# Set Similarity Joins on MapReduce: An Experimental Survey

Fabian Fier
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
fier@informatik.hu-berlin.de

Nikolaus Augsten
Universität Salzburg
Jakob-Haringer-Str. 2
5020 Salzburg, Austria
nikolaus.augsten@sbg.ac.at

Panagiotis Bouros
Johannes Gutenberg
University Mainz
Saarstr. 21
55122 Mainz, Germany
bouros@uni-mainz.de

Ulf Leser
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
leser@informatik.hu-berlin.de

Johann-Christoph Freytag
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
freytag@informatik.hu-berlin.de

## ABSTRACT

Set similarity joins, which compute pairs of similar sets, constitute an important operator primitive in a variety of applications, including applications that must process large amounts of data. To handle these data volumes, several distributed set similarity join algorithms have been proposed. Unfortunately, little is known about the relative performance, strengths and weaknesses of these techniques. Previous comparisons are limited to a small subset of relevant algorithms, and the large differences in the various test setups make it hard to draw overall conclusions.

In this paper we survey ten recent, distributed set similarity join algorithms, all based on the MapReduce paradigm. We empirically compare the algorithms in a uniform test environment on twelve datasets that expose different characteristics and represent a broad range of applications. Our experiments yield a surprising result: All algorithms in our test fail to scale for at least one dataset and are sensitive to long sets, frequent set elements, low similarity thresholds, or a combination thereof. Interestingly, some algorithms even fail to handle the small datasets that can easily be processed in a non-distributed setting. Our analytic investigation of the algorithms pinpoints the reasons for the poor performance and targeted experiments confirm our analytic findings. Based on our investigation, we suggest directions for future research in the area.

**PVLDB Reference Format:**
Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. Set Similarity Joins on MapReduce: An Experimental Survey. *PVLDB*, 11(10): xxxx-yyyy, 2018.
DOI: https://doi.org/10.14778/3231751.3231760

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 44th International Conference on Very Large Data Bases, August 2018, Rio de Janeiro, Brazil.

*Proceedings of the VLDB Endowment,* Vol. 11, No. 10
Copyright 2018 VLDB Endowment 2150-8097/18/06.
DOI: https://doi.org/10.14778/3231751.3231760

## 1. INTRODUCTION

The set similarity join (SSJ) computes all pairs of similar sets from two collections of sets. Two sets are similar if their normalized overlap exceeds some user-defined threshold; the most popular normalizations are Jaccard and Cosine similarity[1]. Applications of SSJ are, for instance, the detection of pairs of similar texts (modeled as sets of tokens) [29], strings or trees (modeled as sets of n-grams resp. pq-grams) [31, 3], entities (modeled as sets of attribute values) [11], the identification of click fraudsters in online advertising [20], or collaborative filtering [6].

Conceptually, the set similarity join between two collections $S$ and $R$ must perform $|S| \cdot |R|$ set comparisons, which is not feasible. Efficient techniques for SSJ use filters to avoid comparing hopeless set pairs, i.e., pairs that provably cannot pass the threshold [6, 7, 34]. We distinguish two classes of filters. *Filter-and-verification* techniques use set prefixes or signatures followed by an explicit verification of candidate pairs (e.g., [6, 34]). *Metric-based* approaches partition the space of all sets such that similar sets fall into the same or nearby partitions (e.g. [13]). The latter methods require the set similarity function to be metric.

For large datasets, which cannot be handled by a single compute node, distributed SSJ algorithms are required. In recent years, a number of solutions for the distributed SSJ have been proposed, most of which are based on the MapReduce paradigm. These solutions include (by publication year) FullFiltering [12], VernicaJoin [30], SSJ-2R [5], FuzzyJoin [1], V-SMART [21], MRSimJoin [27], MG-Join [24], MAPPS [32], ClusterJoin [25], MassJoin [9], MRGroupJoin [10], FS-Join [23], DIMA [28]. While non-distributed solutions have been recently compared in experimental studies [14, 19], we are not aware of any comprehensive comparison of distributed SSJ algorithms. The only exception is the work by Silva et al. [26], which compares FuzzyJoin [1], MRThetaJoin [22], MRSimJoin [22], Vernica [30], and V-SMART [21]. However, many relevant competitors are missing in this benchmark, and the experiments were performed on a single dataset, which limits

[1] Algorithms for edit-based string similarity joins often use SSJ to reduce the number of candidate pairs, e.g. [2].

| | | | CJ | GJ | FF | MG | MJ | MR | S2 | VJ | VS | FS |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ClusterJoin | [25] | CJ | ▨ | | | | | | | =[25] | >[25] | |
| MRGroupJoin | [10] | GJ | > | ▨ | | | | | | | | |
| FullFilteringJoin | [12] | FF | | < | ▨ | | | | <[5] | <[5] | | |
| MGJoin | [24] | MG | > | < | > | ▨ | | | | >[24] | | |
| MassJoin | [9] | MJ | > | < | > | < | ▨ | | | >[9] | | <[23] |
| MRSimJoin | [27] | MR | | < | | < | < | ▨ | | | | |
| SSJ-2R | [5] | S2 | | < | | < | | | ▨ | >[5] | | |
| VernicaJoin | [30] | VJ | > | > | > | > | > | > | > | ▨ | <[21] >[25] | <[23] |
| V-SMART | [21] | VS | | < | | < | < | | | < | ▨ | <[23] |
| FS-Join | [23] | FS | > | < | > | < | > | > | > | < | > | ▨ |

← Our Comparisons →

← Previous Comparisons →

**Figure 1: Overview over previous (upper right triangle) comparisons and those performed in this work (lower left triangle). A ">" indicates that the algorithm in the row is faster than the one in the column according to the respective publication.**

their expressiveness. The empirical evaluations in the original publications of the algorithms provide only an incomplete picture (see Figure 1).

In this paper, we perform a comparative evaluation of the ten distributed SSJ approaches listed in Figure 1 that build on top of the MapReduce framework [8]. To provide a fair experimental setup, we do not tailor parameters, data preprocessing, implementation details, system configuration details or the problem statement to work especially well with one algorithm. We do not include [28] in our study since the DIMA in-memory system (i) builds on top of Spark by extending the Catalyst optimizer and (ii) the proposed approach for similarity joins employs offline distributed indexing. We also exclude FuzzyJoin [1] which focuses on string similarity measures; although arguing that the proposed methods can be adapted for set similarity measures, the authors do not elaborate on this issue. Further, we do not consider MAPSS [32] which is tailored to dense vectors; note that vector representations of sets are typically extremely sparse. Last, we exclude [16, 18, 17], which also focus on vector data and on Euclidian distance rendering the proposed techniques not applicable to set similarity measures.

We base our comparison on twelve datasets (ten real-life and two synthetic datasets) of varying sizes and characteristics. All methods were reimplemented or adapted to remove bias stemming from different code quality. We further removed pre- and/or post-processing steps and thus reduced all methods to their core: the computation of SSJ. Since all tested algorithms are based on the Hadoop implementation of MapReduce, we run all comparisons on the same Hadoop cluster. We repeat experiments from the original works and – where the results differ – discuss reasons for the deviations. We further perform a qualitative comparison of all algorithms. We systematically discuss and illustrate their map and reduce steps and provide an example for most algorithms. We analyze their expected intermediate dataset sizes and distribution and other factors that may have an impact on runtime and scalability. This analysis forms the basis for the subsequent discussion and helps to explain our experimental results.

Our findings are sobering for various reasons. First, the distributed SSJ algorithms are often orders of magnitude slower than their non-distributed counterparts for the same datasets and threshold settings (runtimes as reported by Mann et al. [19]). This can be only partially explained by the overhead of the Hadoop framework, which we measure. Second, we expected the distributed algorithms to scale to very large datasets that cannot be handled by a single machine. However, we observe that all algorithms in our test run into timeouts for at least one of the datasets. This cannot be fixed by increasing the cluster size since the algorithms fail to evenly distribute the workload and individual nodes are overloaded.

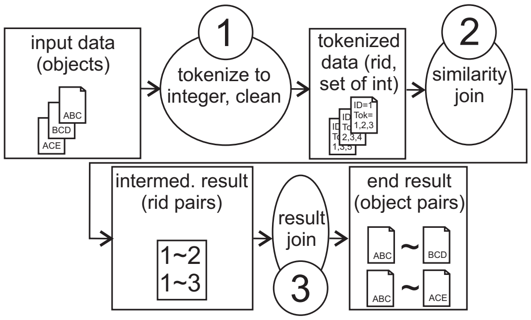


**Figure 2: Computation of a token-based SSJ. Our work focused on step (2), the actual join.**

Summarizing, the contributions of this paper are the following:

- To the best of our knowledge, this is the first comprehensive experimental evaluation of distributed set similarity joins.
- We survey all tested algorithms using a uniform notation, discuss their map and reduce phases, and analyze their expected intermediate result sizes and distribution.
- Our in-depth study of the intermediate data distribution and resource utilization pinpoints the major bottlenecks of the tested algorithms, and inspires our ideas for future work.

The structure of the paper is as follows. Section 2 gives an overview on SSJ and Hadoop. Section 3 surveys all algorithms in our test and analyses their intermediate dataset sizes. Section 4 reports on the experimental results and discusses our findings. Section 5 concludes our study with suggestions for future work.

## 2. BACKGROUND

Given two collections of sets, $S$ and $R$, formed over the same universe $U$ of tokens (set elements), and a similarity function between two sets, $sim : \mathscr{P}(U) \times \mathscr{P}(U) \rightarrow [0,1]$; the *Set Similarity Join* (SSJ) between $S$ and $R$ computes all pairs of sets $(s,r) \in S \times R$ whose similarity exceeds a user-defined threshold $t$, $0 < t \leq 1$, i.e., all pairs $(s,r)$ with $sim(s,r) \geq t$.

Following previous work on SSJ algorithms we hereafter focus on all-pairs self-joins using the inverse Jaccard distance as a similarity function. Thus, all our datasets are a single collection $R$ of sets consisting of sorted tokens. In this survey, we do not introduce new approaches, so we chose a setting that is supported by all tested algorithms: self-join using Jaccard similarity. Figure 2 outlines the typical workflow of an end-to-end set similarity self-join. The input is a collection of objects, i.e., documents. Step (1) transforms each object into a set of integer tokens. The result is a record per object identified by a unique record ID (rid); the tokens in the record are unique integer values. The similarity join Step (2) computes all similar pairs of sets and outputs the respective record ID pairs. Step (3) joins the original objects to the record IDs to produce pairs of objects as the final result.

In our analysis, we focus on the actual similarity join, Step (2), and so, we do not measure the pre- and post-processing cost. The preprocessing step in our experiments is the same for all algorithms in order to ignore the problem of efficient tokenizations [4]. MassJoin does not require an additional join to produce object pairs from ID pairs in Step (3) since the original objects are already present in Step (2). We evaluate this effect in a separate test.

