Parallelizing Set Similarity Joins

Dissertation
zur Erlangung des akademischen Grades
Dr. rer. nat.
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin
von
Dipl.-Inf. Fabian Fier

Präsidentin der Humboldt-Universität zu Berlin
Prof. Dr.-Ing. Dr. Sabine Kunst

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

Gutachter
1. Prof. Johann-Christoph Freytag, Ph.D.
2. Prof. Dr. Felix Naumann
3. Prof. Chen Li, Ph.D.

Tag der mündlichen Prüfung:
23.11.2021

Abstract

One of today's major challenges in data science is to compare and relate data of similar nature. For example, web archives de-duplicate web crawls to present similar web pages only once, data integrators use approximate string matching to relate data from different sources, and online advertisers want to identify collaborative click fraud by finding users with similar usage patterns. Using the *join* operation known from relational databases could help solving these problems and related ones. Given a collection of records, the join operation finds all pairs of records, which fulfill a user-chosen predicate. There are efficient join methods for basic predicates, i. e., equality. However, real-world problems could require more complex predicates: The previous examples require *similarity*. A common way to measure similarity are set similarity functions, such as Jaccard. In order to use set similarity functions as predicates, we assume records to be represented by *sets* of tokens (i. e., words of a text). In this thesis, we focus on the *set similarity join (SSJ)* operation.

The amount of data to be processed today is typically large and grows continually. On the other hand, the SSJ is a compute-intensive operation. It has a quadratic time complexity with respect to the input size. There are sophisticated methods to reduce the practical compute effort if certain data properties are present. The most prominent method is the filter-and-verification approach. However, this approach alone is not sufficient to compute the SSJ on the amount of data, which is common today. To cope with the increasing size of input data, additional means are needed to develop scalable implementations for SSJ. In this thesis, we focus on *parallelization*. We make the following three major contributions to SSJ.

First, we elaborate on the state-of-the-art in parallelizing SSJ. Existing methods use shared-nothing parallelization on the MapReduce programming paradigm. We compare ten approaches from the literature analytically and experimentally. We evaluate them by setting up a fair benchmark and discuss their strengths and limits. Their main limit is surprisingly a low scalability due to too high and/or skewed data replication. None of the approaches could compute the join on large datasets.

Second, we leverage the abundant CPU parallelism of modern commodity hardware, which has not yet been considered to scale SSJ. We propose a novel data-parallel multi-threaded SSJ based on filter-and-verification methods. Our approach provides significant speedups compared to single-threaded executions.

Third, we propose a novel highly scalable distributed SSJ approach. It overcomes the limits and bottlenecks of existing MapReduce SSJ approaches. With a cost-based heuristic

and a data-independent scaling mechanism we avoid 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 join execution and processes much larger datasets than all parallel approaches designed and implemented so far.

Zusammenfassung

Eine der größten Herausforderungen in Data Science ist heutzutage, Daten miteinander in Beziehung zu setzen und ähnliche Daten zu finden. Web-Archive deduplizieren beispielsweise Web-Crawls, um ähnliche Webseiten nur einmal auszugeben. In der Datenintegration wird approximatives String-Matching eingesetzt, um Daten von verschiedenen Quellen zu vereinigen. Für Online-Werbetreibende ist es wichtig, kollaborativen Klickbetrug zu erkennen, was durch das Finden ähnlicher Nutzungsstrukturen erreicht werden kann. Für die genannten sowie verwandte Aufgaben kann der aus relationalen Datenbanken bekannte *Join*-Operator eingesetzt werden. Der Join-Operator findet alle Record-Paare aus einer Eingabemenge von Records, die ein benutzerdefiniertes Prädikat erfüllen. Für einfache Prädikate, wie z. B. Gleichheit, existieren effiziente Berechnungsmethoden. Allerdings erfordern reale Aufgabenstellungen mitunter komplexere Prädikate: So erfordern die eingangs genannten Beispiele das Konzept der *Ähnlichkeit*. Ähnlichkeit wird häufig durch mengenbasierte Ähnlichkeitsfunktionen, wie z. B. Jaccard, gemessen. Um mengenbasierte Ähnlichkeitsfunktionen als Prädikat nutzen zu können, setzt diese Arbeit voraus, dass Records aus *Mengen* von Tokens (z. B. Worte eines Texts) bestehen. Die Arbeit fokussiert sich auf den mengenbasierten Ähnlichkeitsjoin, *Set Similarity Join (SSJ)*.

Die Datenmenge, die es heute zu verarbeiten gilt, ist groß und wächst weiter. Der SSJ hingegen ist eine rechenintensive Operation. Er weist eine quadratische Komplexität bezogen auf die Eingabedaten auf. Wenn bestimmte Voraussetzungen der Eingabedaten erfüllt sind, existieren effiziente Ansätze, um den praktischen Berechnungsaufwand zu reduzieren. Der wichtigste Ansatz ist der Filter-und-Verifikationsansatz. Dieser alleine ist jedoch nicht ausreichend, um heutzutage gängige Datenmengen zu verarbeiten. Um mit größeren Daten umgehen zu können, sind weitere neue Ansätze notwendig. Diese Arbeit fokussiert sich auf das Mittel der *Parallelisierung*. Sie leistet folgende drei Beiträge auf dem Gebiet der SSJs.

Erstens beschreibt und untersucht die Arbeit den aktuellen Stand paralleler SSJ-Ansätze. Existierende Ansätze nutzen verteilte Parallelisierung auf Basis des MapReduce Programmierparadigmas. Diese Arbeit vergleicht zehn Ansätze aus der Literatur sowohl analytisch als auch experimentell. Sie stellt einen fairen experimentellen Benchmark vor und diskutiert Stärken und Schwächen der Ansätze. Der größte Schwachpunkt aller Ansätze ist überraschenderweise eine geringe Skalierbarkeit aufgrund zu hoher Datenreplikation und/ oder ungleich verteilter Daten. Keiner der Ansätze kann den SSJ auf großen Daten berechnen.

Zweitens macht die Arbeit die verfügbare hohe CPU-Parallelität moderner Rechner für den SSJ nutzbar. Sie stellt einen neuen daten-parallelen multi-threaded SSJ-Ansatz basierend auf der Filter-und-Verifikations-Methode vor. Der vorgestellte Ansatz ermöglicht erhebliche Laufzeit-Beschleunigungen gegenüber der Ausführung auf einem Thread.

Drittens stellt die Arbeit einen neuen hoch skalierbaren verteilten SSJ-Ansatz vor. Er beseitigt Einschränkungen existierender MapReduce-basierter Ansätze. Mit einer kostenbasierten Heuristik und einem daten-unabhängigen Skalierungsmechanismus vermeidet er Daten-Replikation und wiederholte Berechnungen. Berechnungskosten werden heuristisch möglichst gleich auf Berechnungsknoten verteilt. Eine Abschätzung des Hauptspeicherbedarfs vermeidet Swapping, was kritisch ist für die Laufzeit. Der Ansatz beschleunigt die Join-Ausführung signifikant und ermöglicht die Ausführung auf erheblich größeren Datenmengen als bisher betrachtete parallele Ansätze.

Acknowledgements

First of all I want to thank my supervisor Johann-Christoph Freytag. He always supported me and encouraged me to follow my research interest. By doing that he truly made set similarity joins *my* topic. The second person I think of related to this work is Matthias J. Sax. He recommended me to apply for a doctoral position at the group of Johann-Christoph Freytag. Without Matthias, I would not have started this adventurous endeavor.

Panagiotis Bouros, whom I got to know when he was a postdoc in the group of Ulf Leser, inspired the thesis topic. Panos' work was related to set similarity joins. We worked together on an experimental paper on MapReduce SSJ approaches with Nikolaus Augsten. I got a lot of critical and constructive feedback on the study from Ulf, Chen Li, and Mike Carey. Erkang Zhu and Tianzheng Wang worked together with me on the multi-threaded join paper.

I thank my partner Xuefeng, my family, and my friends for supporting me. Through my dear friend Zhan I just recently got to know the concepts of *future work* and *rabbit holes*. These concepts, in conjunction with the deadline, helped me a lot on the last few meters.

I got support from the graduate school METRIK, from the research initiative HEADT (Humboldt Elsevier Advanced Data and Text Center), and from a research grant from LexisNexis: Thank you to Flavio Villanustre, Trish McCall, Jessica Lorti, and the entire LexisNexis team.

Thank You to everybody, including the ones I missed to mention, for giving me this chance, supporting me, and working together with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A major challenge in data science today is to compare and relate data of similar nature. For example, web archives de-duplicate web crawls to present only relevant results to users [TSP08]. Data integrators use approximate string matching to join data items from different sources, which do not have a global join key [ABG10]. Online advertisers identify collaborative click fraud by finding similar user behavior [MAEA07].

One way to approach these and related problems is to use 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 the previously mentioned 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 set similarity measure and a user-chosen similarity threshold, the *set similarity join* (SSJ) finds all pairs of records with a similarity above the threshold with respect to this measure.

A naive approach to compute the SSJ involves to compare all possible pairs. Since the complexity of this approach is quadratic, it is not feasible even for small datasets. The most prominent approach in the literature to compute the SSJ more efficiently is the filter-and-verification framework. It does not reduce the worst-case complexity (which is quadratic), but reduces the practical compute effort when favorable input data characteristics are present. The framework first generates candidate pairs and verifies them in a second step. Sophisticated filters keep the number of candidate pairs low. This approach is efficient on single cores [MAB16]. However, it does not scale to large datasets.

To compute the SSJ on large datasets, various MapReduce-based distributed approaches based on the filter-and-verification framework 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 describe existing approaches and analyze them theoretically and practically. Our analysis and experiments show that the amount of data these approaches can process is limited. Users cannot shift the limit by adding more compute nodes due to high and skewed data replication.

One of the most obvious parallelization opportunities, using multiple cores on one computer, has not been considered for the SSJ problem so far. Arguably, the existing MapReduce approaches can be used for intra-node parallelization by spawning multiple executors on one node. However, the approaches replicate data, such as inverted indexes, because they assume a shared-nothing architecture. On one node, multiple threads can share the same data without replication, which is more RAM-efficient and opens caching opportunities. Our novel multicore approach leverages the intra-node data sharing potential. The approach efficiently uses multicore parallelization and achieves significantly lower runtimes compared to single-core executions.

While multicore parallelization reduces the runtime of the SSJ, it cannot scale to large datasets by adding more cores. Based on the knowledge of the shortcomings of the existing MapReduce approaches, we propose a novel distributed SSJ approach. It is highly scalable to hundreds of gigabytes of input. It uses intra-node multicore parallelization, avoids data replication, and uses a cost-based heuristic as well as a data-independent scaling mechanism to distribute the load to thousands of compute nodes if needed.

In the following, we define the problem we address in this thesis, summarize the main contributions, outline the structure of the work, and give an overview on previous work included.

1.1 Problem Definition

This thesis addresses the following question: *How can the all-pairs SSJ be efficiently computed on a potentially large input dataset given a large number of shared nothing nodes of modern commodity hardware?* We subsequently detail each part of this question.

Let us first formalize the *SSJ*. Given a collection of *records* (sets) R, formed over the universe U of tokens (set elements), and a similarity function between two records, $sim : \mathscr{P}(U) \times \mathscr{P}(U) \rightarrow [0, 1]$; the set similarity self-join 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 $sim(r, s) \geq \theta$. We require the result to be free of duplicates. Note that our definition requires the join result to be complete (*all-pairs* or *exact* SSJ). Furthermore, the definition requires a user chosen threshold θ (*threshold-based* SSJ). Without loss of generality and following previous work on SSJ algorithms, we focus on the *self-join* using the Jaccard similarity measure hereafter [MAB16].

