Scaling Up Set Similarity Joins Using A Cost-Based Distributed-Parallel Framework

Fabian Fier and Johann-Christoph Freytag

Humboldt-Universität zu Berlin, Germany
{fier,freytag}@informatik.hu-berlin.de

Abstract. The set similarity join (SSJ) is an important operation in data science. For example, the SSJ operation relates data from different sources or finds plagiarism. Common SSJ approaches are based on the filter-and-verification framework. Existing approaches are sequential (single-core), use multi-threading, or Map-Reduce-based distributed parallelization. The amount of data to be processed today is large and keeps growing. On the other hand, the SSJ is a compute-intensive operation. None of the existing SSJ methods scales to large datasets. Single- and multi-core-based methods are limited in terms of hardware. MapReduce-based methods do not scale due to too high and/or skewed data replication. We propose a novel, highly scalable distributed SSJ approach. It overcomes the limits and bottlenecks of existing parallel SSJ approaches. With a cost-based heuristic and a data-independent scaling mechanism we avoid intra-node data replication and recomputation. A heuristic assigns similar shares of compute costs to each node. A RAM usage estimation prevents swapping, which is critical for the runtime. Our approach significantly scales up the SSJ execution and processes much larger datasets than all parallel approaches designed so far.

1 Introduction

A major challenge in data science today is to compare and relate data of similar nature. One important operation to relate data is the *join* operation known from relational databases. The join operation finds all record pairs from two tables, which fulfill a given predicate. For basic predicates, such as equality, there exist efficient methods to compute the join. However, for many real-world problems the predicate is more complex: it involves similarity. If we assume that records are represented by sets, we could use existing set similarity measures to compare them pairwise. Given a collection of *records* (sets) R, formed over the universe U of tokens (set elements), and a similarity function between two records, $sim : \mathscr{P}(U) \times \mathscr{P}(U) \to [0,1]$; the *set similarity self-join (SSJ)* of R computes all pairs of sets $(r,s) \in R \times R$ whose similarity exceeds a user-defined threshold θ, $0 < \theta \leq 1$, i. e., all pairs (r,s) with $sim(r,s) \geq \theta$. Without loss of generality, we focus on the Jaccard similarity function $sim(r,s) = \frac{|r \cap s|}{|r \cup s|}$ and the self-join.

A naive approach to compute the SSJ compares all possible pairs. Since the complexity of such an approach is quadratic, it is not feasible even for small datasets. The most prominent approaches in the literature to compute the SSJ more efficiently are based on the filter-and-verification framework. Filter-and-verification-based approaches do not reduce the worst-case complexity (which is quadratic), but reduce the practical compute effort when favorable input data characteristics are present. The framework first generates candidate pairs by creating and probing an inverted index [1] and verifies the candidates in a second step. Sophisticated filters such as the prefix filter keep the number of candidate pairs low [2]. This method is efficient on single cores [6]. However, it does not scale easily to large datasets.

We proposed a novel data-parallel filter-and-verification approach using multi-threading [5]. It significantly scales up the SSJ computation. However, the number of available CPU cores limits scalability. The maximum amount of input we could process with this method on our hardware was roughly 25 GB. To compute the SSJ on larger datasets, various MapReduce-based distributed approaches evolved. The MapReduce programming paradigm requires independently computable work shares. The approaches use existing filters from the filter-and-verification framework to replicate and group data into such independent shares. We showed that the amount of data these approaches can process is limited [3]. In our experimental setup, the maximum possible input was roughly 12 GB, which is even smaller than what the multi-threaded approach could process. Users cannot shift the limit by adding more compute nodes due to high and skewed data replication.

The input dataset size and scalability limitations of the previously mentioned approaches motivate our novel distributed-parallel SSJ approach, which pushes these limits significantly. We experimentally show that our new approach scales to hundreds of gigabytes and that it is robust against unfavorable data characteristics[1]. We use existing filter-and-verification techniques as a basis and leverage intra-node multicore parallelization by default. The major advances compared to existing distributed approaches are as as follows. First, our approach *avoids intra-node replication* since replication is the main bottleneck of the MapReduce approaches due to our previous analysis. It assures that each record is present only once in the main memory of each node. Each node runs only one single multi-threaded SSJ instance in order to efficiently share commonly used data, such as the inverted index. Second, it *avoids recomputation*, i. e., the repeated validation of the same candidate record pair. Third, it *removes algorithmic data dependencies* that lead to a skewed execution load as observed in MapReduce approaches using prefix filtering [3].

Our approach solely requires a standard shared nothing architecture for a distributed execution. Our approach is generic, thus it is independent of a specific distributed system. The quadratic nature of the SSJ problem implies that scaling up to larger input dataset sizes may require adding a quadratic number of nodes in the worst case. To avoid the worst case, our distributed-parallel approach

[1] Our implementation is available at `https://github.com/fabiyon/dist-ssj-sisap`.