All algorithms we consider are based on the MapReduce framework [8] and are implemented in Hadoop using its distributed file

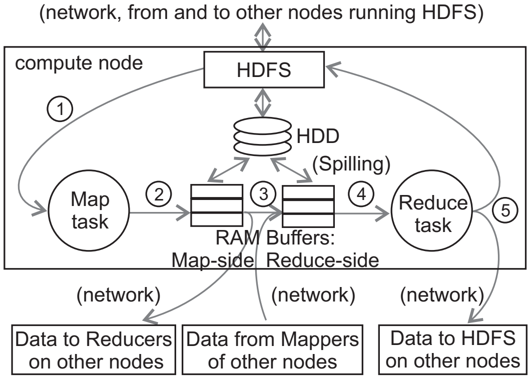

(network, from and to other nodes running HDFS)
compute node
HDFS
1
HDD
(Spilling)
Map task
2
3
4
Reduce task
5
RAM Buffers:
Map-side Reduce-side
(network)
(network)
(network)
Data to Reducers on other nodes
Data from Mappers of other nodes
Data to HDFS on other nodes

**Figure 3: Schematic dataflow between map, reduce, HDFS, RAM, and filesystem in one Hadoop compute node. Dataflow from and to remote map, reduce, and HDFS instances.**

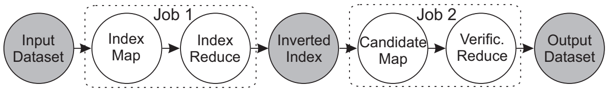
system HDFS [33]. We focus on comparing existing algorithms without introducing new approaches. Thus, we exclude adaptations of the algorithms to other big data platforms such as Flink or Spark, because this would require non-trivial changes in the algorithms. We use the current version 2 of Hadoop, which is based on the resource manager YARN. The system creates *containers* on each node which can run *tasks* such as map or reduce. The number of containers depends on user-defined memory settings. The number of map tasks is by default equal to the number of HDFS data blocks of the input. The number of reduce tasks is user-defined. Both, the concurrent map and reduce tasks are limited by the cluster size.

We refer to each block of map, optionally followed by reduce, as a *job*; most algorithms in our tests consist of multiple jobs. A map or reduce function can use a so-called *setup function*, which is executed once at instantiation time. This is useful to load global settings or data to each map/reduce task and have it available throughout the lifetime of the task. In Figure 3, we provide a simplified overview of relevant data flows in a MapReduce job. Map reads data blocks from HDFS (step 1 in the figure). Each output record of a map is hashed by its key (step 2), buffered on the map side (spilled to disk if a buffer threshold is reached or if not enough reducers are free), and then routed to the reducer responsible for this key. An optional combiner groups the map outputs by key and performs some pre-computations to reduce its size (omitted in the figure for brevity). The input for the reducer is collected from different map tasks (which may be local or remote) and buffered in RAM (step 3). When the buffer is full, the data is spilled to local disk. Map-side buffers can also spill to disk, for example, when the reduce-side buffers are full. After all map tasks finished their execution, the data at the reducers is sorted and grouped by key (*shuffling phase*, step 4). Finally, the reduce function is called for each key (*data group*), and its output is saved to HDFS, possibly serving as an input for subsequent jobs.

# 3. SURVEY AND ANALYSIS

In this section, we review the ten SSJ algorithms of Figure 1, and analyze the size of the intermediate data between maps and reduces. The size of the intermediate data is critical since it often correlates to the I/O cost which dominates the overall execution time.

We denote the input collection with $R \subseteq \{r \mid r \subseteq U\}$. The *global token frequency (GTF)* of a token is the number of records containing this token. We provide the signatures of the map and reduce functions and denote the mapper (combiner, reducer) of job $i$ with $M_i$ ($C_i$, $R_i$). The input and output signatures are $\langle key, value\rangle$ pairs and a (non-empty) list of values is denoted as $value^*$. Last, $rid$ is a record ID, $tok$ is a token, and $l$ is the length (number of tokens) of a particular record.



$M_1$: $\langle rid, tok^*\rangle \rightarrow \langle tok, (rid, l)\rangle$
$R_1$: $\langle tok, (rid, l)\rangle \rightarrow \langle tok, (rid, l)^*\rangle$ // inverted lists
$M_2$: $\langle tok, (rid, l)^*\rangle \rightarrow \langle (rid_1, rid_2), (l_1, l_2, 1)\rangle$ // candidates
$C_2$: $\langle (rid_1, rid_2), (l_1, l_2, 1)\rangle \rightarrow \langle (rid_1, rid_2), (l_1, l_2, par\_olap)\rangle$
$R_2$: $\langle (rid_1, rid_2), (l_1, l_2, par\_olap)\rangle \rightarrow \langle rid_1, rid_2\rangle$ // verification

**Figure 4: FullFilteringJoin Dataflow.**

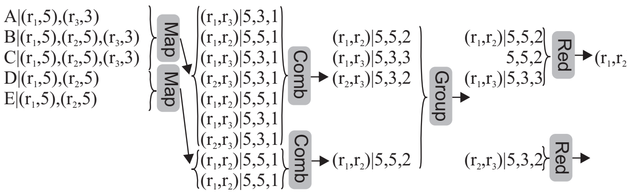


**Figure 5: FullFilteringJoin, Job 2, Example.**

**Running example**: $R = \{r_1, r_2, r_3\}$, $r_1 = \{A, B, C, D, E\}$, $r_2 = \{B, C, D, E, F\}$, $r_3 = \{A, B, C\}$, the similarity function is $sim(r_i, r_j) = |r_i \cap r_j|/|r_i \cup r_j|$ (Jaccard), the threshold is $t = 0.65$. With $sim(r_1, r_2) = \frac{2}{3}$, $sim(r_1, r_3) = \frac{3}{5}$, and $sim(r_2, r_3) = \frac{1}{3}$, the join result is $(r_1, r_2)$.

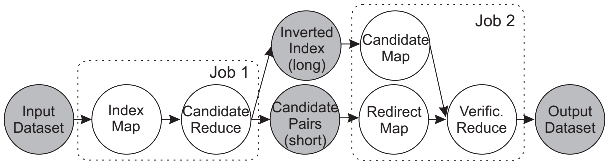
## 3.1 Filter-and-verification based algorithms

**FullFilteringJoin (FF)** [12]. FF computes an inverted index over all tokens in Job 1 (each token maps to all records, which contain this token) and uses the inverted lists in Job 2 to compute the records overlap and the final join result (cf. Figure 4).

We discuss Job 2 (cf. Figure 5). $M_2$ processes the inverted list of a token by generating all record pairs that share the token (i.e., all 2-combinations of the records in the list are produced). The combiner $C_2$ groups record pairs from different lists and computes their partial overlap. Reducer $R_2$ adds this partial overlap for each record pair to get the full overlap. $R_2$ further uses the record lengths, which are stored with the respective records, to compute the Jaccard similarity and verify each record pair. Since each record pair is verified by a single reducer, the output is duplicate free.

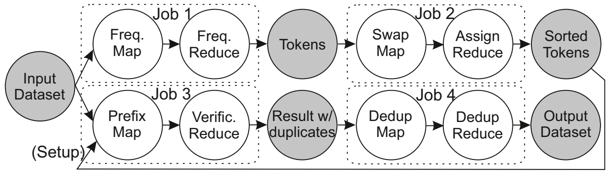
*Discussion.* The output of both $M_1$ and $R_1$ is linear in the input data: $M_1$ produces $|M_1| = \sum_{r \in R} |r| = |R| \cdot \overline{|r|}$ records of 3 integers each ($\overline{|r|}$ is the average record length); $R_1$ produces $|R_1| = |U|$ inverted lists $L = (rid, l)^*$. The maximum list length $|L|$ is given by the maximum GTF. The output of $M_2$ is quadratic in the GTF: for an input record $\langle tok, L\rangle \in R_1$, $|M_2| = \sum_{\langle tok, L\rangle \in R_1} \binom{|L|}{2}$ records of 4 integers each are generated.

**V-SMART (VS)** [21]. VS extends FullFilteringJoin [12] with the idea to split long inverted index lists and replicate them to multiple nodes. Job 1 (cf. Figure 6) computes an inverted index over all tokens. In contrast to FullFilteringJoin, for *short* inverted lists all candidates (2-combinations of the list) are computed already in this first step and are materialized to HDFS. *Long* inverted list are partitioned and replicated to be processed in Job 2. The mappers in Job 2 either generate candidates (long lists) or pass them on to the reducers (short lists). The candidates are pre-aggregated in a combiner and finally verified in the reducer. Due to the similarity to FullFilteringJoin we do not show an example.



$M_1$: $\langle rid, tok^* \rangle \rightarrow \langle tok, (rid, l) \rangle$
$R_1$: $\langle tok, (rid, l) \rangle \rightarrow \langle (rid_1, rid_2), (l_1, l_2, 1) \rangle$ // short lists
$\langle tok, (rid, l) \rangle \rightarrow \langle tok, (rid, l)^* \rangle$ // long lists
$M_2$: $\langle tok, (rid, l)^* \rangle \rightarrow \langle (rid_1, rid_2), (l_1, l_2, 1) \rangle$ // long lists
$C_2/R_2$: see Figure 4

**Figure 6: V-SMART Dataflow**



$M_1$: $\langle rid, tok^* \rangle \rightarrow \langle tok, 1 \rangle$
$R_1$: $\langle tok, 1 \rangle \rightarrow \langle tok, count \rangle$ // global token frequency
$M_2$: $\langle tok, count \rangle \rightarrow \langle count, tok \rangle$
$R_2$: $\langle count, tok \rangle \rightarrow \langle tok \rangle$ // sorted list of tokens, single reducer
$M_3$: $\langle rid, tok^* \rangle \rightarrow \langle \underline{tok}, (rid, tok^*) \rangle$ // prefix inverted lists
$R_3$: $\langle \underline{tok}, (rid, tok^*) \rangle \rightarrow \langle rid_1, rid_2 \rangle$ // verified pairs
$M_4$: $\langle rid_1, rid_2 \rangle \rightarrow \langle (rid_1, rid_2), null \rangle$
$R_4$: $\langle (rid_1, rid_2), null \rangle \rightarrow \langle rid_1, rid_2 \rangle$ // deduplicated pairs

**Figure 7: VernicaJoin Dataflow. Prefix tokens are underlined.**

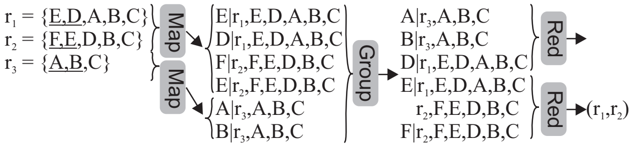


**Figure 8: VernicaJoin, Job 3, Example. Tokens are ordered by GTF, prefix tokens are underlined.**

*Discussion.* Similar to FullFilteringJoin, the intermediate data exchange is dominated by the quadratic number of candidates produced from long inverted lists. Although the candidates for long lists are generated by multiple mappers, the overall burden on the reducers in Job 2 is the same as for FullFilteringJoin.