Next, we define the properties of the *input datasets* we require the SSJ computation to be compatible with. We focus on textual data as input for the SSJ. Such data usually shows a roughly Zipfian token distribution and token universes up to several millions of distinct tokens. These properties are important for the choice of a suitable SSJ approach. Following previous work on single-threaded SSJs, we assume that input datasets are already tokenized and ignore the problem of efficient tokenization [AMNK14, MAB16]. All our datasets are a single collection R of sets consisting of sorted tokens. We consider finding exact duplicates an orthogonal problem and thus require input datasets to be free of duplicates. By *potentially large* we refer to datasets of dozens or hundreds of gigabytes, which have not been processed by any existing SSJ approach so far.

By *efficiently computing the SSJ* we refer to the capability of the SSJ computation to scale by using parallelization resources, such as CPUs and shared nothing compute nodes. According to Amdahl's Law, the speedup of a parallel program is limited by sequential parts of it [Amd67]. We require such sequential parts to be minimal such that the user can achieve desired small runtimes by adding more compute resources. Lastly, we require the SSJ execution to be *robust* against unfavorable data characteristics, such as stop words. Such data characteristics must not lead to straggling or other effects resulting in runtimes, which are unacceptable to users.

By *modern commodity hardware* we refer to standard computers with dozens of cores, dozens of gigabytes of RAM, sufficiently much harddisk space to hold the input and output data, and a fast network interconnection. We consider the parallelization potential of GPUs in this thesis out of scope. By a *large number of compute nodes* we refer to cloud computing where it realistic today to temporarily obtain hundreds or even thousands of nodes to compute one operation.

1.2 Contributions

The primary contributions of this thesis are:

- We analytically and experimentally compare ten MapReduce-based SSJ algorithms. We provide a fair benchmark by (re-)implementing the algorithms and running them on ten real-world and two synthetic datasets. We evaluate the algorithms varying parameters and discuss their strengths and limits. This work was published in [FAB+18].

- We propose a novel multicore-parallel filter-and-verification-based SSJ approach, which significantly speeds up the join runtime compared to using only a single core. We experimentally evaluate the runtime and scalability of this approach using the same datasets as in the MapReduce study. This work was published in [FWZF20].

- We propose a novel highly scalable distributed-parallel SSJ approach. It significantly pushes the amount of input data that can be joined compared to the existing MapReduce and the multicore approaches. It uses a data-dependent cost-based heuristic in conjunction with a data-independent scaling mechanism. We evaluate the approach using the same datasets as in the MapReduce study, enlarged artificially by a factor up to 100. The outline for this approach was published in [Fie17].

1.3 Structure

This thesis is structured as follows. Chapter 2 provides the context of the SSJ we focus on. It describes related set similarity problems, their applications, and their approaches. Furthermore, this chapter describes basic filter-and-verification SSJ concepts and important characteristics of twelve experimental datasets we use throughout this work.

In Chapter 3 we review and analyze ten MapReduce-based SSJ algorithms. We focus in particular on their scalability and runtime. By artificially enlarging the datasets we additionally highlight the limits of the approaches regarding input dataset sizes and discuss the related bottlenecks.

Chapter 4 introduces a novel filter-and-verification-based SSJ approach using multicores. We experimentally show how existing filters affect the runtimes. Additionally, we show the runtime effects of hardware-related implementation optimizations.

Motivated by the scalability limitations of the MapReduce and multicore approaches, we propose a novel distributed SSJ approach in Chapter 5. This approach is based on the insights of the MapReduce study and avoids replication. Based on a cost-based heuristic and a data-independent scaling mechanism it partitions the compute load amongst compute nodes.

This thesis concludes in Chapter 6 with a summary of the contributions. We discuss how they can be used to further improve the SSJ computation in the future.

1.4 Own Prior Work

Chapter 3 presents an experimental study on existing MapReduce SSJ approaches, which has been published in [FAB+18] with multiple authors. The authors' roles can be assigned as follows: Freytag supervised the work. All practical analysis was done by Fier, Fier wrote the manuscript. The description and theoretical analysis of the existing algorithms and the introduction is joint work of Augsten and Fier. The manuscript was critically revised by Augsten, Leser, and Bouros.

Chapter 4 presents a multicore SSJ algorithm, which has been published in [FWZF20] with multiple authors. The authors' roles can be assigned as follows: Freytag supervised the work. All analysis was done by Fier. Wang contributed the description of modern multicore systems and parts of the parallel implementation. Fier wrote the manuscript, which was critically revised by Wang and Zhu.

Chapter 2

Background

This chapter provides context for the set similarity join we focus on. We first give an overview on related similarity problems and their approaches. Throughout this thesis, we use the filter-and-verification approach. We subsequently review this approach including its basic concepts, especially filters. Lastly, we describe experimental datasets and their key characteristics we use for experiments throughout the thesis.

2.1 Related Similarity Problems

In Section 1.1 we defined the *threshold-based all-pairs* SSJ we focus on in this work. Other problems are related to this SSJ. Differences mainly occur in the application and data dimensions. Input data varies, i.e., in the way data is represented (set, multiset, string, vector), the dataset size, or the dataset characteristics, such as the size of the token universe. In the following, we give an overview on related problems, provide application examples, and describe their main approaches. Subsequently, we relate them to the SSJ we focus on.

Top-k Set Similarity Join. Given a collection of records R, the top-k set similarity join computes the k most similar record pairs $\{(r, s)|r, s \in R\}$ with respect to a given set similarity measure. Users specify the number of desired results by k. One application of a top-k set similarity join is, for example, near-duplicate web page detection [Hen06]. Xiao et al. propose a top-k SSJ algorithm based on the prefix filter [XWLS09, CGK06].

Top-k Set Similarity Search. Given a query record q and a collection of records R, the top-k set similarity search finds the k most similar records in R compared to q. JOSIE applies this operation to the search for joinable tables in massive data lakes by translating columns to sets [ZDNM19]. The query record q is a user-chosen column of the query table. A special characteristic of this application is that the input datasets have large dictionary sizes. Large dictionaries lead to large indexing structures in main memory. JOSIE therefore focuses on techniques to skip reading parts of the index. The method uses the prefix and position filters [CGK06, XWLS09].

Set Containment Join. Given a collection of set objects R, the set containment join finds all pairs of objects $\{(r, s)|r, s \in R\}$, which contain each other $r \subseteq s$. For example, the

set containment join could be used in traditional SQL database management systems to evaluate complex SQL queries based on division. LIMIT+ advances the existing prefix tree approach PRETTI by limiting the prefix tree, and by evaluating the join in a two-phase process, candidate generation and verification [BMGT16, JP05]. The work of Kunkel et al. advances this approach by intersecting prefix trees such that the operation can be computed balance-aware in parallel [KRS+16].

Sliding Window Set Similarity Join. Given a collection of (multi-)set objects R, a query document q, a window size w, and a threshold θ, the sliding window SSJ finds all pairs of windows (x, y) such that x is a window of an object $r \in R$ and y is a window of the query object q. One application of a sliding window SSJ is plagiarism detection in texts where parts are copied from other text sources with slight modifications. The approach of Wang et al. uses an extended prefix filter (k-prefix with $k > 1$) to efficiently compute this join [WXQ+16, WLF12].

Approximate/Probabilistic Set Similarity Join. Given a collection of records R and a user-chosen similarity threshold θ, the approximate SSJ finds pairs of records $\{(r, s)|r, s \in R\}$ with a similarity above the threshold with a certain probability. One application of this SSJ is plagiarism detection in textual documents. One approach to compute the approximate SSJ is Locality-Sensitive Hashing (LSH). It uses hash functions to replace the record tokens (or shingles) by much smaller signatures [LRU20]. This approach chooses hash functions such that given two signatures, it could estimate the similarity of the underlying records. The method to find such hashes is called minhashing. Based on the signatures, the LSH approach generates candidate pairs, which are to be verified subsequently. The LSH approach generates false negatives, so the candidate and the result sets might be incomplete. Another approach to compute the approximate SSJ is blocking [Chr07]. This approach creates a set of blocking keys for each record. Subsequently, it matches records by key (using indexes) and verifies the candidates. This method is approximate due to the creation of the blocking keys, which does not assure that all matching records have a matching key.

Approximate String Search. Given a query string, a similarity threshold θ, and a collection of strings, the approximate string search finds all strings in the collection, which are similar to the query string relative to a similarity measure. One application for approximate string search is spell checking. Given an input document, a spellchecker searches for each word whether it is contained in a dictionary and if not, if there is a similar one to recommend. Li et al. propose three algorithms to compute this operation [LLL08]. The paper studies the influence of the common filters to efficiently compute the operation [GIJ+01, CGK06].

Metric Space Similarity Joins. Given a collection of objects represented as vector data in a metric space and a user-chosen similarity threshold θ, the metric space similarity join computes all pairs of such objects, which are above the threshold with regard to a similarity measure. One important prerequisite for this approach is that the similarity measure is a metric and thus fulfills the triangular inequality. One application of this operation is time series analysis, for example, the analysis of stock histories. QuickJoin uses hyperplanes to break down the problem into independently computable pieces [JS08].

Spatio-Textual Similarity Join. Given a collection of objects, which carry both a textual (represented by sets) and a spatial information (represented by latitude and longitude coordinates), a user-chosen textual threshold θ, and spatial threshold ϵ, the spatio-textual join computes pairs of objects, which are spatially close and textually similar with respect to a given set similarity measure. An application for such a join is, for example, friendship recommendation in a social network. Bouros et al. propose ST-SJOIN to compute this join [BGM12]. The approach uses spatial grids and introduces a grouping mechanism to enhance the textual SSJ algorithm PPJoin+ [XWL$^+$11].

All approaches mentioned have in common that they produce candidates before verifying them in a second step. Additionally, top-k set similarity join and search, sliding window set similarity join, and spatio-textual similarity join make use of one or more of the common filters length filter, prefix filter, and position filter. We follow this approach by using the filter-and-verification framework, which we describe subsequently.

2.2 Filter-and-verification Framework

In the following, we describe the filter-and-verification framework for the SSJ, which we use throughout this thesis. We begin with the concept of set similarity, revisit common filters, and then describe the basic algorithmic approach to compute it.

2.2.1 Set Similarity

We require a similarity measure $sim(r, s)$, which returns a value in $[0, 1]$ for two input records r, s. Common similarity measures are, for example, Jaccard, Cosine, and Overlap (while the latter one is not normalized to $[0, 1]$ here, but needed for our discussion):

- $Jaccard(r, s) = \frac{|r \cap s|}{|r \cup s|}$
- $Cosine(r, s) = \frac{|r \cap s|}{\sqrt{|r| \cdot |s|}}$
- $Overlap(r, s) = |r \cap s|$.

Example 2.1. Consider the following records:

r = "do more with structured procrastination"
s = "do more by doing more"

The records from Example 2.1 can be transformed into:

$$r = \{A, B, C, D, E\}$$
$$s = \{A, B, F, G, H\}$$

using the following word-token mapping:

Word	do	more	with	structured	procrastination	by	doing	more2
Token	A	B	C	D	E	F	G	H
Token frequency	2	2	1	1	1	1	1	1

We emphasize that we replaced the second "more" of s with a different token than the first occurrence. This string-to-set transformation is a common method referred to as *counting approach* [AB13]. Considering the *inverse global token frequency* we can sort (*canonicalize*) the tokens within the records, which is an optimization we use in the filter discussion subsequently:

$$r = \{C, D, E, A, B\}$$
$$s = \{F, G, H, A, B\}$$

We can now compute the similarities of the record pair as follows: $Jaccard(r, s) = \frac{2}{8} = 0.25$, $Cosine(r, s) = \frac{2}{\sqrt{5 \cdot 5}} = 0.4$, $Overlap(r, s) = 2$. The choice of the similarity measure is application-dependent and out of scope for this thesis. Let us consider Jaccard from now.