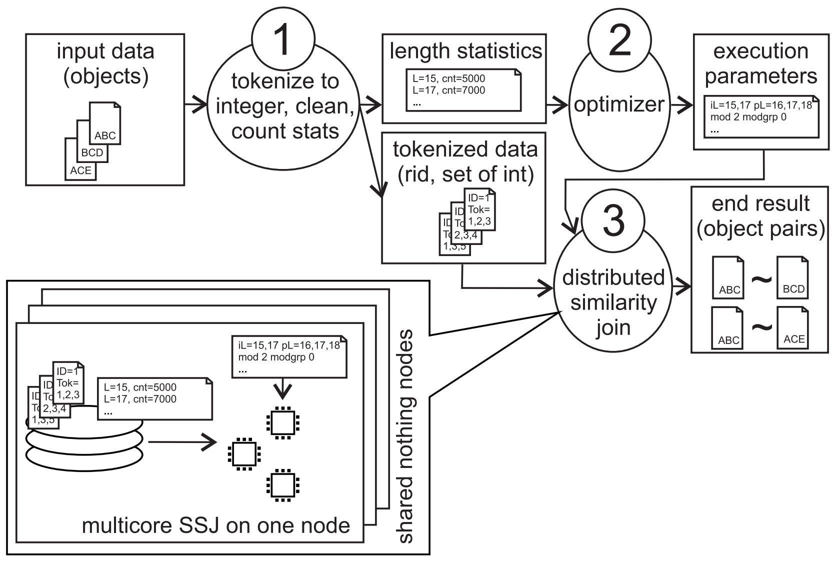

Fig. 1. Schematic dataflow of our distributed-parallel SSJ approach.

uses techniques to distribute the compute load evenly among nodes. However, depending on the dataset size, token distribution, and similarity threshold, the demand for compute nodes might still be high. Modern cloud computing allows to obtain a high number of compute nodes for a limited timeframe. Thus, we may safely assume that it is realistic today to have hundreds or even thousands of compute nodes available for just one operation. The main contributions of this paper are as follows:

– We introduce a cost-based heuristic to break down the SSJ computation into units that are computed independently in parallel.
– We additionally provide a data-independent scaling mechanism that allows to further subdivide each unit if necessary and a RAM usage estimation to avoid swapping.
– We experimentally verify that our distributed SSJ approach scales to hundreds of gigabytes of input data.

In the following section, we introduce our solution in detail. Section 3 experimentally shows its behavior on large datasets and large numbers of compute nodes. Section 4 concludes this paper. We provide an extended version of this paper with an additional description of experimental datasets as well as comprehensive tables and figures of our experimental results [4].

2 Distributing Filter-and-verification-based SSJ

Figure 1 provides an overview on our distributed-parallel SSJ approach. Step (1) preprocesses and tokenizes the raw input data. In addition, we require this step to compute a statistic of the lengths of all records. The length statistic consists of tuples $\{(l, |R_l|)\}$ where l is a record length and $|R_l|$ is the number of the records with this length. Step (2), referred to as optimizer, realizes the major part of our distributed SSJ approach. It generates parameters for each node to distribute the compute workload. Step (3) computes the SSJ based on the parameters of the optimizer. We require the tokenized input data and the length statistics to be available on every compute node. The join is an extension of our multicore SSJ as described in [5]. The extension includes the set of parameters from the optimizer. The parameters limit the records to be indexed and joined on each node such that the result is complete and free of duplicates.

Our solution assumes that each node runs exactly one instance of the multicore SSJ, exclusively using the nodes' hardware resources. By instance, we refer to the main thread of our multicore SSJ together with the worker threads it spawns during execution. We choose this setup to share common data structures such as inverted indexes. As it is common in MapReduce-based distributed systems, SSJ instances cannot communicate with each other and do not share data during execution. The instances have all information for the execution available before the beginning of the join computation. Each instance indexes and probes only subsets of the input dataset to independently compute a partial join result.

In the following, we introduce the optimizer. It runs before the actual join computation and divides the SSJ computation into independently computable units. The optimizer consists of a data-dependent cost-based heuristic and a data-independent scaling mechanism. Furthermore, we provide estimations of RAM demand and cost distribution and a heuristic to find suitable optimizer parameters. We first describe our cost-based heuristic.

2.1 Data-Dependent Cost-Based Heuristic

One goal of our cost-based heuristic is to avoid the cross product by only regarding record pairs with *matching* lengths. Regarding lengths to filter out hopeless pairs is a common technique, which most filter-and-verification approaches use [1]. This filter is effective on datasets with varying lengths and cheap to apply by using the length statistic computed beforehand. As discussed in the introduction, we focus on the Jaccard similarity function and the self-join.

