

Efficient Entity Matching over Multiple Data Sources with MapReduce

Demetrio Gomes Mestre, Carlos Eduardo Santos 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 tasks by reducing the amount of data generated 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, Load Balancing, 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 duplicate entities in a single data source (traditional) and the second one, focus of this work, refers to the special case of finding 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.

Copyright©2013 Permission to copy without fee all or part of the material printed in JIDM is granted provided that the copies are not made or distributed for commercial advantage, and that notice is given that copying is by permission of the Sociedade Brasileira de Computação.

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 according to a specific blocking key and restricting EM to entities of the same block. For instance, it is sufficient to compare entities of the same manufacturer (blocking key) 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. Furthermore, idle but instantiated nodes may produce unnecessary costs because public cloud environments (e.g., Amazon EC2) usually charge per utilized machine hours [Kolb et al. 2012a].

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. It splits larger blocks, according to the load balancing constraints, into smaller chunks based on the input partitions to enable their parallel matching. The approach also uses a heuristic to assign the entire blocks and the smaller chunks to the reduce tasks. 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 addresses 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** (**M**ultiple **S**ource **B**lock**S**licer), 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**'s 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.

This work is organized as follows. Section 2 introduces the MapReduce programming paradigm and describes how MapReduce-based Entity Matching can be processed. Section 3 discusses about related work. Section 4 describes a general MR-based Entity Matching workflow for load balancing, a preprocessing phase used to collect information about the datasets' characteristics and our approach for optimized block-based load balancing EM with MapReduce over multiple data sources, **MSBlockSlicer**. Section 5 presents the performed experiments and evaluation. Finally, Section 6 concludes the article and presents suggestions for future work.

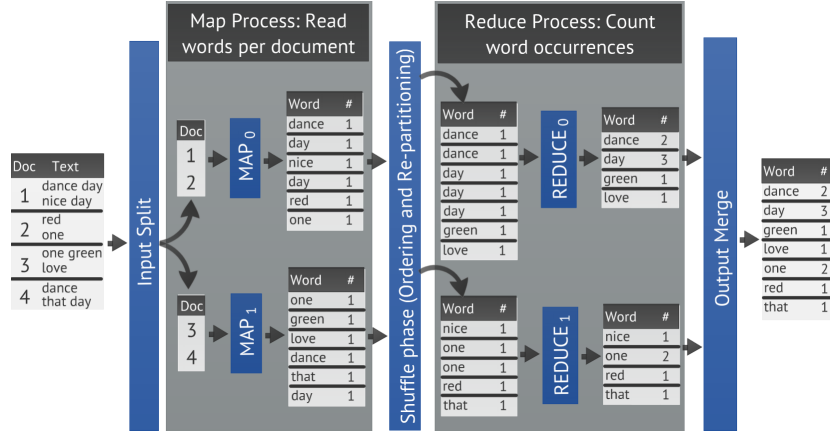


Fig. 1. A MapReduce program example for counting word occurrences in a set of documents (adapted from [Kolb et al. 2012b]).

2. MAPREDUCE AND ENTITY MATCHING

MapReduce is a programming model designed for parallel data-intensive computing in shared-nothing clusters with a large number of nodes [Dean and Ghemawat 2008]. The key idea relies on data partitioning and storage in a distributed file system (DFS). Entities are represented by (key, value) pairs. The computation is expressed with two user-defined functions:

$$\text{map} : (\text{key}_{in}, \text{value}_{in}) \rightarrow \text{list}(\text{key}_{tmp}, \text{value}_{tmp})$$

$$\text{reduce} : (\text{key}_{tmp}, \text{list}(\text{value}_{tmp})) \rightarrow \text{list}(\text{key}_{out}, \text{value}_{out})$$

Each of these functions can be executed in parallel on disjoint partitions of the input data. For each input key-value pair, the map function is called and outputs a temporary key-value pair that will be used in a shuffle phase to sort the pairs by their keys and send them to the reduce function. Unlike the map function, the reduce function is called every time a temporary key occurs as map output. However, within one reduce function only the corresponding values $\text{list}(\text{value}_{tmp})$ of a certain key_{tmp} can be accessed. A MR cluster consists of a set of nodes that run a fixed number of map and reduce jobs. For each MR job execution, the number of map tasks (m) and reduce tasks (r) is specified. The framework-specific scheduling mechanism ensures that after a task has finished, another task is automatically assigned to the released process.