Let us next derive important properties of the Jaccard similarity. We require the Jaccard similarity measure of two records r, s to be larger or equal to the user-defined threshold θ. We can formalize this requirement as follows:

$$Jaccard(r, s) = \frac{|r \cap s|}{|r \cup s|} \geq \theta \tag{2.1}$$

Given that

$$|r \cup s| \geq |r| \quad and \quad |s| \geq |r \cap s|, \tag{2.2}$$

we can infer:

$$Jaccard(r, s) \geq \theta \Rightarrow \theta \cdot |r| \leq |s|. \tag{2.3}$$

Equation 2.3 allows for filtering out records with non-matching lengths as we describe in the following section.

2.2.2 Filters

In this section, we describe common existing filters we use throughout our work.

Length Filter. Arasu et al. first introduced this filter as part of the PartEnum algorithm [AGK06]. Considering that two records r, s to be joined can have different lengths, we can prune records, which do not fulfill the following prerequisite (inferred from Equation 2.3):

$$\theta \cdot |r| \leq |s| \leq \frac{|r|}{\theta} \tag{2.4}$$

Example 2.2. Consider the following records and the threshold $\theta = 0.5$:

$$r = \{A, B, C, D, E\}$$
$$s = \{A, B\}$$

Since $\theta \cdot |r| = 0.5 \cdot 5 \nleq |s| = 2$ we can prune this record pair just by the knowledge of the record lengths, but without accessing the actual record tokens.

Prefix Filter. Chaudhuri et al. first introduced this filter [CGK06]. Subsequently, further SSJ algorithms integrated and refined it with tighter bounds [AGK06, BMS07, XWL+11, MA14]. The prefix filter requires that tokens have an order, which we assume to be integer values. The filter regards only subsets of tokens within each record, so-called *prefixes* to decide if a record pair might be similar. If two records r, s fulfill the similarity threshold $Jaccard(r, s) \geq \theta$ then the overlap of their prefixes is not the empty set. The definition of a prefix filter for Jaccard is as follows:

$$prefixLength(\theta, r) = |r| - \lceil \theta \cdot |r| \rceil + 1 \tag{2.5}$$

Example 2.3. Consider the following records:

$$r = \{E, A, B, C, D\}$$
$$s = \{F, A, B, C, D\}$$

With Formula 2.5 we can compute the prefix lengths for an example similarity threshold $\theta = 0.5$: $prefixLength(0.5, r) = prefixLength(0.5, s) = 5-3+1 = 3$. Let us consider the first three tokens of both records as their prefixes. The prefixes share the tokens A and B, so we can consider both records as candidates ignoring their remaining tokens. The actual Jaccard similarity of these two records is indeed $Jaccard(r, s) = \frac{4}{6} = 0.7 > 0.5$. For another similarity threshold $\theta = 0.9$, the prefix lengths are $prefixLength(0.9, r) = prefixLength(0.9, s) = 5 - 5 + 1 = 1$. The resulting prefixes of both records do not share a common token, so we can safely prune the record pair without considering the rest of their tokens.

Positional Filter. Gravano et al. first introduced this filter, focusing on string data [GIJ+01]. PPJoin adapts the filter to sets [XWL+11]. The underlying idea of the filter is a reasoning about positions over matching positions in the prefixes. We can translate the Jaccard similarity into a *minimum required overlap* between the two records by rearranging the similarity inequation:

$$Jaccard(r, s) = \frac{|r \cap s|}{|r \cup s|} = \frac{|r \cap s|}{|r| + |s| - |r \cap s|} \geq \theta \tag{2.6}$$

$$\frac{|r \cap s|}{\theta} \geq |r| + |s| - |r \cap s| \tag{2.7}$$

$$\frac{(1 + \theta) \cdot |r \cap s|}{\theta} \geq |r| + |s| \tag{2.8}$$

$$|r \cap s| \geq \frac{\theta}{(1 + \theta)} \cdot (|r| + |s|) \tag{2.9}$$

Example 2.4. Let us consider a similarity threshold $\theta = 0.8$ and the following records, which passed the length and prefix filter. We can disregard tokens with a question mark:

$$r = \{C, G, ?, ?, ?, ?, ?, ?, ?, ?\}$$
$$s = \{A, G, ?, ?, ?, ?, ?, ?, ?\}$$

Given a threshold of 0.8, we compute the minimum required overlap between the two records with Equation 2.9 resulting in the value of 9. The prefix length of s is 2, and the one of r is 3. Now, we consider record s. The first matching token with r is at the second position, leaving only seven tokens. Thus, the remaining tokens after the matching one cannot reach the minimum required overlap of 9. Thus, we can prune these two records disregarding further tokens.

In this section, we revisited three important filters commonly used in many SSJ approaches. We refer to the filters throughout the thesis. Next, we describe the basic algorithmic approach, which facilitates the filters.

2.2.3 Base Algorithm

In the following, we describe the basic algorithmic approach of the filter-and-verification framework. The framework follows a two-stage process: *candidate generation* and *candidate verification* [BMS07, XWL+11]. A candidate is a pair of records that has the potential to meet the similarity threshold. To generate candidate pairs, for each set r in the input collection R, the filter-and-verification approach aims to find other sets s in R, which contain tokens from r. An inverted index speeds up this process. The second stage verifies the candidate pairs by computing their similarities. The verification only returns the qualified pairs as final results.

To speed up the SSJ, a line of prior work focused on minimizing the number of candidates, because the exact similarity computation could be expensive. To compute the SSJ over R, $|R| \cdot |R|$ set comparisons need to be performed in the worst case. To reduce the number of candidates, state-of-the-art algorithms typically use the *length filter*, the *prefix filter*, and the *position filter* described in the previous section [AGK06, CGK06, XWL+11]. One important optimization related to the prefix filter is to pre-order the tokens in the input records using the inverted global token frequency. The token order assures that the prefixes contain only the least frequent tokens, but the result stays complete. This method could reduce the number of candidates depending on input data characteristics.

Algorithm 1: Sequential `AllPairs` algorithm.

Data: R, $invertedIndex$, θ
Result: $\{(r,s) | (r,s) \in R \times R, r \neq s, sim(r,s) \geq \theta\}$

```
1 foreach r ∈ R do
2     candidates ← {}
3     foreach token ∈ GetPrefix(r, θ) do
4         foreach s ∈ GetList(invertedIndex, token) do
5             candidates ← candidates ∪ {s}
6     foreach s ∈ candidates do
7         Verify(r, s, θ)
```

Algorithm 1 shows the major steps of AllPairs, which is the foundation for other SSJ algorithms [BMS07]. It assumes the inverted index on the prefix tokens of all records to be computed beforehand. Lines 2–5 compute the candidates by probing the index for each prefix token for each $r \in R$. Lines 6–7 compute the exact similarity of each candidate pair. PPJoin extends AllPairs by using the position filter [XWL+11].

Various other single-core filter-and-verification-based SSJ algorithms emerged based on the AllPairs approach. Since we focus on parallel SSJs, we refer to the experimental study of Mann et al., which revisits single-core algorithms and compares them experimentally [MAB16]. We follow this study and use the same experimental datasets. Subsequently, we describe these datasets and their properties we use in our experimental discussions.

2.3 Datasets

For our experiments, we use ten real-world and two synthetic datasets from the experimental survey on single-core SSJs [MAB16].

We first describe the real-world datasets: **AOL.** This dataset results from a query log of the AOL search engine [Aol]. One set represents a search string and one keyword is represented by one token. **BPOS.** This dataset contains point of sale data [Bpo]. Each set represents a purchase in a shop with tokens representing a product category. **DBLP.** This dataset consists of articles from the DBLP bibliography [Dbl]. One set of tokens represents one publication. Tokens represent q-grams of the concatenated title and author strings. **ENRO.** This data contains e-mail data [Enr]. One set of tokens represents one e-mail where each token represents one word as a concatenation of subject and body. **FLIC.** This dataset contains metadata of images (the original source is not available anymore). One set is a photography and one token is a word from the title or from a tag. **KOSA.** This dataset contains click stream data [Kos]. One set of tokens represents one user interaction recorded on a Hungarian online news portal with each token representing one link the user clicked on. **LIVE.** This dataset contains social media data [Liv]. One token set represents one user and tokens are user interests. **NETF.** This dataset contains social media data [Net]. One set represents one user and tokens are movies rated by the user. **ORKU.** This dataset contains data from a social network [Liv]. One set is one user and tokens are group memberships of the user. **SPOT.** This dataset is from a music streaming service [Spo]. One set is a user and tokens are tracks the user listened to.

Next, we describe the synthetically generated datasets. The tokens of the two synthetically generated datasets are drawn from different distributions (Uniform and Zipfian). The tokens are randomly assigned to the sets until the pre-computed set size (following a Poisson distribution) is reached: **UNI.** This dataset uses a Zipfian token distribution ($z = 1$) with an average record length of 50. **ZIPF.** This dataset uses a Uniform token distribution with an average record length of 10.

Mann et al. preprocessed the datasets [MAB16]. The preprocessing includes integer-tokenizing the datasets, which we consider out of scope for this work. Additionally, the preprocessing orders records by ascending lengths, which is required by most SSJ approaches we consider. Lastly, the preprocessing canonicalizes the records such that the tokens are sorted by increasing global token frequency. Canonicalized tokens are crucial for the efficiency of the prefix filter (cf. Section 2.2.2).

There are four characteristics of these datasets we use throughout our experimental discussions: dataset size, record length, token universe, and token distribution. Table 2.1 gives an overview on the first three of the characteristics. Column $n = 1$ refers to the original sizes of the datasets, which vary between 17MB and 2.5GB. Single-threaded SSJ algorithms efficiently compute the SSJ on such small datasets [MAB16]. Our work aims to compute the SSJ on larger data using parallelism. We artificially increase the datasets by factors $s \in \{5, 10, 25, 50, 100\}$ (in the table we only show $s \in \{1, 10, 100\}$ for brevity). We adopt the procedure from Vernica et al. for the increase [VCL10]. The procedure preserves the original universe sizes and the record lengths. It increases the number of similar record pairs linearly with respect to the increase factor s. We refer to the enlarged datasets by,

i. e., AOL×10 for the AOL dataset increased by $s = 10$. All datasets are free from exact duplicates, because we consider exact duplicate elimination an orthogonal problem.

Table 2.1: Characteristics of experimental datasets.

Dataset	Record length		Universe $\cdot 10^3$		$s = 1$		$s = 10$		$s = 100$	
	max	avg	size	maxFreq	# recs $\cdot 10^6$	size (B)	# recs $\cdot 10^6$	size (B)	# recs $\cdot 10^6$	size (B)
AOL	245	3	3900	420	10	306M	101	2.4G	1005	23G
BPOS	164	9	1.7	240	0.3	17M	3	135M	32	1.2G
DBLP	869	83	6.9	84	0.1	41M	1	389M	10	3.5G
ENRO	3162	135	1100	200	0.3	254M	2	2.5G	25	23G
FLIC	102	10	810	550	1.2	92M	12	816M	124	7.6G
KOSA	2497	12	41	410	0.6	46M	6	390M	61	3.6G
LIVE	300	36	7500	1000	3.1	873M	31	8.3G	306	79G
NETF	18000	210	18	230	0.5	567M	5	5.6G	48	54G
ORKU	40000	120	8700	320	2.7	2.5G	27	25G	273	241G
SPOT	12000	13	760	9.7	0.4	41M	4	380M	44	3.7G
UNI	25	10	0.21	18	0.1	4.5M	1	39M	10	330M
ZIPF	84	50	100	98	0.4	33M	1	303M	10	2.6G

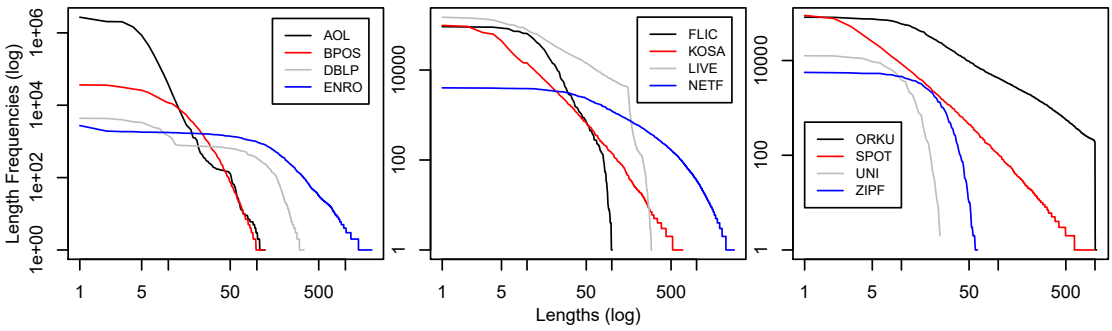


Figure 2.1: Length histograms of the datasets.

