The Journal of Systems and Software 128 (2017) 1–10

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

An efficient spark-based adaptive windowing for entity matching

Demetrio Gomes Mestre[a,b,*], Carlos Eduardo Santos Pires[a], Dimas Cassimiro Nascimento[a,c], Andreza Raquel Monteiro de Queiroz[a], Veruska Borges Santos[a], Tiago Brasileiro Araujo[a]

[a] Data Quality Research Group - Federal University of Campina Grande (UFCG), Aprigio Veloso Street, 882, Campina Grande, Paraiba, Brazil
[b] Coordination of Information Technology and Communication (CTIC) - State University of Paraiba (UEPB), Baraunas Street, 351, Campina Grande, Paraiba, Brazil
[c] Federal Rural University of Pernambuco (UFRPE), Garanhuns, Pernambuco, Brazil

ARTICLE INFO

Article history:
Received 16 August 2016
Revised 28 February 2017
Accepted 3 March 2017
Available online 6 March 2017

Keywords:
Adaptive windowing
Entity matching
Load balancing
Spark

ABSTRACT

Entity Matching (EM), i.e., the task of identifying records that refer to the same entity, is a fundamental problem in every information integration and data cleansing system, e.g., to find similar product descriptions in databases. The EM task is known to be challenging when the datasets involved in the matching process have a high volume due to its pair-wise nature. For this reason, studies about challenges and possible solutions of how EM can benefit from modern parallel computing programming models, such as Apache Spark (Spark), have become an important demand nowadays (Christen, 2012a; Kolb et al., 2012b). The effectiveness and scalability of Spark-based implementations for EM depend on how well the workload distribution is balanced among all workers. In this article, we investigate how Spark can be used to perform efficiently (load balanced) parallel EM using a variation of the Sorted Neighborhood Method (SNM) that uses a varying (adaptive) window size. We propose Spark Duplicate Count Strategy (S-DCS++), a Spark-based approach for adaptive SNM, aiming to increase even more the performance of this method. The evaluation results, based on real-world datasets and cluster infrastructure, show that our approach increases the performance of parallel DCS++ regarding the EM execution time.

1. Introduction

In the past decade, the open source parallel programming models for data-intensive tasks emerged as a new paradigm to run jobs which previously used to be complex and costly to run. These new programming models, such as MapReduce (MR) (Dean and Ghemawat, 2008) and Spark (Zaharia et al., 2012), can run on commodity hardware and provide high scalability, fault tolerance and flexibility to distributed systems. Furthermore, they also provide simple development APIs and hide the complexity of parallel tasks execution and fault-tolerance from the developer user. Such programming models are extensively employed to compose services based on cloud computing (Nascimento et al., 2015).

MapReduce is one of the earliest popular commodity cluster frameworks. Due to its simplicity, which is attractive for developers, MapReduce has become popular quickly. However, the MR framework has several limitations (Shi et al., 2015). Besides the fact that every MR job has to read its input data from the disk, to process it, and then to write it back to the HDFS (disk), iterative applications, such as machine learning and graph analytics, which demand multiple rounds of execution are performed on the same data, thus leading to additional overheads. To overcome these MR limitations, Spark is boosted by the Resilient Distributed Datasets (RDDs) (Zaharia et al., 2012), which provide in-memory (main memory) data structures used to cache intermediate data across subsequent tasks performed by the nodes. Since RDDs can be kept in memory, programs can iterate over RDDs data multiple times efficiently.

In this article, we investigate the Spark-based parallelization of data processing for the problem of Entity Matching (EM) (also known as entity resolution, deduplication, record linkage, or reference reconciliation), i.e., the task of identifying entities referring to the same real-world object (Christen, 2012a). Given the pair-wise comparison nature of the problem, EM is a data-intensive and performance critical task that demands studies on how it can benefit from the Spark framework and its efficient RDDs. This task is a fundamental problem in every information integration (Wang et al., 2013) and data cleansing system (Kolb et al., 2012a), e.g., to find similar product descriptions in databases. It is also essential for other types of systems, such as web pages deduplica-

* Corresponding author.
E-mail addresses: demetriogm@gmail.com (D.G. Mestre), cesp@dsc.ufcg.edu.br (C.E.S. Pires), dimascnf@gmail.com (D.C. Nascimento), andreza.queiroz@ccc.ufcg.edu.br (A.R.M. de Queiroz), veruska.santos@ccc.ufcg.edu.br (V.B. Santos), tiagobrasileiro@copin.ufcg.edu.br (T.B. Araujo).

http://dx.doi.org/10.1016/j.jss.2017.03.003
0164-1212/

tion (Christen, 2012a), plagiarism detection (Cosma and Joy, 2012) and click fraud detection (Metwally and Faloutsos, 2012).

Unfortunately, the expensive detailed comparison of field (attribute) values between entities and the naïve approach of comparing all pairs of entities leads to a performance bottleneck when applying EM on large datasets (Christen, 2012b). In practice, the EM execution is highly costly when the datasets are too large. For example, the matching of two datasets, each one containing one million entities, would result in about one trillion of possible entity pair comparisons. To address this bottleneck, a significant number of indexing techniques have been proposed aiming to minimize the amount of potential comparisons by restricting the entities search space to the entities that present high potential of being considered as matches (Christen, 2012b). In other words, these techniques work by partitioning the input data into groups (blocks) of similar entities and restricting the EM process to entities that belong to the same group. For example, it is sufficient to compare entities of the same manufacturer when matching product offers.

In the last years, robust and modern indexing techniques capable of decreasing even more the number of pair comparisons, such as *DCS++* (i.e., Duplicate Count Strategy) (Draisbach et al., 2012), have been proposed. *DCS++* is based on the well-known Sorted Neighborhood Method (SNM)[1] (Hernández and Stolfo, 1995). It emerged as a response to the major difficulty regarding the appliance of SNM: the definition of the fixed window size (if it is selected too small, some duplicates might be missed, whereas if it is selected too large, it leads to unnecessary comparisons). To overcome this difficulty, *DCS++* adapts the window size according to the number of duplicate entities already identified. The intuition behind this idea is that the more duplicates of an entity are found within a window, the larger becomes the window size. On the other hand, if no duplicate of an entity is found within its neighborhood, then *DCS++* assumes that there are no duplicates or the duplicates are very far away in the sorting order of entities, and thus, the window size is not changed.

