# Exploring Spark-SQL-Based Entity Resolution Using the Persistence Capability

Xiao Chen[1(✉)], Roman Zoun[1], Eike Schallehn[1], Sravani Mantha[2], Kirity Rapuru[1], and Gunter Saake[1]

[1] Otto-von-Guericke-University of Magdeburg, Magdeburg, Germany
{chen,rzoun,eike,saake}@iti.cs.uni-magdeburg.de,
Kirity.Rapuru@st.ovgu.de
[2] German Research Center For Artificial Intelligence, Berlin, Germany
Sravani.Mantha@dfki.de

**Abstract.** Entity Resolution (ER) is a task to identify records that refer to the same real-world entities. A naive way to solve ER tasks is to calculate the similarity of the Cartesian product of all records, which is called pair-wise ER and leads to quadratic time complexity. Faced with an exploding data volume, pair-wise ER is challenged to achieve high efficiency and scalability. To tackle this challenge, parallel computing is proposed for speeding up the ER process. Due to the difficulty of distributed programming, big data processing frameworks are often used as tools to ease the realization of parallel ER, supporting data partitioning, workload balancing, and fault tolerance. However, the efficiency and scalability of parallel ER is also influenced by the adopted framework. In the area of parallel ER, the adoption of Apache Spark, a general framework supporting in-memory computation, still is not widely studied. Furthermore, though Apache Spark provides both low-level (RDD-based) and high-level APIs (Datasets-based), to date, only RDD-based APIs have been adopted in parallel ER research. In this paper, we have implemented a Spark-SQL-based ER process and explored its persistence capability to see the performance benefits. We have evaluated its speedup and compared its efficiency to Spark-RDD-based ER. We observed that different persistence options have a large impact on the efficiency of Spark-SQL-based ER, requiring a careful consideration for choosing it. By adopting the best persistence option, the efficiency of our Spark-SQL-based ER implementation is improved up to 3 times on different datasets, over a baseline without any persistence option or with misconfigured persistence.

**Keywords:** Apache Spark (Spark SQL) · Entity Resolution
Record linkage · Data matching · Parallel computing


S. Kozielski et al. (Eds.): BDAS 2018, CCIS 928, pp. 3–17, 2018.
https://doi.org/10.1007/978-3-319-99987-6_1

## 1 Introduction

A real-world entity may be described in different ways in digital repositories because of typos, diverse formats and others. In order to identify and connect different descriptions that refer to the same real-world entities, Entity Resolution (ER) is required. Ironically, as a process to identify duplicate records, ER itself has many aliases: if records to be identified are only from a single source, it is usually named de-duplication [8]. Otherwise, in computer science it is also typically referred to as data matching, record linkage, duplicate detection, reference reconciliation, or object identification [8]. In the database domain, ER is tightly related to similarity joins.

Today, ER plays a vital role in diverse areas, not only in the traditional applications of census, health data or national security, but also in Internet-based applications of business mailing lists, online shopping, web searches, etc. [8]. It is also an indispensable step in data cleaning, data integration, and data warehousing. The use of computer techniques to perform ER dates back to the middle of the last century. Since then, researchers have developed many techniques and algorithms for ER. Generally speaking, there are two major directions to solve ER problems: pair-wise ER and clustering ER. Pair-wise ER is the most commonly used, it solves ER problems through local evidences by comparing each possible record pair; clustering ER aims to group similar data objects into the same cluster, according to some criteria, and solves ER problems by also using global evidences. Our research focuses on the former one, pair-wise ER. The reason for adopting pair-wise ER is that it is applicable for all types of data and we consider that it might have more applications. However, pair-wise ER faces a big challenge, when the data volume is quite large. Since the time complexity of pair-wise ER is $O(n^2)$, sequential processing to complete a naive ER process is no longer feasible. For instance, in a naive pair-wise approach, an input dataset with 1 million records corresponds to 1 trillion comparisons. Suppose that a comparison takes one microsecond, then we would need 11.6 days to complete this ER task [12]. Therefore, the increasing data volume nowadays and the data expansion inherent to pair-wise approaches make the use of parallel computing necessary to solve pair-wise ER tasks.

