Parallel Entity Resolution with Apache Spark

Min-gang CHEN\textsuperscript{1,2,*} and Hao-jun SUI\textsuperscript{3}

\textsuperscript{1}Shanghai Key Laboratory of Computer Software Testing and Evaluating, Shanghai 201112, China
\textsuperscript{2}Shanghai Development Center of Computer Software Technology, Shanghai 201112, China
\textsuperscript{3}School of Electrical Engineering and Computer Science, Pennsylvania State University, Pennsylvania 16803, USA

Key words: Parallel entity resolution, Apache spark, Inverted index, TF-IDF.

Abstract. Entity resolution (ER) is a problem of matching records from different datasets that refer to the same entities. Accurate and fast entity resolution has broad applications in the field of data management. Although many ER approaches have been proposed, it is still a challenge to design a scalable algorithm to cope with massive datasets. In this paper, we propose a parallel entity resolution algorithm with Spark programming model, which can be executed on the computer cluster in parallel. In our approach, we adopt inverted indices and attribute-related TF-IDF weights of tokens to improve the precision and efficiency of the algorithm. Experimental results show that the proposed parallel ER algorithm achieves good accuracy and performance, and the algorithm is scalable for processing entity resolution of large datasets.

Introduction

Entity resolution (ER) is the problem of identifying, linking, and grouping different manifestations of the same real-world object in the structured or unstructured data. ER is a significant research problem in the field of data management, such as data integration, data cleaning, and data mining [1]. The task of finding matched entity in two different datasets is one of the typical applications of entity resolution, which is a core procedure to the data mining of heterogeneous data [2].

ER techniques usually calculate the similarity between pairs of records, and then compare with a threshold to determine whether two records are matched or not. ER techniques require pairwise comparisons of all records, which are very time-consuming. Therefore, in recent years, researchers have proposed block-based ER techniques, which divide datasets to smaller data blocks according to some kind of features or rules. Then, ER is executed in these blocks to improve the efficiency of the algorithms [3, 4].

ER issues are more challenging in the era of “big data” [5, 6]. First, heterogeneous, unstructured data sets may have different data schemas and manifestations, even exist the problems of data quality. Second, ER algorithms should be scalable and can be executed parallel on the computer cluster. Third, ER algorithms should be efficient in the time-space complexity and communication overhead in order to find matched entities in large scale datasets. Classic ER algorithms focus on the effectiveness of entity recognition, in other words, how to identify the objects that describe the same entity. As far as we know, the research of parallel, scalable ER algorithms for big data is still not much.

This paper proposes a parallel inverted indices-based ER algorithm, which regards entity matching as a record (string) similarity measure. The proposed parallel ER algorithm uses Spark programming model [7] to take advantage of cluster’s computing power to handle large datasets. The main contributions of this paper are as follows:

(1) A parallel, scalable ER algorithm has been proposed and implemented, which can be run on the big data analytics platform of Apache Spark;

(2) The ER algorithm adopted inverted indices data structure and attribute-related TF-IDF weights for each token, which improved the precision of the algorithm.
**Algorithm of Parallel Entity Resolution**

**Description of ER Issue**

The issue of ER in the paper can be defined formally as follows: (1) Data source set \( D = \{d_1, d_2\} \); (2) Entity set \( E = \{e_1, e_2, \cdots, e_{|E|}\} \); (3) Record set \( R = R_1 \cup R_2 \), \( R_1 = \{r_1, r_2, \cdots, r_{|R_1|}\} \), \( R_2 = \{r_1, r_2, \cdots, r_{|R_2|}\} \); (4) Attribute set \( A = \{a_1, a_2, \cdots, a_{|A|}\} \). The input of ER is the record set \( R \) from the heterogeneous data source \( d_1 \) and \( d_2 \), and the output is the matched entity pair set \( E' = \{e_{1\cdots k}(r_i, r_j), r_i \in d_1, r_j \in d_2, k \leq \frac{|E|}{2}\} \). This paper uses Amazon product listing and Google shopping list as two datasets, and data schema is shown in Figure 1. Amazon dataset contains 1363 records, and Google dataset contains 3226 records.

![Figure 1. Data schema of Amazon product list and Google shopping list.](image)

The core of ER algorithm is the computation of the feature’s similarity between \( r_i \) and \( r_j \), and usually the attribute set \( A \) is converted to features. If the similarity is greater than a certain threshold, these two records are considered to belong to the same entity, namely these two records are matched.