Regarding Jaccard similarity and a record r, the length of a similar record s has to be in the interval $[\lceil \theta \cdot |r| \rceil; \lfloor \frac{|r|}{\theta} \rfloor]$. In the self-join case, the probe record set is equal to the index record set. To avoid duplicates and unnecessary recomputation, we subsequently consider only probe records larger than the length of an index record r: $[|r|; \lfloor \frac{|r|}{\theta} \rfloor]$. Figure 2 shows this length relationship for a similarity threshold of $\theta = 0.7$. For each record length on the y axis, it shows on the x axis, which record lengths have to be considered as join candidates. Now consider that we index the lengths on the y axis and probe the lengths on the x axis. Then each square in the figure represents a pair of index and probe lengths (i, p), which has to be joined for a complete result without duplicates. Each square can potentially be joined independently. However, for our heuristic,

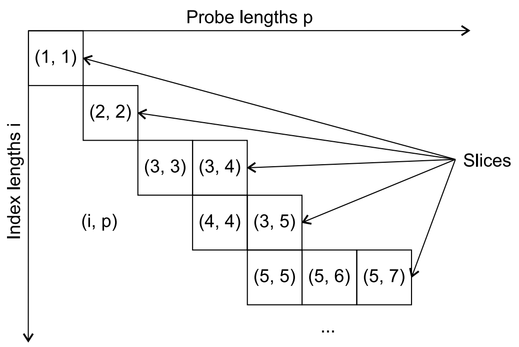

Fig. 2. Example join matrix for $\theta = 0.7$. Squares with the same index length compose one slice.

Table 1. Symbol reference.

R	input dataset
θ	similarity threshold
$\lvert r\rvert$	number of tokens in r
$\lvert R_l\rvert$	number of records with length l
$\mathcal{P}(l)$	prefix length of length l: $\mathcal{P}(l) = l - \lceil \theta \cdot l \rceil + 1$
i	index prefix length
p	probe prefix length
rid	record ID
n	node parameter for cost-based heuristic
m	modulo: data-independent scaling parameter
$modgroup$	group parameter to check if a record is in a sub slice
$indexLengths$	set of index lengths for one SSJ instance
$probeLengths$	set of probe lengths for one SSJ instance

we choose to group squares with the same index lengths together and refer to them as *slices*. For each slice i, we estimate the probe costs $\mathcal{C}(i)$ as follows:

$$\mathcal{C}(i) = \mathcal{P}(i) \cdot |R_i| \cdot \sum_{p=i}^{\lfloor \frac{i}{\theta} \rfloor} \mathcal{P}(p) \cdot |R_p| \tag{1}$$

Table 1 serves as a symbol reference for the symbols we use in the equation and throughout this paper. For the cost estimation we assume that each probe of the inverted index causes a cost of the length of the postings list. We do not know the exact sizes of the postings lists a priori, because they are dependent on the token distribution. Instead, we assume the worst case, where all index records of the probed length are contained in the postings list. With regard to an index length i, the possible probe lengths p are in $[i, \lfloor \frac{i}{\theta} \rfloor]$. The total number of probes of one slice is the sum over the prefix of p (denoted as $\mathcal{P}(p)$) multiplied by the number of records with this length $|R_p|$ for all probe lengths. The number of index tokens of the slice is computed the same way and multiplied.

Table 2. Example of input data lengths, matching probe lengths, number of records, and corresponding slice costs for $\theta = 0.7$.

index length i	probe lengths $\{p\}$	$\lvert R_i \rvert$	$\mathcal{C}(i)$
1	1	10	100
2	2	30	900
3	3,4	80	86 400
4	4,5	500	1 800 000
5	5,6,7	400	1 416 000
6	6,7,8	200	568 000
7	7,8	190	581 400
8	8	150	202 500

Example 1. Table 2 shows the cost computation for a hypothetical dataset. The dataset has eight length values as shown in the first column. The second column shows matching probe lengths for each index length. $|R_i|$ shows the hypothetical length count per index length. Column $\mathcal{C}(i)$ shows the resulting slice costs. □

Example 1 highlights that slices can exhibit uneven costs. Thus, we assign *sets* of slices to compute nodes with the intention to distribute the costs evenly. We use a greedy heuristic to achieve an even cost distribution. We assume that the user chooses a seed number of compute nodes n (the total number of compute nodes for the SSJ computation can be higher depending on further parameters). We sort the slice costs $\mathcal{C}(i)$ in ascending order. Then we assign each slice to each node in a round robin fashion. Thus, the first node receives the slice with the largest cost, the second node receives the second-largest, and when the last node is reached, the first node obtains the next slice again. The following example shows our greedy cost distribution heuristic:

Example 2. Consider again Table 2 and $n = 2$. The highest cost appears for $i = 4$. Thus, we assign this slice to the first node. The next highest cost appears for $i = 5$. Thus, we assign it to the second node. The third one is $i = 7$, assigned to node 1, and so on. This approach generates the following index and probe lengths:
Node 1: index lengths 2,4,7,8, probe lengths 2,4,5,7,8, total costs 2 584 800 and
Node 2: index lengths 1,3,5,6, probe lengths 1,3,4,5,6,7,8, total costs 2 070 500.
□

As discussed before, our cost estimation cannot consider the specific sizes of the postings lists. The estimation assumes that all records with matching lengths are present in the postings lists, which is only the worst case and pessimistic. On the other hand, the heuristic ignores the costs for the verification. The verification is dependent on the number of candidates, which we cannot estimate a priori without actually computing the join. Thus, our heuristic potentially underestimates the costs if a dataset has many candidates. In our experiments, we show the strengths and limits of our approach. Next, we introduce the second part of the optimizer, the data-independent scaling mechanism.

2.2 Data-Independent Scaling Mechanism

The scaling mechanism subdivides each slice (cf. Subsection 2.1) by partitioning its probe records. Our join computation assigns subsequent integer record IDs (*rid*s) to each input record. We use the modulo function to assign a probe record to one partition as shown in the following equation:

$$isRecordInProbeSubset(rid, m, modgroup) = (rid \bmod m \overset{?}{=} modgroup) \quad (2)$$

The user-defined parameter m sets the number of sub slices to generate. The *modgroup* is in the interval $[0, m-1]$ and determines the sub slice a record is assigned to. The following example illustrates how our scaling approach assigns records to sub slices:

Example 3. Assume $m = 2$. One sub slice receives all records where the function returns true for $modgroup = 0$ and another sub slice obtains the ones for $modgroup = 1$. We ordered the records in our input datasets by ascending record lengths. Thus, we expect this approach to be robust against length skew in the input data. It assigns records of all probe lengths to each sub slice round robin. □

The scaling mechanism together with the cost-based heuristic form the main building blocks of the optimizer of our SSJ approach. To find suitable parameter values for m and n, we next discuss how to evaluate the quality of concrete instances of these parameters. We start with an estimation of RAM demand.

2.3 RAM Demand

Our heuristic and the scaling mechanism do not guarantee that the computation of one (sub) slice stays within the RAM size of a given compute node. If the SSJ computation allocates more memory than the system physically provides, swapping occurs. Swapping leads to severe runtime penalties, which we must avoid. The main idea to avoid RAM overutilization is to find optimization parameters m and n such that the RAM usage stays within system limits. With the heuristic from Section 2.1, a concrete value for n, a similarity threshold θ, and the length statistics of a concrete dataset $\{(r, |R_l|)\}$ we compute sets of lengths *indexLengths* and *probeLengths* for each node. We use these length sets for RAM demand estimations subsequently.

We use an extension of our multicore SSJ on each compute node [5]. The extension includes the parameters *indexLengths*, *probeLengths*, *m*, and *modgroup* to limit the index and probe records. Considering the extended multicore SSJ, the *inverted index*, *probe records*, and *candidates* demand the largest parts of main memory. Without loss of generality, we estimate the demands for all three categories for our concrete SSJ implementation. The estimation is applicable to possible other join implementations by adjusting the size factors of the employed data structures.

First, we focus on the inverted index. Our implementation of the inverted index holds the postings list entries in a struct of 12 Bytes. The number of postings list entries is the prefix length times the number of records $\mathcal{P}(l) \cdot |R_l|$ for each index length l. We can estimate the size of the inverted index (in Bytes) as follows:

$$indexRamDemand(indexLengths) = \sum_{l \in indexLengths} \mathcal{P}(l) \cdot |R_l| \cdot 12 \tag{3}$$

Similarly, we estimate the RAM demand for the probe records. One record in our implementation uses 60 Bytes plus each token stored as 4 Byte integer. We estimate the space requirement for the probe records (in Bytes) as follows:

$$probeRamDemand(probeLengths, m) = \sum_{l \in probeLengths} \frac{|R_l| \cdot (60 + l \cdot 4)}{m} \tag{4}$$

Lastly, we focus on the candidate size. Our SSJ uses 12 Bytes to store each candidate record in main memory until verification. Each thread keeps a local list of candidates for its subset of probe records. In the worst case, all indexed records are candidates. However, it is pessimistic to assume that all threads hold all index records as candidates at the same time. In our experiments, we found that it is safe to assume $\frac{1}{3}$ to $\frac{2}{3}$ of the index records to be present on each thread at a time on our datasets. Thus, we include a candidate factor $candFact$ in our estimation. We estimate the candidate RAM demand (in Bytes) as follows:

$$candidateRamDemand(indexLengths, numberThreads, candFact) = \\ \sum_{l \in indexLengths} |R_l| \cdot 12 \cdot numberThreads \cdot candFact \tag{5}$$

To avoid swapping, the sum of all demands must stay below the system limit of a compute node leaving space for other storage needs and the operating system. We found the static space demand to be below 4 GB on the system we run our experiments on and thus consider this value in the following.

