Partition			Overall		
			R	S		R	S	#P
			π_0	π_1	π_2	#E	#E	
Blocks	w	Φ_0	2	1	1	2	2	4
	y	Φ_1	0	1	1	0	2	0
	x	Φ_2	2	1	0	2	1	2
	z	Φ_3	2	2	1	2	3	6

			Match-pairs
Blocks	w	Φ_0	A-I, A-N, D-I, D-N
	x	Φ_2	B-G, F-G
	z	Φ_3	H-C, H-E, K-C,K-E, L-C, L-E

Fig. 1. Example data (up-left), BDM (up-right), and the expected match-pairs (down-center).

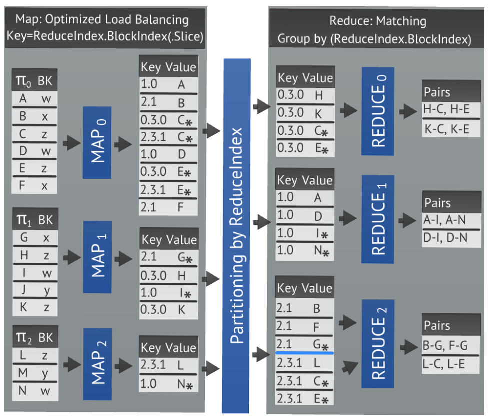


Fig. 2. Example BlockSlicer dataflow for 2 sources.

blocks with the same key in different sources, i.e., $\Phi_{i,R}$ and $\Phi_{i,S}$. The BDM has the same structure for the one-source case but distinguishes between the two sources for each block (see Figure 1).

3.2.1 *MSBlockSlicer Load Balancing.* The **BlockSlicer** approach for two sources (**MSBlockSlicer**) follows the same scheme as for one source, but has two main differences. Firstly, given a block B, the sliced-block sb consists in allowed entities $B_{allowed}$ belonging to the same source (R or S) and the not allowed blocks $B_{notAllowed}$ belonging to the other source. $B_{allowed}$ is extracted from the source that contains the largest number of entities from B. Secondly, the entities belonging to $B_{allowed}$ will no longer iterate comparisons among themselves, only with the entities belonging to $B_{notAllowed}$. In this case, if the permutation of $B_{allowed}$ entities with $B_{notAllowed}$ entities generates a number of pairs above the average workload, then sb is sliced into two new sliced-blocks to enable load balancing. The slicing is processed only on entities belonging to $B_{allowed}$ and the entities belonging to $B_{notAllowed}$ are emitted for each sliced-block generated. The exact slicing is calculated as $\mid B_{allowed} \mid = \frac{\lceil T/r \rceil}{|B_{notAllowed}|}$, where T is the number of the comparisons generated and r is the number of reduce tasks available.

After the slicing phase, the remaining entities of $B_{allowed}$ are submitted to a new average workload constraint verification. If the number of pairs is still above the average workload, the remaining entities of $B_{allowed}$ are submitted to a new slicing process (recursive procedure).

Figure 2 shows the workflow for the example data of Figure 1. The BDM indicates 12 overall pairs so that the average workload is 4 pairs. The largest block Φ_3 is therefore subject to a slice process due to the number of pairs that must be processed (6 pairs). The slice phase results in two new sliced-blocks (the match tasks 3.0 and 3.1). All match tasks are ordered by the number of pairs: 3.0 (4 pairs, $reduce_0$), 0 (4 pairs, $reduce_1$), 2 (2 pairs, $reduce_2$) and 3.1 (2 pairs, $reduce_2$). The dataflow is shown in Figure 2. Partitioning is based on the reduce task index only, for routing all data to the reduce tasks whereas sorting is done based on the entire key. The reduce function is called for every match task $k.(i)$ and compares entities considering only pairs from different entity types (allowed against not allowed). For illustration, Figure 2 shows that H is an allowed entity from match task 3.0 and is only compared with the not allowed entities (marked with an "*"), e.g., the entities C and E.

