How improve Set Similarity Join based on prefix approach in distributed environment

Song Zhu, Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano
Department of Engineering Enzo Ferrari
Università di Modena e Reggio Emilia
Modena, Italy
{song.zhu;luca.gagliardelli;giovanni.simonini;domenico.beneventano}@unimore.it

***Abstract*—Set similarity join is an essential operation to find similar pairs of records in data integration and data analytics applications. To cope with the increasing scale of the data, several techniques have been proposed to perform set similarity join using distributed frameworks (e.g. MapReduce). In particular, it is publicly available a MapReduce implementation of the `PPJoin`, that was experimentally demonstrated as one of the best set similarity join algorithm. However, these techniques produce huge amounts of duplicates in order to perform a successful parallel processing. Moreover, these approaches do not guarantee the load balancing, which generates skewness problem and less scalability of these techniques. To address these problems, we propose a duplicate-free technique called `TTJoin`, that performs set similarity join efficiently by utilizing an innovative filter derived from the prefix filter. Moreover, we implemented `TTJoin` on Apache Spark, that is one of the most innovative distributed framework. Several experiments on real-world datasets demonstrate the effectiveness of proposed solution with respect to either traditional `TTJoin` MapReduce implementation.**

***Index Terms*—Similarity Join, Big Data, Record Linkage**

I. Introduction

With the increasing of the volume of data (i.e. Big Data) managing and extracting valuable information from these huge amount of data has become a hot research topic both in Academy and Industry. Different techniques have been proposed to create tools that are able to explore [1] [2] [3] and to integrate [4] [5] in a efficiently way high volume of data. One of the most promising approach to find similar records on big datasets is the similarity join. Similarity join is used in a wide range of applications to find similar data, ranging from data integration to marketing analysis. Moreover, it can be used in keyword search engines to perform similarity queries [6] [7]. In many scenarios data can be transformed into sets, for example documents can be see as a set of words, or in market analysis user purchases can be considered as a set of tokens related to a specific user. Thus, computation of set similarity can be used in a many applications.

The set similarity join finds pairs of similar sets from one or more set collections. The join between the same collection is also called *self join*, while the join between two different collections (R and S) is called *RSJoin*. Two sets are similar if their overlap (i.e. number of shared elements) exceeds some user-defined threshold. The efficient computation of set similarity join has received much attention from both academia and industry, and different techniques have been developed. In particular, one of the most famous techniques for set similarity join is the `PPJoin` [8], that is also one of the fastest one [9]. `PPJoin` is an improvement of a previous work based on prefix filter, called `SSJoin` [10]. A MapReduce implementation based on Hadoop of `PPJoin` is presented in [11]. This parallel `PPJoin` implementation has two main issues on large datasets:

- generation of duplicate pairs, that are eliminated at the end of the process, slowing the execution;
- Load balancing, due to different frequency of tokens, causing skewness (i.e., workers can have significant different execution times).

To solve these issues we propose a variant of `PPJoin` that avoid the generation of duplicates in a distributed environment, called `PPADJoin` (Avoid Duplicates) and a new load balancing method. Moreover, we propose a novel filter based on the prefix filter principle called Two Tokens Join (`TTJoin`).

From the experimental point of view, we first show the performance improvement of `PPADJoin` and then we show how `TTJoin` outperforms `PPADJoin` in all experiments.

The rest of paper is structured as follows: Section II presents the related works on `SSJoin` and `PPJoin`. The proposed improvements are illustrated in Section III. Then, experimental results are shown in Section IV. Finally, the conclusions are presented in Section V.

II. Preliminaries

A. SSJoin

`SSJoin` proposed by Chaudhuri et al. [10], is based on overlap similarity (i.e. the elements shared by two records). The overlap similarity is defined as follow:

Definition 1 (Overlap similarity): Overlap similarity. Given two sets s1, s2 (representing two records) we consider the overlap similarity, denoted Overlap(s1, s2) as Overlap(s1, s2) = $|s1 \cap s2|$.

In order to generate less candidate pairs the authors proposed an implementation of `SSJoin` using the `prefix-filter`.

A formal principle of `prefix-filter` is given in [8]. To define the prefix of a set its elements have to be ordered by a given ordering.

978-1-5386-7879-4/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCS.2018.00136

Lemma 1 (The Prefix Filter principle:): Given a global ordering O of the token universe U and a collection of sets, each with tokens sorted in the order of O. Let the p-prefix of a set x be the first p tokens of x. If Overlap(x, y) $\geq \alpha$, then the $(|x| - \alpha + 1)$, prefix of x, and the $(|y| - \alpha + 1)$, prefix of y, must share at least one token. Where α is the requested overlapping.