Even using modern blocking techniques, the EM task is still challenging in terms of execution time due to the amount of data utilized by the academy and industry nowadays. For this reason, some works Kolb et al. (2012b), Kolb et al. (2012a), Mestre and Pires (2013), Hsueh et al. (2014) and Mestre et al. (2015) have proposed the parallelization of blocking techniques aiming to explore the advantages of MapReduce-based parallel blocking in the context of EM applied to Big Data scenarios. Some of these approaches also proposed mechanisms to perform the parallel blocking efficiently, treating issues like the load balancing and distributed resources consumption (Kolb et al., 2012b; Mestre and Pires, 2013). However, to the best of our knowledge, the proposition of a model combining *DCS++* and its efficient parallelization using Spark, by taking into account load balancing and distributed resources consumption issues, is still an open gap. To address this gap, we make the following contributions:

- We propose the **S**park-based **D**uplicate **C**ount **S**trategy (*S-DCS++*), a Spark-based approach that provides an efficient parallelization of the *DCS++* technique (Draisbach et al., 2012) by using multiple Spark iterations and applying a customized data replication during data redistribution to allow the resizing of the adaptive window. The proposed approach also addresses the data skew problem[2] with an automatic data partitioning strategy that provides a satisfactory load balancing across all available nodes;
- We evaluate the *S-DCS++* against the Spark-based SNM (Meng et al., 2016) and the MR-based DCS++ (Mestre et al., 2015) and show that the first provides a better performance by diminishing the overall EM execution time. The evaluation employs a real cluster environment and uses real-world data.

This article is structured as follows. Section 2 introduces the EM performance problem and explains how it can be treated with the Spark programming paradigm. Reference 3 discusses related work. Section 4 describes our Spark-based approach for EM adaptive windowing. Section 5 presents the performed experiments and evaluation. Finally, Section 6 concludes the article and provides suggestions for future work.

2. Background

In this work, we consider the problem of EM within one data source. Given a relation E, we denote the schema of the relation as $R_E = (a_1, a_2, \ldots, a_n)$. Each $a_i \in R_E$ corresponds to an attribute. An entity stored in a relation E assigns a value to each attribute. This means that the input data source S contains a finite set of entities $e = [(a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)]$. The task is to identify all pairs of entities $M = \{(e_i, e_k) \mid e_i, e_k \in S \wedge i \neq k\}$ that are regarded as duplicated.

Furthermore, we focus on the following EM challenge: how to minimize the execution time necessary to identify all potential duplicated entities in a given dataset. Thinking this way, our *optimization goal* is: for a given similar identifier operator (matcher) and its inputs, we want to minimize the matching completion time and maintain the detection efficacy considering a given amount of available processing nodes. As process completion time, we mean the time spent by all MapReduce or Spark processes involved in the EM task. In the remainder of the section, we introduce the Spark paradigm and explain how EM can be combined with Spark.

2.1. Adaptive windowing

The EM indexing technique tackled in this work is the *DCS++* (Draisbach et al., 2012), a SNM variant that adapts the window size according to the number of duplicate entities already identified. Let w be the window size, d be the number of detected duplicates within a window, c be the number of comparisons and ϕ_{ddr} be a threshold of duplicate detection rate with $0 < \phi_{ddr} \leq 1$. For each detected duplicate, *DCS++* adds the next $w-1$ adjacent entities of that duplicate to the window while $\frac{d}{c} < \phi_{ddr}$. Entities are added only once to that window and the window is no longer increased when $\frac{d}{c} < \phi_{ddr} \leq \frac{1}{w-1}$. *DCS++* calculates the transitive closure (i.e., entities are regarded as matches if they match another entity directly or through some sequence of intermediate matching entities) to save some of the comparisons: let us assume that the pairs (t_i, t_k) and (t_i, t_l) are duplicates, with $i < k < l$. The calculation of the transitive closure returns the additional duplicate pair (t_k, t_l). Hence, it is not necessary to verify the window $W(k, k+w-1)$, and thus, this window can be skipped. The key idea used in *DCS++* to save unnecessary comparisons is to skip windows (comparisons) by transitive closure. Besides, every time the window slides to the next entity, the size of the window is set to the initial value. An experimental evaluation showing the performance advantages of *DCS++* compared to the traditional *SNM* and *DCS* methods is shown in Draisbach et al. (2012).

Fig. 1 shows an execution example of the Adaptive Windowing (*DCS++*) employing a window size $w_{initial} = 3$. Initially, the window includes the first three entities (*A, D, B*) and generates two pairs of comparisons: $[(A-D), (A-B)]$. Since $A-D$ and $A-B$ are

[1] The SNM sorts all entities using an appropriate blocking key, slides a predefined (and fixed) distance window over the entities and only compares entities within the window.

[2] In this work context, data skewness is related to the window size in the sense that large window sizes lead to a high number of entity comparisons.

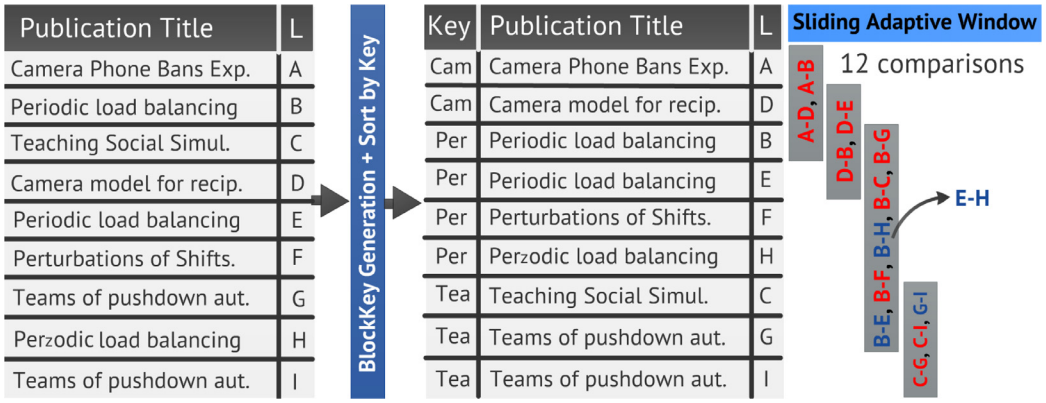


Fig. 1. Execution example of the *DCS++* method with an initial adaptive window size $w_{initial} = 3$.

regarded as non-match and no duplicate entities are identified, there is no need to increase the window size. After that, the window slides to the next entity (B). From B, the next comparison $B - E$ is regarded as a match. Thus, the following relation is tested: if $\frac{d}{c} \geq \frac{1}{w-1}$, such that d is the number of already detected duplicates within the window and c is the number of comparisons already done within the window. Then, w is increased by $w_{initial} - 1$ adjacent entities of that duplicate (E). Since $\frac{1}{1} \geq \frac{1}{3-1}$, then w is increased by two adjacent entities of E. Now, w covers B, E, F and H; the new comparisons generated are $B - F$ and $B - H$. Since $B - F$ is regarded as a non-match, the w increasing test is not satisfied. Then, the comparison $B - H$ is performed.

Since $B - H$ is regarded as a match, the w increasing test is satisfied once more resulting in true ($\frac{2}{3} \geq 0.5$) and the window is increased by two again. This time, the window is increased from H, which is the last duplicate found. Now, w covers B, E, F, H, C and G; B is compared with the remaining entities within the window. The window is no longer increased due to the lack of new matches. Note that the pair $E - H$ is regarded as a match due to the transitive closure assumption. After that, w is set to the initial value (3) and the window slides to the next entity (C). The windows starting with the entities E and F were also skipped due to the transitive closure assumption. Therefore, the number of comparisons performed is 12.