**Algorithm Pipeline of Parallel Entity Resolution**

The goal of ER algorithm is to identify the matched entities in two different sources of datasets. The pipeline of parallel ER algorithm consists of four parts—data loading and preprocessing, TF-IDF computing of records, similarity measure of records, and evaluation and analysis of the algorithm, as shown in Figure 2.
**Data Loading and Preprocessing.** The first step of ER algorithm is data loading and preprocessing, and its result are RDDs of key/value pairs. For each record of data sets, we parse the record to the form of key/value pairs, where the key is ID and the value is a string consisting of the title, description, and manufacturer from the record. The code fragment of this step is as follows:

```scala
loadedData = sc.textFile(filename, 18)
    .map(parseDatafileLine)
    .cache()
```

*textFile* loads data into Spark and create an RDD with eighteen partitions. *parserDatafileLine* parses each record to the form of key/value pairs.

**Attribute-Related TF-IDF Computing.** The core idea behind the ER algorithm is to use record's attribute as its features to measure the similarity between record $r_i$ and $r_j$. We adopt TF-IDF method, which is attribute-related, to measure the weight of each token in the Bag-of-Words. The weight vector is regarded as the record's attribute vector. We first transform the record (string of characters) into tokens using regular expression. Along with this process, we also cross out meaningless stop words (e.g., the, a, is, to).

Our next step then, is to measure the TF (Term Frequency) value for each token. Taking account of the importance of title to the actual contents of the record comparing to the one of description, we multiply a constant to the TF value if the token appears at least once in the title. In this paper, the constant number is 2. Hence, the TF value of the token $t$, which appears in the title is $TF(t) = \frac{2n}{n}$; the TF value of the token $t$, which never appears in the title is $TF(t) = \frac{n}{M}$ . In these two formulas, $n$ is the number of times a token appears in the whole string; $M$ is the total amount of the tokens in the string. Thus, if a token appears more times in a record or it appears in a more important attribute, it has larger weight.

IDF is the short hand for Inverse Document Frequency. $IDF(t) = \log(\frac{N}{n(t)})$ , $N$ is the total number of records in these two datasets, and $n(t)$ is the number of times token $t$ appears in $N$ documents. Therefore, if a token appears less times in the records among all datasets, that token has larger weight. The TF-IDF weight of a token in a record is the product of the TF weight and IDF weight, namely $TF-IDF = TF(t) \times IDF(t)$.

**Computing Record Similarity by Inverted Indices.** The naive ER algorithm requires pairwise similarity measurement among all records which has quadratic time complexity. However, in fact, most pairs of the records do not share common tokens. Therefore, we adopt inverted indices data structure to reduce the algorithm running time to be linear. The specific procedures of the algorithm are shown in Algorithm 1. We first design an *invert* function, which will return lists of
(token, id / url) with given (id / url, TF-IDF weighted token vector), namely inverted indices. Next step then, is to construct computeCommonTokens function, which uses invert function to transform Amazon and Google's TF-IDF weight records into inverted indices, and construct a common token list, which has the format [(id / url), <commonTokensList>], by Spark's transformation methods. In the end, we use recordSimilarity function to measure the similarity between Amazon record and Google record.

The time complexity of this algorithm will be O((m + n)z + max(mn', mn')), in which m is the number of records in Google dataset, n is the number of records in Amazon dataset, z is the average number of tokens in one record, m' is the number of records in Google dataset, which share common tokens with some records in Amazon dataset, and n' is the number of records in Amazon dataset, which share common tokens with some records in Google dataset.

### Algorithm1: Record Similarity Computing by Inverted Indices

1: Inputs:
   - record, an RDD representing the record’s weights of TF-IDF.
   - swap(recordPair), a user-defined function which swaps (token,(ID, URL)) to ((ID, URL), token).
   - cosineSimilarity(commonTokens): a user-defined function which computes the cosine similarity between amazon record and google record.
2: Outputs
   - recordSimilarities, an RDD representing the similarities between amazon record and google record.
3:
4: function invert(record)
5:   invertedPairs ← A list of pairs of token to ID
6:   emit invertedPairs
7: end function
8: function computeCommonTokens(record)
9:   amazonInvertedPairs ← amazonWeights.flatMap(invert).cache()
10:  googleInvertedPairs ← googleWeights.flatMap(invert).cache()
11:  commonTokens ← amazonInvertedPairs.join(googleInvertedPairsRDD)
12:   .map(swap).groupBy().cache()
13:  emit commonTokens
14: end function
15: function recordSimilarity(commonTokens)
16:   recordSimilarities ← commonTokens.map(cosineSimilarity).cache()
17:   emit recordSimilarities
18: end function

### Evaluation and Discussion

We evaluated the effectiveness of the algorithm by “gold standard,” and analyzed the running time of the algorithm under the different level of parallelism. In our experimental environment, Spark was running on a YARN cluster of Cloudera, and the version of Spark was 1.5.0. We deployed four servers in the cluster, in which one server was used as driver node and the other 3 servers were used as worker nodes. We set the running environment parameters as following, --num-executor=3, --executors-cores=2, --executor-memory 2g.