Figure 2.1 contains length histograms of the datasets. The x axis shows the record lengths and the y axis shows the corresponding frequencies. Both axes are in log scale, because all datasets reveal a skew towards many short records, especially AOL. Furthermore, most datasets exhibit a large tail with many different long records lengths, each with only a low frequency.

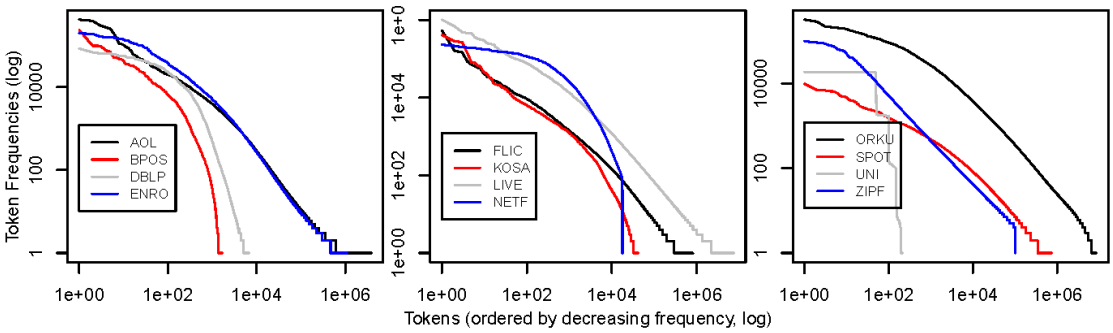


Figure 2.2: Token histograms of the datasets.

Figure 2.2 shows the token distribution of the datasets. Most datasets show a roughly Zipfian distribution. Only BPOS, DBLP, and especially NETF reveal a more uniform token distribution.

Chapter 3

Comparing MapReduce SSJ Algorithms

In this chapter, we compare ten different MapReduce-based SSJ algorithms analytically and experimentally (as published in [FAB+18]). Only little is known about their relative runtime performance, strengths, and weaknesses. Previous comparisons are limited to subsets of the algorithms. Furthermore, different test setups make them hard to compare.

We experimentally compare the ten algorithms in a uniform test environment. For the experiments, we employ twelve input datasets with varying characteristics from a broad range of applications (cf. Section 2.3). The results are surprising. All algorithms fail to scale for at least one dataset due to sensitivity to long sets, frequent set tokens, low similarity thresholds, or a combination thereof. Some algorithms even cannot compute the SSJ on small datasets, which can be processed by single-threaded algorithms. We analyze the algorithms to uncover the reasons for this limitation.

3.1 Background

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) Full-Filtering [ELO08], VernicaJoin [VCL10], SSJ-2R [BDFML10], FuzzyJoin [ASM+12], V-SMART [MF12], MRSim-Join [SR12], MG-Join [RLW+13], MAPPS [WMP13], Cluster-Join [SHC14], Mass-Join [DLH+14], MRGroupJoin [DLWF15], FS-Join [RLS+17], and DIMA [SSL+17, SSL+19]. Jiang et al. and Mann et al. compared non-distributed solutions in their experimental studies [JLFL14, MAB16]. However, there is no comprehensive comparison of distributed SSJ algorithms, except the work by Silva et al. [SRB+16], which compares FuzzyJoin [ASM+12], MRThetaJoin [OR11], MRSimJoin [OR11], Vernica [VCL10], and V-SMART [MF12]. However, the benchmark does not include several important competing algorithms. Furthermore, the validity of the experiments is limited, because the authors used only a single dataset. The empirical evaluations in the original publications of the algorithms provide only an incomplete understanding (see Figure 3.1 on the right upper part).

			CJ	GJ	FF	MG	MJ	MR	S2	VJ	VS	FS
ClusterJoin	[SHC14]	CJ								=[SHC14]	>[SHC14]	
MRGroupJoin	[DLWF15]	GJ	>									
FullFilteringJoin	[ELO08]	FF		<					<[BDFML10]	<[BDFML10]		
MGJoin	[RLW+13]	MG	>	<	>					>[RLW+13]		
MassJoin	[DLH+14]	MJ	>	<	>	<				>[DLH+14]		<[RLS+17]
MRSimJoin	[SR12]	MR		<		<	<					
SSJ-2R	[BDFML10]	S2		<		<				>[BDFML10]		
VernicaJoin	[VCL10]	VJ	>	>	>	>	>	>	>		<[MF12] >[SHC14]	<[RLS+17]
V-SMART	[MF12]	VS		<		<	<			<		<[RLS+17]
FS-Join	[RLS+17]	FS	>	<	>	<	>	>	>	<	>	

← Previous Comparisons

← Our Comparisons →

Figure 3.1: Overview of 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.

In the following, we perform a comparative evaluation of ten distributed SSJ algorithms as listed in Figure 3.1, which require the MapReduce programming framework [DG04]. To provide a fair experimental setup, we do not tailor parameters, data preprocessing, implementation details, or system configuration details to work especially well with one algorithm. We do not include DIMA in our study since (i) it builds on top of Spark by extending the Catalyst optimizer and (ii) the proposed approach for similarity joins employs offline distributed indexing [SSL+17, SSL+19]. We also exclude FuzzyJoin, which focuses on string similarity measures; although Afrati et al. argue that the proposed methods can be adapted for set similarity measures, they do not elaborate on this issue [ASM+12]. Furthermore, we do not consider MAPSS, which is tailored to dense vectors; note that vector representations of sets are typically extremely sparse [WMP13]. Last, we exclude HDSJ, which also focuses on vector data and on Euclidian distance rendering the proposed techniques not applicable to set similarity measures [LTMN12, MMW16, MJZ17].

We base our comparison on twelve datasets (ten real-world and two synthetic datasets) - as discussed before - 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.

In this chapter, we do not introduce new approaches. Thus, we chose a setting that is supported by all algorithms tested: self-join using Jaccard similarity. Figure 3.2 outlines the typical workflow of an end-to-end set similarity self-join. The input is a collection

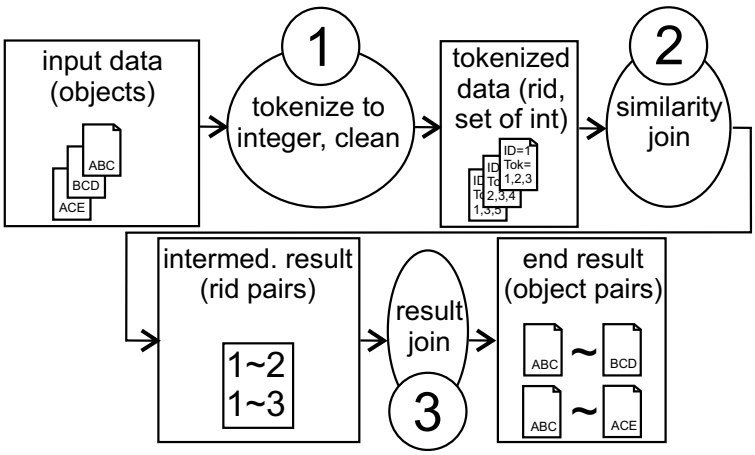


Figure 3.2: Computation of a token-based SSJ. Our work focused on step (2), the actual join.

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.

We focus in our analysis on the join, Step (2), thus not measuring the pre- and post-processing cost. The preprocessing step in our experiments is the same for all algorithms to ignore the problem of efficient tokenization [AMNK14]. 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 [DG04] and are implemented in Hadoop using its distributed file system HDFS [Whi12]. 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. Such adaptations would require non-trivial changes in the algorithms. We use Version 2 of Hadoop, which is based on the resource manager YARN. The system creates *containers* on each system node, which executes *tasks*, such as map or reduce. The number of containers depends on user-defined memory settings. For example, if the user allows the system nodes to use 10GB for Hadoop and sets the container size to 4GB, the system spawns two containers on each node. By default, the number of map tasks is 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 number of containers available.

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 may use a so-called *setup function*, which Hadoop executes once at instantiation time. Such function is useful to load global settings or data to each map/reduce task to make it available throughout the lifetime of the task. In Figure 3.3, we provide a simplified overview of relevant data flows

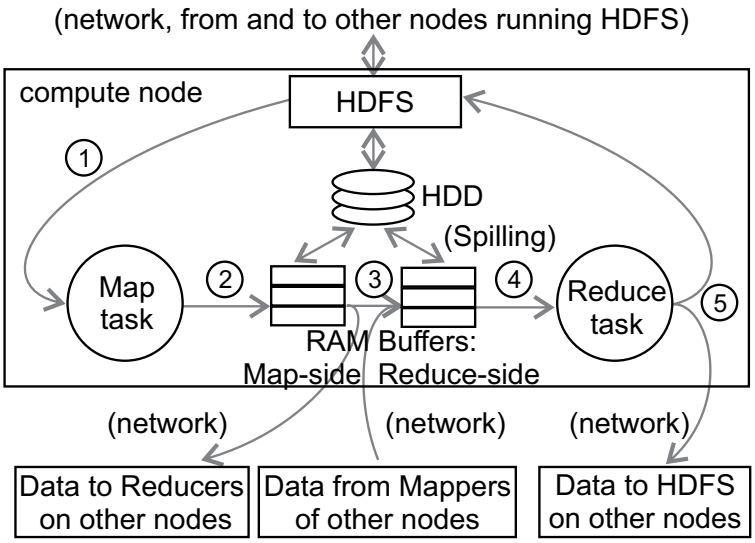


Figure 3.3: Schematic dataflow between map, reduce, HDFS, RAM, and filesystem in one Hadoop compute node. Dataflow from and to remote map, reduce, and HDFS instances.

in a MapReduce job. Map reads data blocks from HDFS (step 1 in the figure). The system hashes each output record of a map by its key (step 2), buffers it on the map side (spills it to disk if a buffer threshold is reached or if not enough reducers are free), and then routes it to the reducer responsible for its 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 system collects the input for each reducer from different map tasks (which may be local or remote) and buffers it in RAM (step 3). When the buffer is full, the systems spills data to local disk. Map-side buffers may also spill to disk, for example, when the reduce-side buffers are full. After all map tasks have finished their execution, the system sorts and groups the data at the reducers by key (*shuffling phase*, step 4). Finally, the system calls the reduce function for each key (*data group*), and saves its output to HDFS, possibly serving as an input for subsequent jobs.

3.2 Survey and Analysis

This section reviews the ten SSJ algorithms of Figure 3.1, and analyzes 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.

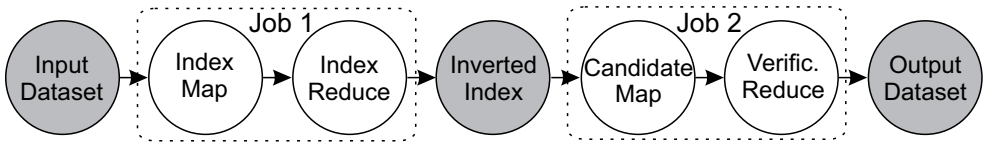
We denote the input collection with $R \subseteq \{r \mid r \subseteq U\}$. The *global token frequency (GTF)* of a token is the number of records containing this token. We provide the signatures of the map and reduce functions and denote the mapper (combiner, reducer) of job i with M_i (C_i, R_i). The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

Example 3.1. We use the following records as a 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), and the threshold is $\theta = 0.65$. 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).