2.2. Spark and adaptive windowing

Apache Spark is a fast and general-purpose cluster computing engine with APIs in Scala, Java and Python as well as libraries for streaming, graph processing and machine learning (Zaharia et al., 2012). Unlike the two-stage MapReduce (Dean and Ghemawat, 2008) topology, it has an advanced DAG (Directed Acyclic Graph) execution engine that supports cyclic data flow. Spark presents a programming model and offers a functional API based on Resilient Distributed Datasets (RDDs) (Zaharia et al., 2012). The RDD abstraction provides transformations (e.g., map, flatmap, reduce, reduceByKey, filter, group-by and join) and actions (e.g., count and collect) that operate on datasets partitioned over a cluster of nodes. A conventional Spark program performs a sequence of transformations that ends with an action and returns a result value (e.g., a collected list of similar entities referenced by the RDD) to the Spark "driver" program, which can then start another sequence of RDD transformations.

The Spark "driver" program runs as a central coordinator and operates on RDDs through references. A driver program can be a stand-alone Scala program or a user operation through the Spark terminal. In both cases, RDD references lazily evaluate transformations by returning a new RDD reference that is specific to the transformation operation on the partitioned input RDD(s). Partitions of a dataset are materialized on demand when they are used in a parallel operation (e.g., by passing the block of a file through a flatmap function), and are discarded from memory after use. Actions start the evaluation of an RDD reference and all RDD transformations leading up to it. By employing this strategy, Spark optimizes the engine for low latency.

A parallel EM implementation using windowing approaches with Spark can be designed as follows. In a simple way, the Sorted Neighborhood Method (SNM) can be implemented by using the SlidingRDD model from MLlib[3] (Meng et al., 2016), i.e., the main Apache Spark's scalable machine learning library. The Spark-based SNM implementation, named S-SNM, utilizes the SlidingRDD model to perform a Spark-based fixed window over a partition of entities sorted according to a specific blocking key. As mentioned in Section I, the major difficulty when using the S-SNM is to define the fixed window size. *DCS++* addresses the problem of the fixed window sizes by adapting the window size according to the amount of already identified duplicates. However, it introduces a new problem to parallel computing. It generates different window sizes which leads to data skewness caused by windows of different sizes. The main problem is that this data skewness can lead to situations in which the execution time may be dominated by few workers (generating load unbalances), and thus, situations in which serious bottlenecks can emerge.

3. Related work

Entity Matching (EM) is an important research topic (Christen, 2012a) and has been investigated over forty years. Many EM approaches have been proposed and evaluated as described in a recent survey (Christen, 2012b). As modern databases are becoming larger, detecting duplicates or matching them require increasingly massive amounts of computing power and storage resources. Researchers have begun to investigate how modern parallel and distributed computing environments can be employed to reduce the time required to conduct large-scale entity matching projects (Kirsten et al., 2010). A considerable number of approaches that consider parallel EM have been proposed in recent years (Christen, 2012a), but there is still much to be done in the era of Big Data.

Since MR became a popular commodity cluster framework quickly, a high number of contributions were made addressing two well-known distributed data management problems: load balancing and data skew handling. The authors of Kirsten et al. (2010), Kolb et al. (2012a), Mestre and Pires (2013), Yan et al. (2013) and Hsueh et al. (2014) propose models to parallel entity matching

[3] http://spark.apache.org/mllib/

taking memory and load balancing requirements into account. Nevertheless, the usage of Hadoop MapReduce as a platform for EM has some drawbacks (Shi et al., 2015). Hadoop MapReduce is not designed for an interactive usage. It is made for Big Data batch processing because it lacks the ability to cache intermediate results (in main memory) to be used for multiple alternatives. For these reasons, it is common to find models of MR-based EM approaches using more than one MapReduce job (Kolb et al., 2012a).

In a previous work (Mestre et al., 2015), it was proposed an approach for the MapReduce-based Duplicate Count Strategy. The solution provides an efficient parallelization of the *DCS++* method (Draisbach et al., 2012) by using multiple MR jobs. However, the solution presented in Mestre et al. (2015) neither considered RDDs nor broadcast variables,[4] since MR does not provide such features. Their usage can decrease the number of jobs employed by MR approaches. We compared the MR-based *DCS++* (*MR-DCS++*) approach (Mestre et al., 2015) (state of the art) with our work in an experimental evaluation (Section V).

Apache Spark emerged as an alternative to provide features like interactive usage and in-memory (main memory) data structures and broadcast variables. However, to the best of our knowledge, no work involving Spark-based adaptive windowing technique for EM has been proposed.

4. Spark-based adaptive windowing

An interesting line of reasoning when we deal with Spark-based EM is to define an efficient approach (with load balancing handling) by knowing previously the entity comparisons generated by the serialized blocking (windowing) method. However, how do we define an efficient Spark-based approach when the blocking (windowing) method adapts according to the duplicate detection rate (like the *DCS++* strategy)? How do we assign entity comparisons to the proper workers with load balancing handling without previously knowing all the necessary entity comparisons? To answer these research questions, we propose our Spark-based *DCS++* for EM processing using multiple transformation steps. Each step is detailed with examples in the following sections.

4.1. Spark-based DCS++ (S-DCS++)

To provide an efficient workload distribution among the workers and also enable the *DCS++* method to be efficiently executed in the Spark workflow, we perform the adaptive windowing EM processing employing three transformation steps as illustrated in Fig. 2. The key idea is to improve the load balancing by fixing the same number of entities at each worker input partition. By doing this, the number of entities processed by each worker will be the same, leading the worker to perform a maximum number of window's slides (due to the fixed partition size). This mechanism improves the load balancing because the definition of a maximum number of window's slides avoids the overload of a single worker.

Thereby, the first step of the approach is a Spark mapper that receives as input from each worker a partition of the dataset. As mentioned in Section 2.1, *DCS++* assumes an ordered list of entities based on their blocking keys. Since the input dataset consists in an unordered entity collection, the mapper step determines the blocking key of each entity and outputs a key-value RDD pair containing an output key = (blocking key) and value = (entity) to enable the combination and sorting of entities by their key. During this mapper step, the key-value RDD pairs are partitioned based on the average (a) number of entities per available reducers (in the second step), $a = \frac{\Sigma|\Pi|}{r}$, such that Π represents the set of partitions of the data set and r corresponds to the number of reducers (workers). The aim of this step is to avoid the data skewness problem by allocating the same number of entities to each worker.