For example, the data flow of a MapReduce computation is shown in Figure 1. The MapReduce process counts the number of term occurrences in a given input (set of documents), which is a common task in Information Retrieval. In the example of Figure 1, the input data is partitioned in two map tasks to show how the parallelism idea works. Although the input data has a tiny size, normally map tasks processes larger datasets. Figure 1 shows that the two instances of map functions read all words existing in the dataset and emits a list of (term, 1) pairs hash-partitioned on the key. Thereafter, during the intermediate phase between the map process and the reduce process, the (key, value) pairs are sorted by their key and then sent across the cluster in a shuffle phase, which means that each (key, value) pair will be accessed for the proper reduce task. In the example, all keys (words) starting with a letter from *a* to *m* are sent to the first reduce task and the rest of the keys are sent to the second reduce task. Thus, each reduce function instance accounts and outputs the number of occurrences per key (word).

Although there are several frameworks that implement the MapReduce programming model, in the scientific community, Hadoop is the most popular implementation of this paradigm [Dean and Ghe-

mawat 2008]. We are therefore implementing and evaluating our approach with Hadoop.

Parallel EM implementation using blocking approaches with MR can be done without major difficulties. In the **Basic** implementation [Kolb et al. 2012a], already mentioned in section 1, the map process defines the blocking key for each input entity and outputs a key-value pair (blockingKey, entity). Thereafter, the default hash partitioning in the shuffle phase can use the blocking key to assign the key-value pairs to the proper reduce task. The reduce process is responsible for performing the entity matching computation for each block. An evaluation of the **Basic** approach showed a poor performance due to the data skewness caused by varying size of blocks [Kolb et al. 2012a]. The data skewness problem occurs when the match work of large blocks of entities is assigned to a single reduce task. It can lead to situations in which the execution time may be dominated by a few reduce tasks and thus enable serious memory and load balancing problems when processing too large blocks. Therefore, concerns about lack of memory and load imbalances become necessary.

3. RELATED WORK

Entity Matching is a very studied research topic. Many approaches have been proposed and evaluated as described in recent surveys [Elmagarmid et al. 2007; 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.

In this context, when we deal with MapReduce-based large-scale Entity Matching, two well-known data management problems must be treated: load balancing and data skewness handling. MR has been criticized for having overlooked the skew issue [DeWitt and StoneBraker 2008]. Parallel hash join processing [Lin 2009], a skew handling mechanism available in parallel database systems, share many similarities with our problem. However, the real performance gain when dealing with MapReduce comes from the injection of skew handling into user-defined functions (map, reduce). Our work does not aim to prioritize skew handling, but minimize its effects on performance by efficiently distributing entity replication among all nodes.

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 for generality 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 addresses 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 a two-step 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

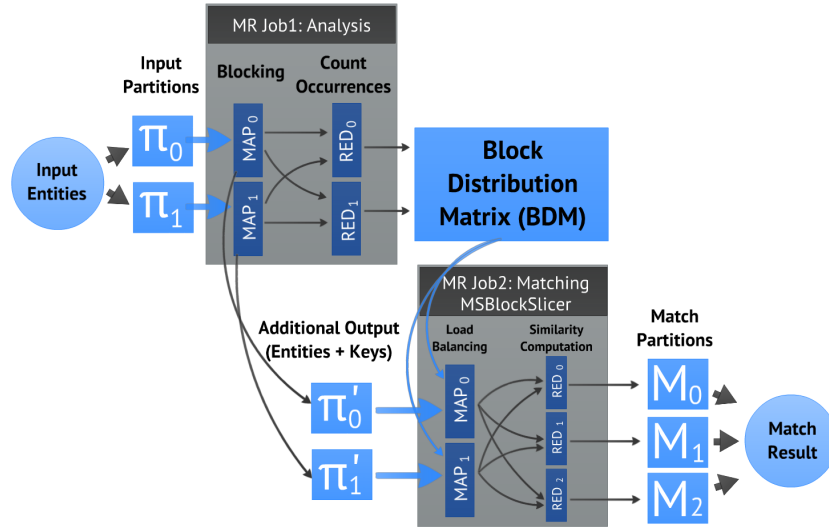


Fig. 2. Overview of the MR-based matching process with load balancing (adapted from [Kolb et al. 2012a]).