By using this principle, it is possible to find pairs with at least one common element in the prefix. Hence, it generates fewer pairs than the cartesian product, this means that there are less pairs to verify. Another interesting point of this algorithm is the ordering, since using the rare tokens of a set as prefix tokens can minimize the number of generated candidate pairs. So the order of tokens is usually given by the document frequency (i.e. in how many documents a token appears).

It is also possible to use `SSJoin` with other similarity measures (e.g. Edit distance, Dice, Cosine, Jaccard) that can be re-conducted to the overlap one.

B. PPJoin

`PPJoin` proposed by Xiao et al. in [8] is an improvement of `SSJoin`. It adds a new filter called `position-filter` that prune more efficiently the pairs, reducing the number of candidate pairs. Therefore, the new algorithm use both `prefix-filter` and `position-filter`, it is called `PPJoin`. Also, in the same paper the authors proposed an other filter called `suffix-filter`. However, in [9] it was demonstrated that the `suffix-filter` is not convenient, since it is too complex and expensive in term of time.

C. Other similarity join derived from Prefix-Filter

In literature there are several join techniques derived from `prefix-filter`:

- GroupJoin [12];
- MPJoin [13];
- MPJoin-PEL [14];
- AdaptJoin [15].

Each of these joins has its specific quality and application domains, it is possible to find a comparison of them in [9]. From this evaluation `PPJoin` emerges as the most promising techniques, so we choose to use it for our work.

D. MapReduce and Spark

MapReduce is a programming model and an associated implementation for processing and large datasets with a parallel, distributed algorithm on a cluster of machines [16]. The principle is simple, it uses the "divide and conquer" strategy to process a large amount of data. The large datasets are stored into GFS (Google File System), which is a distributed file system; this means that the dataset is stored into many machines that compose the GFS. With the MapReduce paradigm, large input data are divided into smaller partitions and processed in distributed fashion.

It allows to write distributed programs in a simple way, since it provides an high level of abstraction, hiding the parallelization details, and providing a fault tolerance system. An open source implementation of GFS and MapReduce is available with the name of Apache Hadoop, and its distributed file system is called HDFS.

With Apache Hadoop, the open source community has a powerful framework to develop distributed application. However, the framework has several limitations. Applications such as machine learning and graph analytics iteratively process the data, this means multiple rounds of map reduce iterations are performed on the same data. In MapReduce, every job reads its input data from HDFS, processes it, and then writes it back to HDFS. For the subsequent job to consume the output of a previously run job, it has to repeat the read, process, and write cycle. For iterative algorithms, which want to read once, and iterate over the data many times, the MapReduce model poses a significant overhead. To overcome the above limitations of MapReduce, *Apache Spark* uses Resilient Distributed Datasets (RDDs) [17], which implements in-memory data structures used to cache intermediate data across a set of nodes. Since RDDs can be kept in memory, algorithms can iterate over RDD data many times very efficiently [18]. A performance comparison and analysis between the two frameworks is presented in [18].

E. PPJoin by using MapReduce framework

In [11], authors proposed a MapReduce implementation of `PPJoin`. During map and shuffle phase, the `prefix-filter` is exploited to generate blocks (i.e. cluster of sets): sets that have at least one common token are grouped in the same block. During reduce phase, in each block, the sets are combined to obtain candidate pairs, and the pairs are pruned using the `position-filter`.

This `PPJoin` implementation have two main issues: generates duplicate pairs, and the workload is unbalanced.

a) Generation of duplicate pairs: One of the main problem is the generation of many duplicates, due to multiple overlapping tokens in the prefix.

The problem is presented in Figure 2. Records 1 and 21 have in common tokens A and B. This leads, records 1 and 21 are together in block A and B. Thus, both blocks generate the pair $(1, 21)$.

Duplicates causes performance issues, since their elimination is an expensive operation.

b) Record skew: The main workload of `PPJoin` is in the reducing phase, where the sets are combined and filtered. The computation time of this phase depends by the number of sets contained in each block, more sets are grouped in a block, more time is required to compute that block. The number of sets in the blocks depends by the token frequencies in the prefixes. The workload balancing is a very important task for distributed applications.