In the second step, the key-value RDD pairs of each reducer input partition are sorted by the key (according to *DCS++*) and the reducer performs the adaptive sliding window over the sorted partition. The algorithm emits a partial result of the EM processing, saves the transitive closure information in a broadcast variable (that will be accessed by workers during the third step) and stores as a mapper output the entities of the partition that could not be processed during the sliding window due to the overlapping assumptions. By overlapping assumptions, we mean the cases in which the window may need to slide or to be increased (when the window achieves the end of the current partition) in order to cover entities belonging to another reducer input partition. In this case, the EM processing stops and the entities belonging to the last window are stored in the memory as a mapper output to be recovered by the third step. From the subsequent reducer input partition, only the $w_{initial}$ entities are stored as a mapper output. In other words, one of the reducers of the third step will process the rest of the incomplete window processing attached to the $w_{initial}$ entities from the subsequent reducer input partition. This strategy ensures that, if a window needs to increase even more due to a new match found, the transitive closure information generated by the sliding window of the subsequent reducer input partition will cover the new window increased due to this new match.

In the running example shown in Fig. 3, the mapper reads the entities and generates (to each entity) a key-value pair with a map output key = (blocking key) and value = (entity) during the first step. For instance, the map output key of *E* is *cam* since *E*'s blocking key is *cam*. The first key (of *E*) is assigned to the second reducer because the amount of entities assigned to the first reducer reached the average number of entities per reducer ($a = \frac{4+4+4}{3} = 4$). Once all entities are positioned, the reduce phase starts sliding the adaptive window. In the example, we use $\Phi_{max} = 0.9$ (the similarity threshold that indicates if a pair of entities matches) and $w = 3$, which implies that, according to the *DCS++* strategy, the increasing condition threshold due to the duplicate detection rate is $\Phi_{ddr} = \frac{1}{w-1} = \frac{1}{3-1} = 0.5$.

Thus, in the reducer 0, the first generated window covers the entities *A, B* and *C*. This results in the following comparisons: $A - B$ and $A - C$. $A - B$ and $A - C$ are regarded as non-matches and since no duplicate entities are identified, there is no need to increase the window size. Therefore, the window slides to the next entity (*B*). From *B*, the next comparison $B - C$ is regarded as a match (framed). Thus, the following relation is tested: if $\frac{d}{c} \geq \frac{1}{w-1}$, where d is the number of already detected duplicates within the window and c is the number of comparisons already done also within the window, then w is increased by $w_{initial} - 1$ adjacent entities of that duplicate (*C*). Since $\frac{1}{1} \geq \frac{1}{3-1}$, w is increased by two adjacent entities of *C*. However, since the reduce partition does not have sufficient entities to cover an increasing of size 2, the execution of the EM process stops, the entity *C* is marked with an "*" and all the entities of the window will be processed in the third step.

In the third step, the reducer 0 reads the last window performed by the reducer 0 of the second step (*B, C* and *D*) and increases the reduce partition with the $w_{initial} - 1$ first entities (*E* and *F*) of the subsequent reduce partition (reducer 1 of the second step). Since the window execution during the second step ended in *C* (marked with an "*"), in the third step, the execution starts from *D* (the next entity). Now, w covers entities *B, C, D* and *E*; the new comparisons generated are $B - D$ and $B - E$. Since $B - D$ is regarded as a match, the w increase test is loaded once

[4] Broadcast variables allow the program execution to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks.

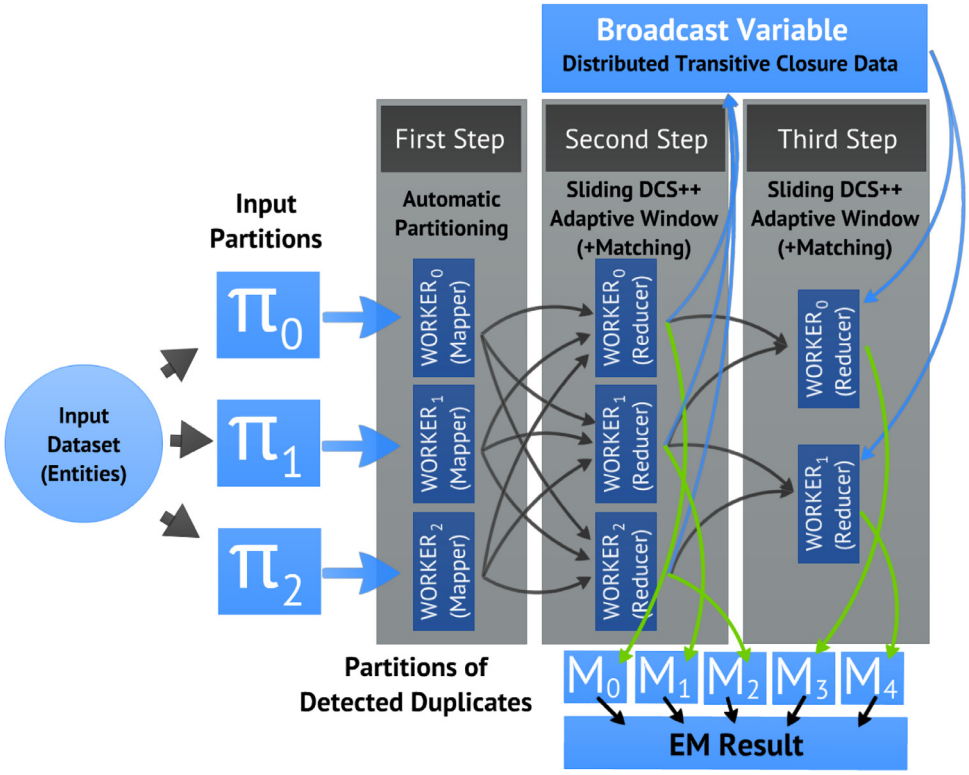


Fig. 2. Overview of the *S-DCS++* matching process workflow.

more resulting in true ($\frac{2}{2} \geq 0.5$) and the window is increased by two again. In turn, w covers entities *B, C, D, E* and *F*; the new generated comparison is $B - F$. Since $B - E$ and $B - F$ are regarded as non-matches and no duplicate entities are identified, there is no need to increase the window. However, if $B - E$ or $B - F$ were regarded as matches, the transitive closure broadcast variable would indicate if the entities *E* and *F* match another entities during the second step (processed by worker 1).

5. Evaluation

In the following, we evaluate the *S-DCS++* against *MR-DCS++* and *S-SNM* (the Spark-based SNM that uses the slidingRDD from the library MLlib) approaches, regarding three performance critical factors: degree of skewness, the number of nodes available (n) in the cluster environment and the tradeoff between the matching quality and execution time. In the experiment, we evaluate the algorithms aiming to investigate their robustness against data skew, how they can scale with the number of nodes available and their robustness in maintaining the EM quality while their execution time decreases.

