Performance Comparison of Three Spark-Based Implementations of Parallel Entity Resolution

Xiao Chen(✉), Kirity Rapuru, Gabriel Campero Durand, Eike Schallehn, and Gunter Saake

Otto-von-Guericke-University of Magdeburg,
Universitaetsplatz 2, Magdeburg, Germany
{xiao.chen,kirity.rapuru,campero,eike,saake}@ovgu.de

Abstract. During the last decade, several big data processing frameworks have emerged enabling users to analyze large scale data with ease. With the help of those frameworks, people are easier to manage distributed programming, failures and data partitioning issues. Entity Resolution is a typical application that requires big data processing frameworks, since its time complexity increases quadratically with the input data. In recent years Apache Spark has become popular as a big data framework providing a flexible programming model that supports in-memory computation. Spark offers three APIs: RDDs, which gives users core low-level data access, and high-level APIs like DataFrame and Dataset, which are part of the Spark SQL library and undergo a process of query optimization. Stemming from their different features, the choice of API can be expected to have an influence on the resulting performance of applications. However, few studies offer experimental measures to characterize the effect of such distinctions. In this paper we evaluate the performance impact of such choices for the specific application of parallel entity resolution under two different scenarios, with the goal to offer practical guidelines for developers.

Keywords: Entity resoution · Apache Spark · Parallel computation
High/low-level APIs

1 Introduction

Undoubtedly, we are in the era of data. Along with computers being a pervasive companion in our daily lives, and Internet services connecting the world, data volumes and their variety have exploded in recent years. In this situation, big data processing frameworks are used to fulfill the needs of processing such large-scale data and supporting analysis tasks. With the help of such frameworks, developers can abstract the complexities of distributed programming, failure handling and data distribution, allowing them to focus on core domain tasks. Entity Resolution (ER) is a typical application that requires big data processing

M. Elloumi et al. (Eds.): DEXA 2018 Workshops, CCIS 903, pp. 76–87, 2018.
https://doi.org/10.1007/978-3-319-99133-7_6

frameworks. The reason is as follows: Its task is to identify digital records that refer to one real-world entity for one input dataset or to link and merge them when more than one input dataset exists. Its straightforward and common solution is pair-wise ER, which means it compares all possible combinations of input records and calculates their similarity scores. Based on their scores we can decide whether the pair of records refer to the same entity or not. As a result ER can be a time consuming process, which can be treated as a Cartesian Product on the input datasets. Therefore, we can see that ER is a typical case that requires big data processing frameworks, comparing to other applications that do not need quadratic time to perform.

Therefore, so far there has been a large majority of research that explores using big data frameworks to support parallel ER [3]. In recent years parallel ER using Apache Spark has received attention from the data management community. Spark is one of the most popular frameworks nowadays. Compared to earlier frameworks, such as MapReduce, its programming model is more flexible without the need of abstraction to "map" and "reduce" phases and it could provide good speeds by supporting in-memory computation without storing intermediate results to disk. In addition, MapReduce provides low-level programming, while Apache Spark supports both low-level and high-level programming because of the integration with the SQL library. However, almost all existing Spark-based parallel ER research uses Spark's low-level API: the RDD API [10,12,14], and to date, there's little research tackling parallel ER through the high-level APIs: DataFrame and Dataset (available for Scala and Java language. In this paper, APIs mean APIs for Java language) except for our research in [4], which introduces a detailed ER process using DataFrame API. In Apache Spark 2.0, Dataset and DataFrame APIs are unified and DataFrame is considered as Dataset<Row>. In this paper, we implement parallel ER with all three APIs and compare their performance under two different scenarios. These two scenarios mean two parallel ER processes: one has five steps including its last step: the evaluation step, the other one contains only the same first four steps without the evaluation step. The reasons why we have such two scenarios will be explained in Subsect. 3.2.

The rest of the paper is structured as follows: In Sect. 2, we provide some necessary knowledge of Apache Spark. In Sect. 3, we describe our employed ER process, and then explain our implementation considerations. Consequently, in Sect. 4, we introduce our experiments to compare and discuss three implementations. We conclude our paper in Sect. 5.