III. PPJoin improvements

A. PPJoin variant: Avoid Duplicates

We designed a variant of `PPJoin` to avoid the generation of duplicates, i.e. `PPADJoin` (Avoid Duplicates) As men-

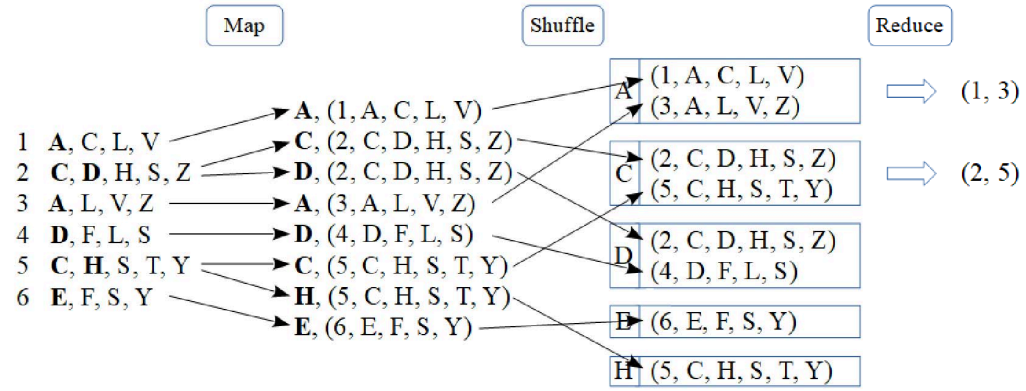


Fig. 1: MapReduce implementation of `PPJoin`.

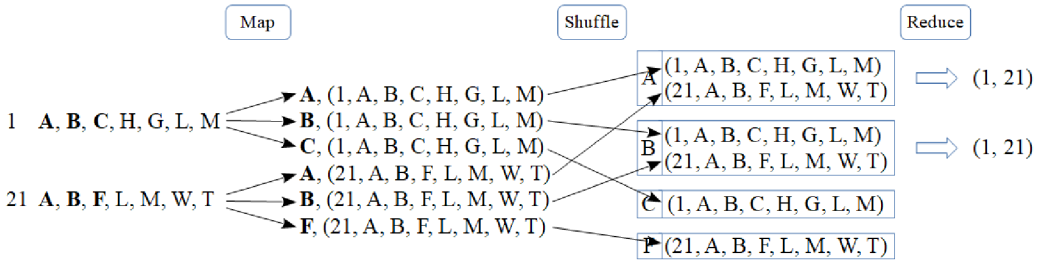


Fig. 2: Duplicate generation example

tioned the generation of duplicate candidate pairs is caused by records that have more than one overlapping token in the prefix. As shown in the example the pair ⟨1,21⟩ is present two times in the results, due to multiple overlap in prefix tokens. Our solution is to generate candidate pairs only from one overlapping prefix token .

To achieve this, when `PPADJoin` combines the records in a block checks if the last common token in the prefix of a pair corresponds to the token that has generated that block, and only if this is true emits the pair. In this way, if two tokens co-occur in more than one block, will be emitted only once, by their last common token. A prerequisite of this approach is that the tokens in the sets have to be sorted based on a global order.

The pseudo-code of candidate pairs generation is shown in Algorithm 1.

B. Load Balancing

In a distributed environment one of the most important issue is load balancing; if the workload is not well distributed there can be skew problems, that means that some workers will finish their jobs before others, remaining idle, wasting computation resources. The operation with the largest computation time is the main target to balance. In the case of `PPJoin` is the candidate pairs generation step, since `PPJoin` performs the partition of records based on the common prefix tokens, these prefix tokens can have very different frequencies. As

Algorithm 1: `PPADJoin` - Candidate pairs generation

```
input : PrefixToken token that generates the block
input : R_i sets of the first dataset (R) that contains the
        PrefixToken. r_i ∈ R_i is (r_id, r_tokens)
input : S_i sets of the second dataset (S) that contains the
        PrefixToken. s_i ∈ s_i is (s_id, s_tokens)
input : t the request similarity threshold
output: CandidatePairs
CandidatePairs ← {}
for r ∈ R_i do
    for s ∈ S_i do
        if passLengthFilter(|r_tokens|, |s_tokens|, t)) then
            lastOverlapToken ←
              getLastOverlap(r_tokens, s_tokens, t)
            if PrefixToken = lastOverlapToken then
                overlaps ←
                  getOverlaps(r_tokens, s_tokens, t)
                if overlaps > 0 then
                    CandidatePairs ←
                      CandidatePairs ∪ (r_id, s_id)
                end
            end
        end
    end
end
```

consequence, the computation time can be vary a lot. To distribute the workload, a custom partitioner is designed. The input of this step are tuples $(t_{id}, [(rid, [tokens], position)])$, since `PPJoin` performs a group by t_{id}, and the results are tuples with a t_{id} and an array of $(r_{id}, [tokens], position)$ with same t_{id}. `PPJoin` has to assign t_{id} to task based on the workload of candidate pair generation. The complexity of this work load is based on the number of comparisons of records of each t_{id}, i.e. NC. For *self join* the number of comparison is:

$$NC = rSize * (rSize - 1)/2 \tag{1}$$

Where $rSize$ is the size of Array.