We ran our experiments on a 20-node Intel(R) Xeon(R) cluster. Each node has one Intel Xeon processor 4 × 1.0GHz CPU, 4GB memory and runs the 64-bit Debian GNU/Linux OS with a 64-bit JVM, Apache Spark 1.4 and Hadoop 2.6. Each node runs at most two mappers and reducers in parallel. We utilized three real-world datasets. The first dataset (DS1) is a sample of the Ask's database that contains about 214,000 question records. The second dataset (DS2) is by an order of magnitude larger and contains about 1.46 million publication records. We utilized the first three letters as the blocking key in the experiments involving DS1 (questions) and DS2 (publication titles). The similarity between two entities was computed using the Jaro–Winkler distance (Cohen et al., 2003a) of their comparing attributes (i.e., the question for DS1 and the publication title for DS2). We utilized the Jaro–Winkler distance for the experiments involving DS1 and DS2 (largest datasets) due to its efficacy and simplicity (low-cost matcher) to process string similarities (Cohen et al., 2003b). Thus, those pairs with a similarity $\Phi_{max} \geq 0.7$ were regarded as matches. The third dataset (DS3) is small and contains about 7800 DBLP and Google Scholar publication records (extracted from Kopcke et al., 2010). DS3 was utilized due to the absence of a gold standard for DS1 and DS2, which is necessary to evaluate the approaches' matching quality.

5.1. Robustness: degree of skewness

In this experiment, we study the robustness of the load balancing approaches to handle data skew. In our context, data skew occurs when there is a region of high similarity that enables the increasing of the adaptive window. This window increase generates new entity comparisons and consequently leverages the EM execution time during the window sliding on that region. Since the adaptive window increases only when a new duplicate entity is found, for this investigation, we control the degree of data skewness by modifying the number of duplicates in the dataset. The objective is to intentionally replace non-duplicated entities by duplicated ones according to an input percentage. Given a fixed number of entities e in the dataset, the number of duplicate entities d is equal to $d = e \cdot s$, such that $s \geq 0$ is the percentage that represents the degree of data skewness (the degree of duplication). To exemplify, suppose that we have 100 entities (e) and we set $s = 0$ (no skewness) then there are no duplicate entities in the dataset. In turn, if we want 20% of duplicate entities, we need to set $s = 0.2$, and thus, $d = 100 * 0.2 = 20$. To compare the load balancing approaches for different degrees of data skewness, we are interested at the execution time of the approaches when the dataset presents regions with high similarity detection.

The execution time of the approaches for different degrees of data skewness of DS1 ($n = 20$, $w = 1000$) is shown in Fig. 4. As we can see, the execution time of *S-SNM* is similar in all scenarios.

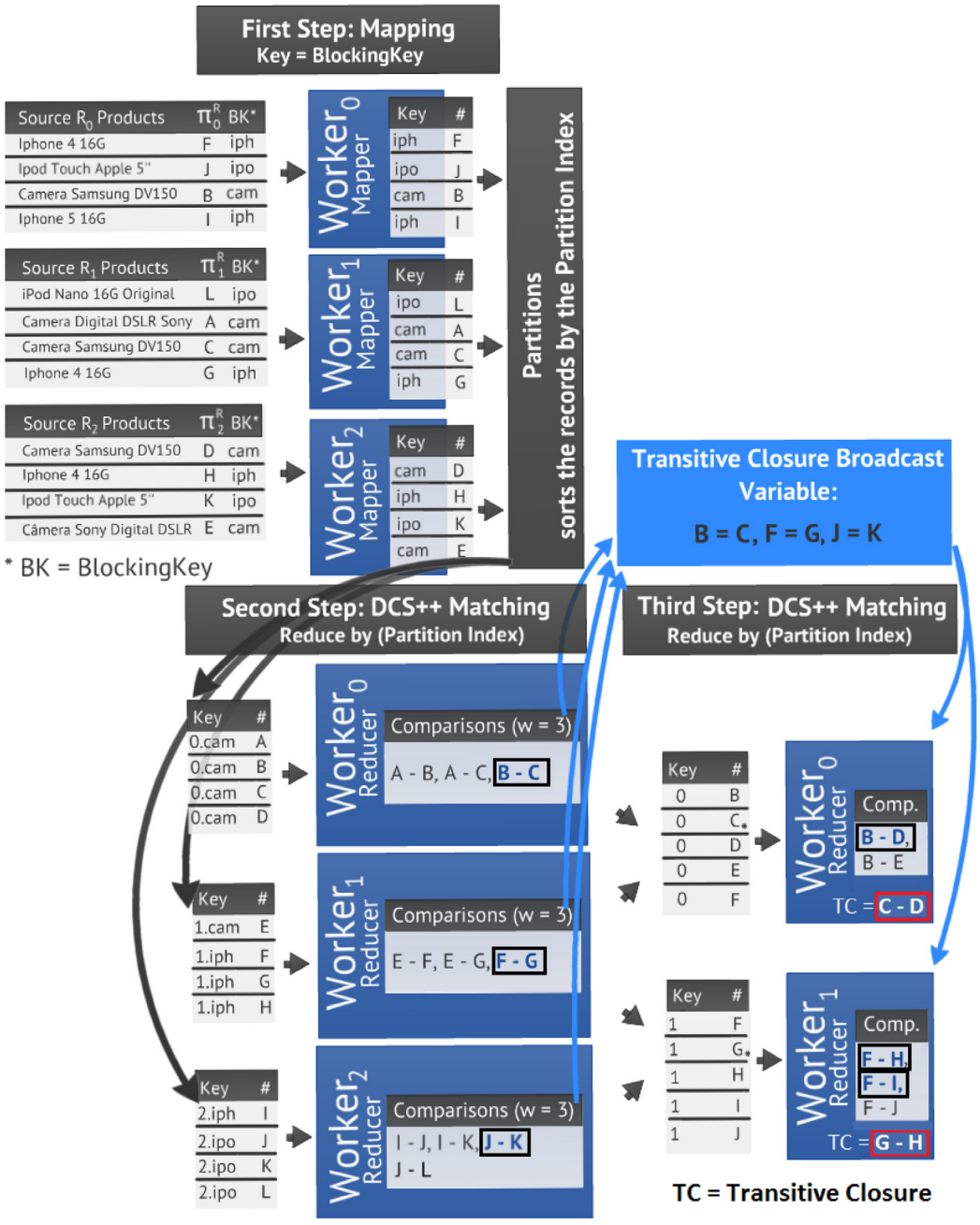


Fig. 3. Example of dataflow for the *S-DCS++* strategy with $w_{initial} = 3$.

On the other hand, since *S-SNM* employs a fixed window size, the workload balance is uniform. Also note that *S-DCS++* and *MR-DCS++* outperform *S-SNM* in the scenarios which there is duplicate detection. Only in the case that there are no duplicates in the dataset, the execution time of the approaches was similar. This result confirms that even with the increasing of the window size in regions of high duplicate detection (which leads to more entity comparisons), the usage of the transitive closure mechanism works by minimizing the execution time. This means that the time saved due to the reduction of unnecessary comparisons (promoted by the transitive closure mechanism) is higher than the time spent by the additional entity comparisons (generated due to the increasing of the window size). Considering the comparison between the *DCS++* parallel approaches, the main reason why *S-DCS++* outperforms *MR-DCS++* in all scenarios is due to the usage of two features regarding the spark programming model. As discussed in Section 2.2, the first feature is the usage of RDDs. Since RDDs can be kept in memory, programs can iterate over RDDs data multiple times very efficiently. The second one is related to the number of steps to execute all the EM process. *MR-DCS++* needs six steps to perform the whole EM process, i.e., three MR jobs containing two phases (Mapper and Reducer) each, whereas *S-DCS++* needs only three (one Mapper and two Reducers). Therefore, the results we have carried out indicate that there is no scenario in which *S-SNM* or *MR-DCS++* outperform *S-DCS++* regarding the execution time due to unbalanced workloads.