To implement parallel ER, there have been two main research directions so far: the first one is with parallel DBMSs, the other way is to employ a distributed computation framework to help with the implementation [7]. The solution of using parallel DBMSs proposed more than two decades ago has some shortcomings for individuals or small and medium-sized enterprises with ER tasks. Parallel DBMSs are traditionally expensive [13], ill-suited for heterogeneous environments and have limited fault tolerance in large deployments [1]. As a result, users with limited budget and strong fault tolerance requirements find it difficult to solve their ER tasks with parallel DBMSs. The second solution: employing a distributed computation framework has become very popular in recent years, since most frameworks are open source, free to use, and also provide straightforward programming models to ease the management of distributed programs. Hadoop MapReduce is the most popular programming model used and

has almost dominated research in parallel ER since 2008. However, it has the following disadvantages: first, it is relatively difficult to program and the process has to be abstracted with "map" and "reduce" phases; second, it is disk-oriented, which may lower the performance. Therefore, we explored to use another computation framework: Spark, since it supports acyclic data flow (i.e., a more expressive processing model) and both in-memory and on-disk computing. Due to these characteristics Spark offers more promising performance and tuning choices. Furthermore, it integrates with several libraries such as Spark SQL, GraphX and MLlibs, making it possible to express ER in terms of relational databases, graphs and machine learning [2]. As we will describe in Sect. 3.2, Apache Spark provides both low-level (RDD-based) and high-level APIs (Datasets-based). RDD is short for resilient distributed dataset, which is the essential abstraction in Spark core. It stands for a collection of data partitioned across the cluster. Datasets are an abstraction from the Spark SQL library. Most previous research employing Spark for parallel ER is based on RDD APIs, instead, in this paper, we investigate using Spark-SQL to implement parallel ER and explore the persistence capability of Spark to see how it affects the performance. We summarize our contributions in the following:

- We explore an efficient way of using Spark to implement parallel ER tasks. A baseline workflow is implemented with Spark SQL, considering suitable and scalable algorithms for our experimental datasets, conforming to the tenet of high efficiency, but on the premise of satisfactory effectiveness.
- We analyze our baseline workflow with the Spark web UI. As a result, we find that some data should be persisted with a suitable strategy to improve performance. Therefore, we optimize the baseline workflow by exploring different persistence options. We evaluate efficiency and speed-up of our work and report that the runtime of our baseline Spark-SQL-based ER is optimized up to 3 times when employing the best persistence option.
- For comparison, we also implement a same ER process with Spark-RDD in order to evaluate and contrast to our proposed Spark-SQL-based ER, and we report that our Spark-SQL-based ER can achieve in average 2.2 times the efficiency of Spark-RDD-based ER.

We structure the rest of our paper in the following way: In Sect. 2, we review related work. Then Sect. 3 provides background knowledge to understand the paper; In Sect. 4, we introduce our Spark-SQL-based ER process and our method used to explore the persistence capability. Subsequently, we describe our experiments and evaluation results in Sect. 5; At last, we conclude our work and giving an outlook on future work in Sect. 6.

## 2 Related Work

The use of computer techniques to perform ER dates back to the middle of the last century. Since then, researchers have developed various techniques and algorithms for ER, fueled by its applications in many fields. Several books and

surveys can be recommended for overviews on ER-related techniques, e.g., [8,11]. To implement parallel ER, the use of parallel DBMSs is one important alternative, e.g., [4]. Another choice is use of big data processing frameworks. Research in parallel ER has been dominated by the use of the MapReduce programming model e.g., [16,19]. We give an overview and classification on the current state of parallel ER in our survey paper [7].

Along with the growing popularity of Apache Spark in recent years, there have been several implementations based on Apache Spark, which are the most similar to our approach. Therefore, we briefly introduce work in this area, pointing out the difference between our work and previous studies. Pita et al. [18] implemented a Spark-RDD-based approach for probabilistic ER of healthcare data. Their evaluation covers execution time and compares results to OpenMP-based implementations. Chen et al. [6] focused on the Top-k similarity join problem, which is akin to ER, and implement a Spark-RDD-based algorithm for massive multidimensional data, focusing on a more efficient distance function using locality sensitive hashing (LSH) and determining the top-k closest pairs. Wang and Karimi [21] focus on using a k nearest neighbors (kNN) algorithm for the classification step with Spark and propose a method to minimize the cross cluster kNN search. Mestre et al. [17] propose S-DCS++, a Spark-RDD-based ER with an adaptive-window Sorted Neighborhood blocking method and load balancing strategies. Our work in this paper distinguishes itself from the described publications, since the research is RDD-based, while we explore how to efficiently use Spark-SQL for implementing parallel ER, which supports high-level APIs and is able to outperform the performance of RDD-based approaches. Some reasons for our selection of Spark-SQL are, on the one hand based on the ease of adoption, considering that our test datasets are already structured and that Spark SQL-based APIs are also high level rather than low-level; on the other hand our choice is based on opportunities to improve the performance by leveraging the API features, since Spark SQL includes a cost-based optimizer called Catalyst, a columnar storage called Tungsten and is able to use Kryo serialization replacing traditional Java serialization to improve efficiency [15]. In addition, our ER process is designed to be generic and can be easily adapted for different kinds of data.