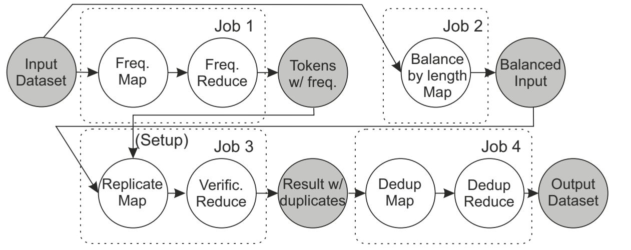
**VernicaJoin (VJ)** [30]. VJ is based on the *prefix filter* [7], a technique successfully applied in non-distributed SSJ algorithms. The $k$-prefix of a set are its $k$ first elements in an arbitrary yet fixed order. With appropriate prefix sizes, a candidate pair of sets can be pruned if their prefixes have no common element [19]. To improve the pruning power of the prefix filter, the sets are ordered by ascending GTF, i.e., rare tokens appear first in the prefix. The prefix size depends on the similarity threshold (small prefix for high similarity), the record length, and the similarity function. The prefix size for Jaccard is $k = |r| - \lceil |r| \cdot t \rceil + 1$.

Figure 7 gives an overview of VJ. Jobs 1 and 2 count and sort (in a single task of $R_2$) the tokens by GTF, respectively. Mapper $M_3$ loads the resulting sort order in the setup function and creates the inverted index on the tokens in the prefix (we underline prefix tokens, $\underline{tok}$); $R_3$ generates candidate pairs from the inverted lists that are immediately verified. Since different reducers may generate identical result pairs, a final deduplication step is required (Job 4). Figure 8 illustrates Job 3 for our running example.

Similar to FullFilteringJoin, VJ builds an inverted index on tokens and generates candidate pairs from the records in the inverted lists. However, VJ differs as follows. First, VJ builds the inverted index only on prefix tokens, thus reducing the length of the lists. Second, VJ does not generate all possible pairs from an inverted list, but applies filters proposed in the non-distributed PPJoin+ [34] algorithm to reduce the candidate set (length, positional, and suffix filter). Third, the inverted lists store – in addition to the record ID – all tokens of the original record; this is necessary for verification.

*Discussion.* The amount of data exchanged is dominated by Job 3. $M_3$ generates an inverted list entry $\langle \underline{tok}, (rid, tok^*) \rangle$ for each token $\underline{tok}$ that appears in some prefix. With the prefix length $|r| - \lceil |r| \cdot t \rceil + 1$ for Jaccard, $|M_3| = (1-t) \cdot |R| \cdot \overline{|r|} + |R|$. The list entry stores all tokens of the original record and is of length $|r| + 2$ for record $r$, thus the output size of $M_3$ is $O(|R| \cdot \overline{|r|}^2)$. The output of $R_3$ is quadratic in the frequency of the tokens in the prefix: for an inverted list $L$, $\binom{|L|}{2}$ pairs are generated and verified in the worst case. This is a pessimistic upper bound since the filters may reduce the number of candidate pairs.

**MGJoin (MG)** [24]. MG extends VernicaJoin with two ideas. First, in addition to GTF-ordered prefixes, other prefix orders are also applied: GTF-ordered prefixes are indexed to generate candidates (like in VernicaJoin). Two additional prefix orders different from GTF are used to filter the resulting candidates before verification. Second, a load balancing job groups the input records into partitions with a similar length distributions before the inverted index on the GTF prefixes is computed.



$M_1/R_1$: see Figure 7 // compute global token frequency
$M_2$: $\langle rid, tok^* \rangle \rightarrow \langle rid, tok^* \rangle$ // balance records by length
$M_3$: $\langle rid, tok^* \rangle \rightarrow \langle \underline{tok}/l, (rid, tok^*, \underline{tok}^*) \rangle$ // prefix inv. lists
$R_3$: $\langle \underline{tok}/l, (rid, tok^*, \underline{tok}^*) \rangle \rightarrow \langle rid_1, rid_2 \rangle$ // verification
$M_4/R_4$: see Figure 7 // deduplication

**Figure 9: MGJoin Dataflow.**

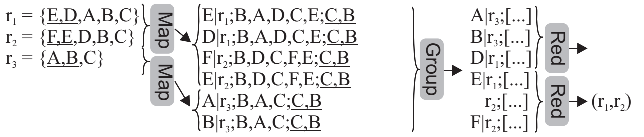
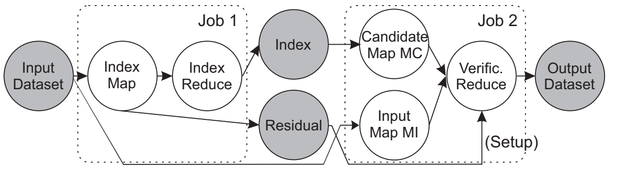


**Figure 10: MGJoin, Job 3, Example. Prefix tokens are underlined. Indexed prefix ordered by GTF, non-indexed prefix by reverse GTF, random order is B,A,D,C,F,E.**

Figure 9 illustrates MG. Job 1 counts token frequencies to establish a global order, which is loaded in the setup function of Job 3. Job 2 (map-only) distributes the records to HDFS files such that each file contains a mixture of short and long records. Subsequent mappers of Job 3 use the file boundaries as input split (by system default), so each mapper operates on a mixture of short and long records for load balancing. Job 3 (mapper) creates an inverted index on the GTF-ordered prefixes. A $\langle \underline{tok}/l, (rid, tok^*, \underline{tok}^*) \rangle$ list entry stores the record ID ($rid$), all tokens of the record in random order, and finally the prefix in reverse GTF. The record length $l$ is used as a secondary key to sort the tokens within each inverted list;



$$
\begin{aligned}
M_1\colon\quad & \langle rid, tok^*\rangle \rightarrow \begin{cases} \langle rid,\ \underline{tok}^*\rangle & \text{// residuals}\\ \langle \underline{tok}, (rid, tok, l)\rangle & \text{// inv. list entries}\end{cases}\\
R_1\colon\quad & \langle \underline{tok}, (rid, tok, l)\rangle \rightarrow \langle \underline{tok}, (rid, tok, l)^*\rangle \quad \text{// prefix index}\\
MC_2\colon\quad & \langle \underline{tok}, (rid, tok, l)^*\rangle \rightarrow \langle (rid_1, rid_2), (rid_2, tok, l_2)\rangle\\
& \qquad\qquad \text{// candidates generated from index}\\
MI_2\colon\quad & \langle rid, tok^*\rangle \rightarrow \langle (rid, null), tok^*\rangle \quad \text{// load input for join}\\
R_2\colon\quad & \langle (rid_1, rid_2|null), tok^*|(rid_2, tok, l_2)\rangle \rightarrow \langle rid_1, rid_2\rangle
\end{aligned}
$$

**Figure 11: SSJ-2R Dataflow.**

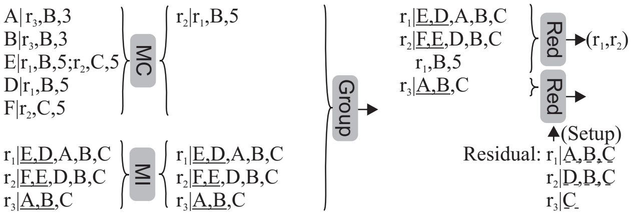


**Figure 12: SSJ-2R, Job 2, Example.**

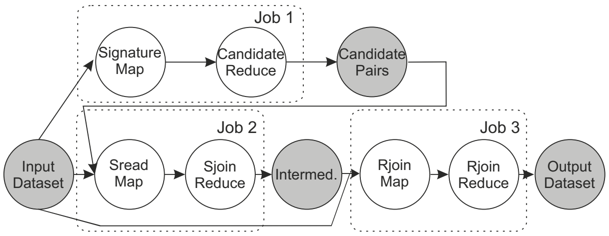
the reducer generates candidate pairs from the inverted lists using a length filter (i.e., pairs that cannot reach the similarity threshold based on their length difference are not considered). Before verifying a pair, the overlap of the random prefixes and the reverse GTF prefixes is computed. A candidate pair needs verification only if all prefixes have non-zero overlap. Job 4 removes duplicates.

*Discussion.* The amount of data exchanged is similar to VernicaJoin, except that the entries in the inverted lists are larger since they contain an additional prefix.

**SSJ-2R (S2)** [5]. Similar to VJ, S2 uses a prefix index to generate candidates, but addresses the problem of large entries in the inverted lists. VernicaJoin must replicate the entire record in each entry of the inverted list for verification. S2 splits the records into a prefix and a residual (mapper $M_1$ in Figure 11). The prefixes are indexed, and an inverted list entry contains the record ID, the last token in the prefix, and the record length. Mapper $MC_2$ generates candidate pairs $(rid_1, rid_2)$ such that the last prefix token in $rid_1$ is larger than the last prefix token in $rid_2$. Mapper $MI_2$ reads all input records, which are then joined on $rid_1$ of the candidate pairs (group step before $R_2$). For $rid_2$ only the residuals are required: the overlap between record $rid_1$ and the prefix of $rid_2$ is the number of candidates pairs $(rid_1, rid_2)$. The residuals are loaded to each reducer task using the setup function.

*Discussion.* The index generated by $R_2$ has the same cardinality as the index in VernicaJoin, but each list entry consists of only 4 integers such that the overall index size is limited to $O(|R| \cdot \overline{|r|})$. The mapper $MC_2$ outputs all 2-combinations of an inverted list $L$, i.e., the number of candidates for $L$ is quadratic in $|L|$ (like for VernicaJoin). The residuals consist of $|R|$ records of average length $t \cdot \overline{|r|}$; for large similarity thresholds $t$ the residuals may be almost as large as the input dataset. The residuals must be loaded by each task of $R_2$, which is infeasible for large datasets.