5.2. Execution time of the adaptive windowing: Spark vs. Hadoop's mapreduce

To analyze the scalability of the approaches, we vary the number of nodes from 1 up to 20. Following the Hadoop's docu-

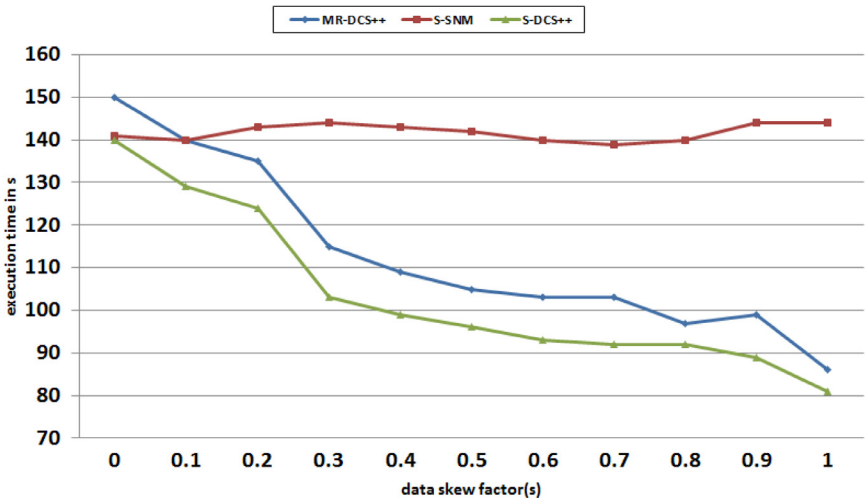


Fig. 4. Execution times for different data skews using $w = 1000$ (n = 20).

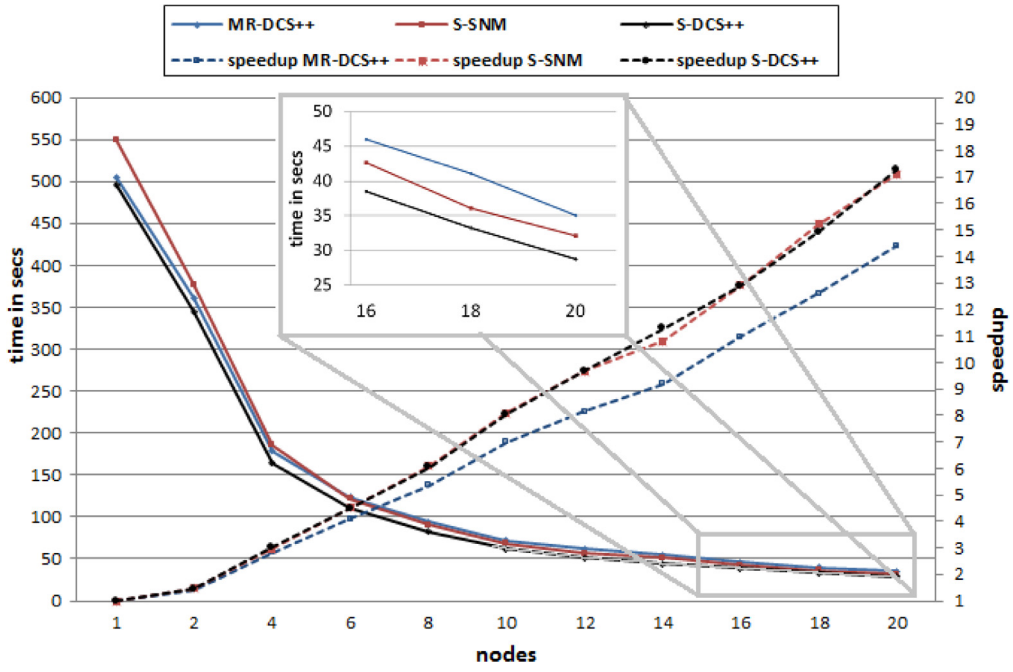


Fig. 5. Execution times and speedup of the approaches using DS1 ($w = 200$).

mentation, for n nodes, the number of mappers is set to $m = 2 \cdot n$ and the number of reducers is set to $r = m$. The values of the execution times are shown in Fig. 5 (DS1) and Fig. 6 (DS2). Since the number of comparisons also grows with the window size (increasing the execution time), we defined the same window size for the approaches according to the magnitude of the dataset size to avoid benefiting a specific approach. For DS1 and DS2, we utilized $w = 200$ and $w = 1000$, respectively, aiming to verify the scalability of the two approaches when performing a small and large ($w \geq 400$ (Kolb et al., 2012b)) window size.

The three approaches scale almost equally for the smaller and larger datasets DS1 and DS2, respectively. These results show their ability to evenly distribute the workload across the workers. However, due to the difficulties presented by the traditional SNM (*S-SNM*) related to the fixed (and difficult to configure) window size, its performance is depreciated by the execution of many unnecessary comparisons. This problem is the main reason why the *S-SNM* approach always performs slower than *S-DCS++* even presenting by design a uniform load balancing mechanism. However, in some cases, the *S-SNM* approach outperforms *MR-DCS++* (e.g., when the number of nodes is higher than 4). Despite the better uniform load balancing mechanism of *S-SNM*, this result highlights the advantage of using Spark to perform EM tasks. Even with a higher number of comparisons, *S-SNM* takes advantage of the RDD features[5] provided by Spark. It is noteworthy that the RDD caching is much more efficient than other low-level caching approaches such as the HDFS caching utilized by Hadoop, which in turn can only reduce disk I/O.