## 3 Preliminaries

### 3.1 A Common Entity Resolution Process

A common ER process includes five major steps in total: data preprocessing, blocking, pair-wise comparison, classification, and evaluation [8]. Data preprocessing is required as the first step to clean and standardize the input data. Before the core step of pair-wise comparison, blocking is required to reduce the search space of ER, since the original search space of ER is quite large, each input record would need to be compared with all other records, making the time complexity quadratic on the input dataset. With the blocking step, the input

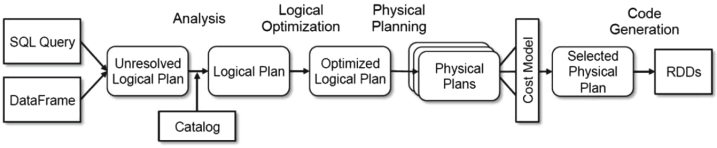



**Fig. 1.** Phases of query planning in Spark SQL [3]

dataset is divided into blocks and candidate pairs are generated. Then similarity functions are employed to estimate the similarity of candidate pairs, in the pair-wise comparison step. After all intermediate results are obtained, the classification step estimates whether each pair of records refers to the same real-world entity or not. The last step, evaluation, is optionally used for estimating the effectiveness of the ER process.

### 3.2 Apache Spark

Apache Spark is designed for fast processing of large-scale data. It supports acyclic data flow and in-memory computing. Its main abstraction is a resilient distributed data set (RDD), which is a collection of data partitioned across the nodes of a cluster that can be operated on in parallel, supporting in-memory computing and providing fault tolerance. RDDs offer two kinds of operations: transformations and actions. Transformations only create new RDDs from existing RDDs, while actions run a computation on RDDs and return values. In order to enable Spark to run more efficiently, transformations in Spark are all lazy, which means transformations for datasets are only scheduled but not computed right away. Besides, persisting data in memory is one of the most important capabilities in Spark. There are several persistence levels available, such as MEMORY_ONLY, MEMORY_ONLY_SER, MEMORY_AND_DISK. These are distinguished by the location for data persistence (memory or disk), or whether to serialize the data before persisting it. Spark SQL is the module of Apache Spark for structured data, which enables people to query structured data inside Spark programs, using either SQL queries or the corresponding Datasets APIs. Spark SQL does not force users to decide for a relational or a procedural API, but enables users to mix both of them [3]. Spark SQL provides the possibility to have logical and physical optimizations before the real execution. Figure 1 shows the four phases using its cost-based optimizer Catalyst to optimize the application. It first analyzes a logical plan and optimizes it through equivalence rules, then it chooses the best physical plan based on statistics and cost estimates, lastly it generates code to compile the query into Java bytecode [3]. Besides, it also has a columnar storage called Tungsten, and is able to use Kryo serialization to replace traditional Java serialization to minimize storage cost and improve efficiency [15]. All above-introduced features make Spark SQL a more promising option to process structured or semi-structured data.

# 4 Spark-SQL-Based Entity Resolution

In this section, we will first present our baseline Spark-SQL-based ER process and describe possible strategies, algorithms, and corresponding parameters used within the process of ER, then we introduce the persistence concept to optimize our baseline implementation.

## 4.1 Spark-SQL-Based Entity Resolution Process

In our Spark-SQL-based ER process, input data is abstracted to Dataset<Row>, which is distributed data with schema and is quite similar to a relational table. Figure 2 shows how data is transformed through the different processing steps and how we obtain matches and evaluate results through action operations in Spark SQL. The left part of Fig. 2 shows the data transformation process and the right part corresponds to steps, which cause transformations of data. The input data is first loaded into Spark Dataset<Row> with a balanced distribution to all