2 Apache Spark

Apache Spark is designed for fast and general processing of large-scale data. It supports acyclic data flow and in-memory computing, which make Spark run programs up to 100 times faster than Hadoop MapReduce in memory, or 10 times faster on disk [1]. Its main abstraction is resilient distributed data (RDD), which corresponds to the first kind of API in Spark: the RDD API, is a collection of immutable data partitioned across the nodes of the cluster that can

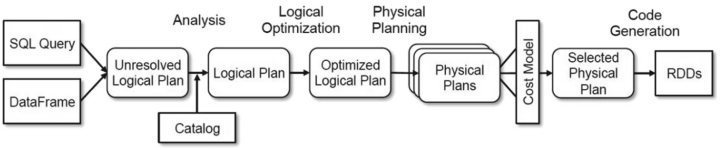

Fig. 1. Phases of query planning in Spark SQL [2]

be operated on in parallel, supports in-memory computing and provides fault tolerance. RDD supports 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. 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, which differentiate them from the location to persist data (memory or disk), whether to serialize data before persisting.

Spark also integrates four libraries: Spark SQL, MLlib for machine learning, GraphX for graphs and graph-parallel computation and Spark streaming for building scalable fault-tolerant streaming applications [1]. Spark SQL is the module of Apache Spark for structured data, which enables developers to query structured data inside Spark programs. Before Spark 1.6, there were only two APIs: RDD and DataFrame. The RDD API corresponds to the low-level API in Spark and the DataFrame API is the structured API in Spark SQL, which can benefit from a series of optimizations. Figure 1 shows the four phases using its cost-based optimizer Catalyst to optimize the application. It first analyzes a logical plan from references and optimize it, then it chooses the best physical based on costs, and last generates code to compile the query to Java code [2]. Besides, it also has a columnar storage called Tungsten, and is able to use Kryo serialization or Encoder to replace traditional Java serialization to minimize storage cost and improve efficiency [9]. However, a disadvantage of using the DataFrame API compared to the RDD API is that the DataFrame API has no type-safety for analysis errors, i.e., analysis errors cannot be caught during compile time. In order to overcome this shortcoming and at the same time keep the advantage of those optimizations, the Dataset API was introduced in Spark 1.6. Therefore, there are three kinds of APIs after Spark 1.6. In Spark 2.0, two structured APIs: the DataFrame API and the Dataset API are unified to a single Dataset API, and the DataFrame API is named Dataset<Row>, and the return type of a SQL query or SQL-based API is Dataset<Row> [1]. Spark does not force users to decide between a relational or a procedural API, since Spark SQL supports both of them [2].

3 Parallel Entity Resolution with Three Spark Java APIs

In this section, we first present the ER process used to compare the performance of different implementations in Subsect. 3.1. Next, we introduce several global design considerations in Subsect. 3.2.

3.1 The Entity Resolution Process Used for Comparison

To compare the performance of each implementation, we employ the following ER process [5]. The first step is data preprocessing, it is used to clean and standardize the input data. As a stand-in for more complex pre-processing we simply added null value replacement, for easing the similarity calculation.

After preprocessing, blocking is often adopted, which splits the whole input dataset into blocks and allows only to compare records within the same block, to reduce the search space for ER such that the corresponding computational complexity can be improved. Standard blocking techniques are used in our approach. In this step we first define a blocking key. Then, records with the same key are assigned to the same block. Standard blocking is straightforward, efficient and able to achieve reasonable reduction ratio and completeness. The selection of a suitable blocking key is essential for standard blocking, and it is a trade-off to choose it in a more general vs. specific way, based on requirements for efficiency and completeness [11]. In our experiments for datasets with personal information, we experimentally tested different blocking keys and the following blocking key is defined and determined as our final blocking key: first applying the double metaphone algorithm on attributes "surname" and "given_name" as the blocking key. The reason for using double metaphone is that, the double metaphone algorithm is designed to capture common transcription mistakes that people might make when recording information based on what they hear from speakers. In this algorithm, the letters that share a similar pronunciation are transformed to the same representation, this helps to recognize true matches. The decision of using this blocking key is because it reduces the search space for pair-wise comparison by a factor of 3000 and at the same time reaches a satisfactory completeness, which is a balanced solution for both goals.