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** provides 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.

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

The goal of this work is to improve the load balancing of the blocking-based EM with MapReduce over multiple data sources. As mentioned in section 3, the state of the art approach (**BlockSplit**) is performed with two MR jobs (two-step). The first MR job collects and stores information about the entities distribution, blocks and input partitions in the Block Distribution Matrix (BDM). The second MR job performs the load balancing strategy in the map phase and the comparisons between the entities during the reduce phase. To achieve our goal, we made an improvement in the state of the art process aiming to enhance the EM performance in terms of execution time. Basically, the improvement consists in the replacement of the second MR job, where the load balancing strategy and the entities comparisons are processed, by a new approach that, besides carrying out the BDM reading only in the map phase, provides a better load balancing uniformity. The two MR jobs needed to perform our load balancing approach are illustrated in Figure 2.

The first job, presented in [Kolb et al. 2012a], generates the so-called Block Distribution Matrix (BDM) that specifies the number of entities belonging to each input partition. The BDM is an essential mechanism to support the load balancing strategy (in the second MR job) because it provides information about the entity distribution for parallel matching of blocks with different sizes. Note that an additional output is generated having the same entities of the original input partitions followed by their corresponding blocking key. Thus, it is ensured that the BDM is representative since the second job receives the same partitioning of the input data as the first job. This additional output, as well as the BDM, is stored in the distributed file system.

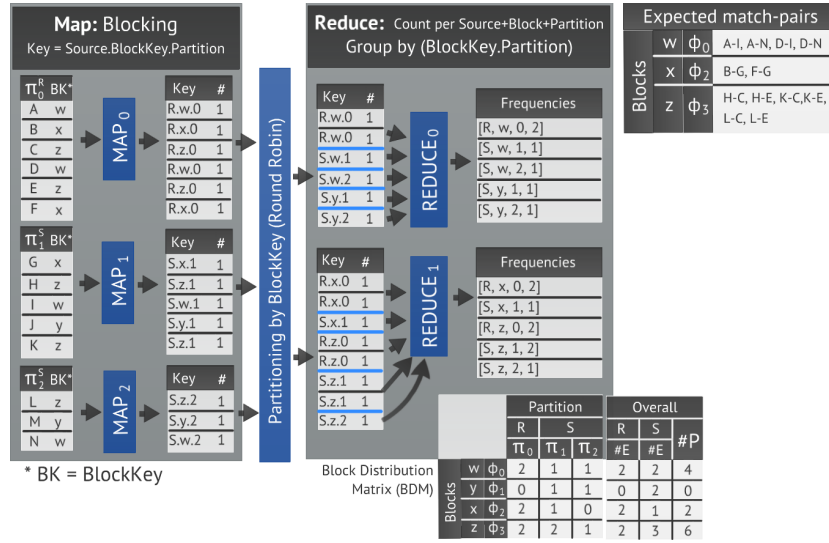


Fig. 3. Dataflow example for computing the Block Distribution Matrix.

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 involved data sources, this section describes an extension of **BlockSlicer** [Mestre and Pires 2013] for matching two sources R and S . To achieve effective performance optimization, **MSBlockSlicer** performs the load balancing in the map phase and performs the similarity computation in the reduce phase. Our approach follows the idea that map generates a systematic constructed composite key aiming to allow entity distribution in order to enable load balancing optimization in the matching process. This composite key combines information about the target reduce task(s), the entity blocking key and the entity itself. Since MR partitioning, in addition to routing map outputs, also groups together key-value pairs sharing the same blocking key (component of the composite key), it is ensured that only entities of the same block are compared within the reduce phase when the local EM is performed.

4.1 First Job: Analysis (Block Distribution Matrix)

The Block Distribution Matrix (BDM) is a simple preprocessing step to determine some datasets' characteristics. The BDM consists in a $b \times m$ matrix that specifies the number of entities of b blocks across m input partitions. Its computation using MR is illustrated in Figure 3 for an example dataset. As shown in the example, we utilize the entities $A-N$ and the blocking keys $w-z$. Each entity belongs to one of the two data 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 (3 in the example). Each entity has a blocking key ($w-z$) denoted as index. For example, the map output key of H is $S.z.1$ because H 's blocking key equals z and H appears in the second partition of source S (partition index=1). The entities sharing this key ($S.z.1$) are assigned to the last reduce task which in turn outputs $[S, z, 1, 2]$ because there are 2 entities in the second partition for blocking key z . The combined reduce outputs correspond to the BDM. To assign blocking keys to rows of the BDM, we use the order of the blocks from the reduce output, i.e., we assign the first block (key w) to block index position 0, and so on.