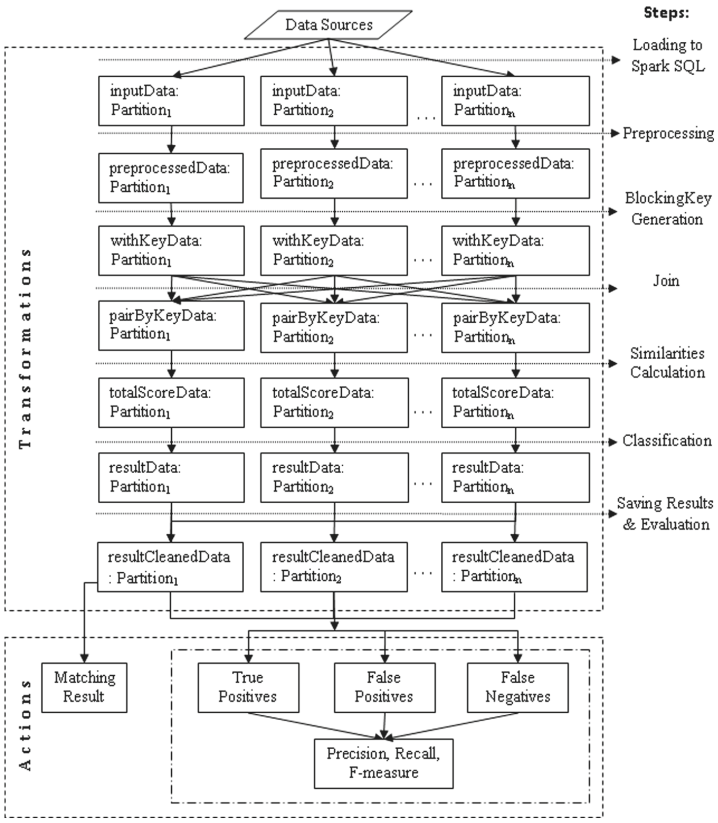



**Fig. 2.** Spark-SQL-based ER

available nodes. Before the real process begins, a preprocessing step is performed to clean and standardize the data. Because of different characteristics of input data, some common measures are suggested, such as replacing null values with "0" and removing stop words, to improve the input data quality for the sake of reducing the negative impacts that noise may bring to the real process. Then the blocking step is performed, which includes "blockingkey generation" and "join" (Fig. 2) substeps. Our blocking method is standard blocking, which is straightforward and efficient but may lead to false negatives. In order to mitigate this negative effect, some approximate approaches can be used. The first possibility of approximation is during the preparation of blocking keys, instead of directly using some attribute values as the blocking key, attribute values can be first generalized. For instance, for our experimental datasets with personal information, surname and given name are first transformed using the double metaphone algorithm. With this algorithm, letters that share a similar pronunciation are transformed to the same representation to handle with common transcription mistakes that people might make when recording information based on what they hear from speakers. Another possibility of approximation is during the join operation. After knowing the blocking key, in order to get all necessary candidate pairs, a join operation is performed. Instead of an equi-join, the join condition can be set that the similarity degree between blocking keys of two records of a pair is higher than a given threshold. With these two sub-steps, data is repartitioned and candidate pairs are distributed through the cluster. Next, the similarity on each attribute is calculated for the candidate pairs, and then aggregated into a total similarity score. Similarities only need to be calculated when the ID of the first record is smaller than that of the second record, since the similarity between a pair of records is reflexive and symmetric. The algorithms used to calculate similarity scores should be also chosen according to attribute properties. For instance, the Jaro-Winkler distance is suitable for string attributes, because Jaro-Winkler has been proved to be an efficient edit-distance metric for short strings, such as those required for name matching [10]. Based on the total similarity score, whether a pair is match or non-match can be judged through a preset threshold. Choosing a suitable threshold is requirement-dependent. Tuning this feature is a sensitive matter, since a higher threshold translates into a higher precision but lower recall and vice versa (i.e., higher thresholds lead to fewer matches and more non-matches). Until now, all steps cause only transformations of data and they are lazily evaluated until actions are triggered. Actions in our ER process include saving the matching result and calculating true positives, false positives and false negatives in order to evaluate results, given that precision, recall and F-measures rely on those three values. To calculate true positives, false positives and false negatives we require ground truths. The ground truth can normally be accessed through an external file or can be implicitly contained in the data itself. For example, in our synthetic datasets, the ground truth can be obtained through removing useless information of the attribute "ID". Therefore, in Fig. 2, resultData is transformed to resultCleanedData in preparation for the evaluation.

In order to make the process perform efficiently in a distributed environment, unnecessary repartitioning should be avoided and only the data which is useful for the following steps should be kept. Data which only helped the previous operations should be discarded as soon as possible to avoid unnecessary costs. For instance, after the join operation the blockingkey can be discarded, and after calculating their corresponding similarity scores all original attributes except the identifiers can be discarded.

### 4.2 Exploring Persistence Capability to Optimize Our Baseline Implementation

As described in Subsect. 3.2, lazy evaluation is an important strategy in Spark to optimize the performance and achieve fault tolerance. However, because of this property, the related data might need to be re-evaluated again each time an action is triggered, which may lower the performance when no suitable persistence option is used. In this section, we introduce our method to find the right persistence option to improve the efficiency of our baseline implementation. There are two aspects that need to be considered to find the right persistence option. The first one is which data should be persisted, the other one is which persistence level to use for persisting the data. Since choosing a persistence level depends on the data size and the cluster hardware conditions, we will introduce our decision in Sect. 5. In this subsection, we only introduce the approach we used to decide which data should be persisted.

Configuring persistence can bring benefits only when the performance improvements due to persistence are larger than the potential overheads from persisting the data itself. We considered this for determining which data was relevant for persistence.