3.2.1 Filter-and-verification based algorithms

FullFilteringJoin (FF). FF computes an inverted index over all tokens in Job 1 such that each token maps to all records, which contain this token. The algorithm subsequently uses the inverted lists in Job 2 to compute the pairwise record overlaps and the final join result (cf. Figure 3.4).

We discuss Job 2 (cf. Figure 3.5). 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 obtain 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.



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 3.4: FullFilteringJoin Dataflow.

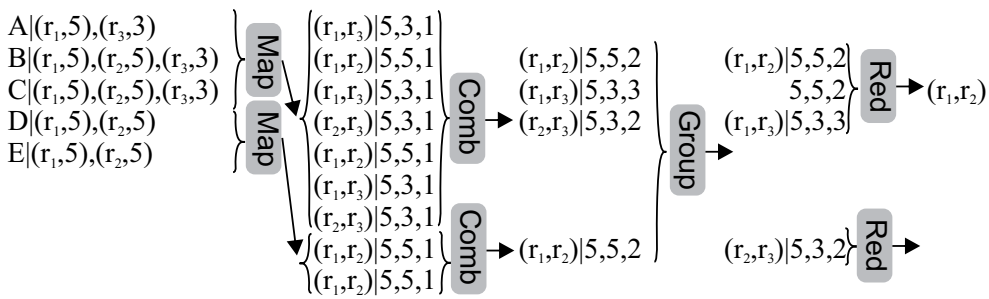
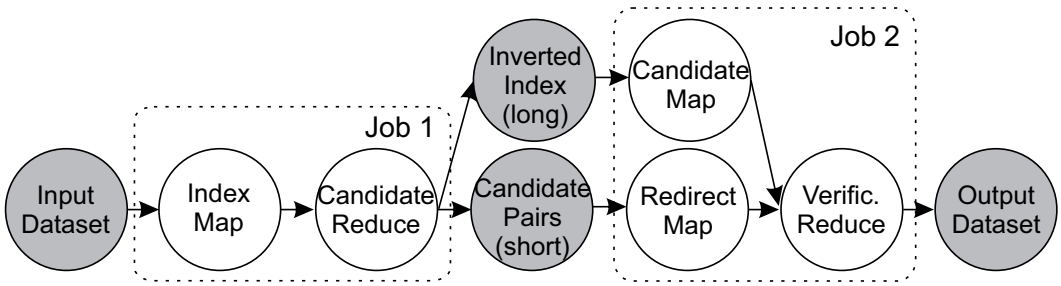


Figure 3.5: FullFilteringJoin, Job 2, Example.

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 three 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: each input record $\langle tok, L\rangle \in R_1$ generates $|M_2| = \sum_{\langle tok,L\rangle \in R_1} \binom{|L|}{2}$ records of four integers.

V-SMART (VS). VS extends FullFilteringJoin by splitting long inverted index lists and replicating them to multiple nodes [ELO08]. Job 1 (cf. Figure 3.6) computes an inverted index over all tokens. In contrast to FF, VS computes all candidates (2-combinations) for *short* inverted lists already in this first step and materializes them to HDFS. On the other hand, VS partitions and replicates *long* inverted lists and processes them in Job 2. The mappers in Job 2 either generate candidates (long lists) or pass them on to the reducers (short lists). A combiner pre-aggregates the candidates. Finally, a reducer verifies them. Due to the similarity to FF 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 3.4

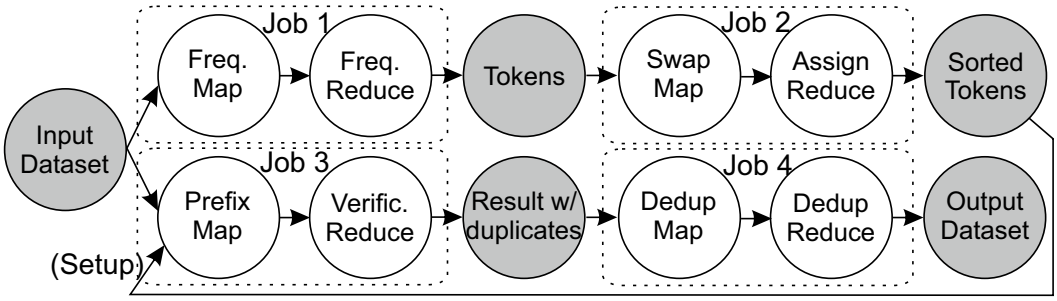
Figure 3.6: V-SMART Dataflow.

Discussion. Similar to FF, the quadratic number of candidates produced from long inverted lists dominate the intermediate data exchange. Although multiple mappers generate the candidates for long lists, the overall burden on the reducers in Job 2 is the same as for FF.

VernicaJoin (VJ). VJ is based on the *prefix filter* (cf. Section 2.2.2). Figure 3.7 gives an overview on 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 de-duplication step is required (Job 4). Figure 3.8 illustrates Job 3 for our running example.

Similar to FF, 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+ algorithm to reduce the candidate set (length, positional,

and suffix filter) [XWL+11]. Third, the inverted lists store – in addition to the record ID – all tokens of the original record, which is necessary for verification.



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: $\langle (rid_1, rid_2), null \rangle \rightarrow \langle rid_1, rid_2 \rangle$ // de-duplicated pairs

Figure 3.7: VernicaJoin Dataflow. Prefix tokens are underlined.

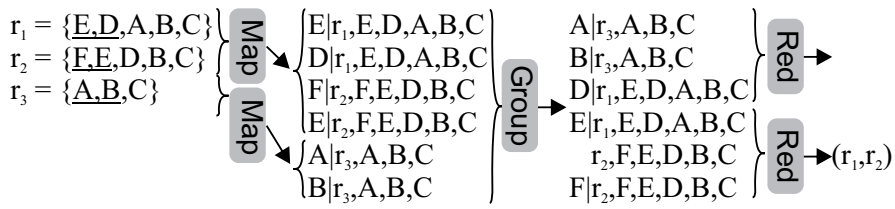
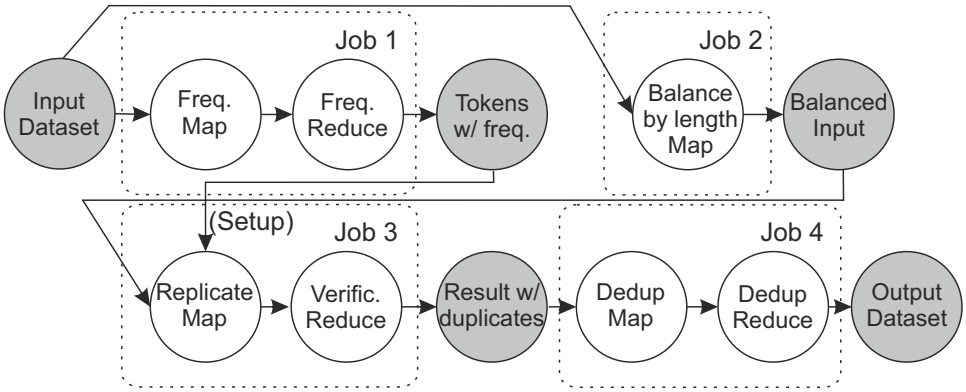


Figure 3.8: VernicaJoin, Job 3, Example. Tokens are ordered by GTF, prefix tokens are underlined.

Discussion. Job 3 dominates the amount of data exchanged. 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 $|r| - \lceil |r| \cdot \theta \rceil + 1$ for Jaccard, $|M_3| = (1 - \theta) \cdot |R| \cdot \overline{|r|} + |R|$. 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, VJ generates and verifies $\binom{|L|}{2}$ pairs in the worst case. This upper bound is pessimistic since the filters may reduce the number of candidate pairs.

MGJoin (MG). MG extends VJ twofold. First, in addition to GTF-ordered prefixes, MG also applies other prefix orders: it indexes GTF-ordered prefixes to generate candidates (similar to VJ). In addition, MG uses two prefix orders different from GTF to filter the resulting candidates before verification. Second, a load balancing job groups the input records into partitions with a similar length distribution before the inverted index on the GTF prefixes is computed.



M_1/R_1: see Figure 3.7 // 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 3.7 // de-duplication

Figure 3.9: MGJoin Dataflow.

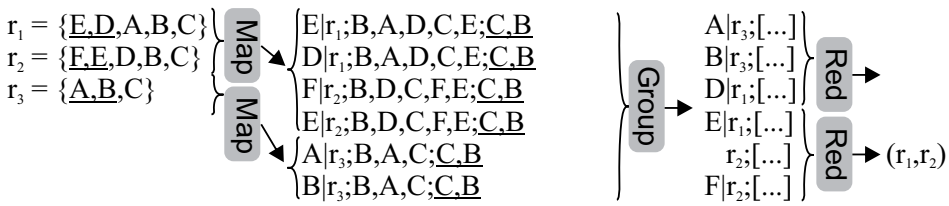


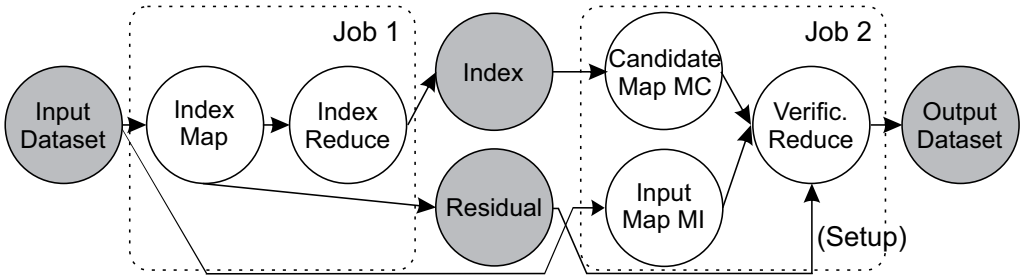
Figure 3.10: MGJoin, Job 3, Example. Prefix tokens are underlined. Indexed prefix ordered by GTF, non-indexed prefix by reverse GTF, random order is B,A,D,C,F,E.

Figure 3.9 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; 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. MG exchanges an amount of data similar to VJ, except that the entries in the inverted lists are larger since they contain an additional prefix.

SSJ-2R (S2). 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 3.11). S2 indexes prefixes with inverted list entries containing the record IDs, the last tokens in the prefix, and the record lengths. 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 and a group step before R_2 joins the candidate pairs on rid_1. 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). A setup function loads the residuals to each reducer.



$$M_1: \quad \langle rid, tok^* \rangle \rightarrow \begin{cases} \langle rid, \underline{tok}^* \rangle & // \text{ residuals} \\ \langle \underline{tok}, (rid, tok, l) \rangle & // \text{ inv. list entries} \end{cases}$$

$$R_1: \quad \langle \underline{tok}, (rid, tok, l) \rangle \rightarrow \langle \underline{tok}, (rid, tok, l)^* \rangle \quad // \text{ prefix index}$$

$$MC_2: \quad \langle \underline{tok}, (rid, tok, l)^* \rangle \rightarrow \langle (rid_1, rid_2), (rid_2, tok, l_2) \rangle$$
$$// \text{ candidates generated from index}$$

$$MI_2: \quad \langle rid, tok^* \rangle \rightarrow \langle (rid, null), tok^* \rangle \quad // \text{ load input for join}$$

$$R_2: \quad \langle (rid_1, rid_2 | null), tok^* | (rid_2, tok, l_2) \rangle \rightarrow \langle rid_1, rid_2 \rangle$$

Figure 3.11: SSJ-2R Dataflow.

Discussion. R_2 generates an index with the same cardinality as the index in VJ, but each list entry consists of only four 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 $|L|$ (similar to VJ). The residuals consist of $|R|$ records of average length $\theta \cdot \overline{|r|}$; for large similarity thresholds θ the residuals may be almost as large as the input dataset. Each task of R_2 must load the residuals, which is infeasible for large datasets.