The block sizes ($\#E$) for each data source in the example vary from 1 (x from data source S) to 3 (z from data source S) entities. The match work ($\#P$) to compare all entities per block against each other thus ranges from 2 (x) to 6 (z) comparisons; the largest block with key z entails 50% (6

of 12) of all comparisons although it contains only 30% (5 of 14) of all entities. The match work of any b block ($\#P_b$) is calculated as follows: $\#P_b = |\Phi_b^R| \times |\Phi_b^S|$, where $|\Phi_b^R|$ is the number of entities of block b belonging to the data source R and $|\Phi_b^S|$ is the number of entities of block b belonging to the data source S . Thus, as the z block contains 5 entities, 2 from the data source R and 3 from the data source S , the $\#P_z$ (2×3) is equal to 6 comparisons. The expected match-pairs for this running example is shown in the upper part of Figure 3.

As illustrated in Figure 2, the map processes produce an additional output Π_0^i per input partition that contains the original entities annotated with their blocking keys. This output is shown in Figure 2 as the input of the second MR job.

4.2 Second Job: Matching (MSBlockSlicer)

The block-based approach **MSBlockSlicer** is responsible for creating match tasks per block and distributing these match tasks among the reduce tasks effectively. To be more specific, our approach uses the following key ideas:

- Smaller blocks are processed within a single match task. Larger blocks are sliced according to a cutting strategy aiming at performing the Cartesian product of such blocks in parallel and ensuring that all comparisons of the original block will be computed;
- MSBlockSlicer** determines the number of comparisons per matching task and assigns match tasks to the reduce tasks using a greedy algorithm for optimization purposes. Thus, a load balancing optimization is ensured.

Our approach makes use of the BDM as well as the composite map output keys. Each map function generates a well-defined composite key that (together with the associated partition) allows an optimized load balancing distribution. The composite key thereby combines information about the target reduce task(s) and the entity block. If this entity is supposed to be processed by multiple reduce tasks, the map function generates multiple keys per entity and the information about the entity itself is added to its key. The reduce phase performs the EM by computing match similarities between entities of the same block. Since the reduce phase consumes the vast majority of the overall runtime (more than 90%, according to our experiments), our load balancing approach solely focuses on the data redistribution for reduce tasks.

MSBlockSlicer's mappers output key-value pairs with key = ($reduceIndex \odot blockIndex$) and value = (entity). If one entity has to be processed by multiple reduce tasks, as mentioned before, its key is composed by $reduceIndex \odot blockIndex \odot slice$. The reduce task index has a value between 0 and $r - 1$, where r is the number of reduce tasks available, and is used by the MR function part to perform assignments to the reduce tasks. Since the block index is part of the key and the MR function *group* takes the entire key into account, it is ensured that each reduce function only receives entities of the same block. The slice value indicates what kind of match task has to be performed by the reduce function, i.e., whether a complete block or sliced-blocks need to be processed.

At the beginning of the execution, the map phase reads the BDM and computes the number of comparisons per block as well as the number of comparisons T over all b blocks Φ_k :

$$T = \sum_{k=0}^{b-1} |\Phi_k^R| \times |\Phi_k^S|. \quad (1)$$

For each block Φ_k , the map phase also checks if the number of comparisons is above the average reduce task workload, i.e., if