Example 4. Consider the dataset ORKU with scaling factor 100, $\theta = 0.6$, $m = 64$, $n = 8$, and $numberThreads = 24$. Over all slices, we can compute a maximum index RAM demand of 21 GB, 2 GB for the probe records, and up to 10 GB for candidates. We estimate the total demand including the static demand to be 37 GB. In fact, on our system with 32 GB RAM, this parameter combination leads to heavy swapping. The runtime of each slice is above 12 hours. When we changed the parameters to $m = 16$ and $n = 32$ (which equals the total number of nodes in the previous configuration, 512) the total estimated RAM demand decreases to 24 GB. The maximum runtime per slice in this configuration is 300 seconds and no swapping occurs. The example motivates that it is crucial to find a suitable parameter configuration, which keeps the memory demand below the system limit to achieve an acceptable runtime for the join operation. □

Note that our data-independent scaling approach focuses only on probe records. In case the set of *indexLengths* contains solely one length and the corresponding *indexRamDemand* exceeds the available main memory, our approach does not provide a means to further reduce the index size. However, if an index exceeds available main memory it is possible to partition the index records, i. e., with a modulo function in the same way as we applied it to the probe records. We do not elaborate on further reducing the index size, because we cannot observe such an extreme index skew within our experiments even on highly enlarged datasets. Next, we discuss the cost distribution among the compute nodes.

2.4 Cost Distribution Quality

Even without swapping, the choice of parameter n might be crucial for the runtime depending on the length distribution of the input dataset. Example 5 illustrates and motivates the need for an appropriate parameter choice.

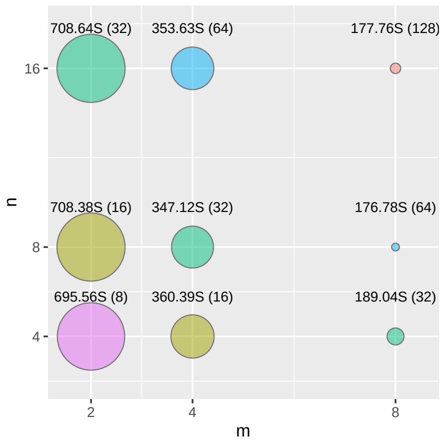

Fig. 3. AOL×10 runtimes.

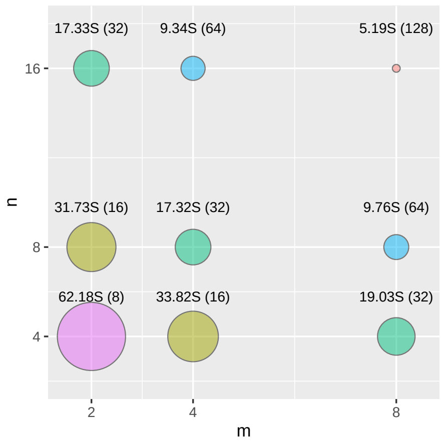

Fig. 4. ENRO×10 runtimes.

Example 5. Figures 3 and 4 visualize the runtimes of AOL and ENRO, both increased with scaling factor 10, for $\theta = 0.6$ varying both parameters m and n. The circle sizes represent the runtime. The same color marks combinations of parameters with the same total number of nodes. For example, the parameter combination $m = 8$ and $n = 4$ uses 32 nodes in total. Parameter combination $m = 4$ and $n = 8$ also uses 32 nodes and therefore has the same color assigned. The numbers above the circles are the maximum runtimes over all slices in seconds followed by the total number of nodes in brackets. For ENRO×10 a higher n is beneficial for an improved runtime. That is, the runtime with parameters $m = 2$ and $n = 16$ is lower than with parameters $m = 8$ and $n = 4$ for the same total amount of nodes of 32. On the other hand, for AOL×10, a higher value of n does not lead to improved runtimes. A higher m parameter is effective for both datasets. The effectiveness of parameter m on both datasets is expected, because it linearly scales the number of probe records. □

Table 3. Example for input data length skew. Columns show hypothetical input data lengths, matching probe lengths, and the number of records for AOL and ENRO for $\theta = 0.6$.

index length i	probe lengths $\{p\}$	AOL $\|R_i\|$	ENRO $\|R_i\|$
1	1	2705785	149
2	2,3	2026952	361
3	3,4,5	2051010	594
4	4,5,6	1457075	814
5	5,6,7,8	849944	1029
6	6,7,8,9,10	445489	1141
7	7,8,9,10,11	225401	1301
8	8,9,10,11,12,13	117962	1386