For *RSJoin* the number of combination is:

$$NC = rSize * sSize \tag{2}$$

Where $rSize$ is the number of records of the first dataset in the array, and $sSize$ is the number of records of the second dataset in the array.

`PPJoin` uses the NCs to check the computation cost to balance workload. The custom partitioner starts to distribute t_{id} with higher NC, and allocates groups of records to the partition with the lowest workload. The workload of each partition is calculated as the sum of NC of t_{ids} already allocated in the partition. The pseudo-algorithm used to distribute blocks is shown in Algorithm 2.

Algorithm 2: Blocks assignments

input : $blocks$ list of blocks with blockID and NC, sorted by NC
input : n number of partitions
output: $blockAssignments$
$blockAssignments \leftarrow \{\}$
$partitionsWorkloads \leftarrow list[n]$
for $p \in partitionsWorkloads$ **do**
  $p \leftarrow 0$
end
for $b \in blocks$ **do**
  $pIndex \leftarrow getIndexWithMinValue(partitionsWorkloads)$
  $partitionsWorkloads[pIndex] \leftarrow partitionsWorkloads[pIndex] + b_{NC}$
  $blockAssignments \leftarrow blockAssignments \cup (pIndex, b_{blockID})$
end

C. TTJoin

`PPJoin` exploits `prefix-filter` and `position-filter`. We propose a new filter method based on `prefix-filter`. `prefix-filter` is based on the analysis of the prefix, the size of the prefix is given by Equation 3.

$$PrefixSize = |x| - \lceil t \cdot |x| \rceil + 1 \tag{3}$$

The equation 3 is employed during the blocking phase, since it is not possible to know a priori the cardinality of y. However, when candidate pairs are generated by the `prefix-filter`, it is possible to use a more restrictive formula to calculate the prefix.

Two Token Filter is designed to be more restrictive, and it is defined starting from the relation between Overlap and Jaccard similarities, demonstrated in [8]. This relation is shown in Equation 4.

$$J(x,y) \geq t \iff O(x,y) \geq \alpha = \frac{1}{1+t} \cdot (|x| + |y|) \tag{4}$$

In the Equation 4: x and y are two ordered sets, $J(x,y)$ is their Jaccard similarity; $O(x,y)$ is their overlap similarity; t and α are the thresholds of Jaccard and Overlap similarity; $|x|$ and $|y|$ are the sizes of sets x and y. α represents the number of minimum overlap to have the condition $J(x,y) \geq t$.

DT (Different Token) is an element in x and not contained in y. $MaxNDT(x)$ is the maximum number of DT that can be present in x without violate the constraint $J(x,y) \geq t$. $MaxNDT(x)$ can be calculated as:

$$|DT(x)| = |x| - |x \cap y| \Rightarrow |x \cap y| = |x| - |DT(x)|$$

Since:

$$O(x,y) = |x \cap y| \geq \alpha \Rightarrow |x| - |DT(x)| \geq \alpha$$

$$\Rightarrow MaxNDT(x) = |x| - \alpha \tag{5}$$

The same reasoning can be applied to the elements of y.

$$MaxNDT(y) = |y| - \alpha \tag{6}$$

If the constraint $J(x,y) \geq t$ is verified, in a subset of x with $MaxNDT(x)+2$ elements, it is possible to deduce that there are at least two elements which have to be present in y. Thus, with x and y ordered, the firsts $MaxNDT(x) + 2$ elements from x (this set of elements is denoted with $TTPrefix(x)$) and the firsts $MaxNDT(y) + 2$ elements from y (this set of elements is denoted with $TTPrefix(y)$) must have at least 2 elements in common.