The main method we used to find the relevant data is through observing the Spark web UI and analyzing our ER process. Straightforwardly, the data involved in multiple actions should be relevant. According to the transformations and actions shown in Fig. 2, resultData which contains the pair-wise comparison result and resultCleanedData which contains the ground truth that can be used for the evaluation, are counted as relevant data. If this data is not persisted, each time a related action is triggered, they have to be obtained through a series of transformations from the original input data. TotalScoreData is not involved because it is only used in an early step of the transformation process. However, since it has one attribute less than resultData, it could be a good persistence candidate (since it might incur in less overheads to persist it). Therefore, we also evaluate the assumption that persisting totalScoreData may have a better effect than straightforwardly persisting resultData. In the next section, based on the above-described ideas we designed several experiments to evaluate different persistence options and figure out the best alternative.

# 5 Experiments

## 5.1 Experimental Setting

In this subsection, we introduce our experimental setting. It relates to two aspects. On the one hand, we describe the datasets that we used to evaluate our application; on the other hand, we demonstrate our Spark YARN cluster based on Hortonworks Data Platform (HDP).

**Table 1.** Datasets used in experiments

| Datasets | Property | | |
|---|---|---|---|
| | ID | Name | Input size |
| Synthetic datasets | 1.1 | $5 * 10^5 + 5\%$ | 57.3M |
| | 1.2 | $5 * 10^5 + 50\%$ | 79.9M |
| | 1.3 | $10^6 + 5\%$ | 114M |
| | 1.4 | $10^6 + 50\%$ | 160M |
| Real datasets | 2 | Facebook | 82.7M |

**Datasets for Experiments.** We use both synthetic datasets and real datasets for our experiments. The purpose of using synthetic datasets is that we can easily know the ground truth and we can control characteristics of the datasets, such as sizes or duplicate percentages. In addition, our generated data is based on real-world data and can follow similar characteristics to the real data it is based on [9]. It is generated by a data generator called GECO [20]. GECO consists of GEnerator and COrruptor, which is specifically designed for generating ER datasets. First, the generator is responsible for generating original data, whose attribute values are based on frequency look-up files and functions provided by users. Afterwards, the corruptor modifies some attribute values of the generated data from the first step to generate duplicate records. In this way, a synthetic dataset with original records and their duplicate records is generated. For each record, an identifier is assigned, which can show the ground truth of the dataset. For the sake of data diversity and for being able to cover different cases, we generated the datasets stored in csv files using GECO, which all contain personal information with the following 14 attributes: rec-id, gender, given-name, surname, postcode, city, telephone-number, credit-card-number, income-normal, age-uniform, income, age, sex, and blood-pressure. The datasets used are shown in the upper part of Table 1. Their names can reflect their sizes and duplicate percentages: the first part stands for the number of original records that a dataset contains, for example $10^6$ means there are 1 million (10 power 6) original records. The second part conveys the duplicate percentage based on the original records, for example 5% corresponds to the number of duplicates as a percentage of the number of original records. Because of privacy reasons, it is very hard to

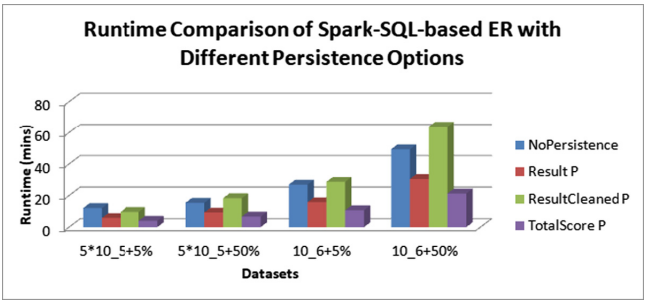



**Fig. 3.** Runtime of implementation with or without persistence on synthetic datasets

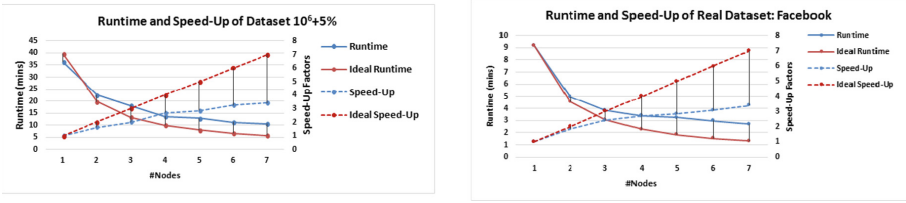
find real datasets that contain personal information. The real data we have for the experiments are parsed public data from Facebook [5], which are people's names including duplicates. We cannot know the ground truth for this dataset, so we conducted experiments with real datasets to evaluate efficiency and speed-up our workflow without the evaluation step. Our real dataset, which is shown in Table 1, includes one million records used for evaluating efficiency. For the blocking step, the blocking key is defined by concatenating the first two letters of double metaphone codes of attributes "given-name" and "surname". For the pair-wise comparison step, Jaro-Winkler and absolute difference similarity functions are used to calculate similarities of strings and numbers separately. For the classification step, a threshold of 0.75 is used to judge the match or non-match state of a pair, which is reasonable for both precision and recall.