**MassJoin (MJ)** [9]. MJ uses signatures based on the pigeon-hole principle and extends the non-distributed Pass-Join [15]. For each record, MJ generates a set of signatures such that two matching records must share at least one signature. Record pairs with a common signature are candidates that must be verified. Mapper $M_1$ (cf. Figures 13) computes an inverted index on signatures. The list entries are record IDs with some additional information for pruning. Reducer $R_1$ generates candidate pairs (all 2-combinations) from an inverted list; the pruning info is leveraged to decrease the candidate set. The output format is a record ID with a list of candidates (in Figure 14 the candidate lists are of length 1). Jobs 2 and 3 join the input to the candidate pairs to verify the candidates in $R_3$.



$$
\begin{aligned}
M_1\colon\ & \langle rid, tok^*\rangle \rightarrow \langle signature, (rid, pruneinfo)\rangle \quad \text{// inverted list entry}\\
R_1\colon\ & \langle signature, (rid, pruneinfo)\rangle \rightarrow \langle rid_1, rid_2^*\rangle \quad \text{// candidate list of } rid_1\\
M_2\colon\ & \text{(identity)}\\
R_2\colon\ & \langle rid_1, (rid_2^*|tok^*)\rangle \rightarrow \langle rid_1, (rid_2^*, tok^*)\rangle \quad \text{// get tokens of } rid_1\\
M_{3a}\colon & \langle rid_1, (rid_2^*, tok^*)\rangle \rightarrow \langle rid_2, (rid_1, tok^*)\rangle \quad \text{// prepare join on } rid_2\\
M_{3b}\colon & \text{(identity)}\\
R_3\colon\ & \langle rid_2, (rid_1, tok^*)|tok^*\rangle \rightarrow \langle rid_1, rid_2\rangle \text{ // get tokens of } rid_2\text{, verify}
\end{aligned}
$$

**Figure 13: MassJoin Dataflow.**

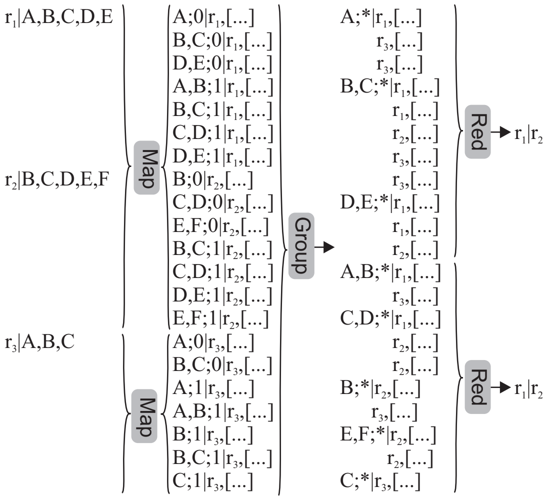
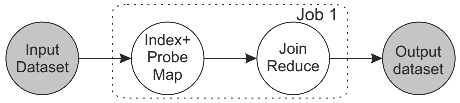


**Figure 14: MassJoin, Job 1, Example.**

*Discussion.* The amount of exchanged data is dominated by $M_1$'s output. In [9], the authors show that $M_1$ generates $|M_1| = \sum_{r\in R} \frac{(1+t^3)(1-t)^3}{t^3} \cdot |r| \cdot C + \frac{1-t}{t} \cdot |r|$ ($C$ is a constant) signature records of $\frac{|r|}{\frac{1-t}{t}\cdot|r|+1} + 35$ integers each. The size of $M_1$'s output grows with the record length and decreasing similarity thresholds.

**MRGroupJoin (GJ)** [10]. GJ groups records by length and partitions the records in each group into subrecords containing a disjunctive subset of the tokens. To generate candidates, a probe record $r$ is probed against all groups of records containing subrecords of potentially matching lengths. A candidate record $s$ must share at least one subrecord with $r$, which is ensured by the pigeonhole principle. GJ requires only a single MapReduce job (cf. Figures 15, 16). $M_1$ partitions a record $r$ into sub-partitions $par$ for the index length $len = |r|$ and the probe lengths $len \in [t * |r|, |r|]$. The key is the pair $(par, len)$, the value is the record $r$ and an index/probe flag. $R_1$ computes and verifies candidates. Candidates are computed as the cross product of all index records with all probe records for a given key $(par, len)$.



$M_1$: $\langle rid, tok^* \rangle \rightarrow \langle (par, len), (rid, tok^*, flag) \rangle$
$R_1$: $\langle (par, len), (rid, tok^*, flag) \rangle \rightarrow \langle rid_1, rid_2 \rangle$

**Figure 15: MRGroupJoin Dataflow.**

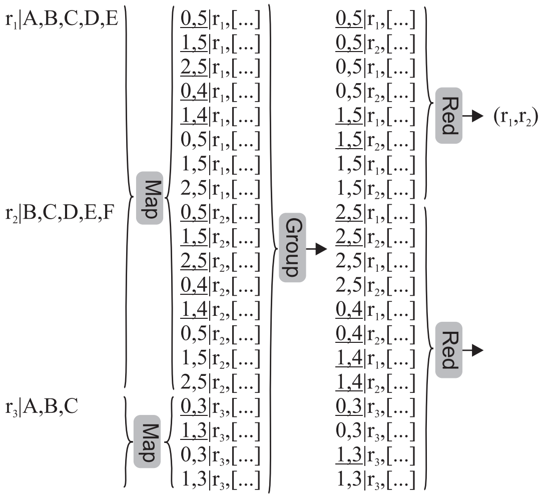


**Figure 16: MRGroupJoin, Example. Index keys are underlined. Non-underlined keys are probe keys.**

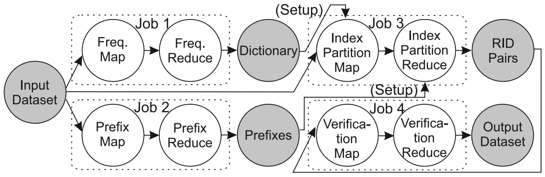
*Discussion.* The number of records produced by the mapper is $|M_1| = \sum_{r \in R} (\frac{1-t}{t} \cdot |r| + \sum_{t \cdot |r| \leq s \leq |r|} (\frac{1-t}{t} \cdot s))$, the record size is $|r| + 4$ integers.

**FS-Join (FS)** [23]. FS-Join sorts the input records in GTF order and splits them into disjoint *segments* using so-called pivot tokens as separators (*vertical partitioning*). The order number of a segment is its key. All segments with the same key are grouped into *fragments*. Segments from different fragments have zero overlap. Thus, each fragment is joined independently and produces a set of record ID pairs with a partial overlap. The fragment join uses the prefixes of the input records, the length filter, and some segment specific pruning techniques to decrease the output. To verify a record pair, the partial overlaps of all its segments are summed up. An optional length-based *horizontal partitioning* allows distributing a fragment to different nodes at the cost of replicating data. Job 1 (cf. Figures 17, 18) computes the global token frequency, which is used in $M_3$ to choose good pivot tokens. $R_3$ loads the prefixes (computed in Job 2), joins the fragments, and outputs candidate pairs. Job 4, finally, verifies the candidate pairs.

*Discussion.* Job 3 dominates the runtime. $M_3$ produces $|M_3| = |R| \cdot (p+1)$ segments, where $p$ is the number of pivots; the overall output size is $|R| \cdot (|r| + p + 1)$ integers since each segment has a key and no data is replicated. $|R_3| = (p+1) \cdot |R|^2$ records of length 5 in the worst case. This upper bound is pessimistic since the data is sparse and filters reduce the output size. The prefixes of all records must be loaded by each task of $R_3$.

## 3.2 Metric partitioning based algorithms

**MRSimJoin (MR)** [27]. MR parallelizes the non-distributed QuickJoin algorithm of [13]. QuickJoin uses pivots to partition the metric space with hyperplanes. The resulting partitions are joined independently in main memory. If a partition does not fit into main memory, it is further partitioned, i.e., by a hash function. Records that fall into border areas are replicated into dedicated window partitions that must be joined separately. MR is illustrated in Figure 19. Before Job 1 starts, random pivot records are drawn. The pivots are used by mapper $M_1$ to assign each input record to its base partition. If a record is too close to another partition, it is replicated to the respective window partition. Reducer $R_1$ processes and joins the partitions that fit into main memory and outputs the other partitions (that are too large) to HDFS. MR is recursively called on the intermediate partitions until no partition is left.



$M_1/R_1$: see Figure 7 // compute global token frequency
$M_2$: $\langle rid, tok^* \rangle \rightarrow \langle (rid, len), \underline{tok^*} \rangle$ // compute prefixes
$R_2$: (identity)
$M_3$: $\langle rid, tok^* \rangle \rightarrow \langle frag, (rid, \underline{tok^*}) \rangle$ // compute segments
$R_3$: $\langle frag, (rid, \underline{tok^*}) \rangle \rightarrow \langle (rid_1, l_1, rid_2, l_2), par\_olap \rangle$ // seg. overl.
$M_4$: (identity)
$R_4$: $\langle (rid_1, l_1, rid_2, l_2), par\_olap \rangle \rightarrow \langle rid_1, rid_2 \rangle$ // aggreg. & verify

**Figure 17: FS-Join Dataflow. Prefixes and segments underlined.**

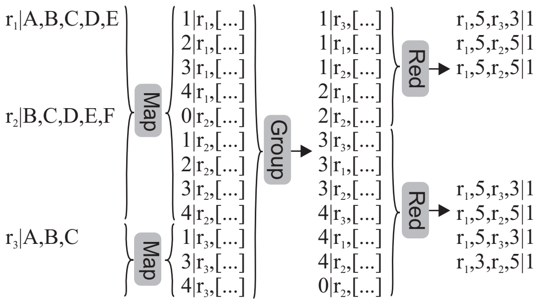
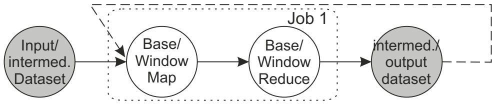


**Figure 18: FS-Join, Job 3, Example. Pivots are B,D,E,F.**

*Discussion.* Each record is assigned to one of the $p$ base partitions. In addition, it may be replicated to at most $p-1$ window partitions, so the maximum number of intermediate records is $|M_1| = p \cdot |R|$. The output records of $M_1$ contain a partition ID, the ID of a record $r$, and all tokens of $r$.

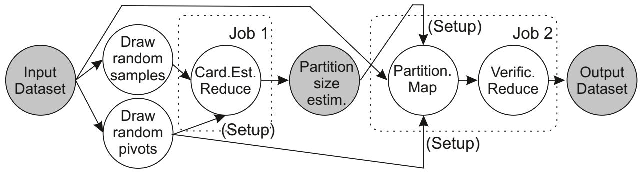
**ClusterJoin (CJ)** [25]. Similar to MR, CJ uses random pivots to split the data into disjoint partitions and to replicate border objects to window partitions. But, to avoid iterations for large partitions, CJ estimates the partition sizes in a preprocessing step, and then replicates partitions that exceed a user-defined threshold, following the Theta-Join approach [22]. For Jaccard similarity, the authors discuss a length filter to reduce the candidate size [25]. Similar to MR, CJ uses random pivots to split the data into disjoint partitions and to replicate border objects to window partitions.

Figure 20 illustrates the dataflow of MR. Before Job 1 starts, random pivot records *and* random sample records are drawn. Job 1 estimates the partition cardinalities from these two datasets. Mapper $M_2$ assigns the records to their partitions. If the estimated partition size exceeds a user-defined threshold, the records are hashed and replicated into sub-partitions such that all record pairs appear in at least one sub-partition. $R_2$ verifies the pairs that can be formed within each (sub-)partition.



$M_1$: $\langle rid, tok^* \rangle \rightarrow \langle partition, (rid, tok^*) \rangle$
$R_1$: $\langle partition, (rid, tok^*) \rangle \rightarrow \langle rid_1, rid_2 \rangle$

**Figure 19: MRSimJoin Dataflow.**



$M_1$: $\langle rid, tok^* \rangle \rightarrow \langle partition, size \rangle$ // estimate partition sizes
$M_2$: $\langle rid, tok^* \rangle \rightarrow \langle partition, (rid, tok^*) \rangle$
$R_2$: $\langle partition, (rid, tok^*) \rangle \rightarrow \langle rid_1, rid_2 \rangle$

**Figure 20: ClusterJoin Dataflow until no intermediate data is left.**

*Discussion*. The size of the intermediate results is at least $M_2 = |R|$ (if no records fall into a window partition and all partitions are small). In addition, there may be at most $(p-1) \cdot |R|$ records in window partitions. Finally, large partitions are split and replicated; the size of all sub-partitions is quadratic in the partition size, which may substantially increase the intermediate result size.