$$|TTPrefix(x)| = MaxNDT(x) + 2 = |x| - \alpha + 2 \tag{7}$$

$$|TTPrefix(y)| = MaxNDT(y) + 2 = |y| - \alpha + 2 \tag{8}$$

To demonstrate this, considering $|x| \geq |y|$ and $|TTPrefix(x) \cap TTPrefix(y)| < 2$, thus:

$$|TTPrefix(x) \cap TTPrefix(y)| = 1 \tag{9}$$

To satisfy $O(x,y) \geq \alpha$:

$$|x \cap y| \geq \alpha \Rightarrow$$
$$|(x - TTPrefix(x)) \cap (y)| + |TTPrefix(x) \cap y| \geq \alpha \Rightarrow$$
$$|(x - TTPrefix(x)) \cap (y)| +$$
$$|TTPrefix(x) \cap y - TTPrefix(y)| +$$
$$|TTPrefix(x) \cap TTPrefix(y)| \geq \alpha$$

Now, applying the condition of Equation 9 will results:

$$|(x - TTPrefix(x)) \cap (y)| + |TTPrefix(x) \cap y - TTPrefix(y)| \geq \alpha - 1$$

ro (remain overlaps) is the left part of above equation:

$$ro = |(x - TTPrefix(x)) \cap (y)| + |TTPrefix(x) \cap y - TTPrefix(y)|$$

Hence, tokens in x and y are sorted and $|x| \geq |y|$, it is possible to prove that ro cannot be $> max(|(x - TTPrefix(x))|, |y - TTPrefix(y)|)$, since, $|x| \geq |y|$, this means that with the maximum value of ro will be $|(x - TTPrefix(x))|$, thus, to satisfy $O(x, y) \geq \alpha$:

$$|(x - TTPrefix(x))| \geq \alpha - 1 \Rightarrow$$
$$|x| - |TTPrefix(x)| \geq \alpha - 1$$

Since:

$$|TTPrefix(x)| = |x| - \alpha + 2 \Rightarrow$$
$$|x| - (|x| - \alpha + 2) \geq \alpha - 1$$

This demonstrate that it is impossible to satisfy $O(x, y) \geq \alpha$ if $|TTPrefix(x) \cap TTPrefix(y)| < 2$. Thus, TTPrefix(x) and TTPrefix(y) must have at least 2 elements in common to satisfy $J(x, y) \geq t$.

Given a candidate pair of ordered sets (x, y), Two Tokens filter (TTF) checks $TTPrefix(x)$ and $TTPrefix(y)$, discarding the pair if there are no at least 2 overlap elements.

This filter can be used in the reduce phase to cut down the number of candidate pairs.

IV. Experiments

A. Datasets

To test `PPJoin` and its improvements four real-world datasets are used:

- Enron Mails is a dataset of emails which was collected and elaborated by CALO Project (A Cognitive Assistant that Learns and Organizes)[1]. The corpus contains about 500 thousands emails;
- PubMed Abstracts is a collection of abstracts of biomedical publications from MEDLINE[2]. Contains about 14 million abstracts;
- Citations is composed by two datasets which contain data extracted from Citeseer and DBLP. For each publication are reported: title, authors and name of the journal in which was published; however, the title is the only attribute that is always available in all records. The missing information can cause significant variation of the similarity measure, thus we chose to keep only the title for our tests. The $RSJoin$ is tested with this dataset.

Table I resume the datasets characteristics; where *rows* is the number of documents in the datasets, *max* is the maximum number of distinct tokens in a document, *min* is the minimum number of tokens in a documents, and *AVG* and *std* are respectively the average and the standard deviation.

[1]http://www.ai.sri.com/project/CALO

[2]https://www.nlm.nih.gov/bsd/pmresources.html

TABLE I: Datasets statistics

Dataset	Rows	Max	Min	AVG	Std dev
Enron Mails	0.5 million	47064	1	141.01	253.30
PubMed Abstract	13 million	1082	1	104.20	36.49
Citations - Citeseer	1.8 million	51	1	7.67	4.35
Citations - DBLP	2.5 million	38	1	9.24	3.53

Enron Mails is the smallest dataset in terms of number of documents, however each document has 141 on average distinct token. Citations datasets have more documents, through, these datasets have less distinct tokens, whereby, the computation on Citations is the fastest. PubMed Abstracts is the largest dataset, with 13 millions of documents. Since, `PPJoin` cannot complete the process for the whole dataset, a subset of PubMed Abstracts, 20% of the whole dataset, is created to evaluate `PPJoin` ; this subset is named PubMed020.

B. Configuration

The evaluation was performed on a cluster where each node owns two Intel Xeon E5-2697 v4 at 2.3GHz (36 cores overall) and 128 Gb of RAM. As framework Apache spark 2.1.0 is used, the code is developed in Scala 2.11.8.