$$|\Phi_k^R| \times |\Phi_k^S| > T/r. \quad (2)$$

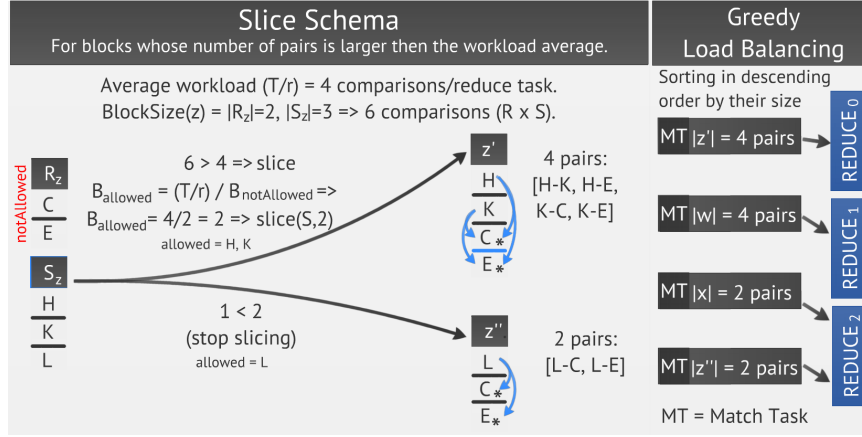


Fig. 4. Slice schema illustration.

If $|\Phi_k^R| \times |\Phi_k^S|$ is not above the average workload Φ_k can be processed within a single match task (as mentioned before, the map output key only contains information about the reduce index and the block index k). Otherwise, the block is sliced into n sliced-blocks.

Each sliced-block is generated according to a special cutting strategy based on the average reduce task workload. The basic idea is defined as follows. 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 entities $b_{notAllowed}$ belonging to the other source. $b_{allowed}$ is extracted from the source (R or S) that contains the largest number of entities from b . Thereafter, the entities belonging to $b_{allowed}$ will no longer iterate comparisons among themselves, but 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 all the entities belonging to $b_{notAllowed}$ are emitted for each new sliced-block generated. To differentiate entities belonging to $b_{allowed}$ from $b_{notAllowed}$ sliced-blocks, all entities belonging to $b_{notAllowed}$ are marked with an "*" in the map output value. The exact slicing is calculated as:

$$|b_{allowed}| = \frac{\lceil T/r \rceil}{|b_{notAllowed}|}. \quad (3)$$

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).

For illustration purposes, according to the workflow example of Figure 3, the BDM indicates 12 overall pairs. The largest block Φ_3 (z) has 5 entities, 2 entities belonging to source R and 3 entities belonging to source S . The number of comparisons generated by this block is 6 (2×3) and the average workload is equal to 4 as shown in Figure 4. Note that, in the example, the number of comparisons generated by block z is above the average workload ($6 > 4$) and thus z is sliced and transformed into two sliced-blocks: z' and z'' . The sliced-block z' is generated by keeping the two allowed entities calculated in the first cut (H and K) and all entities from $b_{notAllowed}$ (C and E) marked with an "*". Thereafter, the sliced-block z'' is generated by keeping the remaining allowed entities of the first cut (L) and all entities replicated from $b_{notAllowed}$ (C and E) also marked with an "*". Figure 4 shows that $b_{allowed}$ value is 2, in this case, only the first 2 entities (F and G) are allowed to perform comparisons in this block with all $b_{notAllowed}$ entities. The second sliced-block generated (z'') is subjected to a new average workload constraint verification and since its T ($T = 2$) is not above the average workload ($2 < 4$) there is no need to slice z'' .