# 4. COMPARATIVE EVALUATION

We next present our experimental analysis. We implemented all algorithms from Section 3 (FF, GJ, MG, MJ, VS) or adapted existing code if available (CJ, FS, MR, S2, VJ). We evaluated the algorithms using 12 datasets. Our analysis focuses on runtime, but we also discuss data grouping, data replication, and cluster utilization.

## 4.1 Setup

**Hadoop**. We deployed all methods on Hadoop 2.7 (using YARN, cf. Section 2). The experiments run on an exclusively used cluster of 12 nodes equipped with two Xeon E5-2620 2GHz of 6 cores each (with Hyper-threading enabled, i.e., 24 logical cores per node), 24GBs of RAM, and two 1TB hard disks. All nodes are connected via a 10GBit Ethernet connection. We configured Hadoop according to Table 1. We assigned twice as much memory to reduce compared to map tasks, because a reducer needs to buffer data. The number of mappers is limited to the number of HDFS blocks of the input; by default, the HDFS block size is 64MBs or 128MBs but as our input data is usually smaller, we set this value to 10MBs. The maximum number of reduce tasks is set to 4 reducers per node, which underutilizes the available memory slightly. This is recommended, because other Hadoop system tasks (especially HDFS) need memory as well. We vary these memory settings and the number of reducers in our experiments. The speculative task execution allows Hadoop to start an already running part of a job (for example, a reduce task) on another node in parallel. The faster job wins, the slower one is killed. Since we run each test three times and report the mean of the measured runtimes, we disable this feature to ensure consistent results. By default we also disable map output compression since the bottleneck turns out to be reduce-side buffering, not network traffic; we run a separate experiments to test the effect of enabling compression.

**Table 1: Hadoop configuration.**

| Parameter | Value | Parameter | Value |
|---|---|---|---|
| Map task memory | 4GB | Min vcores/container | 1 |
| Reduce task mem. | 8GB | Max vcores/container | 32 |
| Reduce tasks/node | 4 | Min mem/container | 2GB |
| Compute nodes | 12 | Max mem/container | 8GB |
| HDFS replication | 3 times | Speculative task exec. | disabled |
| HDFS block size | 10MB | Map output compr. | disabled |

**Table 2: Characteristics of the experimental datasets.**

| Dataset | # recs | Record length | | Universe $\cdot 10^3$ | | Size (B) |
|---|---|---|---|---|---|---|
| | $\cdot 10^5$ | max | avg | size | maxFreq | |
| AOL | 100 | 245 | 3 | 3900 | 420 | 396M |
| BPOS | 3.2 | 164 | 9 | 1.7 | 240 | 17M |
| DBLP | 1.0 | 869 | 83 | 6.9 | 84 | 41M |
| ENRO | 2.5 | 3162 | 135 | 1100 | 200 | 254M |
| FLIC | 12 | 102 | 10 | 810 | 550 | 92M |
| KOSA | 6.1 | 2497 | 12 | 41 | 410 | 46M |
| LIVE | 31 | 300 | 36 | 7500 | 1000 | 873M |
| NETF | 4.8 | 18000 | 210 | 18 | 230 | 576M |
| ORKU | 27 | 40000 | 120 | 8700 | 320 | 2.5G |
| SPOT | 4.4 | 12000 | 13 | 760 | 9.7 | 41M |
| UNI | 1.0 | 25 | 10 | 0.21 | 18 | 4.5M |
| ZIPF | 4.4 | 84 | 50 | 100 | 98 | 33M |

**Datasets**. We use 10 real-world and 2 synthetic datasets from the non-distributed experimental survey in [19]; Table 2 summarizes the characteristics of these datasets. Records in AOL are very short, but draw tokens from a large universe. In contrast, BPOS and DBLP have a small universe, but short and long records, respectively. The token frequency roughly follows a Zipfian distribution in all datasets, i.e., there is a large number of infrequent tokens (less than 10 occurrences). As an exception, NETF involves very few infrequent tokens. By *maxFreq*, we denote the maximum frequency of the tokens in a dataset. LIVE has a high maxFreq, while SPOT has a very low maxFreq. Synthetic datasets UNI and ZIPF are generated following a uniform and a Zipfian token distribution, respectively. ORKU is the only dataset above 1GB, which is still small enough to compute SSJ without parallelization, i.e., using methods from [19]. Last, all datasets are free from exact duplicates, because exact duplicate elimination is a different problem from similarity joins and it makes our results comparable to the existing non-distributed study [19].

**Tests**. To compare the performance of the investigated algorithms, we conducted three types of tests. First, we applied all methods to compute a self-join of the datasets in Table 2. Second, we investigated the scalability of the algorithms by artificially increasing the size of the datasets. Third, we describe the effects when varying other parameters such as memory settings, which determine the number of YARN containers. Subsequently, we discuss how the algorithms replicate and distribute intermediate data, show results of repeated experiments from the literature, and summarize our findings for each algorithm.

## 4.2 Performance and Robustness

**Performance**. Table 3 reports the join runtime of the examined algorithms while varying the Jaccard similarity threshold inside $\{0.6, 0.7, 0.8, 0.9, 0.95\}$. For practical reasons, we consider a timeout of 30mins after which the execution of an algorithm is terminated. Our timeout is higher than 3 times the highest runtime amongst the winners over all datasets and all thresholds of the non-distributed study (494 seconds for NETF threshold 0.6) [19]. Inside each table cell, we report the lowest observed runtime in seconds followed by the corresponding algorithm (underlined); note that below this "winner", we also list the algorithms (if any) that came

**Table 3: Fastest algorithms; runtime in seconds, timeout 30mins. Fastest algorithm underlined.**

| Dataset | Jaccard threshold | | | | |
|---|---|---|---|---|---|
| | 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
| AOL | 166<br>VJ<br>MG | 155<br>VJ<br>MG | 84<br>GJ | 68<br>GJ | 64<br>GJ |
| BPOS | 123<br>VJ<br>MG | 116<br>VJ<br>MG | 101<br>GJ | 101<br>GJ | 106<br>VJ<br>GJ, MJ |
| DBLP | 342<br>VJ | 174<br>VJ | 129<br>VJ | 112<br>VJ<br>MG | 111<br>FS<br>VJ |
| ENRO | 323<br>VJ | 230<br>MG<br>VJ | 161<br>MG<br>FS | 130<br>FS<br>MG VJ | 127<br>FS<br>MG, VJ |
| FLIC | 234<br>MG<br>VJ | 163<br>MG<br>VJ | 119<br>GJ<br>MG | 86<br>GJ | 85<br>GJ |
| KOSA | 138<br>VJ<br>MG | 121<br>VJ<br>MG | 117<br>VJ<br>MG | 113<br>VJ<br>FS, MG | 112<br>VJ<br>FS, MG |
| LIVE | 313<br>VJ | 285<br>VJ | 278<br>VJ<br>MG | 254<br>VJ<br>MG | 243<br>VJ<br>GJ, MG |
| NETF | T | T | 527<br>VJ | 215<br>VJ | 161<br>VJ |
| ORKU | T | 1592<br>MG | 941<br>VJ<br>MG | 761<br>GJ<br>VJ | 681<br>VJ |
| SPOT | 128<br>MG<br>FS | 120<br>FS<br>MG | 119<br>FS<br>MG | 118<br>FS<br>MG, VJ | 114<br>FS<br>MG, VJ |
| UNI | 89<br>GJ | 74<br>GJ | 70<br>GJ | 45<br>GJ | 39<br>GJ |
| ZIPF | 114<br>VJ<br>MG | 109<br>VJ<br>FS, MG | 105<br>FS<br>MG, VJ, | 103<br>FS<br>GJ, MG<br>VJ | 59<br>GJ |

**Table 4: Gap factors: deviation from best runtime.**

| | FS | GJ | MG | MJ | VJ |
|---|---|---|---|---|---|
| mean | 3.85 | 4.91 | 1.31 | 8.32 | 1.18 |
| median | 1.97 | 2.21 | 1.07 | 2.52 | 1.00 |
| maximum | 21.63 | 16.59 | 3.65 | 139.19 | 2.67 |

**Table 5: Timeouts (30mins) per algorithm, dataset, and threshold.**

| FS | 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
|---|---|---|---|---|---|
| AOL, DBLP, LIVE, UNI | T | | | | |
| ORKU | T | T | | | |
| NETF | T | T | T | | |
| GJ | 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
| DBLP | T | T | | | |
| KOSA, LIVE | T | T | T | | |
| ENRO, NETF, ORKU, SPOT | T | T | T | T | T |
| ZIPF | T | | | | |
| MJ | 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
| DBLP, FLIC, ZIPF | T | | | | |
| ENRO, NETF, ORKU | T | T | T | T | |
| KOSA, LIVE | T | T | | | |
| MG | 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
| NETF | T | T | T | T | |
| ORKU | T | | | | |
| VJ | 0.6 | 0.7 | 0.8 | 0.9 | 0.95 |
| ORKU, NETF | T | T | | | |

out as at most 10% slower. We mark the enforcement of the timeout by the letter "T". We observe that VJ is the clear winner of the tests; VJ reported the lowest runtime 27 times, followed by GJ with 15, FS with 9, and MG with 6. Notice that neither the filter-and-verification algorithms FF, VS, S2 nor the metric-based algorithms MR, CJ ever appear in Table 3, as they failed to produce competitive runtimes (i.e., at most 10% above the best) or timed out. We elaborate on the reasons behind this behavior in Section 4.5.

Figure 1 summarizes our findings on the relative performance of the algorithms compared to the results reported on the corresponding publications. Our experiments confirm that VJ is faster than FF from [5], VJ is faster than VS [25], and FS is faster than VS [23]. However, in our experiments, VJ is faster than CJ (equal runtime in [25]), VJ is faster than MG (contrary to [24]), VJ is faster than MJ (contrary to [9]), VJ is faster than S2 (contrary to [5]), VJ is faster than VS (contrary to [21]), and VJ is faster than FS (contrary to [23]). In Section 4.6, we investigate these inconsistencies by repeating experiments from the original publications.

**Robustness**. We next analyze the robustness of the algorithms; we omit the results on CJ, FF, MR, S2, VS, which timed out on more than 60% of our experiments. We adopt the notion of the *gap factor* employed in [19]; more specifically, we measure the average, median, and maximum deviation of an algorithm's runtime from the best reported runtime. Table 4 reports the deviation factors for FS, GJ, MG, MJ, and VJ over all datasets and all thresholds in $\{0.6, 0.7, 0.8, 0.9, 0.95\}$. We excluded experimental runs with timeouts in the calculation. The most robust algorithm is VJ. On average, it shows 1.18 times the runtime of the winner (including the cases when VJ records the best runtime), 1.0 time in the median, and only 2.67 times maximum. The second most robust algorithm is MG which in the worst case has 3.65 times the runtime of the fastest algorithm. Finally, Table 5 summarizes for which combinations of algorithm, threshold, and dataset, a timeout occurred.