The third step, pair-wise comparison, is the essential one of the ER process, which also turns ER to a compute-intensive and time-consuming task. Before we calculate the similarity between two records of a pair, we need to first get all candidate pairs, whose similarities are needed to calculate. This can be achieved by a join operation using the blocking key as the join attribute. During this operation, almost half of the pairs can be removed due to the transitive property, for example, the result of the join operation contains two pairs (a, b) and (b, a), one of them can be removed, because the similarity calculation is transitive. Then we have the final candidate pair set. The next step is to calculate the similarities of all pairs. Almost all datasets have more than one attribute, similarity functions need to be applied on each attribute to get a total comparison score afterwards. Similarity functions should be chosen based on the attribute properties. In our case, we applied Jaro-Winkler distance for such attributes that

are strings (including number strings), because Jaro-Winkler has been proved to be a good and efficient edit-distance metric for short strings, such as for name matching [7]; While for numbers, absolute difference functions are chosen because of their similarity and understandability.

After all intermediate results are obtained from pair-wise comparisons, the classification step is executed to decide whether each pair of records refers to the same real-world entity or not. In the classification phase, we classified each record pair to match or non-match based on the similarity scores of their attributes using a threshold-based method, which sums up all similarity scores to a total score and judges whether the score is higher than the threshold or not. If the score of a pair is higher than the threshold, this pair would be recognized as a match pair. In our experiments, 0.75 is determined because it serves the best trade-off between precision and recall. After the classification step, we save the results, which contain all candidate pairs, their similarity scores and the match or non-match decision.

The last step is evaluation. This step is optional based on the user's requirement or the availability of a ground truth. For our performance comparison, we take both scenarios into consideration, i.e., an ER process with evaluation or without evaluation. The reason will be explained in the next subsection. For the scenario with evaluation, we evaluate precision, recall and F-measure of our results. The ground truth in our synthetic data is provided by their record IDs. We count the number of true positive, false positive and false negative through comparing our obtained match or non-match result to the analyzed ground truth. Then precision, recall and F-measure are calculated based on true positive, false positive and false negative values, and we also save their values.

3.2 Global Design Considerations

Next, we will list several global considerations to design the comparison of Spark ER with three APIs.

Choice of Two ER Scenarios for Evaluating Impact of Data Reuse: The reason, why we have two aforementioned scenarios in this paper: one with the evaluation step (Scenario 1) and one without the evaluation step (Scenario 2), is two fold. On one hand, two processes stand for two kinds of use cases. In some cases, people are concerned about only an approximate result of which records refer to the same entity without the need of knowing the exact precision and recall. Another case is without available ground truth, it is not possible to have the evaluation step. These two cases lead to an ER process without an evaluation step. For other cases, in which people need to know the result quality of ER and the ground truth is available, they lead to the ER process with an evaluation step. Talking a technical perspective, an ER process with an evaluation step involves the case of data reuse, while the process without an evaluation step does not involve data reuse. We want to know how data reuse affects the performance of different APIs.

Principle for DataFrame- and Dataset-Based Implementation: High-level APIs DataFrame and Dataset in Spark have been unified to Dataset API in version 2.0 for ease of learning, in which DataFrame was considered as a special Dataset with a row type and renamed to Dataset<Row> [1]. Nonetheless, for ease of expression in this paper we still use DataFrame API to stand for API of Dataset<Row>, while Dataset API means Dataset<T> API not including the special Dataset<Row> API. The difference between Dataset API (Dataset<T>) and DataFrame API (Dataset<Row>) is explained in the following. The return type of all SQL queries or SQL-like operation is Dataset<Row>, i.e., DataFrame, our DataFrame-based implementation is actually SQL-based implementation. And for our Dataset-based implementation, after we loaded the data as Dataset<T>, we don't use any API of Dataset<Row>, but other possible APIs for the general Dataset<T>.

Optimizations on Each Implementation: We optimize each implementation by tuning the following parameters: the level of parallelism and possible persistence options. Choosing a suitable level of parallelism for RDD is crucial to reach a good performance. The default level of parallelism for RDD is based on the input data size. For an ER task, for the case that the input data size is small, the data size for pair-wise comparison can be very large. Therefore, if the level of parallelism is determined by the input size, it cannot fulfill the parallelism requirement for steps after blocking. As a result, we have to tune the level of parallelism parameter to reach a good performance (After test experiments on different levels of parallelism are conducted, 320 is finally chosen for the RDD-based implementation). For Dataset-based APIs, the default level of parallelism is 200, which is proved to be a sufficient number in our experience and needs no specific tuning considerations.