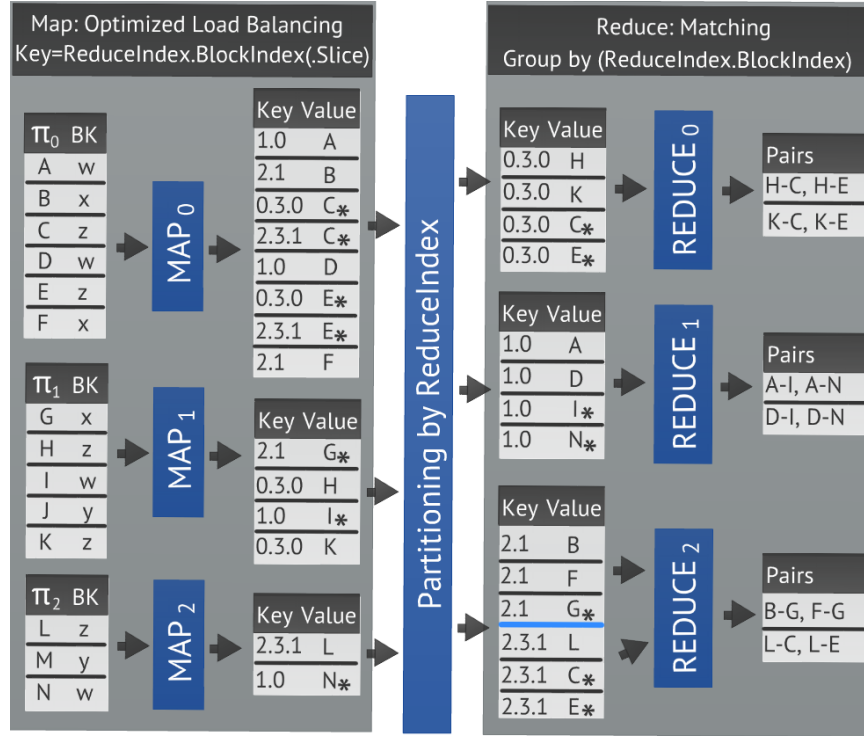


Fig. 5. Example of a MSBlockSlicer dataflow for 2 sources.

Thus, the slice phase results in two new sliced-blocks: the match tasks 3.0 (z') and 3.1 (z''). Since all match tasks are defined both for non-sliced blocks and sliced-blocks, we apply a greedy algorithm to optimally determine the reduce task for each match task, as shown in Figure 4. Firstly, all match tasks are sorted in descending order of their number of pairs (comparisons): 3.0 (4 pairs, $reduce_0$), 0 (4 pairs, $reduce_1$), 2 (2 pairs, $reduce_2$) and 3.1 (2 pairs, $reduce_2$). Thereafter, match tasks are assigned to reduce tasks in this order so that the current match task is assigned to the reduce task with the lowest number of already assigned pairs. In the following, we denote the reduce task index for match task $k.x$ as $R(k.x)$.

The dataflow is shown in Figure 5. 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 5 shows that H is an allowed entity from match task 3.0 and is only compared with the not allowed entities (marked with an "*"), i.e., the entities C and E .

5. EVALUATION

In the following, we evaluate² **MSBlockSlicer** against **BlockSplit**'s extension approaches. The latter was implemented according to the pseudo-code available in [Kolb et al. 2012a], regarding a 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.

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

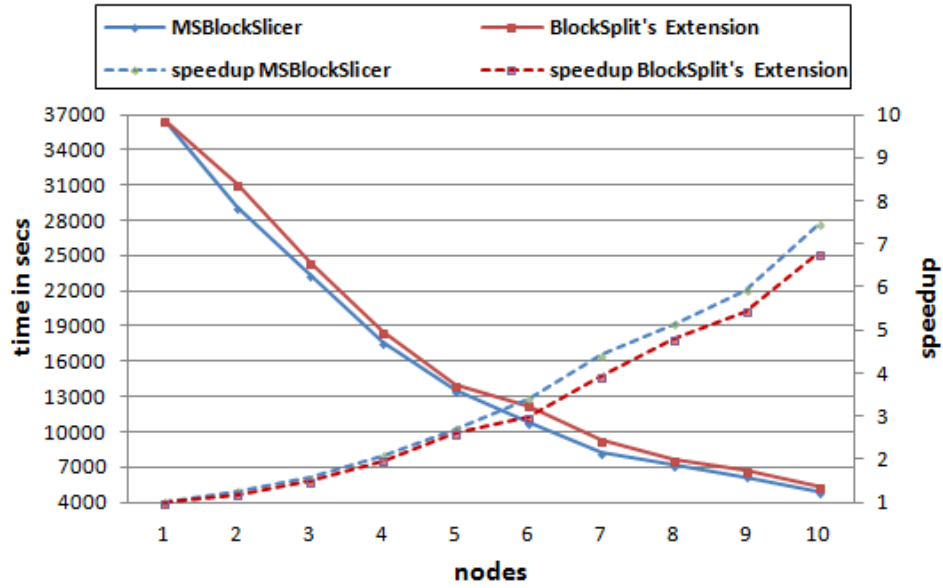


Fig. 6. Execution times and speedup for both approaches using DS1.

We ran our experiments on a 10-node HP Pavilion P7-1130 cluster. Each node had one Intel I5 processor with four cores, 4GB of RAM, and one 1TB of hard disk. Thus the cluster consists of 40 cores and 10 disks. On each node, it was installed the Windows 7, 64-bit, JAVA 1.6, cygwin and Hadoop 0.20.2. In order to maximize the parallelism and minimize the running time, we made the following changes to the default Hadoop configuration: we set the block size of the distributed file system to 128MB and disable the speculative task execution feature. Each node runs at most two map and reduce tasks in parallel.