In Example 5, the length distributions of the datasets are essential for the efficiency of parameter n regarding runtime. AOL shows significantly more short records than ENRO. For example, in AOL there are 1.4 to 2.7 million records with the lengths 1 to 4, which corresponds to roughly 80 percent of the total number of records in AOL. ENRO has only 149 to 814 records in this length range, which corresponds to less than 1 percent of the records in ENRO. Table 3 lists matching probe lengths and record counts of AOL and ENRO for a low similarity threshold $\theta = 0.6$. The slices of AOL for $i \in 1, 2, 3, 4$ are large in relation to the number of total records, while the slices of ENRO remain small. The cost-based heuristic is less effective for AOL due to its skewed record lengths. Furthermore, depending on the choice of n, this length skew results in cost skew over the slices. In this example, the costs for AOL are less skewed for $n = 4$ compared to higher values of n.

To evenly distribute the compute costs over the nodes, we aim to find the best n out of a given value range regarding a distribution quality function. Given one n, we can compute the maximum cost deviation over all slices with $\max\{\mathcal{C}(i)\} \div \min\{\mathcal{C}(j)\}$ for $i, j \in [0; n-1]$. Given a *valueRange* for n, we can then minimize this deviation as follows:

$$\min_{n \in valueRange} = \left\{ \max_{i \in [0;n-1]} \{\mathcal{C}(i)\} \div \min_{j \in [0;n-1]} \{\mathcal{C}(j)\} \right\} \qquad (6)$$

Example 6. Consider AOL×10, $\theta = 0.6$, and $n \in \{4, 8, 16, 32\}$. Using Equation 6, $n = 4$ has the lowest maximum cost deviation of 4.16. For higher values of n the deviation varies between 200 and 230 000. For ENRO×10 and the same parameters, the lowest deviation is 1.02 for $n = 4$, followed by 1.05 for $n = 8$, 1.09 for $n = 16$, and 1.21 for $n = 32$. For both datasets, our cost distribution quality estimation chooses a good value for parameter n. Our estimation might not necessarily lead to the optimal parameter value regarding runtime, but it avoids unfavorable values. □

In the following subsection, we discuss how to use these cost distribution considerations together with the RAM estimation to find suitable parameter values m and n.

2.5 Finding Suitable Parameter Values

Our approach uses the two parameters m and n. Based on the previous discussion about RAM demand and cost distribution we propose the following strategy to determine parameter values, which avoid RAM overutilization and cost skew. We assume that the user chooses a total number of compute nodes t as a seed, which should preferably be a power of two for practical reasons. For each possible m and n (such that $m \cdot n = t$) we compute the estimated demand for RAM (cf. Section 2.3) and the minimum and maximum cost over all slices (cf. Section 2.4). We can prune all parameter combinations with a RAM demand above the system limit. We then choose the parameter combination (m, n) with the lowest cost deviation. In case all parameter combinations are pruned, we set the total number of nodes $t = t \cdot 2$ and re-run the previous computation until a suitable combination is found. If the resulting t is above the number of available compute nodes, the computation should be split into subsequent phases. The described strategy finds only the minimum m parameter value with respect to t. Users may increase m to achieve lower runtimes. In our experiments, we show the applicability of our approach to find suitable parameters.

3 Experiments

This section presents our experimental analysis. We focus on scalability, varying the parameters m and n, the input dataset sizes, and the similarity threshold θ. Based on the shortcomings of manually choosing parameter values, we subsequently discuss our strategy to find suitable parameter values m and n.

To compute the join on one slice we use a multicore C++ SSJ implementation running it on each compute node by extending our previous multicore SSJ with the parameters *indexLengths*, *probeLengths*, m, and *modgroup*. By default, we run the multicore SSJ with the optimal parameters [5]. We enable the position filter and set the number of threads to 24, which is optimal on our hardware: Each node is equipped with two Xeon E5-2620 2GHz of 6 cores each (with hyper-threading enabled, i. e., 24 logical cores per node), 24 GBs of RAM, and two 1 TB hard disks. Whenever we report runtimes, we refer to the maximum runtime over all slices since the maximum runtime determines the overall runtime.

As input datasets, we use the 10 real-world and two synthetic datasets (cf. extended paper [4]). Since we focus on larger datasets, we use only increased datasets with the scaling factors 10, 25, 50, and 100. We start our experiments with a scaling factor of 10, because these are the largest datasets joinable with both the MapReduce and the multicore approaches so far. Our novel distributed approach is able to compute the join on much larger datasets as we show subsequently.