## 4.3 Scalability

Practically, the datasets of Table 2 can be processed by state-of-the-art non-distributed algorithms in main memory; [19] provides a competitive experimental analysis under this setup. In fact, non-distributed algorithms outperform Hadoop-based solutions in the majority of the datasets (cf. Table 4 in [19]). This behavior comes as no surprise due to the overhead induced by the MapReduce framework for starting/stopping jobs and transferring data between the cluster nodes. Figure 21 reports this data-independent overhead for all algorithms using a sample of 100 records of AOL.

However, distributed algorithms should be able to process much larger datasets than non-distributed ones. In this spirit, we report on the scalability of the algorithms in settings that justify the need for MapReduce. We focus only on FS, GJ, MG, MJ, and VJ as the other methods failed to handle even the small datasets of Table 2. For our scalability tests, we artificially increased the size of our datasets. We adopted the procedure from [30], which preserves the original universe size and the record lengths, but increases the number of similar record pairs linearly with respect to an increase factor $n$. Each record is copied $n$ times while every token in a record is shifted by $n$ positions in the global token frequency. Thus, the number of records for each dataset is increased $n$ times as well as the maximum frequency of each token is roughly increased $n$ times. The record lengths and the universe size do not change. We used moderate values $n = 5$ and 10 for the enlargement factor and a Jaccard threshold of 0.95.

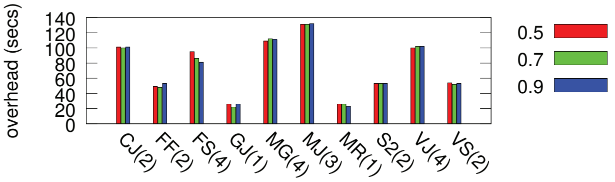


**Figure 21: Data-independent overhead for Jaccard threshold inside $\{0.5, 0.7, 0.9\}$; number of MapReduce steps given in brackets. For very small datasets and high similarity thresholds the overhead takes a large share of the overall runtime (cf. Table 3).**

**Table 6: Timeouts (120mins) on scalability tests; Jaccard similarity threshold $t = 0.95$.**

| Dataset | FS | | | GJ | | | MG | | | MJ | | | VJ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1x | 5x | 10x | 1x | 5x | 10x | 1x | 5x | 10x | 1x | 5x | 10x | 1x | 5x | 10x |
| AOL | | | T | | | | | | | | | | | | |
| BPOS | | | | | | | | | | | | | | | |
| DBLP | | | | | | | | | | | | | | | |
| ENRO | | | | T | T | T | | | | | | | | | |
| FLIC | | | | | | | | | | | | | | | |
| KOSA | | | | | | | | | | | | | | | |
| LIVE | | | T | | | | | | | | | | | | |
| NETF | | | | T | T | T | | | | | | | | | |
| ORKU | | T | T | T | T | T | | | T | | T | T | | | T |
| SPOT | | | | T | T | T | | | | | | | | | |
| UNI | | | | | | | | | | | | | | | |
| ZIPF | | | | | | | | | | | | | | | |

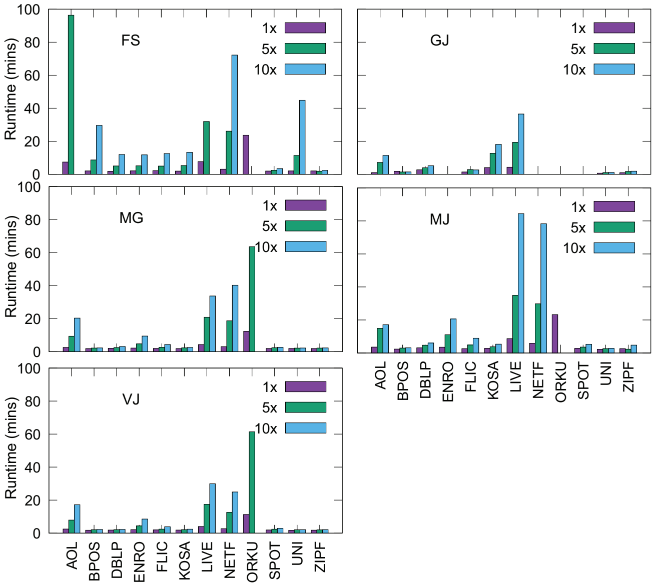


**Figure 22: Runtimes on scalability tests; Jaccard similarity threshold $t = 0.95$; timeouts excluded.**

Figure 22 reports the runtimes of the algorithms for each dataset. Table 6 shows for which combinations of algorithm and dataset timeouts (120mins in this setting) occurred. We observe that VJ, MJ, and MG better coped with the size increase for the majority of the datasets; an exception rises only for ORKU, where all algorithms timed out. FS and GJ also timed out on a number of other datasets. In Section 4.5, we discuss reasons for these results.

## 4.4 Varying the Cluster Configuration

*Compression.* We test the effect of enabling map output compression. On the small datasets (1x, Table 2), VJ, FS, and MG benefit from compression (13-19% shorter runtime), while the runtimes of GJ and MJ do not change. The runtime advantages occur in the join phases of the algorithms (job 3 of VJ and MG, jobs 3 and 4 of FS). On larger datasets (5x, 10x), enabling compression increases the runtimes of all algorithms except MJ (same runtime). Compression decreases the network load, but the bottleneck for large datasets is the reducer memory. When the transferred data does not fit the reducer memory, it must be spilled to disk such that increasing the network transfer rate does not help. The increase in runtime can be attributed to the overhead of compression. MJ produces smaller data groups per reducer and is not affected.

*Number of reducers.* The number of reducers is controlled by the memory per reducer and is in addition bound by a parameter $max$ for the maximum number of reducers. We decrease the memory per reducer from 8 GB (our default) to 4 GB, $max = 48$. With these settings, the utilization reaches the maximum of 48 reducers. For the small datasets, all algorithms profit from the larger number of reducers (17-26% shorter runtimes). With the 5x dataset sizes, VJ, MJ, and MG gain performance (resp. 30%, 30%, and 17%), while GJ runs slower by 9-17%. Increasing the maximum number of reducers to $max = 60$ increases the utilization up to 60, but does not change the runtimes. Further decreasing the available memory per reducer to 2 GB does not affect the runtimes of VJ, GJ, MG, and MJ, while the runtime of FS gets worse. On the 10x datasets, only VJ and MG gain from setting the reduce memory to 4 GB (14-25%), all other algorithms show similar (FS, GJ) or worse runtimes (MG 14%). Overall we note that the memory per reducer (and the resulting number of reducers) has some impact on the runtime, but the effect is small compared to the differences resulting from the use of different algorithms.

## 4.5 Analysis and Discussion

We next analyze the distributed execution of the algorithms and provide insights for their runtime behavior observed in the previous sections. We discuss intermediate data replication and distribution relative to characteristics of the input data and the similarity threshold, how well the computation load is distributed over time (cluster utilization), and specific limitations of each algorithm.

Hadoop-style MapReduce requires intermediate data to be buffered on the reducers until all mappers finish their execution (Figure 3). For the runtime, it is crucial that none of the reduce buffers spill to disk. The execution time of only one straggling reducer can dominate the overall runtime. All algorithms presented in Section 3 use replication to achieve a high level of parallelization. Further, they attach a key to every intermediate record; these records are then grouped and each reducer is assigned a particular set of keys (and corresponding records). This key assignment is also crucial for the runtime, because it determines whether all reduce tasks get a balanced share of the overall computation. For most algorithms, the key generation and the replication depends on data characteristics such as the maximum global token frequency without considering memory restrictions of the execution system.

We consider our experimental results and the intermediate data exchange discussions in Section 3. In our setup, each reducer has 8GBs of memory and so, the *total amount of main memory* (TMM) is $48 \cdot 8 = 384$GBs. Recall that the number of map tasks depends on the number of HDFS input blocks, and the number of reduce tasks is at most $12 \cdot 4 = 48$ for 12 nodes with 4 reducers per node. Figure 23 reports our measurements on cluster utilization. Due to lack of space, we only show a fraction of the conducted tests and

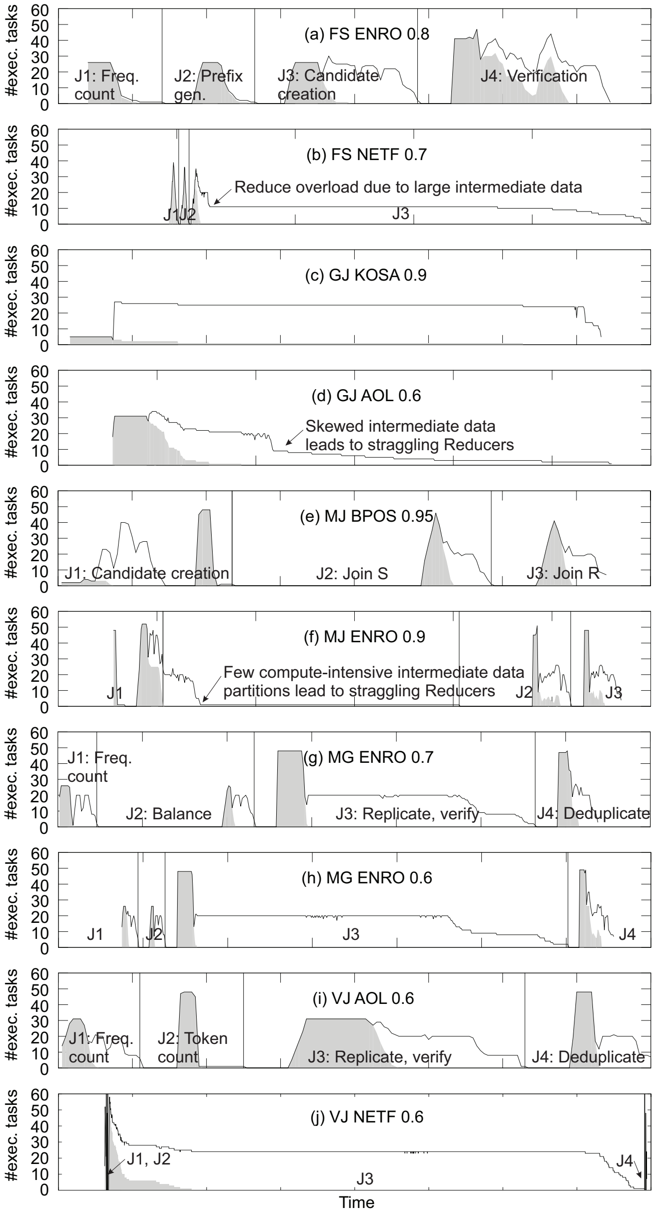


**Figure 23: Cluster utilization for each algorithm. Vertical lines divide jobs as described in Section 3. Runtimes relative to each job. Number of tasks is sum of map and reduce tasks. The grey areas mark the map share of the tasks.**

focus on FS, GJ, MG, MJ, and VJ; recall that the other algorithms timed out on more than 60% of our tests.