4. EVALUATION

In the following, we evaluate[2] **MSBlockSlicer** against **BlockSplit**'s extension approach, which was implemented according to the pseudo-code available in [Kolb et al. 2012a], regarding one performance critical factor: the number of available nodes (n) in the cloud environment. In each experiment we evaluate the algorithms aiming to investigate their behavior when dealing with the resources consumption caused by the use of many map and reduce tasks and how they can scale with the number of available nodes.

We ran our experiments on a 10-node HP Pavilion P7-1130 cluster (with Hadoop 0.20.2) and utilized one real-world dataset. The dataset DS1 (DBLP) contains about 1.46 million publication records and was divided into two data sources (R and S) containing about 730,000 entities each. The first three letters of the publication title form the default blocking key. Two entities were compared by computing the Jaro-Winkler [Cohen et al. 2003] distance of their comparing attributes and those pairs with a similarity ≥ 0.7 were regarded as matches.

4.1 Scalability: Number of nodes

As mentioned in the introduction, scalability is important for many reasons and one of them is the financial. The number of nodes should be carefully estimated since distributed infrastructure suppliers usually charge per hired machines even if they are underutilized. To analyze the scalability of the two approaches, we vary the number of nodes from 1 up to 10. Following the Hadoop's documentation, for n nodes, the number of map tasks is set to $m = 2 \cdot n$ and the number of reduce tasks is set to $r = 4 \cdot n$, i.e., adding new nodes leads to additional map and reduce tasks. The resulting execution times values are shown in Figure 3 (DS1) and the respectively number of generated key-value pairs are illustrated in Figure 4.

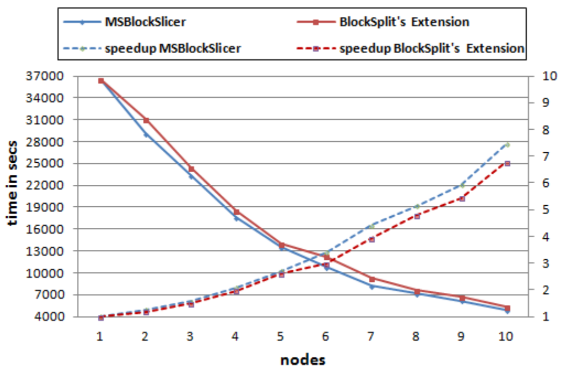


Fig. 3. Execution times and speedup for both approaches using DS1.

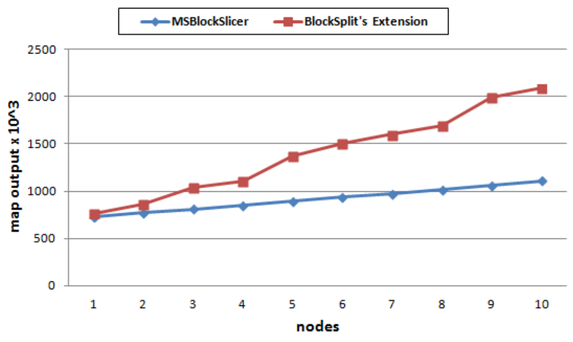


Fig. 4. Number of generated key-value pairs by map for DS1 varying the number of nodes n (m=2·n, r=4·n).

We can note that both **MSBlockSlicer** and **BlockSplit**'s extension scale almost linearly showing their ability to evenly distribute the workload across reduce tasks and nodes. However, due to the difficulties already discussed, i.e., high dependency of the input partitions, **BlockSplit**'s execution time is compromised by the deficiency of properly split larger blocks with few input partitions as we can see in Figure 3 when we vary n from 5 to 6. This deficiency is the main reason of **BlockSplit**'s extension approach to perform around 1000 seconds slower than **MSBlockSlicer** on experiments with 10 nodes (m=20 and r=40). The difference is highlighted by the speedup (it's like **MSBlockSlicer** has almost one extra node working). Furthermore, Figure 4 shows, by the map output generation,