**Spark YARN Cluster.** We have implemented our Spark-SQL-based ER in Java and conducted all experiments in our Spark YARN cluster with Spark version 2.0. We use HDP to deploy our Spark YARN cluster, which is an open source Apache Hadoop distribution based on a centralized architecture (YARN) [14]. The cluster has ten Spark clients, which includes one node to access the cluster, two nodes as HDFS NameNodes (one active and one standby) and seven executor nodes as HDFS DataNodes, which can also be used to run Spark applications. Each of them runs on virtual machines, with VMWare ESXi as hyper-visor. In the following, we introduce hosts' hardware for seven executor nodes. Each of them has four CPU cores, 16 GB RAM (6 GB is available for executor nodes to run Spark applications) and 150 GB hard disk. However, our cluster is heterogeneous, since four of nodes are with four Intel Xeon E5-2650 @2.00 GHz cores, two nodes are with four Intel Xeon E5-2650v2 @2.60 GHz cores, the last nodes are with four Intel Xeon E5520 @2.27 GHz cores. All hosts are connected by a 10 GBit Ethernet in a star topology. We submitted our applications as jar files to the Spark Yarn cluster and conducted a series of experiments to evaluate the runtime and speedup (w.r.t our baseline implementation) of our implementations with persistence. We ran each experiment five times and after dropping the highest and lowest result, our final result was obtained by averaging the remaining three.

### 5.2 Evaluation: Persistence

**Experiment Design.** According to the analysis in Subsect. 4.2, we have the following persistence options: baseline, resultPersistence, resultCleanedPersistence and totalScorePersistence. The baseline implementation is our Spark-SQL-based ER without any persistence. The other three options are persisting these three relevant data, respectively. We ran these implementations with different persistence options on different datasets with our 7-executors-cluster and recorded the runtimes separately. Persistence level "memory_only" was chosen to provide the most efficient persistence, because our cluster has sufficient memory to store relevant data.

**Results and Discussion.** Figure 3 shows the runtime of four different implementations on different datasets. As we can see from it, for all four datasets, it reflects a similar trend on different implementations. There seems to be limited gains from only persisting resultCleanedData, as in most cases it takes longer for the implementation. By persisting resultData, an improvement of a factor of 1.6 to 2 can be achieved. By persisting totalScoreData instead of the straightforward choice (resultData), the efficiency improvement can even reach a factor of 2.3–3x. Therefore, we have the following conclusion: persisting totalScoreData is our best persistence option, which can shorten the total runtime and improve efficiency by mostly 3x. The straightforward choice to persist is not always the best, when the data that is used for multiple actions is quite large, instead of directly persisting it, we can judge whether it is possible to persist its former data that is much smaller than it and when the computation between them is not so time-consuming and their data sizes differ much, it is normally beneficial to persist the former one instead. For instance, in our experiment, it is more beneficial to persist totalScoreData than resultData. Besides, not all the data that will be reused needs persistence, such as when resultCleanedData is persisted, we need even more time to complete the application. In conclusion, it is important and necessary to persist certain data in Spark SQL, but this needs to be judiciously used, carefully deciding which data to persist for achieving the most benefits.




(a) Synthetic Dataset: $10^6$+5%

(b) Real Dataset

**Fig. 4.** Runtime and speed-up of best-persistence implementation (Color figure online)

### 5.3 Runtime and Speed-Up Evaluation

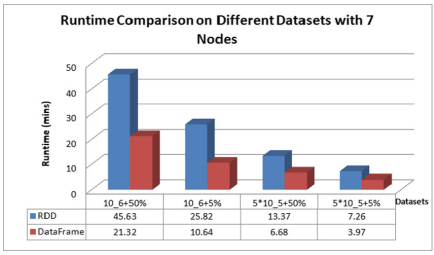
**Experiment Design.** To evaluate the efficiency and speed-up of our best-persistence implementation, we designed the following experiments: we use datasets with IDs 1.1–1.4, and 2 to run our implementations with different number of executor nodes (from one to seven) to get their total runtime, then based on runtime we can calculate their speed-up. These datasets cover different sizes and properties of datasets and also synthetic and real datasets. Since we do not have the ground truth for the real datasets, we evaluated speed-up of our implementations without evaluation steps.