In the evaluation with DS1, *MR-DCS++* and *S-DCS++* saved 102,038 comparisons (compared with *S-SNM*) due to the adaptive window and the feature of skipping comparisons based on the information of transitive closure, which resulted in a decrease of execution time. On the other hand, the usage of a small window size combined with the benefits of the RDDs and the uniform load balancing mechanism lead the *S-SNM* to execute faster than

[5] Spark is tuned to cache the key-value pairs as RDDs in order to decrease both CPU (i.e., processing text parsing) and disk I/O overheads for subsequent iterations

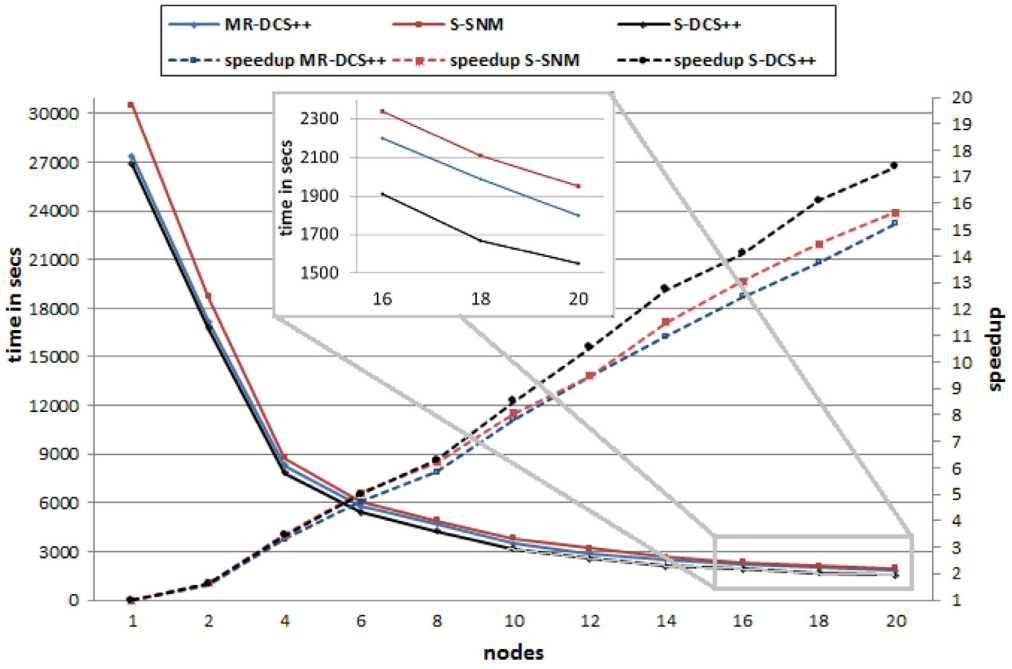


Fig. 6. Execution times and speedup of the approaches using DS2 ($w = 1000$).

MR-DCS++. Also, note that with the increase in number of nodes, the difference in terms of execution time between the approaches decreases due to a better division of the unnecessary comparisons among the nodes performed by *S-SNM* approach.

In turn, the evaluation with DS2 (Fig. 6) shows that the *MR-DCS++* outperforms *S-SNM* in terms of execution time, despite the slight gain of speedup in favor of *S-SNM*. This result shows that large datasets and window sizes can depreciate fairly the execution of *S-SNM*. This experiment also shows the performance gain of *S-DCS++* (highlighted by the speedup). Note that the performance difference between the *S-DCS++* and *MR-DCS++* is almost two nodes in favor of *S-DCS++*, according to the speedup with n = 20.

5.3. Matching quality vs. execution time

In this experiment, we study the tradeoff between the matching quality, in terms of F-Measure (i.e., the harmonic mean of precision and recall (Christen, 2012a)), and the execution time. As mentioned earlier, we utilized another publication dataset (DS3) that contains about 7800 records, and along with the dataset, there is the gold standard needed to compute the precision, recall and F-Measure metrics. Two window sizes $w = 500$ and $w = 1000$ are employed for the three approaches. This time, we employed as blocking key the first word of the publication title. Two entities are considered duplicated if their titles have a q-gram similarity (Christen, 2012a) $\Phi_{max} \geq 0.75$. We ran our experiments on 4 nodes ($m = r = 8$) due to the small size of the dataset. If a large number of nodes were utilized, due to the small size of dataset, the executions would be instantaneous and it would be difficult to highlight the differences between the execution times. Fig. 7 shows the observed execution times along with the number of comparisons and the precision, recall and F-measure collected values for the execution of the approaches with $w = 500$ and $w = 1000$. Precision, recall and F-Measure are computed including all pairs that have been directly identified and those generated by transitive closure.

For all cases, Fig. 7 shows that the number of comparisons performed by *S-SNM* is significantly higher than the amount performed by *S-DCS++* and *MR-DCS++* (under the same window size condition). In practice, the main difference between *S-DCS++* and *MR-DCS++* is the execution time since they both perform the *DCS++* method. As we can see, *S-SNM* employing $w = 1000$ performs about 2.7 million more comparisons than *S-DCS++*. In this case, the difference is also highlighted by the execution time given that *S-DCS++* (89 s) outperforms *S-SNM* (110 s) by 20.9%. This result indicates that *S-DCS++* performs efficiently the distribution of tasks among the nodes given that *S-SNM* has a natural load balancing mechanism (due to the fixed window size).

In terms of matching quality, although the slight decrease in the precision metric, mainly due to the false negatives generated by the wrong transitive closure assumptions, *S-DCS++* outperforms *S-SNM* in all cases with respect to recall and F-Measure. The reason is the *S-DCS++* ability to increase the window in regions of high similarity and perform better guesses about which comparisons must not be performed (by skipping them). Furthermore, the recall of *S-SNM* approach is compromised given that it misses some real duplicates due to the usage of a fixed window size. Thus, the results confirm that *S-DCS++* can achieve a high match quality even with smaller window sizes and that it can outperform *S-SNM* not only in match quality but also in terms of efficiency due to its transitive closure feature.

6. Summary and outlook

In this article, we proposed *S-DCS++* (Spark-based *DCS++*), a novel Spark-based approach for solving the problem of the adaptive SNM parallelization. The solution provides an efficient parallelization of the *DCS++* method (Draisbach et al., 2012) by using three Spark transformation steps, employing the broadcast variables Spark feature and applying a tailored data replication during data redistribution to allow the resizing of the adaptive window. The approach also addresses the data skewness problem with an automatic mechanism of data partitioning that enables a satisfactory load balancing across all available nodes. Our evaluation on a cluster environment using real-world datasets demonstrated that *S-DCS++* scales with the number of available nodes. We compared our approach against two existing state of the art algorithms (i.e., *S-SNM* and *MR-DCS++*) and verified that *S-DCS++* overcomes *S-SNM* and *MR-DCS++* in terms of performance (execution time).

Approaches						
	S-SNM		MR-DCS++		S-DCS++	
Window size	w = 500	w = 1,000	w = 500	w = 1,000	w = 500	w = 1,000
#Comparisons	≈3.7•10^6	≈7.1•10^6	≈2.4•10^6	≈4.4•10^6	≈2.4•10^6	≈4.4•10^6
Execution time (in secs)	79	110	78	105	64	89
Matching Quality						
Precision	92.3%	91.5%	88.9%	88.2%	88.9%	88.2%
Recall	63.9%	64.6%	70.4%	71.0%	70.4%	71.0%
F-Measure	75.5%	75.7%	78.6%	78.7%	78.6%	78.7%

Fig. 7. Comparison of quality and execution time of the approaches using different window sizes.

For future work, we intend to investigate how other EM indexing methods can take advantage of the iterative features of Spark. We also intend to verify how our approach can be adapted to address other Spark-based adaptive techniques for different kinds of data-intensive tasks. Another research direction is to compare *S-DCS++* with other parallel technologies, e.g., SNM on GPUs.