3.1 Impact of Cost-based Heuristic

In this experiment, we show how the runtime develops varying parameter n. We do not set parameter m. Thus, the probe records per slice remain complete with

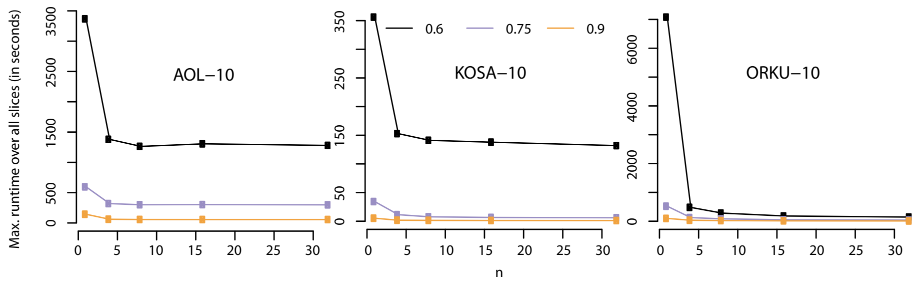

Fig. 5. Maximum runtimes over all slices for $n \in \{4, 8, 16, 32\}$ for three exemplary datasets AOL, KOSA, and ORKU. $n = 1$ represents the multicore SSJ without distributed parallelization. Thresholds $\theta \in \{0.6, 0.75, 0.9\}$.

regard to the *probeLengths* computed with the heuristic from Section 2.1. We use all datasets increased by factor 10, $\theta \in \{0.6, 0.75, 0.9\}$, $n \in \{4, 8, 16, 32\}$ and compare it to the non-distributed multicore SSJ (cf. Figure 5).

For all datasets and all thresholds, $n = 4$ significantly reduces all runtimes compared to $n = 1$. The speedups vary between 1.8 (AOL$\times$10, $\theta = 0.75$) and 13.9 (ORKU$\times$10, $\theta = 0.6$). The average speedup over all datasets and thresholds is 3.7. For higher values of n the speedups decrease. Adding more than 8 or 16 nodes leads to only small runtime decreases for most datasets and thresholds. This effect is due to the nature of our heuristic. Recall that one slice consists of an index length and all its possible probe lengths. The length skew of the input datasets and the similarity threshold determine the largest and potentially slowest slice, which cannot be further partitioned with the heuristic. AOL$\times$10 is exemplary for this circumstance. As we discussed in Section 2.4, AOL has roughly 80 percent of its records within the length range 1 to 4. n values higher than 4 are not beneficial for this dataset. Other datasets show different length distributions, which lead to optimal n values higher than 4.

KOSA$\times$10 also shows a limited scalability for $\theta = 0.6$, but for a different reason than length skew. We observe that amongst all slices for each n there exists one slice with a runtime between 130 and 150 seconds, while all other slices have lower runtimes. The reason for the outlier slices in KOSA$\times$10 are their high number of candidates compared to all other slices. The runtimes of KOSA$\times$10 show a limitation of our heuristic. It optimizes the runtime based on length information and is thus not robust against candidate skew by design.

3.2 Impact of Data-independent Scaling Mechanism

In this experiment, we study how the scaling parameter m influences the runtimes. We continue to use the datasets using scaling factor 10 and fix parameter n to 8, since this parameter setting showed good runtimes in the previous experiment. We again use $\theta \in \{0.6, 0.75, 0.9\}$ and vary $m \in \{2, 4, 8\}$. The results indicate that $m \geq 2$ is beneficial to achieve a lower runtime for all datasets and

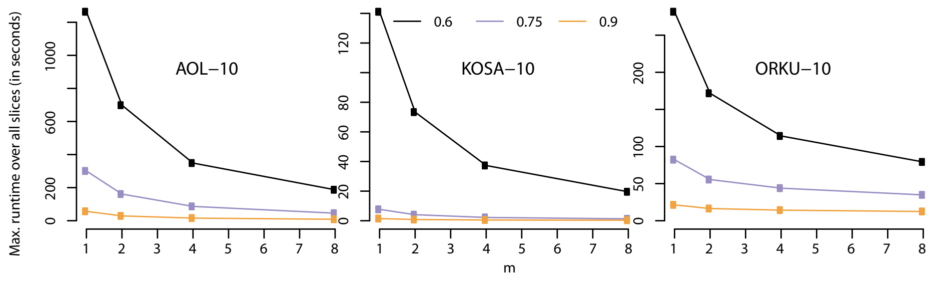

Fig. 6. Maximum runtimes for exemplary datasets over all slices for $n = 8$, $\theta \in \{0.6, 0.75, 0.9\}$, $m \in \{2, 4, 8\}$. $m = 1$ indicates runtimes without the scaling mechanism.

thresholds, including AOL×10 and KOSA×10, which showed scalability boundaries for $n \geq 4$ in the previous experiment (cf. Figure 6).