We adopted the common measures of information retrieval, which are accuracy, recall, and F-measure, to evaluate the parallel ER algorithm. Precision is the ratio of the true same entities to the marked same entities,

\[
Precision = \frac{True \ same \ entities}{Marked \ same \ entities}
\]

Recall indicates the proportion of the true same entities that is found in all the true same entities,
Recall = \frac{\text{True same entities}}{\text{All the true same entities}}

F-measure is the harmonic mean of the precision and the recall,

\[ F - \text{measure} = 2 \cdot \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \]

The precision, recall and the F-measure are the function of similarity threshold. We set different similarity value (0-1), and the precision, recall and the F-measure are shown in the Figure 3.

Spark creates a task for each partition, and then schedule and run of tasks on the cluster. Each task is executed by a core in the cluster. The level of parallelism determines the data partition and task's scheduling, so it is an important parameter for the optimization of Spark application. The reasonable level of parallelism can make full use of the parallel computing power of the cluster [8].

We recorded the execution time of the algorithm under different levels of parallelism, and the experimental results are shown in Table 1. We found that with the increase of the level of parallelism from 1 to 18, the running time of the algorithm was getting shorter and shorter, and the performance was optimal when the level of parallelism was set with 18. However, when the level of parallelism was too large, such as 1024 in our experimental environment, the performance of the algorithm was reduced because of the overhead of the switch of task's threads. In fact, the Spark official document also recommends 2-3 tasks per CPU core in the cluster.

| Level of Parallelism | 1  | 3 | 6 | 18 | 64 | 256 | 512 | 1024 |
|----------------------|----|---|---|----|----|-----|-----|------|
| Running Time (s)     | 302| 159| 130| 103| 109| 113 | 167 | 285  |

**Conclusion**

Due to the widespread existence of complex, heterogeneous data, ER has important applications in the field of data management. This paper proposes and implements a parallel, scalable ER solution on Spark platform, describes the detail process of the algorithm, and analyzes the complexity of the algorithm. Experiment results show the effectiveness of the algorithm. The paper also analyzes the efficiency of the algorithm by setting a different level of parallelism. When the scale of the data sets becomes larger, the parallel ER can meet the requirements of efficiency by allocating more computing nodes in the Spark cluster.
The parallel ER solution can only cope with static datasets and cannot support the update of the inverted indices in real time. In the future work, we will study the incremental ER algorithm to support the fast processing of dynamic heterogeneous datasets.

Acknowledgments

This work was supported in part by Science and Technology Commission of Shanghai Municipality Program (No.16511101202), and the National Natural Science Foundation of China (No.61502299).

References

[1] L. Getoor and A. Machanavajjhala, “Entity resolution: theory, practice & open challenges”, Proceedings of the VLDB Endowment, vol. 5, no. 12, (2012), pp. 2018-2019.

[2] X. Ren, A. Kishky, C. Wang, F. Tao, C. R. Voss, H. Ji and J. Han, “Clustype: Effective entity recognition and typing by relation phrase-based clustering”, Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, Australia, (2015) August 10-13.

[3] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederee and W. Nejdl, “A Blocking Framework for Entity Resolution in Highly Heterogeneous Information Spaces”, IEEE Transactions on Knowledge & Data Engineering, vol. 25, no. 12, (2013), pp. 2665-2682.

[4] B. Kenig, A. Gal, “MFIBlocks: An effective blocking algorithm for entity resolution”, Information Systems, vol. 38, no. 6, (2013), pp. 908-926.

[5] L. Kolb, E. Rahm, “Parallel entity resolution with dedoop”, Datenbank-Spektrum, vol. 13, no. 1, (2013), pp. 23-32.

[6] L. Getoor and A. Machanavajjhala, “Entity resolution for big data”, Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Chicago, USA (2013) August 11-14.

[7] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets”, Usenix Conference on Hot Topics in Cloud Computing, Boston, USA, (2010) June 20-23.

[8] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, “Learning Spark: Lightning-Fast Big Data Analysis”, O'Reilly Media, Inc., California, (2015).