[2]The datasets and codes are available in https://sites.google.com/site/demetriomestre/activities

an overgrowth of output entities during the **BlockSplit**'s map phase. This overgrowth leads to the associated overhead already discussed earlier, i.e, unnecessary data transfer (network resources), sorting large partitions and OS memory management when processing reduce tasks, and thereby causes the deterioration of the execution time. Besides, depending on the entities length or dataset size, such situation can cause serious problems of lack of memory. Note that, for 10 nodes, the amount of output generated by **BlockSplit**'s extension approach is almost twice the amount generated by the **MSBlockSlicer** one. This limits **BlockSplit**'s extension to be used with huge datasets sustainably.

5. SUMMARY AND OUTLOOK

We proposed an improved load balancing approach, **MSBlockSlicer**, for distributed blocking-based entity matching over multiple data sources using a well-known MapReduce framework. The solution is by design efficient to provide load balancing to an entity matching process of huge datasets without depending on the data distribution or order of the input partitions with a significant improved reduction of the information emitted from map to reduce phases (map output). It is able to deal optimally with skewed data distributions and workload among all reduce tasks by slicing large blocks. Our evaluation on a real cloud environment using real-world data demonstrated that **MSBlockSlicer** scale with the number of available nodes without relying on any extra architecture configuration. We compared our approach against an existing one (**BlockSplit**'s extension) and we verified that **MSBlockSlicer** overcomes **BlockSplit**'s extension in performance terms.

In future work, we will investigate how we can improve our solution to also address the horizontal skew problem (entities with disproportionate lengths). Also, we will further investigate how our load balancing approach can be adapted to address other MapReduce-based techniques for different kinds of data-intensive tasks, such as join processing or data mining.

REFERENCES

Baxter, R., Christen, P., and Churches, T. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD '03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*. pp. 25–27, 2003.

Cohen, W. W., Ravikumar, P., and Fienberg, S. E. A comparison of string distance metrics for name-matching tasks. In *IIWeb*. pp. 73–78, 2003.

Dean, J. and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM* 51 (1): 107–113, Jan., 2008.

Kim, H.-s. and Lee, D. Parallel linkage. In *Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*. CIKM '07. ACM, New York, NY, USA, pp. 283–292, 2007.

Kirsten, T., Kolb, L., Hartung, M., Gross, A., Kopcke, H., and Rahm, E. Data Partitioning for Parallel Entity Matching. In *8th International Workshop on Quality in Databases*, 2010.

Kolb, L., Thor, A., and Rahm, E. Load balancing for mapreduce-based entity resolution. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*. ICDE '12. IEEE Computer Society, Washington, DC, USA, pp. 618–629, 2012a.

Kolb, L., Thor, A., and Rahm, E. Multi-pass sorted neighborhood blocking with mapreduce. *Comput. Sci.* 27 (1): 45–63, Feb., 2012b.

Kopcke, H. and Rahm, E. Frameworks for entity matching: A comparison. *Data Knowl. Eng.* 69 (2): 197–210, Feb., 2010.

Mestre, D. G. and Pires, C. E. Improving load balancing for mapreduce-based entity matching. In *Proceedings of the Eighteenth IEEE Symposium on Computers and Communications*. ISCC '13. IEEE Computer Society, 2013.

Okcan, A. and Riedewald, M. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. SIGMOD '11. ACM, New York, NY, USA, pp. 949–960, 2011.

Vernica, R., Carey, M. J., and Li, C. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. ACM, New York, NY, USA, pp. 495–506, 2010.

Wang, C., Wang, J., Lin, X., Wang, W., Wang, H., Li, H., Tian, W., Xu, J., and Li, R. Mapdupreducer: detecting near duplicates over massive datasets. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. SIGMOD '10. ACM, New York, NY, USA, pp. 1119–1122, 2010.