**VJ**. Replication and verification step (Job 3, Figure 7) dominates the runtime of VJ. The utilization graphs for VJ follow the pattern of Figure 23(i). Recall that VJ replicates the full input records for each prefix token. Low similarity thresholds and long records lead to more prefix tokens. For 1xNETF and a similarity threshold of 0.6, the amount of intermediate data already grows up to approximately 212GBs. Since the local inverted indices on the reducers additionally buffer these data, memory overload occurs in the execution phase of reduce after shuffling, which is captured by a high utilization which decreases very slowly in Figure 23(j). The efficient execution of VJ is thus limited to a combination of similarity threshold and record lengths so that the intermediate data does not exceed roughly half of the TMM. This issue could be solved by adding more nodes. However, each intermediate data group size is determined by the frequency its key token occurs within all prefixes. This frequency is likely to grow for increasing dataset sizes so that data groups hit the memory limit of single reducers. This is a hard limit which cannot be solved by more nodes.

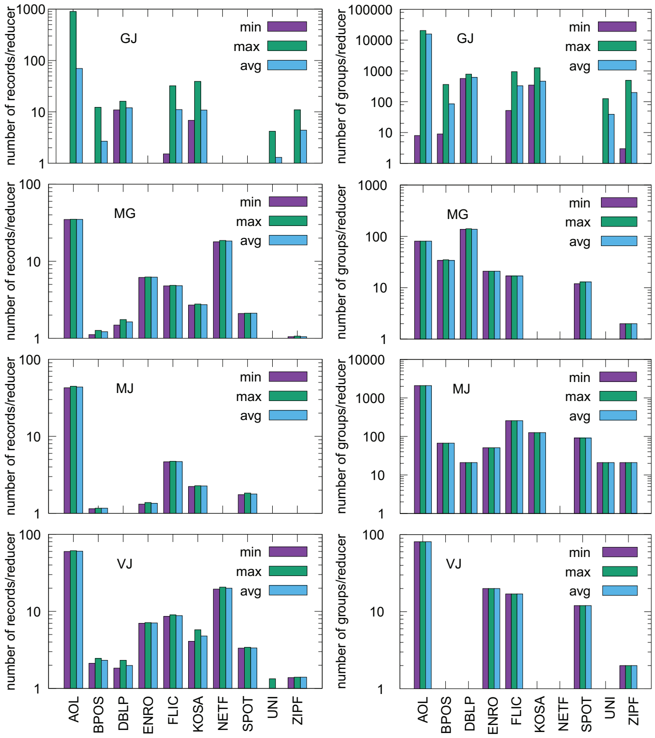


**Figure 24: Data grouping and replication (number of records or groups $\times 1000$) at the reduce step computing the join; sim. threshold $t = 0.95$; all datasets 10x synthetically enlarged.**

*Summary.* Our tests suggest that VJ is both the fastest and the most robust technique for distributed set similarity joins. Still, VJ is sensitive to long records and/or frequent tokens, where the memory on individual reducers becomes a bottleneck and limits scalability.

**GJ**. GJ consists of only one MapReduce job. A high and stable cluster utilization for large values of the similarity threshold is observed for GJ on most datasets (Figure 23(c)). The level of utilization decreases sharply at the end of the join evaluation, which reflects positively on the total runtime. On the other hand, Figure 23(d) illustrates the straggling reducer effect, which occurs on the AOL, DBLP, ENRO, LIVE datasets for a similarity threshold below 0.7. On ORKU and NETF, GJ runs into timeouts. Figure 24 shows the data distribution of GJ for varying datasets (omitting datasets where timeouts occurred). The minimum and maximum number of records per reducer greatly varies, which explains the straggler effect. Recall that GJ splits the input records into subgroups, which are matched by a group key in the reducer. The size of these groups depends on the order and distribution of the tokens in the input records, which is not discussed in [10]. GJ is thus limited to input datasets with a "good" token order that does not lead to overloaded reducers. Furthermore, the algorithm replicates each input record $\frac{1-t}{t} \cdot |r| + 1$ times (for indexing) and roughly $|r| - t \cdot |r|$ times (for probing). Each intermediate record contains approximately 4 integers plus the original data. Now, consider a Jaccard similarity threshold of 0.6 and 10x-ORKU, which consists of $2.7 \cdot 10^7$ records with an average length of 120. Every record will be roughly replicated $\frac{1-0.6}{0.6} \cdot 120 + 1 + 120 - 0.6 \cdot 120 = 129$ times, which results in $2.7 \cdot 10^7 \cdot 120 \cdot 129 \approx 390$GBs of intermediate data exceeding TMM. The high replication limits the applicability of GJ to short input records and high similarity thresholds.

*Summary.* GJ is the runner-up in the number of wins on the tested dataset/threshold settings. The algorithm benefits from high similarity thresholds and datasets with short records. However, GJ is not robust: it times outs even for small datasets when the similarity threshold is small or the records are long. Note that we are the first to test GJ under a distributed setup since the original paper [10] evaluates only the non-distributed scenario.

**S2 and FS**. Recall from Section 3 that one reducer on both algorithms needs to load the prefix/residual file into main memory; the size of this file grows linear with the input cardinality. This step limits the applicability of S2 and FS. Consider for example S2 on 10x-ORKU, which contains $2.7 \cdot 10^7$ records with an average record length of 120 and assume a similarity threshold of 0.9. The residual length is $0.9 \cdot 120 = 108$ and the residual file roughly occupies $2.7 \cdot 10^7 \cdot 108 \cdot 4 \approx 11$GBs (with 4 bytes for an integer), which exceeds the available memory on a reducer. Another limitation specific to FS is the fragment size (=number of intermediate records per reducer), as each reducer computes the costly (despite all filters) cross join on its fragment.

In Figure 23(a), we show a typical execution of FS without overload, while (b) shows the effect of (evenly) overloaded reducers in Job 3. Fragment sizes in our setup are 30-40% of the number of input records for SPOT (runtime winner, short input records of length 13 on average), 87-95% for ENRO (good runtimes, medium large records of length 135), and 80-90% for NETF (timeouts, long records of length 210). This indicates that increasing record lengths have a negative impact on the runtime. The fragment sizes could decrease by adding more pivots and reduce nodes, but the authors of FS suggest to use the number of nodes minus one as the number of pivots, so that each reducer gets exactly one fragment. Further, the token order determines whether a record participates in a fragment. The algorithm uses the inverse GTF, which does not explicitly optimize the fragment assignment. In the worst case, each record participates in every fragment.

*Summary.* FS was the third fastest algorithm. The novel vertical partitioning manages to reduce data replication. However, the prefix list must be loaded into the main memory of each reducer, which clearly limits the scalability of FS. In contrast to VJ, S2 does not replicate the input records in the inverted list index. Unfortunately, each reducer must load the entire residual file to the main memory, which limits the scalability of S2.

**MG**. Recall that MG works similar to VJ, but balances input records by length and transfers multiple prefixes in intermediate data to accelerate the verification, leading to larger records. Job 3 (Figure 9), which replicates the input and verifies candidates, dominates the runtime. For the majority of our tests, MG demonstrates a stable and high utilization in this phase (Figure 23(g)). We observe that the number of tasks remains constant while only the range of this constant utilization in the join phase varies (Figure 23(h)). Compared to VJ, the reduce compute load is distributed more evenly, so the additional balancing step of MG has a positive impact on the even distribution of the compute load. The distribution of the compute load is only loosely coupled with the distribution of the intermediate records.

Figure 24 shows the number of records and data groups per reducer for MG and VJ; both algorithms show a comparably even intermediate data distribution. Yet, MG does not solve the limitations described for VJ previously.

*Summary.* MG sticks out as a robust algorithm. Although it is usually slower than VJ, MG is often among the fastest algorithms and wins for low similarity thresholds on some datasets.

**MJ**. For the majority of our experiments, MJ exhibited a utilization level similar to Figure 23(e). But, on datasets of long records (NETF) or of high maxFreq (AOL, BPOS, DBLP, ENRO, FLIC, KOSA, LIVE, NETF, ORKU) combined with a similarity threshold below 0.7, some reducers in Job 2 straggle (Figure 23(f)). A low value of maxFreq as in SPOT can compensate for a high maximum record length. Our experiments on 10x-ENRO showed that MJ is able to evaluate the join for a similarity threshold of $t = 0.95$, but times out when $t < 0.9$. To investigate this behavior, we experimented with more threshold values inside $[0.9, \ldots, 0.99]$. The lowest runtime was recorded for a threshold of 0.93. Most importantly though, the volume of the intermediate data increases roughly from 110GBs to 259GBs within the $[0.9, \ldots, 0.99]$ interval, which means that intermediate data is not the dominating factor for the overall runtime. Figure 24 shows data grouping and replication for MJ. Despite the even distribution of intermediate records and keys among the reduce tasks, straggling reducers occur as shown in Figure 23 (f). For a similarity threshold above 0.93, the execution of the reducers dominates the entire join computation, while for thresholds below 0.93, signature creation in the mappers takes increasingly more time. We repeated our experiments on MJ with more compute nodes (24 nodes). In this setup, map congestion disappears, but straggling reducers are still present. The main reason is that the signature creation only assures a good intermediate data distribution, which does not necessarily lead to an even distribution of the compute load.

*Summary.* MJ scales to large datasets, but fails to compete with the previous algorithms on performance and robustness, due to its expensive signature creation and verification. Although MJ evenly distributes the *number* of intermediate data records, the workload still varies among the nodes which leads to straggling reducers.

**FF and VS**. Consider 1xAOL. From Table 2, maxFreq $= 4.2 \cdot 10^5$, hence, for the most frequent token, a particular candidate mapper (Job 2 in Figure 4 and 6) emits $\binom{4.2 \cdot 10^5}{2} = 88 \cdot 10^9$ records. Each record contains 4 integers for FF or 5 for VS, all of 4 bytes, so this mapper produces overall $88 \cdot 10^9 \cdot 4 \cdot 4 \approx$ 1,311GBs or $88 \cdot 10^9 \cdot 5 \cdot 4 \approx$ 1,639GBs of data, respectively. The volume of these intermediate data already exceeds the TMM without considering the remaining universe tokens. Although both algorithms use combiner functions, in practice they fail to shrink the intermediate data sent to the subsequent reducer sufficiently. Increasing the number of cluster nodes will not address this problem, because the volume of the intermediate data grows quadratic with maxFreq. As a result, both FF and VS can only process datasets of low maxFreq.

*Summary.* FF and VS operate in a similar manner; their data replication factor is quadratic, i.e., the algorithms are sensitive to frequent tokens. They cannot compete with the other filter-and-verification methods, time out frequently, and do not scale.