C. Methods

In the experiments `PPJoin` and `TTJoin` are evaluated on the datasets described in the previous section. Both self join and RSJoin are tested. Enron Mails and PubMed20 datasets are used to test self join, while the RSJoin is executed on Citations.

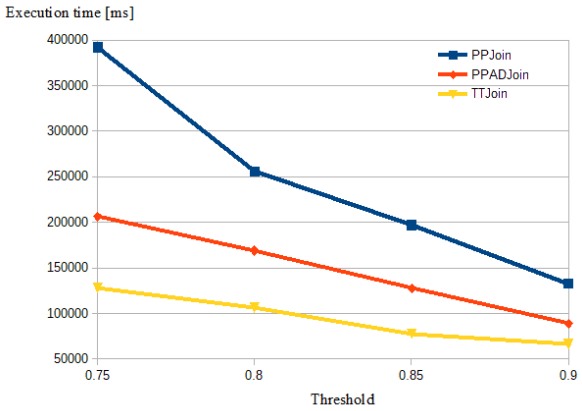
The experiments are performed using Jaccard similarity as similarity measure, however it is possible to use different similarity measures (e.g. Dice, Cosine, Edit, etc.). The comparison tests are executed first varying the threshold, and then with the variation of number of workers nodes, from 2 to 10.

Firstly, two version of `PPJoin`are compared, then more detailed tests are executed on `TTJoin` , evaluating it on the largest dataset, Pubmed Abstract. The scalability is evaluated measuring its execution time from 10 to 20 worker nodes, and also varying the threshold. Furthermore, `TTJoin` is tested with subsets of PubMed Abstract, these datasets are generated by sampling PubMed Abstract by 10, 20 and 50 percent of its original size.

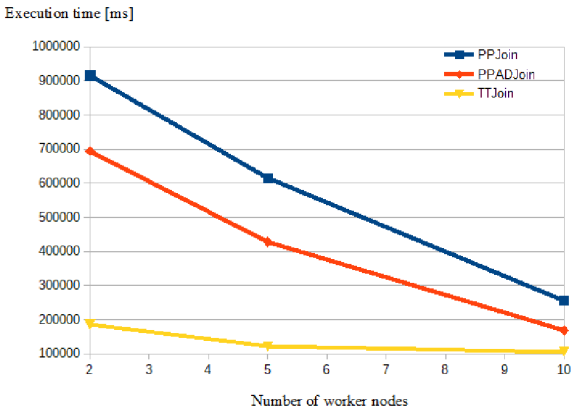
D. Results

Figures 3, 4 and 5 reports the results obtained among the different algorithms. These tests compare the scale-out of examined algorithms and the behaviors of algorithms with variance of threshold. The scalability on small datasets, like Citations, is limited since the workload is small, and the distribution on high number of worker nodes does not increase the performance. While, the scalability is enhanced on larger datasets (Enron Mails and PubMed20).

Varying the threshold it is possible to observe how the workload is reduced with the increasing of the threshold. This

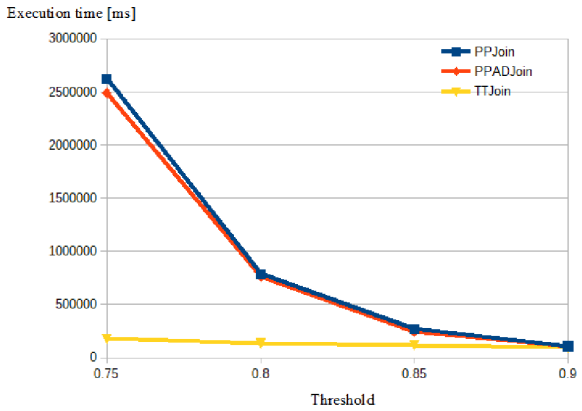


(a) Threshold variation

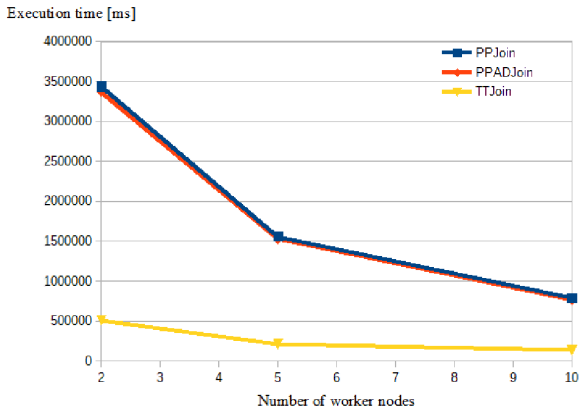


(b) Number of nodes scalability

Fig. 3: Enron Mails execution time



(a) Threshold variation



(b) Number of nodes scalability

Fig. 4: PubMed20 execution time

happens because the size of the prefixes is increased with the lowering of the threshold, causing in an increasing of the workload.