Regarding persistence options, for Scenario 1 with the evaluation step, which involves data reuse, we designed different persistence options, which persist different relevant data. These persistence options are tested experimentally and for performance comparison, we take the best persistence option for each implementation and compare their runtime. To achieve the best performance in Spark, the persistence level taken is MEMORY_ONLY for all experiments. For Scenario 2 without the evaluation step, because it does not involve any data reuse case, persistence is not necessary.

4 Experiments

4.1 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; on the other hand, we demonstrate our cluster based on Hortonworks Data Platform (HDP).

Table 1. Datasets used in experiments

Datasets	Name	Input size
1	$5 * 10^5 + 5$ %	57.3M
2	$5 * 10^5 + 50$ %	79.9M
3	$10^6 + 5$ %	114M
4	$10^6 + 50$ %	160M

Datasets. We use synthetic datasets for our experiments. By using synthetic datasets, we can easily know the ground truth and we can control the property of the datasets, such as sizes or duplicate percentages of datasets. In addition, our generated data is based on real-world data and can follow similar characteristics to the real data [6]. It is generated by a data generator called GECO [13]. GECO consists of GEnerator and COrruptor, which is specifically designed for generating ER datasets. For each record, an identifier is assigned, which can show the ground truth of the dataset. For the sake of data diversity and being able to cover different cases in reality, 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 parts mean the number of original records that a dataset contains, such as 10^6 means there are 1 million original records. The second part stands in for the duplicate percentage based on the original records, such as 5% means the number of duplicates are 5% of the number of original records and are inserted into the dataset. Because of privacy reasons, it is very hard to find real datasets that contain personal information.

Cluster on Hortonworks Data Platform. Our cluster used for all experiments is deployed by HDP (Hortonworks Data Platform)-2.6, which is an open source Apache Hadoop distribution based on a centralized architecture (YARN) [8]. The cluster has ten hosts, which includes one node to manage 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 MapReduce, Hive, Tez and Spark applications. Each of them runs on virtual machines, whose hyper-visor is VMWare ESXi. Each host has four CPU cores, 16 GB RAM and 150 GB hard disk. However, our cluster is heterogeneous, since four 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 with a star topology. We submitted our Spark applications with jar files to our cluster and conducted a series of experiments to evaluate the runtime for different frameworks. The corresponding Spark version is 2.2.0 and Java version is 1.8.0. To

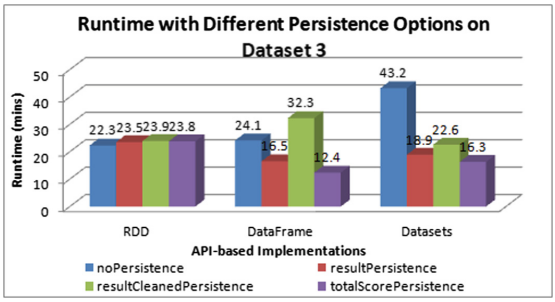

Fig. 2. Overview of runtime with four different persistence options on dataset 3

ensure a successful run of all submissions and that constant cluster resources are used for each submission, we have the following configurations: The driver has 1 GB memory for all experiments. Each executor is allocated with 6 GB memory, 2 GB space for executor memory overhead and 4 cores. We run each experiment five times and after dropping the highest and lowest result, our final result was obtained by averaging the remaining three results.

4.2 Scenario 1: With the Evaluation Step

Persistence Options for Three Implementations: For the first scenario, before we compare the runtime for each implementation, we need to test those persistence options we designed to find the best persistence option for all implementations. The persistence options are designed with the help of analyzing our ER process and observing the Spark web UI. Straightforwardly, data that is involved in multiple actions needs to be persisted to avoid repetitive computation. For our ER process, Result data which contains the pair-wise comparison results and ResultCleaned Data which contains the ground truth that can be used for the evaluation are data that need to be persisted with this straightforward thinking. If such data is not persisted, each time a related action is trigged, they have to be obtained through a series of transformations from the original input data. The TotalScore data is not involved by considering the above explained straightforward thought, but we consider to persist it because of the following consideration: The number of rows in totalScore and result data is quite large, the extra time needed due to persisting result data should be much more than that for totalScore data. Therefore, we have the assumption that persisting totalScore data may have a better effect than straightforwardly persisting result data. Based on the explanations above, we designed the following four persistence options: noPersistence, resultPersistence, resultCleanedPersistence and totalScorePersistence. We submitted all ER implementations with all four persistence options to our entire cluster (seven executors) to record all the runtime on datasets to find the best persistence option for them.