Acknowledgements

The results of this work have been partially funded by EUBra-BIGSEA (690116), a Research & Innovation Action (RIA) funded by the European Commission under the Cooperation Programme, Horizon 2020 and the Ministério de Ciência, Tecnologia e Inovação (MCTI), RNP/Brazil (grant GA-0000000650/04).

References

Christen, P., 2012a. Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection. Springer Publishing Company, Incorporated.

Christen, P., 2012b. A survey of indexing techniques for scalable record linkage and deduplication. IEEE Trans. Knowl. Data Eng. 24 (9), 1537–1555. doi:10.1109/TKDE.2011.127.

Cohen, W., Ravikumar, P., Fienberg, S., 2003. A comparison of string metrics for matching names and records. In: Kdd workshop on data cleaning and object consolidation, Vol. 3, pp. 73–78.

Cohen, W., Ravikumar, P., Fienberg, S., 2003. A comparison of string metrics for matching names and records. In: Kdd workshop on data cleaning and object consolidation, Vol. 3, pp. 73–78.

Cosma, G., Joy, M., 2012. An approach to source-code plagiarism detection and investigation using latent semantic analysis. IEEE Trans. Comput. 61 (3), 379–394. doi:10.1109/TC.2011.223.

Dean, J., Ghemawat, S., 2008. Mapreduce: simplified data processing on large clusters. Commun. ACM 51 (1), 107–113. doi:10.1145/1327452.1327492.

Draisbach, U., Naumann, F., Szott, S., Wonneberg, O., 2012. Adaptive windows for duplicate detection. In: Proceedings of the 2012 IEEE 28th International Conference on Data Engineering. IEEE Computer Society, Washington, DC, USA, pp. 1073–1083. doi:10.1109/ICDE.2012.20. http://dx.doi.org/10.1109/ICDE.2012.20.

Hernández, M.A., Stolfo, S.J., 1995. The merge/purge problem for large databases. SIGMOD Rec. 24 (2), 127–138. doi:10.1145/568271.223807.

Hsueh, S.-C., Lin, M.-Y., Chiu, Y.-C., 2014. A load-balanced mapreduce algorithm for blocking-based entity-resolution with multiple keys. Parallel Distrib. Comput. 3.

Kirsten, T., Kolb, L., Hartung, M., Gross, A., Kopcke, H., Rahm, E., 2010. Data partitioning for parallel entity matching. 8th International Workshop on Quality in Databases.

Kolb, L., Thor, A., Rahm, E., 2012a. Load balancing for mapreduce-based entity resolution. In: Proceedings of the 2012 IEEE 28th International Conference on Data Engineering. IEEE Computer Society, Washington, DC, USA, pp. 618–629. doi:10.1109/ICDE.2012.22. http://dx.doi.org/10.1109/ICDE.2012.22.

Kolb, L., Thor, A., Rahm, E., 2012b. Multi-pass sorted neighborhood blocking with mapreduce. Comput. Sci. 27 (1), 45–63. doi:10.1007/s00450-011-0177-x.

Kopcke, H., Thor, A., Rahm, E., 2010. Evaluation of entity resolution approaches on real-world match problems. Proc. VLDB Endow. 3 (1–2), 484–493. doi:10.14778/1920841.1920904.

Meng, X., Bradley, J., Yavuz, B., Sparks, E., Venkataraman, S., Liu, D., Freeman, J., Tsai, D., Amde, M., Owen, S., Xin, D., Xin, R., Franklin, M.J., Zadeh, R., Zaharia, M., Talwalkar, A., 2016. Mllib: machine learning in apache spark. J. Mach. Learn. Res. 17 (1), 1235–1241.

Mestre, D.G., Pires, C.E., 2013. Improving load balancing for mapreduce-based entity matching. In: Proceedings of the Eighteenth IEEE Symposium on Computers and Communications. IEEE Computer Society, pp. 618–624. doi:10.1109/ISCC.2013.6755016.

Mestre, D.G., Pires, C.E., Nascimento, D.C., 2015. Adaptive sorted neighborhood blocking for entity matching with mapreduce. In: Proceedings of the 2015 ACM Symposium on Applied Computing. ACM doi:10.1145/2695664.2695757.

Metwally, A., Faloutsos, C., 2012. V-SMART-Join: a scalable mapreduce framework for all-pair similarity joins of multisets and vectors. Proc. VLDB Endow. 5 (8), 704–715. doi:10.14778/2212351.2212353.

Nascimento, D.C., Pires, C.E., Mestre, D.G., 2015. A data quality-aware cloud service based on metaheuristic and machine learning provisioning algorithms. In: Proceedings of the 30th Annual ACM Symposium on Applied Computing. ACM, New York, NY, USA, pp. 1696–1703. doi:10.1145/2695664.2695753. http://doi.acm.org/10.1145/2695664.2695753.

Shi, J., Qiu, Y., Minhas, U.F., Jiao, L., Wang, C., Reinwald, B., Ozcan, F., 2015. Clash of the titans: mapreduce vs. spark for large scale data analytics. Proc. VLDB Endow. 8 (13), 2110–2121. doi:10.14778/2831360.2831365.

Wang, F., Wang, H., Li, J., Gao, H., 2013. Graph-based reference table construction to facilitate entity matching. J. Syst. Softw. 86 (6), 1679–1688.

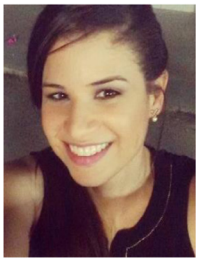
Yan, W., Xue, Y., Malin, B., 2013. Scalable and robust key group size estimation for reducer load balancing in mapreduce. In: Big Data, 2013 IEEE International Conference on, pp. 156–162. doi:10.1109/BigData.2013.6691568.

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I., 2012. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, p. 2.

Demetrio Gomes Mestre received the bachelor's and master's degrees in computer science from Federal University of Campina Grande, Brazil. He is currently a PhD student at Federal University of Campina Grande. His research interests include data quality, Big Data and cloud computing.

Carlos Eduardo Pires received the bachelor's and master's degrees in computer science from Federal University of Campina Grande, Brazil. He also finished his PhD from Federal University of Pernambuco, Brazil. He is currently a full Professor at Federal University of Campina Grande. His research interests include data quality and Big Data.

Dimas C. Nascimento received the bachelor's and master's degrees in computer science from Federal University of Campina Grande, Brazil. He is currently a PhD student at Federal University of Campina Grande. His research interests include data quality, machine learning and cloud computing.



Andreza Raquel Monteiro de Queiroz is currently a B.S.C. student at Federal University of Campina Grande. Her research interests include data quality and databases.

Veruska Borges Santos is currently a B.S.C. student at Federal University of Campina Grande. Her research interests include data quality and databases.

Tiago Brasileiro Araujo received the bachelor's and master's degrees in computer science from Federal University of Campina Grande, Brazil. He is currently a PhD student at Federal University of Campina Grande. His research interests include data quality, Big Data and cloud computing.