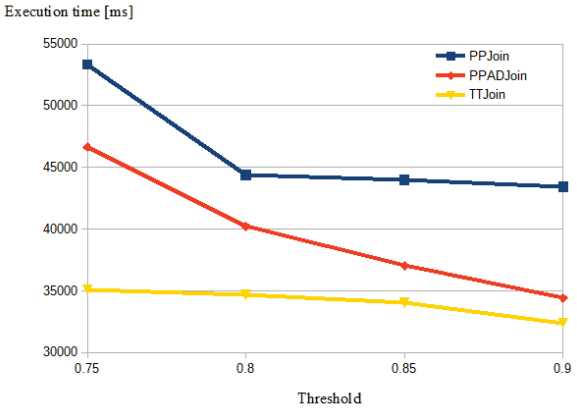
a) PPJoin comparison: `PPJoin` as implemented in [11] is compared with our version that avoids the duplicate generation `PPADJoin`. The results show that `PPADJoin` takes advantage when the number of candidate pairs is higher. On PubMed2020 the number of candidate pairs is small with respect to other datasets, for this reason on this dataset the performance of `PPJoin` and `PPADJoin` are similar. On the other datasets `PPADJoin`outperforms the standard `PPJoin` implementation.

b) TTJoin vs PPJoin: Our new join algorithm `TTJoin`is more efficient with respect the `PPJoin`and `PPADJoin`, since the use of the TTFilter let to prune more pairs, generating less candidate pairs. Furthermore, as shown in Figure 6 `TTJoin` is able to process the whole PubMed Abstract dataset, that cannot be processed with `PPJoin`.

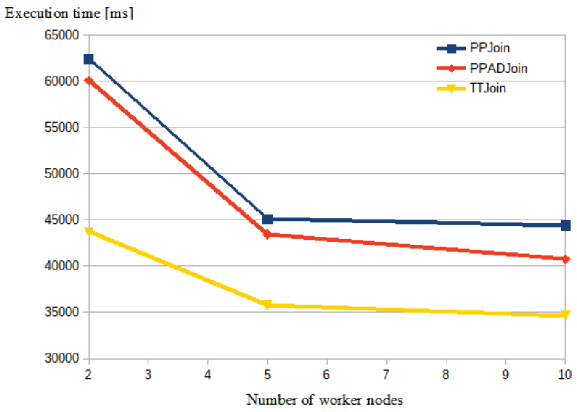
c) TTJoin - scalability: The scalability of `TTJoin` is evaluated on the whole PubMed Abstracts dataset. Figure 6a shows the scalability of `TTJoin` varying the threshold on 10, 15 and 20 worker nodes. With a higher threshold the execution times are very close to each other, this because the workload is very reduced, since the length of the prefix depends from the threshold. With a lower threshold there is a significative improvements at the increasing of the number of workers in term of execution time, since the data to process are much more.

The last evaluation is on the performance of `TTJoin` varying the datasets size. This evaluation is performed with PubMed10, PubMed20, PubMed50 and the whole PubMed abstract. The workload increases when increasing the volume of dataset. The results of tests are shown in Figure 6b. With the increase of volume of data, the performance with high number of worker nodes are better, this is due to limit of resources when the number of worker nodes is low. Thus, with a small dataset the performances are similar, while with a larger dataset, the performances with more worker nodes are higher.

From these experiments it is possible to conclude that `TTJoin` is able to scale almost linearly with the increasing