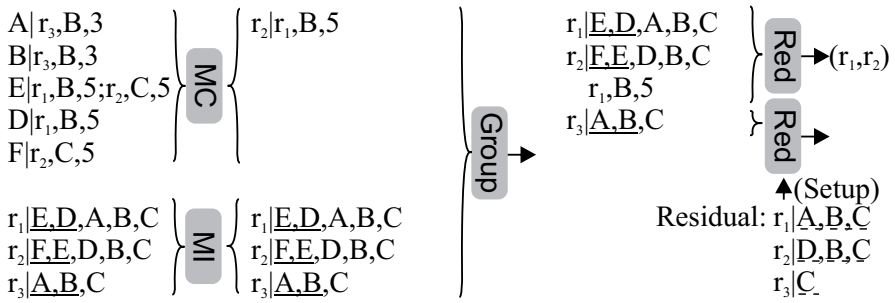
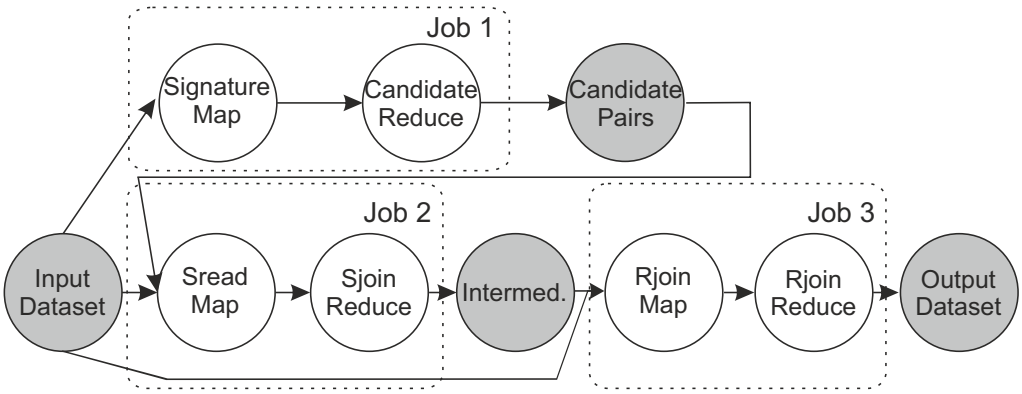


Figure 3.12: SSJ-2R, Job 2, Example.

MassJoin (MJ). MJ uses signatures based on the pigeon-hole principle and extends the non-distributed Pass-Join [LDWF11]. 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. Mapper M_1 (cf. Figures 3.13) 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. It leverages the pruning information to decrease the candidate set. The output format is a record ID with a list of candidates (in Figure 3.14 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.



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

Figure 3.13: MassJoin Dataflow.

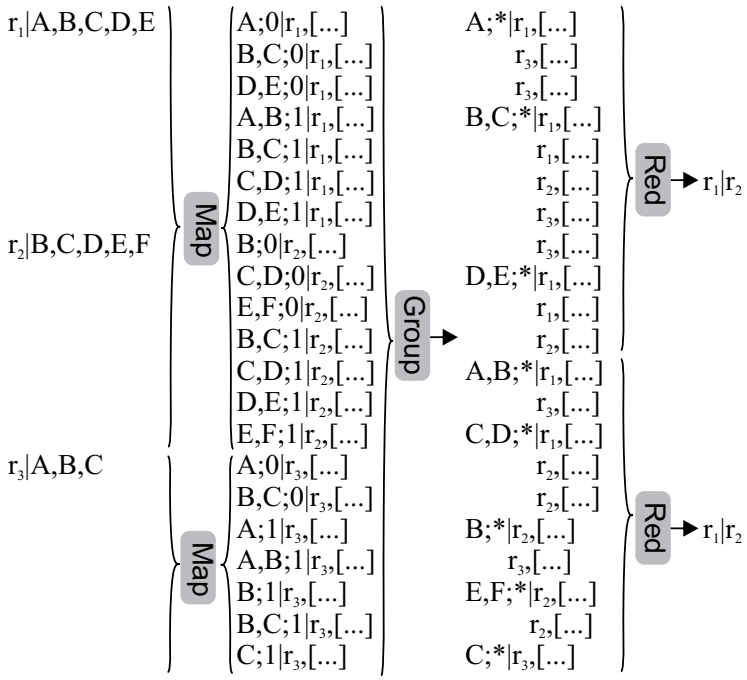
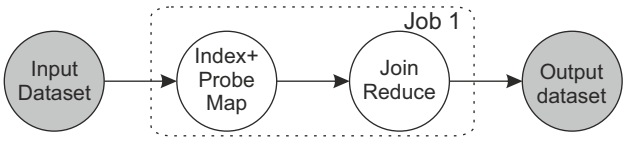


Figure 3.14: MassJoin, Job 1, Example.

Discussion. The output of M_1 dominates the amount of data exchanged. Deng at al. show that M_1 generates $|M_1| = \sum_{r \in R} \frac{(1+\theta^3)(1-\theta)^3}{\theta^3} \cdot |r| \cdot C + \frac{1-\theta}{\theta} \cdot |r|$ (C is a constant) signature records of $\frac{|r|}{\frac{1-\theta}{\theta} \cdot |r| + 1} + 35$ integers each [DLH+14]. The size of M_1's output grows with the record length and decreasing similarity thresholds.

MRGroupJoin (GJ). GJ groups records by length and partitions the records in each group into subrecords containing a disjunctive subset of the tokens. To generate candidates, GJ probes a record r 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 3.15, 3.16). M_1 partitions a record r into sub-partitions par for the index length $len = |r|$ and the probe lengths $len \in [\theta * |r|, |r|]$. The key is the pair (par, len), the value is the record r and an index/probe flag. R_1 computes and verifies candidates. To compute the candidates, the reducer creates the cross product on 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 3.15: MRGroupJoin Dataflow.

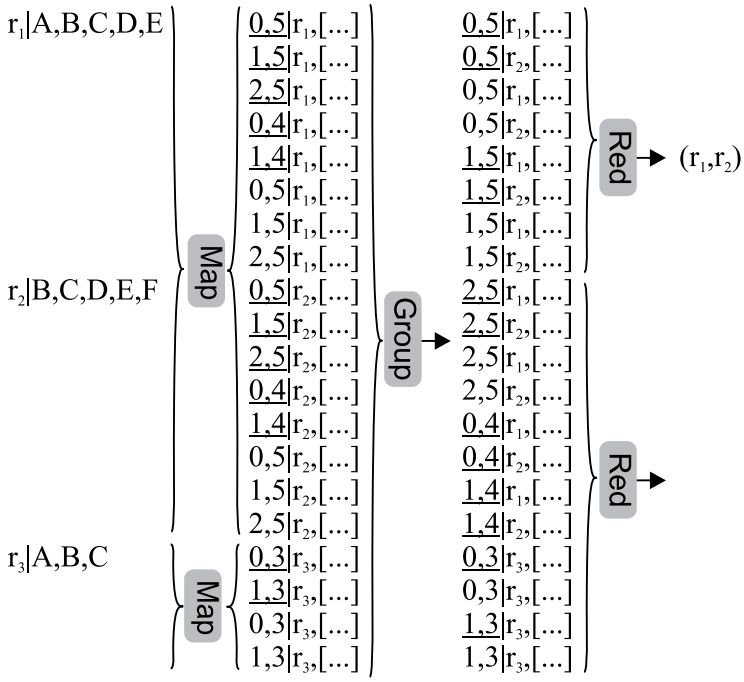
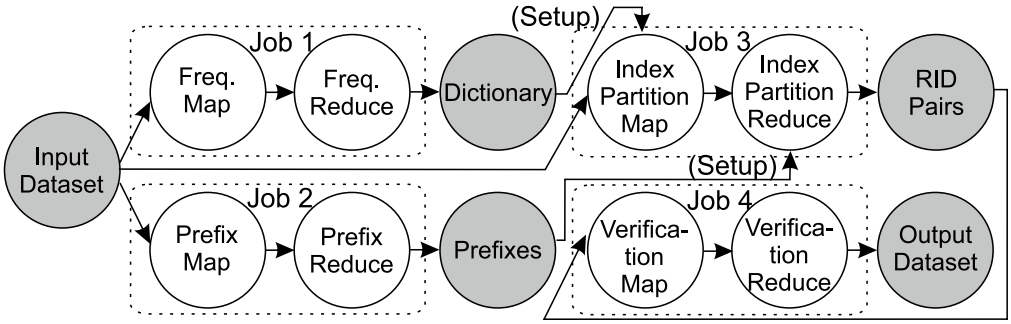


Figure 3.16: MRGroupJoin, Example. Index keys are underlined. Non-underlined keys are probe keys.

Discussion. The mapper produces $|M_1| = \sum_{r \in R} (\frac{1-\theta}{\theta} \cdot |r| + \sum_{\theta \cdot |r| \leq s \leq |r|} (\frac{1-\theta}{\theta} \cdot s))$ records. The record size is $|r| + 4$ integers.

FS-Join (FS). 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. FS groups all segments with the same key into *fragments*. Segments from different fragments have zero overlap. Thus, FS joins each fragment 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, FS sums up the partial overlaps of all its segments. An optional length-based *horizontal partitioning* allows distributing a fragment to different nodes at the cost of replicating data. Job 1 (cf. Figures 3.17, 3.18) 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 verifies the candidate pairs.



M_1/R_1: see Figure 3.7 // 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 3.17: FS-Join Dataflow. Prefixes and segments underlined.

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 five in the worst case. This upper bound is pessimistic since the data is sparse and filters reduce the output size. Each task of R_3 must load the prefixes of all records.

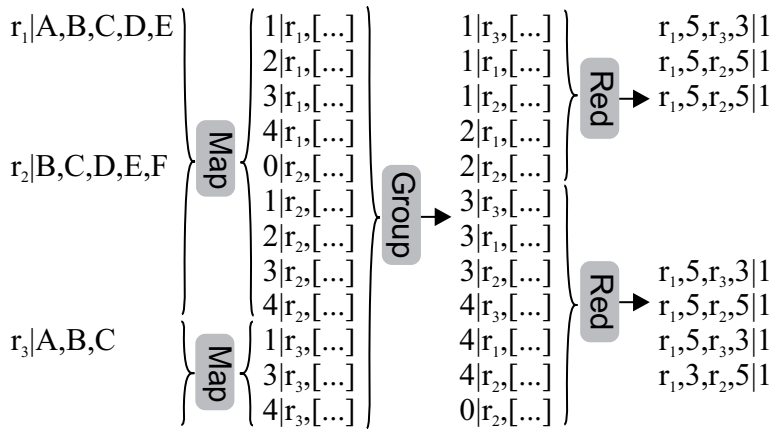
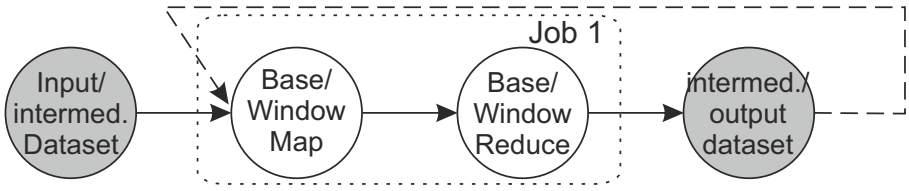


Figure 3.18: FS-Join, Job 3, Example. Pivots are B,D,E, and F.

3.2.2 Metric partitioning based algorithms

MRSimJoin (MR). MR parallelizes the non-distributed QuickJoin algorithm [JS08], which uses pivots to partition the metric space with hyperplanes. MR joins resulting partitions independently in main memory. If a partition does not fit into main memory, MR partitions it further, i.e., by a hash function. The approach replicates records that fall into border areas into dedicated window partitions to be joined separately. Figure 3.19 illustrates MR. Before Job 1 starts, MR draws random pivot records. The mapper M_1 uses the pivots to assign each input record to its base partition. If a record is too close to another partition, the mapper replicates it 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 calls itself recursively on the intermediate partitions until no partition is left.