**Results and Discussion.** Figure 4a and b show the runtimes and the corresponding speed-ups of our best-persistence implementation for the synthetic dataset: $10^6 + 5\%$ and our real dataset respectively, when we increased the number of executor nodes from one to seven. Since all the other synthetic datasets show similar results, we represent only one to save space and make the figure clear. The red lines in it are the ideal runtime and speed-up. The results of both implementations are similar and the differences between our results and a linear improvements increases proportionally with number of nodes. For example, with seven nodes, the ideal speed-up (for an optimal parallelization) should be equal to seven, while the real speed-up of both implementations achieved is only around 3.4. Figure 4b shows the runtime and corresponding speed-up of our application without the evaluation step for the real dataset, when we increase the number of executor nodes from one to seven. In general, it displays a similar trend of speed-up compared to synthetic datasets. In conclusion, we can see from the result that our Spark-SQL-based ER shows a reasonable speed up and the runtime for all implementations with different datasets is reduced along with increasing the number of nodes, although with a gap from ideal ones because of unavoidable communication costs, synchronization and scheduling overheads, negative effects due to our heterogeneous cluster and an existing biased workload.

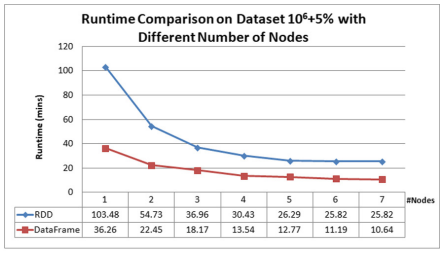
### 5.4 Comparison to RDD-Based Implementation

**Experiment Design.** As the state-of-the-art research has implemented several SQL-RDD-based ER applications, we have also implemented a Spark-RDD-based ER for comparison, which follows the same ER process from our Spark-SQL-based ER. For the Spark-RDD-based implementation, we also ran experiments with an optimized configuration, such as the level of parallelism, and its best persistence options. The experiments were conducted on datasets 1.1–1.4 in our cluster with a variable number of executor nodes (from one to seven).

**Results and Discussion.** Figure 5b shows the runtime of both implementations for datasets 1.1–1.4, when we run them with a seven-node-cluster. Figure 5a shows the runtime of both implementations for the synthetic dataset: $10^6 + 5\%$.




(a) on Datasets 1.1-1.4 with 7 Nodes




(b) on Dataset: $10^6$+5% with 1-7 Nodes

**Fig. 5.** Runtime comparison of SQL-based and RDD-based ER

The results when we run the same experiments with other datasets or other number of node are similar to the case with seven nodes. As we can see from the figure, our SQL-based implementation runs around 1.8 to 3 times faster than the RDD-based implementation. By averaging all speed-up factors with all number of nodes, our SQL-based implementation can speed up ER by a factor of 2.2. Therefore, we conclude that, due to the possible optimizations in Spark SQL and the leveraging of persistence options, Spark-SQL-based implementations can achieve much better efficiency for parallel ER than RDD-based implementations and it is a good option to use Spark SQL to perform parallel ER.

### 5.5 Threats to Validity

The experimental results are subject to several critical considerations. Regarding the **data sets** we focused on highly structured and flat data, which are typical in ER applications. Nevertheless, approaches for less structured data such as document data or more complex data such as graph data may show a better potential for more low-level APIs like RDD. Still we expect some of the basic considerations regarding persistence options to hold. Regarding the **experimental environment** most of all the limited number of nodes available and the size of the data sets that were chosen accordingly restrict the possible interpretation regarding speed-up and scalability for large scale applications. Here, the experiments indicate a promising trend that would have to be verified for large-scale settings. Finally, the presented conceptual ER process as well as its implementation may be subject to **further optimization**. We provided a straightforward implementation of a general process for ER involving data reuse and applied common optimizations to the best of our knowledge in a way, that was unbiased toward all the compared approaches. Nevertheless, further optimizations regarding the environment settings are conceivable, and advanced techniques such as improved load-balancing should be subject of further research. The comparison result may be changed for **an ER process without the evaluation step**, i.e. without data reuse, because we observed an important reason that our Spark-SQL-based implementation is faster than Spark-RDD-based implementation is Spark-SQL-based implementation can benefit from the best persistence option.

## 6 Conclusion

In this paper, we explored an efficient way to use Spark to implement parallel ER. We implemented our baseline parallel ER application with the Spark SQL library due to its high performance and high level APIs. Then we explored different persistence options in order to optimize our baseline implementation. At last, we confirmed experimentally that the best persistence option requires a trade-off between the overheads for persisting the data and the gains from avoiding its re-computation. We evaluate the efficiency and speed-ups of persistence options over different synthetic and real datasets. Our results show that the implementation with the best persistence option can achieve a performance improvement by 2.3 to 3 times over the baseline implementation. Despite of the limits of our heterogeneous cluster, we still achieved an acceptable speed-up for our Spark-SQL-based implementation in all datasets, which suggests that this approach is robust to data with different duplicate percentages and sizes. In addition, we implemented a Spark-RDD-based ER and compared both cases with experiments. Our results show that for our comparison experiment, our best-persistence Spark-SQL-based ER is about 2.2 times faster than RDD-based ER for different datasets. In conclusion, our work shows that Spark SQL provides another alternative to implement the ER process. In the adoption of Spark SQL for ER tasks, developers are susceptible to decrease the runtime performance by incorrectly using the persistence options of Spark. We believe that there is a need either to propose a "best practice" approach for nudging developers into using the best solution, or to add these carefully adopted persistence optimizations into the Catalyst optimizer, by using statistics and cost estimations similar to relational databases.