**MR and CJ**. Both algorithms draw a number of $|P|$ random pivots from the dataset and then replicate every input record up to $|P|$ times; in fact the replication factor can be even higher for CJ in case the hash-based replication is additionally used. Consider 10x-NETF and assume that 1000 pivot elements are selected. Each intermediate record contains $|r| + 7$ integers, so based on Table 2, the intermediate data occupies $1000 \cdot 4.8 \cdot 10^6 \cdot (210{+}7) \cdot 4 \approx$ 3,880GBs, which exceeds our TMM by one order of magnitude. There is additional replication for the window partitions in the same order of magnitude for our (high-dimensional) datasets, as the hyperplanes do not partition the high-dimensional space effectively; the data points are too close. The tests in [25] and [10] suggest that the algorithms perform better on data with a low number of dimensions.

*Summary.* The metric-based approaches did not perform well in our tests; CJ and MR often time out even for small inputs. This is due to their high level of data replication.

## 4.6 Reproducing Previous Results

We repeated core experiments for VJ [30], S2 [5], MJ [9], and FS [23]. It was not possible to repeat tests for CJ [25], as the experimental setup (parameters of the method, hardware setting) is not specified and a publicly unavailable dataset was used. The experimental parameters for FF were not given on [12] either, while for VS [21], a larger cluster than ours and a publically unavailable dataset were used. Also, MG [24] used a larger cluster and a DBLP dataset which was tokenized/preprocessed in a way we could not reproduce, leading to large deviations in maxFreq. Finally, GJ [10] was never tested on a distributed setup, and for MR [27], a different similarity function (Euclidean) was used. Unless stated otherwise, our Hadoop cluster is configured according to Table 1. Our tests can reproduce the results of **VJ** and **S2**. Due to lack of space, we only report on algorithms with deviating results.

**MJ**. Contrary to the original paper, our experiments showed that VJ is faster than MJ. In [9], MJ is compared to VJ on ENRO with a 10 nodes cluster, varying the Jaccard similarity threshold. As MJ computes an end-to-end non-self R×S join, the dataset is split into two equal-sized parts. Figure 25 reports the results of this comparison. On the other hand, our focus is on self-joins; hence, we generated an input by sampling 50% of ENRO. Also, we only join record IDs as our focus is not on an end-to-end computation. Figure 26 reports our results. Our implementation of both MJ and VJ recorded lower absolute runtimes compared to [9]. This is expected as we do not compute an end-to-end join. We also observe that VJ outperforms MJ in our test, which contradicts the original results. There are two potential reasons for this behavior. First, as already discussed, MJ is optimized for non-self joins. Second, MJ is designed to perform an end-to-end join; reporting full records in the results comes for free as MJ needs the full records to perform the verification step. In an effort to conduct a fairer comparison, we repeated this test as an end-to-end non-self R×S join. Our VJ implementation labels each input record by “R” or “S”, while during the join phase, record pairs of the same label are pruned. Further, VJ involves an additional layer which joins the record ID based results with the input records. With these modifications, although VJ’s runtime increased compared to the self-join, VJ remained faster than MJ for similarity thresholds below 0.9.

*Discussion.* We could not reproduce the runtimes of MJ; in our experiments the runtimes are higher for a threshold of 0.75 and lower for 0.8(5). The competing VJ shows runtimes of nearly one magnitude more in the original experiments, which are unclear us.

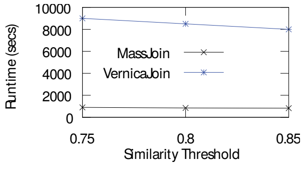


Figure 25: Original results

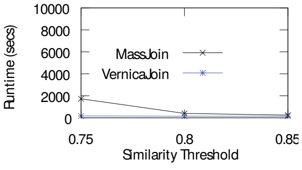


Figure 26: Our results

**FS**. Contrary to the original paper, our experiments show that VJ is faster than FS. Rong et al. [23] compared FS to VJ on ENRO varying the Jaccard similarity threshold. The test ran on an 11 nodes cluster; each node had 15GBs of RAM while 3 reduce tasks were allowed per node. Figure 27 reports the results. To repeat this experiment, we considered a similar cluster setup of 12 nodes with 12GBs of RAM each and 4 reducers per node. However, there is a difference in the employed tokenization strategy. According to [23], the input dataset contains $517k$ records, while the record length varies from 51 to $148k$ tokens. With our tokenizer, a record contains from 1 to $3k$ tokens; we refer to this tokenization setup as E1. To address this issue, we used the tokenizer offered by Rong et al. in their publicly available source code. Yet, we were not able to reproduce exactly the same ENRO dataset used in [23]; this setup (named E2) contains records of 37-76$k$ tokens. Figure 28 reports our results for setups E1 and E2. E1: The runtime of FS is similar to the original result (Figure 27), but VJ is at least one order of magnitude faster in our experiments; this is expected as VJ’s prefix filter benefits from the shorter records of E1. E2: Surprisingly, FS is slower than in the original result, while VJ is again faster.

*Discussion.* We could reproduce the runtimes of the original experiment using the publicly available dataset and our tokenizer. However, the characteristics of our tokenized data differs from the original paper; we were not able to reproduce the same characteristics by using the publicly available tokenizer of the original paper. The high runtimes of VJ in the original experiments remain unclear.

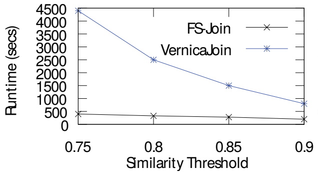


Figure 27: Original results

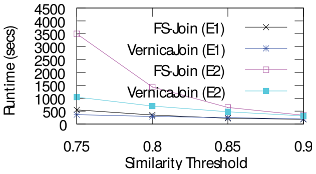


Figure 28: Our results

## 5. CONCLUSIONS AND FUTURE WORK

We conducted an experimental study on ten recent Hadoop-based SSJ algorithms focusing on runtime. We considered a fair experimental environment with a harmonized problem statement, common Hadoop settings, input preprocessing, and equal implementation optimizations. The winning algorithm concerning runtime and robustness w.r.t. various data characteristics is VJ. This refutes experimental results of previous papers, where VJ was reported to be outperformed by its competitors. We repeated experiments from previous papers and discussed reasons for the diverging results. Winner number two is GJ, which was not compared to any other algorithm so far. Number three is the most recent FS algorithm.

The motivation to use distributed computing for the SSJ problem are large data volumes that cannot be handled by a single node. None of the algorithms in this survey scales to large input datasets. We analyzed the reasons both analytically and based on measurements. The main bottleneck are straggling reducers due to high and/or skewed data replication between the map and the reduce tasks. This effect is triggered by specific characteristics of the input data. Adding more nodes does not solve this problem.

For the future, it would be interesting to adapt the approaches to newer platforms such as Flink or Spark. They provide the possibility to create more complex dataflow programs without the need to separate programs into several MR jobs that write their (intermediate) output to disk. Furthermore, these systems provide more complex operators such as joins. However, the challenge of efficiently grouping and replicating intermediate data remains and is an interesting direction for future research.

## 6. ACKNOWLEDGMENTS

We thank Willi Mann [19] for providing the datasets we used. This work was supported by the Humboldt Elsevier Advanced Data and Text (HEADT) Center, by a grant for the DFG Graduate School GRK 1324, and the Austrian Science Fund (FWF): P29859-N31.

## 7. REFERENCES

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *28th International Conference on Data Engineering (ICDE)*, pages 498–509. IEEE, 2012.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, pages 918–929. VLDB Endowment, 2006.

[3] N. Augsten, M. Böhlen, and J. Gamper. The pq-gram distance between ordered labeled trees. *ACM Transactions on Database Systems (TODS)*, 35(1):4, 2010.

[4] N. Augsten, A. Miraglia, T. Neumann, and A. Kemper. On-the-fly token similarity joins in relational databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 1495–1506. ACM, 2014.

[5] R. Baraglia, G. De Francisci Morales, and C. Lucchese. Document similarity self-join with mapreduce. In *10th International Conference on Data Mining (ICDM)*, pages 731–736. IEEE, 2010.

[6] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, pages 131–140. ACM, 2007.

[7] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 5–5. IEEE, 2006.

[8] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, pages 137–150, 2004.

[9] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *IEEE 30th International Conference on Data Engineering (ICDE)*, pages 340–351. IEEE, 2014.

[10] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *Proceedings of the VLDB Endowment*, 9(4):360–371, 2015.

[11] D. Dey, S. Sarkar, and P. De. A distance-based approach to entity reconciliation in heterogeneous databases. *IEEE Transactions on Knowledge and Data Engineering*, 14(3):567–582, 2002.

[12] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, pages 265–268. Association for Computational Linguistics, 2008.

[13] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Transactions on Database Systems (TODS)*, 33(2):7, 2008.

[14] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[15] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *Proceedings of the VLDB Endowment*, 5(3):253–264, 2011.

[16] W. Luo, H. Tan, H. Mao, and L. M. Ni. Efficient similarity joins on massive high-dimensional datasets using mapreduce. In *13th IEEE International Conference on Mobile Data Management, MDM*, pages 1–10, 2012.

[17] Y. Ma, S. Jia, and Y. Zhang. A novel approach for high-dimensional vector similarity join query. *Concurrency and Computation: Practice and Experience*, 29(5), 2017.

[18] Y. Ma, X. Meng, and S. Wang. Parallel similarity joins on massive high-dimensional data using mapreduce. *Concurrency and Computation: Practice and Experience*, 28(1):166–183, 2016.

[19] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 9(9):636–647, 2016.

[20] A. Metwally, D. Agrawal, and A. El Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. In *Proceedings of the 16th international conference on World Wide Web*, pages 241–250. ACM, 2007.

[21] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.

[22] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 949–960. ACM, 2011.

[23] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and scalable distributed set similarity joins for big data analytics. In *IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1059–1070. IEEE, 2017.

[24] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. Tung. Efficient and scalable processing of string similarity join. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2217–2230, 2013.

[25] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *Proceedings of the VLDB Endowment*, 7(12):1059–1070, 2014.

[26] Y. N. Silva, J. Reed, K. Brown, A. Wadsworth, and C. Rong. An experimental survey of mapreduce-based similarity joins. In *International Conference on Similarity Search and Applications*, pages 181–195. Springer, 2016.

[27] Y. N. Silva and J. M. Reed. Exploiting mapreduce-based similarity joins. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 693–696. ACM, 2012.

[28] J. Sun, Z. Shang, G. Li, D. Deng, and Z. Bao. Dima: A distributed in-memory similarity-based query processing system. *PVLDB*, 10(12):1925–1928, 2017.

[29] M. Theobald, J. Siddharth, and A. Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 563–570. ACM, 2008.

[30] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[31] S. Wandelt, D. Deng, S. Gerdjikov, S. Mishra, P. Mitankin, M. Patil, E. Siragusa, A. Tiskin, W. Wang, J. Wang, et al. State-of-the-art in string similarity search and join. *ACM SIGMOD Record*, 43(1):64–76, 2014.

[32] Y. Wang, A. Metwally, and S. Parthasarathy. Scalable all-pairs similarity search in metric spaces. In *Proceedings of SIGKDD international conference on Knowledge discovery and data mining*, pages 829–837. ACM, 2013.

[33] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[34] 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)*, 36(3):15, 2011.