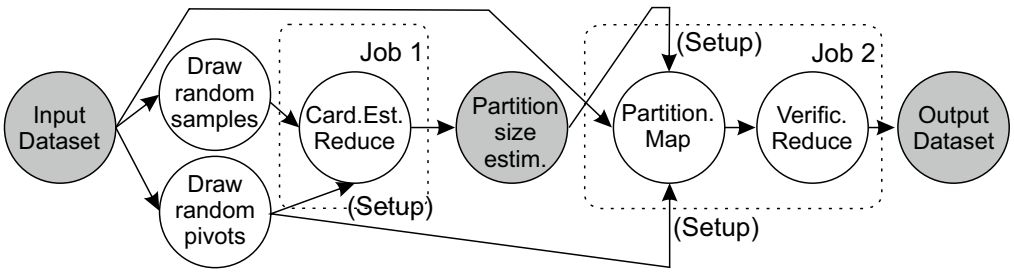
M_1: $\langle rid, tok^* \rangle \rightarrow \langle partition, (rid, tok^*) \rangle$
R_1: $\langle partition, (rid, tok^*) \rangle \rightarrow \langle rid_1, rid_2 \rangle$

Figure 3.19: MRSimJoin Dataflow.

Discussion. MR assigns each record to one of the p base partitions. In addition, it might replicate the record 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). 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 [OR11]. For Jaccard similarity, the authors discuss a length filter to reduce the candidate size [SHC14]. Similar to MR, CJ uses random pivots to split the data into disjoint partitions and to replicate border objects to window partitions.

Figure 3.20 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, CJ hashes and replicates records 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: $\langle rid, tok^* \rangle \rightarrow \langle partition, size \rangle$ // 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 3.20: 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, the approach splits and replicates large partitions; the size of all sub-partitions is quadratic in the partition size, which may substantially increase the intermediate result size.

3.3 Experiments

For our experimental analysis, we implemented[1] all algorithms from Section 3.2 (FF, GJ, MG, MJ, VS) or adapted existing code if available (CJ, FS, MR, S2, VJ). We evaluated the algorithms using 12 datasets (cf. Section 2.3). Our analysis focuses on runtime. However, we also discuss data grouping, data replication, and cluster utilization.

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

Table 3.1: 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

3.3.1 Setup

Hadoop. We deployed all methods on Hadoop 2.7 (using YARN, cf. Section 3.1). 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 3.1. 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 four reducers per node, which underutilizes the available memory slightly. This setup 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 experiment to test the effect of enabling compression.

Datasets. We use 10 real-world and 2 synthetic datasets from the non-distributed experimental survey in [MAB16]. We describe them in Section 2.3.

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 2.1. 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.

3.3.2 Performance and Robustness

Performance. Table 3.2 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 30 minutes 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) [MAB16]. 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 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 3.2, 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 3.3.5.

Figure 3.1 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 [BDFML10], VJ is faster than VS [SHC14], and FS is faster than VS [RLS+17]. However, in our experiments, VJ is faster than CJ (equal runtime in [SHC14]), VJ is faster than MG (contrary to [RLW+13]), VJ is faster than MJ (contrary to [DLH+14]), VJ is faster than S2 (contrary to [BDFML10]), VJ is faster than VS (contrary to [MF12]), and VJ is faster than FS (contrary to [RLS+17]). In Section 3.3.6, 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 [MAB16]; more specifically, we measure the average, median, and maximum deviation of an algorithm's runtime from the best reported runtime. Table 3.3 reports the deviation factors for FS, GJ, MG, MJ, and VJ over all datasets and all thresholds $\theta \in \{0.6, 0.7, 0.8, 0.9, 0.95\}$. 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 3.4 summarizes for which combinations of algorithm, threshold, and dataset, a timeout occurred.

3.3.3 Scalability

State-of-the-art non-distributed algorithms can process the SSJ on our datasets in memory. Mann et al. provide a competitive experimental analysis under this setup [MAB16]. In fact, non-distributed algorithms outperform Hadoop-based solutions in the majority of the datasets (cf. Table 4 in [MAB16]). 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 3.21 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

Table 3.2: Fastest algorithms; runtime in seconds, timeout 30 minutes. Fastest algorithm underlined.

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

Table 3.3: 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 3.4: 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			

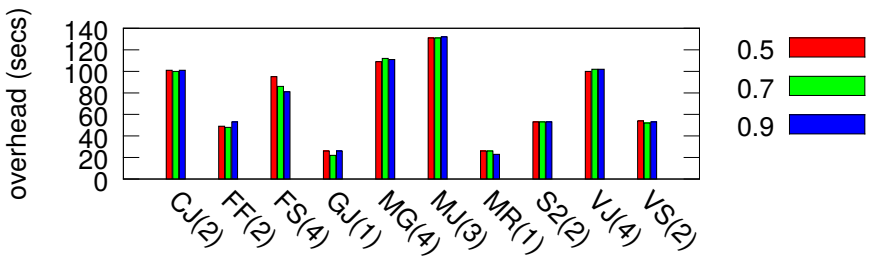


Figure 3.21: Data-independent overhead for $\theta \in \{0.5, 0.7, 0.9\}$; number of MapReduce steps given in brackets. For small datasets and high similarity thresholds the overhead takes a large share of the overall runtime (cf. Table 3.2).

Table 3.5: Timeouts (120mins) on scalability tests; Jaccard similarity threshold $\theta = 0.95$.

Dataset	FS			GJ			MG			MJ			VJ		
	1×	5×	10×	1×	5×	10×	1×	5×	10×	1×	5×	10×	1×	5×	10×
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															

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 2.1. For our scalability tests, we artificially increased the size of our datasets by factors $s = 5$ and 10 (cf. Section 2.3). Furthermore, we use a high Jaccard threshold $\theta = 0.95$.

Figure 3.22 reports the runtimes of the algorithms for each dataset. Table 3.5 shows for which combinations of algorithm and dataset timeouts (120 minutes 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 3.3.5, we discuss reasons for these results.

3.3.4 Varying the Cluster Configuration

In the following, we report results when varying parameters, such as map size compression and memory settings.

Compression. We test the effect of enabling map output compression. On the small datasets (1×, Table 2.1), 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 (5×, 10×), 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, Hadoop spills it 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 depends on the configured memory per reducer and on the parameter *max* for the maximum number of reducers. We decrease

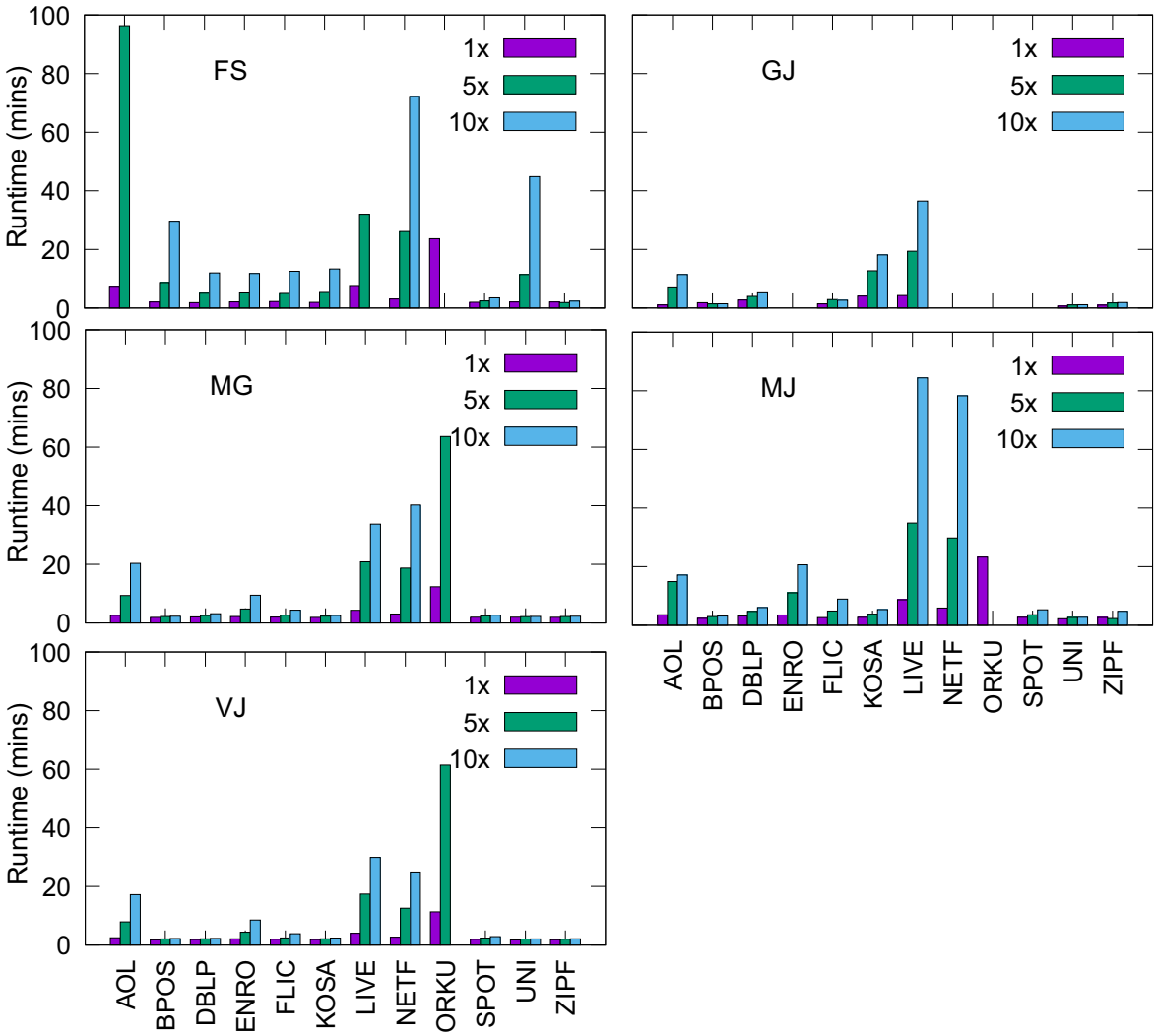


Figure 3.22: Runtimes on scalability tests; Jaccard similarity threshold $\theta = 0.95$; timeouts excluded.

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 5× 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. On the other hand, it negatively affects the runtime of FS. On the 10× 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.

3.3.5 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 3.3). 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 3.2 use replication to achieve a high level of parallelization. Furthermore, 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 obtain a balanced share of the overall computation. For most algorithms, the key generation and the replication depend 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 3.2. 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 3.23 reports our measurements on cluster utilization. We show only a fraction of the conducted tests and focus on FS, GJ, MG, MJ, and VJ; recall that the other algorithms timed out on more than 60% of our tests.

VJ. The replication and verification step (Job 3, Figure 3.7) dominates the runtime of VJ. The utilization graphs for VJ follow the pattern of Figure 3.23(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 NETF$\times$1 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 this data, memory overload occurs in the execution phase of reduce after shuffling, which is captured by a high utilization, which decreases slowly in Figure 3.23(j). The combination of similarity threshold and record lengths decides on the size of intermediate data. The execution of VJ is only efficient if the intermediate data does not exceed roughly half of the TMM. Adding more nodes could solve this issue, because it increases the TMM. However, each intermediate data group size depends on 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. The resulting memory usage imposes a hard limit, which users cannot push by adding more nodes.