Figure 2 shows an overview of runtime on three implementations with four different persistence options on dataset 3 for the scenario with the evaluation step. The results on other datasets show a similar trend, therefore, here we only show the result on dataset 3 due to space limitation. Surprisingly, there are no obvious performance difference with those four persistence options for the RDD-based implementation. Even with persisting the relevant data, Spark runs the RDD-based implementation in the same way, which leads to similar runtime. And persisting data for the RDD-based implementation cannot avoid overhead, for the RDD-based implementation, we did not use any persistence for it. In contrast, DataFrame- and Dataset-based implementations can benefit from persistence. As we can see from Figure 2, by persisting result data, an improvement of a factor up to 1.5 and 2.3 for DataFrame-based and Dataset-based implementation can be achieved, respectively. By persisting totalScore data instead of the straightforward choice: result data, efficiency improvement can reach a factor up to 2x and 2.7x for them. We conclude that 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 previous data when the computation between them is not time-consuming and their data sizes differ much, it is normally beneficial to persist the former one instead. Therefore, we take persisting totalScore data as the best persistence option for DataFrame- and Dataset-based implementations. Since the RDD-based implementation cannot benefit from persistence options, for runtime comparison we also took DataFrame- and Dataset-based implementations without any persistence into consideration.

Runtime Comparison. Based on the above discussion on persistence options, for Scenario 1, we have five different cases to compare their runtime: RDD-based without persistence, DataFrame-based without persistence, DataFrame-based with best persistence, Dataset-based without persistence, Dataset-based with best persistence. Each case is submitted to our Spark cluster with seven available nodes on different datasets and their corresponding runtime is recorded. Figure 3 shows the comparison result. As we can see from it, the DataFrame-based implementation with best persistence is the most efficient one, which is up to 2.5x and 1.6x faster than the RDD-based implementation and the Dataset-based implementation, respectively. However, if without persisting any data, the RDD-based implementation has a similar speed as the DataFrame-based implementation, and sometimes is even slightly faster than the DataFrame-based implementation, but both of them are much faster than the Dataset-based implementation. In conclusion, by using a proper persistence option, the DataFrame-based and the Dataset-based implementation are able to outperform the RDD-based implementation due to the obtained benefits from persistence. The RDD-based implementation did not change its running plan with the persistence options and cannot get benefits from persistence in Scenario 1.