We 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.

5.1 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 time values are shown in Figure 6 and the respective number of generated key-value pairs are illustrated in Figure 7.

We can note that both **MSBlockSlicer** and **BlockSplit's** extension approaches scale almost linearly showing their ability to evenly distribute the workload across reduce tasks and nodes. However, by design, **BlockSplit's** extension approach hugely depends on the input (map) partitions and thus its ability to split large blocks can be affected if most of the entities belonging to the same block are present in only one input partition. This is what mainly happened in Figure 6 when the experiment varied from 7 to 8 nodes. Sixteen map tasks were needed to split a large block significantly

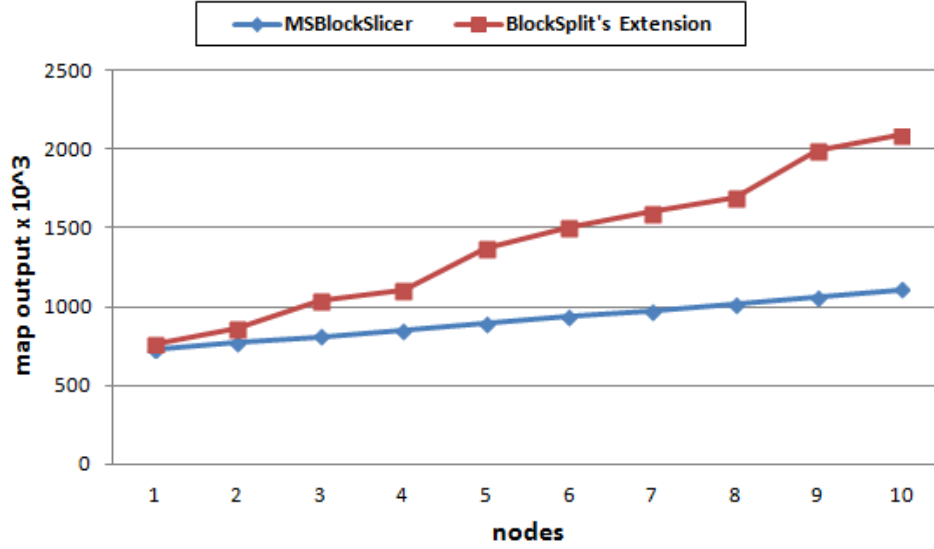


Fig. 7. Number of generated key-value pairs by map for DS1 varying the number of nodes n ($m=2 \cdot n$, $r=4 \cdot n$).

and consequently improve the load balancing. Thus, **BlockSplit**'s execution time is compromised by the deficiency of properly split larger blocks with few input partitions. 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 (**MSBlockSlicer** has almost one extra node working).

Furthermore, Figure 7 shows, by the map output generation, an overgrowth of output entities during the **BlockSplit**'s map phase. This map output study is important because too many emitted key-value pairs lead the MapReduce process to an associated overhead, i.e., extra network resources consumption due to additional data transfer, deterioration of the reduce tasks execution by sorting large data streams and high OS memory management. Such overhead does not significantly impact the execution times up to a moderate size of the utilized cloud infrastructure due to the fact that the matching process in the reduce phase is by far the dominant factor of the overall execution time. However, when dealing with really huge volumes of data, the cloud infrastructure can suffer to manage the memory used and the data transfer, and thus deteriorate 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 the **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.

6. 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 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 *IWeb*. pp. 73–78, 2003.
- DEAN, J. AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Commun. ACM* 51 (1): 107–113, Jan., 2008.
- DEWITT, D. AND STONEBRAKER, M. Mapreduce: A major step backwards, http://www.cs.washington.edu/homes/billhowe/mapreduce_a_major_step_backwards.html, 2008. [Online; accessed 10-June-2012].
- ELMAGARMID, A. K., IPEIROTIS, P. G., AND VERYKIOS, V. S. Duplicate record detection: A survey. *IEEE Trans. on Knowl. and Data Eng.* 19 (1): 1–16, Jan., 2007.
- 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. *CoRR* vol. abs/1006.5309, 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.
- LIN, J. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. In *7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, 2009.
- 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. Mapduplicator: 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.