Since the modulo function evenly distributes different probe lengths among sub slices we expect the runtimes to scale linearly with m, which experimental results partially confirm. Regarding the minimum, maximum, and average speedups for $m \in \{2, 4, 8\}$ in relation to $m = 1$, grouped by θ, there is a maximum speedup close to the optimum m for each threshold group. The averages over all thresholds for $m = 2$ are close to the optimum 2. The average speedups for larger values for m decrease.

3.3 Impact of Dataset Size

In this subsection, we investigate how the runtimes evolve when increasing the dataset size by scaling factors $s \in \{10, 25, 50, 100\}$. We statically set $n = 8$ and $m = 64$. We compare the maximum runtimes per slice for $s \in \{25, 50, 100\}$ relative to maximum runtime for $s = 10$.

In many cases, the runtime does not increase linearly with the dataset size. A non-linear runtime increase is expected, because the SSJ has a quadratic complexity. A perfectly linear runtime relative to $s = 10$ would be $\frac{s}{10}$ for $s \in \{25, 50, 100\}$. Only few combinations of datasets, θ, and s fall in this category. For ENRO and $\theta = 0.9$, ORKU and $\theta = 0.9$, and SPOT (all thresholds) the relative runtimes for $s \in \{25, 50, 100\}$ are better than linear. ENRO and $\theta = 0.75$, FLIC and $\theta \in \{0.75, 0.9\}$, LIVE and $\theta = 0.9$, ZIPF and $\theta \in \{0.75, 0.9\}$ are close to linear. We can observe that the runtimes of higher thresholds increase more linearly than the ones of lower thresholds relative to s. This runtime behavior can be explained by the prefix filter, which is more effective for higher thresholds.

With our approach, it is possible to compute the SSJ on all datasets of all sizes in our evaluation and all thresholds except ENRO-100 and $\theta = 0.6$. We manually stopped the computation after 12 hours. In Section 2.3, we discussed that for ORKU×100 the parameter combination $n = 8$ and $m = 64$ is not optimal, because it causes swapping. We next discuss our proposed parameter finding strategy.

3.4 Discussion of Parameter Finding Strategy

The previous experiment on enlarged datasets highlights that the manually assigned parameters $m = 64$ and $n = 8$ are not suitable for ORKU$\times$100 and $\theta = 0.6$, because the runtime exceeds 12 hours. In Section 2.3, we discussed the same example and concluded that swapping occurs. When we apply the parameter strategy from Section 2.5 to the equal number of total nodes as before ($t = 8 \cdot 64 = 512$), it suggests $m = 32$ and $n = 16$. The runtime of this parameter combination is 1314 seconds, so the strategy avoids the worst case. We furthermore expect the strategy to choose the parameter combination with the smallest cost deviation. In the example in Section 2.4, we discussed that for AOL$\times$10 $\theta = 0.6$ $n = 4$ is better than a larger n. Running the parameter finding strategy for $t = 16$, it indeed suggests the parameter value $n = 4$.

4 Conclusion

In this paper we introduced our novel distributed SSJ approach. We showed experimentally that it scales the computation to potentially hundreds of compute nodes if needed. Our method computes the SSJ on our hardware on datasets up to roughly 240 GB, which is much larger than the ones which could be computed with existing parallel methods so far. We discussed how to a priori estimate limits of parameter values from which we cannot expect an efficient execution, especially regarding main memory usage. We proposed a parameter finding strategy, which avoids poor parameter values leading to either RAM overutilization or a skewed cost distribution. One remaining challenge is to better estimate or manipulate the maximum number of candidates of each slice, which occur at one instance of time.

Acknowledgements This work was supported by a research grant from LexisNexis Risk Solutions.

References

1. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. Proceedings of the International Conference on World Wide Web (2007)
2. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. International Conference on Data Engineering (ICDE) (2006)
3. Fier, F., Augsten, N., Bouros, P., Leser, U., Freytag, J.C.: Set similarity joins on MapReduce: an experimental survey. Proceedings of the International Conference on Very Large Data Bases (PVLDB) (2018)
4. Fier, F., Freytag, J.C.: Scaling up set similarity joins using a cost-based distributed-parallel framework [extended paper] (2021). https://doi.org/10.18452/23209
5. Fier, F., Wang, T., Zhu, E., Freytag, J.C.: Parallelizing filter-verification based exact set similarity joins on multicores. Proceedings of the International Conference on Similarity Search and Applications (SISAP) (2020). https://doi.org/10.1007/978-3-030-60936-8_5
6. Mann, W., Augsten, N., Bouros, P.: An empirical evaluation of set similarity join techniques. Proceedings of the International Conference on Very Large Data Bases (PVLDB) (2016)