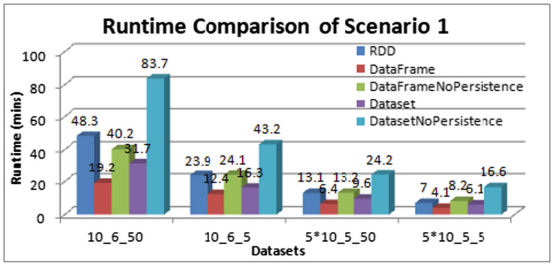

Fig. 3. Runtime comparison for Scenario 1

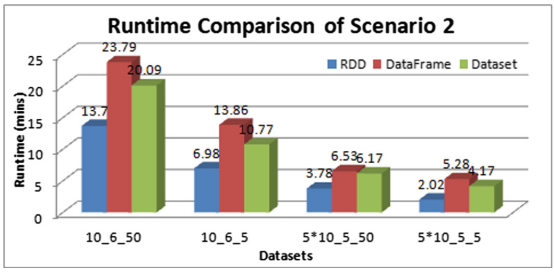

Fig. 4. Runtime comparison for Scenario 2

4.3 Scenario 2: Without the Evaluation Step

For the second scenario, our ER process does not include the evaluation step and we removed this step from Scenario 1 and kept the rest of steps as Scenario 2. Since there is no data reuse in Scenario 2, it is not necessary to have any persistence options. We submitted three implementations to our 7-nodes-cluster on the four datasets introduced in Table 1. Figure 4 shows the results. For all datasets, it shows a consistent result: among three implementations, the RDD-based implementation is the fastest, up to 2.6x and 2.1x faster than the DataFrame-based and the Dataset-based implementation, respectively. The DataFrame-based implementation is the slowest, even slower than the Dataset-based implementation, which is the opposite side to Scenario 1, when we consider the best persistence case for all of them. In conclusion, expected physical and logical optimizations do not help the DataFrame-based and the Dataset-based implementations much. The RDD-based implementation shows the best efficiency for running a general ER process without the evaluation step (without data reuse).

4.4 Threats to Validity

Our experimental results can be subject to several critical considerations.

As internal threats, our results might be affected by unstable network transmission speed between cluster nodes, or other factors in resource usage that

influence the causal behavior we observe between performance and choices for API/persistence configurations.

As external threats, first the synthetic datasets facilitate the ER process and the algorithms we use, specially they lead to a specific evaluation process; if we select another dataset, the results might be different. Our choices for calculation of similarity, blocking and threshold-based matching are selected to represent performance-wise a general use case, they also match well the datasets we use. Other configurations, using more complex similarity measures, different blocking techniques, and an alternative matching process (e.g., with classification) could lead to different performance observations for our experiments. Secondly, our cluster is heterogeneous and with the limited number of nodes available, which restricts the possible interpretation regarding speed-up and scalability for large scale applications. In our experiments, results indicate a promising trend that would have to be verified for large-scale settings. The cluster is deployed on HDP 2.6, the Spark version 2.2 and Java version 1.8.0 are installed on it. The results may change when using different platforms, Spark or Java versions. Third, finely tuned implementations might possibly lead to other observations, but for generality we adopted a straightforward solution.

5 Conclusions

In this paper we compare the performance of parallel ER using three APIs: RDD, DataFrame and Dataset in Spark, for two scenarios of a general ER process. For Scenario 1, the RDD-based implementation runs faster than the other two implementations, without consideration of any persistence option for them. However, the DataFrame and Dataset implementations benefit from persistence and the RDD implementation does not. Therefore, with the best persistence option for DataFrame and Dataset implementations, they are able to outperform the RDD implementation. For Scenario 2, the RDD implementation is the fastest, which conforms to the case in Scenario 1 without consideration of any persistence option.

High-level APIs are expected to be more convenient for developers. Furthermore they can be expected to outperform their low-level counterparts, given the physical and logical optimizations that they can introduce to the code, we observe that this is not the case for parallel ER, since their competitive edge only appears when persistence options are possible. Though it can be expected that future versions of Spark will overcome such limitations, we report a snapshot of the current state-of-the-art in using this framework for ER.

Acknowledgment. This work was supported by China Scholarship Council [No. 201408080093].

References

1. Apache: Apache spark. http://spark.apache.org/. Accessed 10 April 2018
2. 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)
3. 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)
4. Chen, X., Zoun, R., Schallehn, E., Mantha, S., Rapuru, K., Saake, G.: Exploring spark-SQL-based entity resolution using the persistence capability. In: International Conference: Beyond Databases, Architectures and Structures (2018, Forthcoming)
5. Christen, P.: Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. DCSA. Springer Science & Business Media, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31164-2
6. 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). https://doi.org/10.1145/2505515.2507815
7. 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)
8. Hortonworks: Hortonworks data platform. https://hortonworks.com/products/data-platforms/. Accessed 25 June 2018
9. Karau, H., Warren, R.: High Performance Spark. O'Reilly Media, Sebastopol (2017)
10. 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)
11. Papadakis, G., Svirsky, J., Gal, A., Palpanas, T.: Comparative analysis of approximate blocking techniques for entity resolution. Proc. VLDB Endow. **9**(9), 684–695 (2016). https://doi.org/10.14778/2947618.2947624
12. 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)
13. 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). https://doi.org/10.1145/2505515.2508207
14. Wang, C., Karimi, S.: Parallel duplicate detection in adverse drug reaction databases with spark. In: EDBT, pp. 551–562 (2016)