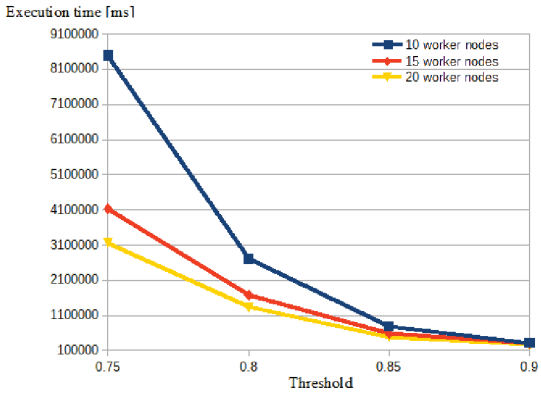


(a) Threshold variation

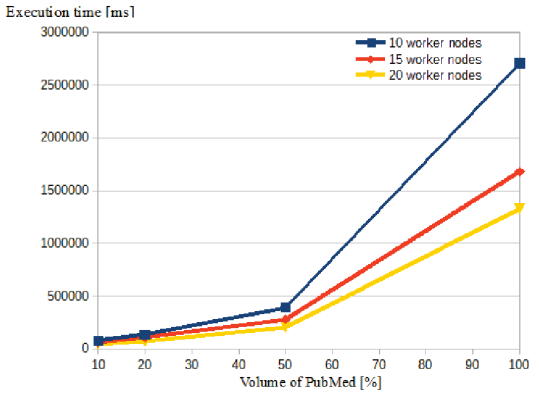


(b) Number of nodes scalability

Fig. 5: Citations execution time



(a) Threshold variation



(b) Number of nodes scalability

Fig. 6: TTJoin scalability

of the worker nodes.

V. Conclusion and Future Works

In this work improvements are proposed to increase the performance of the state-of-the-art of set similarity join. We started from the study and implementation of `PPJoin`, one of the most popular and cited set similarity join algorithm in literature.

Apache Spark is employed to implement `PPJoin`, the scale-out of this implementation is tested by using four real world datasets and compared with proposed solutions `PPADJoin` and `TTJoin`.

Experiments show the performance improvement of `PPJoin` without generation of duplicates. Furthermore, `TTJoin` outperforms `PPJoin` in all experiments. This proves the effectiveness of TTF as filter for set similarity join.

We also demonstrated the scalability of TTJoin with the variation of the volume of dataset and the numbers of work nodes.

In conclusion, our work put forward new methods to improve set similarity join, with the hope it can contribute researches on similarity join optimization.

References

[1] G. Simonini and S. Zhu, "Big data exploration with faceted browsing," in *High Performance Computing & Simulation (HPCS), 2015 International Conference on*. IEEE, 2015, pp. 541–544.

[2] S. Bergamaschi, D. Ferrari, F. Guerra, G. Simonini, and Y. Velegrakis, "Providing insight into data source topics," *Journal on Data Semantics*, vol. 5, no. 4, pp. 211–228, 2016.

[3] G. Simonini, S. Bergamaschi, and H. Jagadish, "Blast: a loosely schema-aware meta-blocking approach for entity resolution," *Proceedings of the VLDB Endowment*, vol. 9, no. 12, pp. 1173–1184, 2016.

[4] G. Simonini, G. Papadakis, T. Palpanas, and S. Bergamaschi, "Schema-agnostic Progressive Entity Resolution." *ICDE '16*, 2016.

[5] F. Benedetti, D. Beneventano, S. Bergamaschi, and G. Simonini, "Computing inter-document similarity with context semantic analysis," *Information Systems*, 2018.

[6] S. Bergamaschi, F. Guerra, and G. Simonini, "Keyword search over relational databases: Issues, approaches and open challenges," in *Bridging Between Information Retrieval and Databases*. Springer, 2014, pp. 54–73.

[7] F. Guerra, G. Simonini, and M. Vincini, "Supporting image search with tag clouds: a preliminary approach," *Advances in Multimedia*, vol. 2015, p. 4, 2015.

[8] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, "Efficient similarity joins for near-duplicate detection," *ACM Transactions on Database Systems (TODS)*, vol. 36, no. 3, pp. 1–41, 2011.
[9] W. Mann, N. Augsten, and P. Bouros, "An empirical evaluation of set similarity join techniques," *Proceedings of the Vldb Endowment*, vol. 9, no. 9, pp. 636–647, 2016.
[10] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *International Conference on Data Engineering*, 2006, pp. 5–5.
[11] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using mapreduce," in *ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, Indianapolis, Indiana, Usa, June*, 2010, pp. 495–506.
[12] P. Bouros, G. Shen, and N. Mamoulis, *Spatio-textual similarity joins*. VLDB Endowment, 2013.
[13] L. A. Ribeiro and T. Hrder, "Generalizing prefix filtering to improve set similarity joins ," *Information Systems*, vol. 36, no. 1, pp. 62–78, 2011.
[14] W. Mann and N. Augsten, "Pel: Position-enhanced length filter for set similarity joins." in *Grundlagen von Datenbanken*, 2014, pp. 89–94.
[15] J. Wang, G. Li, and J. Feng, "Can we beat the prefix filtering?: an adaptive framework for similarity join and search," in *ACM SIGMOD International Conference on Management of Data*, 2012, pp. 85–96.
[16] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "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, 2012, pp. 2–2.
[18] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 13, pp. 2110–2121, 2015.