For future work, we will adopt available load balancing strategies to further improve efficiency. Moreover, since our current evaluation is based on the specific workflow we introduced in this paper, in the future, we will consider different scenarios for ER and adopt other blocking techniques.

**Acknowledgments.** The authors would like to thank China Scholarship Council [No. 201408080093] to fund our work. Besides, we are very grateful to Gabriel Campero Durand, David Broneske and Yusra Shakeel to provide us valuable feedback.

## References

1. Abouzeid, A., Bajda-Pawlikowski, K., Abadi, D., Silberschatz, A., Rasin, A.: HadoopDB: an architectural hybrid of MapReduce and DBMS technologies for analytical workloads. Proc. VLDB Endow. **2**(1), 922–933 (2009)
2. Apache: Apache spark. http://spark.apache.org/. Accessed 10 Apr 2017
3. Armbrust, M., et al.: Spark SQL: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383–1394. ACM (2015)
4. Benjelloun, O., et al.: D-Swoosh: a family of algorithms for generic, distributed entity resolution. In: 27th International Conference on Distributed Computing Systems, ICDCS 2007, p. 37. IEEE (2007)

5. Bowes, R.: Facebook names dataset. http://academictorrents.com/details/e54c73099d291605e7579b90838c2cd86a8e9575. Accessed 15 June 2017
6. Chen, D., Shen, C., Feng, J., Le, J.: An efficient parallel top-k similarity join for massive multidimensional data using spark. Int. J. Database Theory Appl. **8**(3), 57–68 (2015)
7. Chen, X., Schallehn, E., Saake, G.: Cloud-scale entity resolution: current state and open challenges. Open J. Big Data (OJBD) **4**(1), 30–51 (2018)
8. Christen, P.: Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31164-2
9. Christen, P., Vatsalan, D.: Flexible and extensible generation and corruption of personal data. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management, CIKM 2013, pp. 1165–1168. ACM, New York (2013)
10. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string metrics for matching names and records. In: KDD Workshop on Data Cleaning and Object Consolidation, vol. 3, pp. 73–78 (2003)
11. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: a survey. IEEE Trans. Knowl. Data Eng. **19**(1), 1–16 (2007)
12. Getoor, L., Machanavajjhala, A.: Entity resolution for big data. In: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, p. 1527. ACM (2013)
13. Hameurlain, A., Morvan, F.: Big data management in the cloud: evolution or crossroad? In: Kozielski, S., Mrozek, D., Kasprowski, P., Małysiak-Mrozek, B., Kostrzewa, D. (eds.) BDAS 2015-2016. CCIS, vol. 613, pp. 23–38. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-34099-9_2
14. Hortonworks: Hortonworks data platform. https://hortonworks.com/products/data-center/hdp/. Accessed 10 July 2017
15. Karau, H., Warren, R.: High Performance Spark. O'Reilly Media, Sebastopol (2017)
16. Kolb, L., Thor, A., Rahm, E.: Dedoop: efficient deduplication with Hadoop. Proc. VLDB Endow. **5**(12), 1878–1881 (2012)
17. Mestre, D.G., Pires, C.E.S., Nascimento, D.C., de Queiroz, A.R.M., Santos, V.B., Araujo, T.B.: An efficient spark-based adaptive windowing for entity matching. J. Syst. Softw. **128**, 1–10 (2017)
18. Pita, R., Pinto, C., Melo, P., Silva, M., Barreto, M., Rasella, D.: A spark-based workflow for probabilistic record linkage of healthcare data. In: EDBT/ICDT Workshops, pp. 17–26 (2015)
19. Rong, C., Lu, W., Du, X., Zhang, X.: Efficient duplicate detection on cloud using a new signature scheme. In: Wang, H., Li, S., Oyama, S., Hu, X., Qian, T. (eds.) WAIM 2011. LNCS, vol. 6897, pp. 251–263. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23535-1_23
20. Tran, K.N., Vatsalan, D., Christen, P.: GeCo: an online personal data generator and corruptor. In: Proceedings of the 22nd ACM International Conference on Information & Knowledge Management, CIKM 2013, pp. 2473–2476. ACM, New York (2013)
21. Wang, C., Karimi, S.: Parallel duplicate detection in adverse drug reaction databases with spark. In: EDBT, pp. 551–562 (2016)