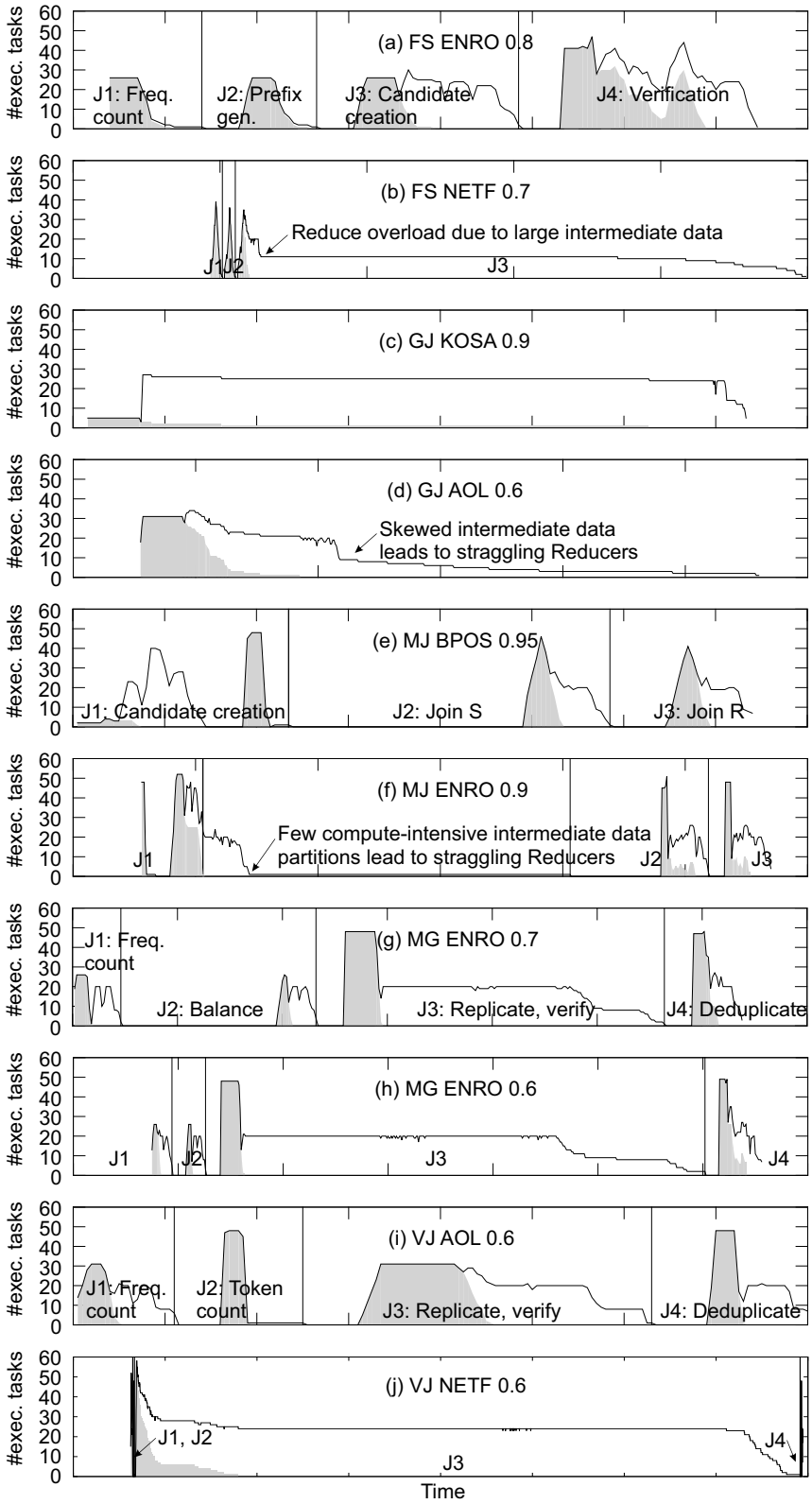


Figure 3.23: Cluster utilization for each algorithm. Vertical lines divide jobs as described in Section 3.2. Runtimes relative to each job; absolute runtimes not shown for brevity. The number of tasks is the sum of map and reduce tasks. Grey areas mark the map share of the tasks.

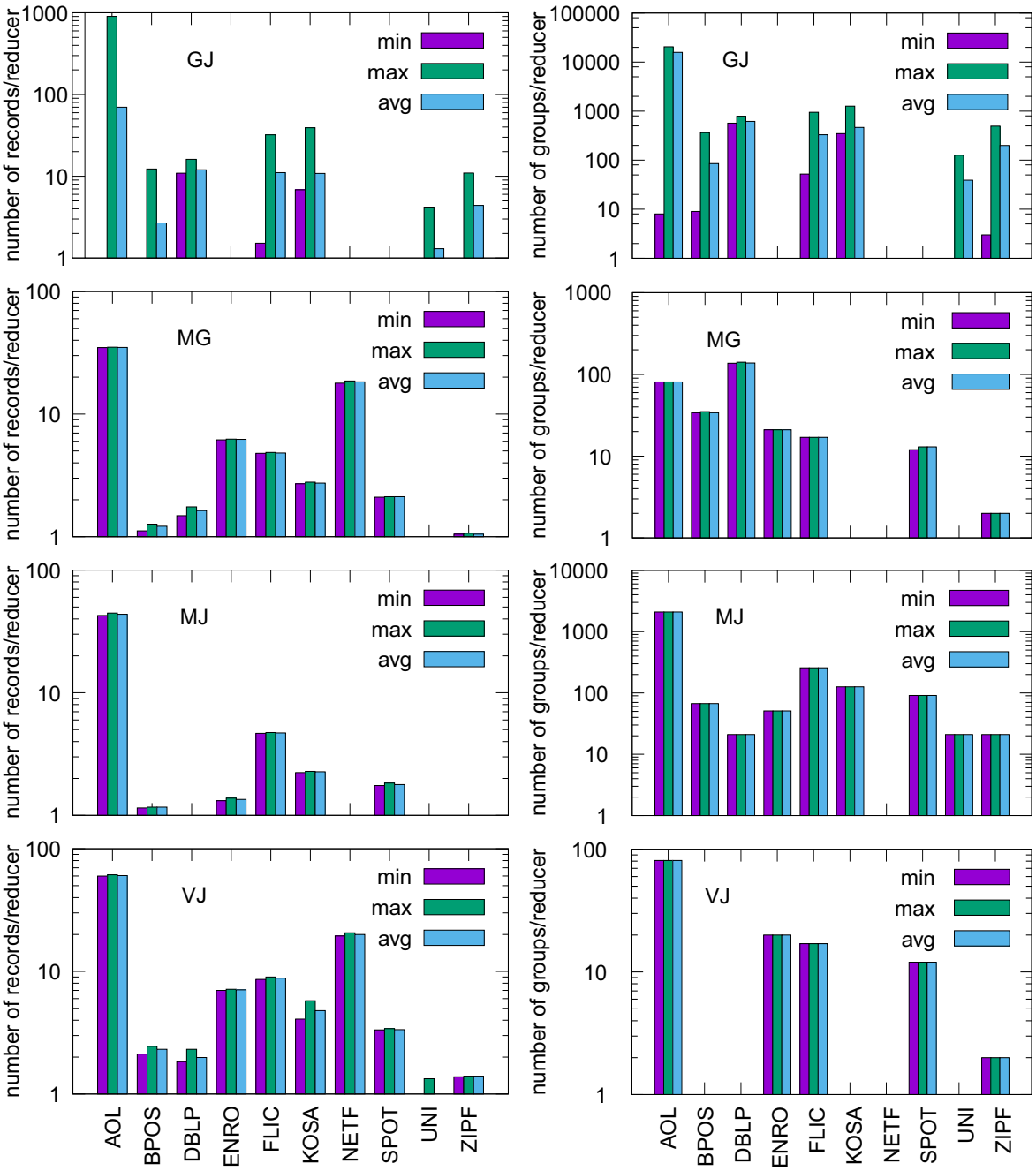


Figure 3.24: Data grouping and replication (number of records or groups ×1000) at the reduce step computing the join; sim. threshold $\theta = 0.95$; all datasets 10× synthetically enlarged.

To summarize, 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. We observe a high and stable cluster utilization for large values of the similarity threshold on most datasets (Figure 3.23(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 3.23(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 3.24 shows the data distribution of GJ for varying datasets (omitting datasets where timeouts occurred). The minimum and maximum number of records per reducer considerably varies, which explains the straggler effect. Recall that GJ splits the input records into subgroups and matches them 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 [DLWF15]. 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 $\frac{1-\theta}{\theta} \cdot |r| + 1$ times (for indexing) and roughly $|r| - \theta \cdot |r|$ times (for probing). Each intermediate record contains approximately four integers plus the original data. Now, consider a Jaccard similarity threshold of 0.6 and ORKU×10, which consists of $2.7 \cdot 10^7$ records with an average length of 120. Every record is 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.

To summarize, 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 evaluates only the non-distributed scenario [DLWF15].

S2 and FS. Recall from Section 3.2 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 ORKU×10, 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 3.23(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). These results indicate that increasing record lengths has 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 obtains exactly one fragment. Furthermore, 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.

To summarize, 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 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 3.9), 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 3.23(g)). We observe that the number of tasks remains constant while only the range of this constant utilization in the join phase varies (Figure 3.23(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 3.24 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.

To summarize, 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 3.23(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 3.23(f)). A low value of maxFreq as in SPOT can compensate for a high maximum record length. Our experiments on ENRO$\times$10 showed that MJ is able to evaluate the join for a similarity threshold of $\theta = 0.95$, but times out when $\theta < 0.9$. 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 3.24 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 3.23 (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 assures only a good intermediate data distribution, which does not necessarily lead to an even distribution of the compute load.

To summarize, 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 AOL×1. From Table 2.1, maxFreq $= 4.2 \cdot 10^5$, hence, for the most frequent token, a particular candidate mapper (Job 2 in Figure 3.4 and 3.6) emits $\binom{4.2 \cdot 10^5}{2} = 88 \cdot 10^9$ records. Each record contains four integers for FF or five for VS, all of four bytes, so the mapper produces overall $88 \cdot 10^9 \cdot 4 \cdot 4 \approx 1{,}311$GBs or $88 \cdot 10^9 \cdot 5 \cdot 4 \approx 1{,}639$GBs of data, respectively. The volume of 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 does not address this problem, because the volume of the intermediate data grows quadratic with maxFreq. As a result, both FF and VS can process only datasets of low maxFreq.

To summarize, 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 NETF×10 and assume that 1000 pivot elements are selected. Each intermediate record contains $|r| + 7$ integers, so based on Table 2.1, the intermediate data occupies $1000 \cdot 4.8 \cdot 10^6 \cdot (210 + 7) \cdot 4 \approx 3{,}880$GBs, 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 Das Sarma et al. and Deng et al. suggest that the algorithms perform better on data with a low number of dimensions [SHC14, DLWF15].

To summarize, the metric-based approaches did not perform well in our tests. Due to their high level of data replication, CJ and MR often time out even for small inputs.

3.3.6 Reproducing Previous Results

We repeated core experiments for VJ, S2, MJ, and FS. It was not possible to repeat tests for the remaining algorithms. The authors auf CJ do not specify the experimental setup (parameters of the method, hardware setting) and use a non-public dataset [SHC14]. Similarly, the necessary experimental parameters for FF are missing [ELO08]. For VS, a larger cluster than ours and a publically unavailable dataset were used [MF12]. The authors of MG also 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 [RLW+13]. GJ has not been tested on a distributed setup so far [DLWF15]. The authors of MR use a different similarity function (Euclidean) [SR12].

Unless stated otherwise, our Hadoop cluster is configured according to Table 3.1. Our tests can reproduce the results of **VJ** and **S2**. We report only on algorithms with deviating results.

Algorithm MJ. Contrary to the original paper, our experiments showed that VJ is faster than MJ. The authors compare MJ to VJ on ENRO with a 10 node cluster, varying the Jaccard similarity threshold [DLH+14]. As MJ computes an end-to-end non-self R×S join, the dataset is split into two subsets of equal size. Figure 3.25 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 join only record IDs as our focus is not on an end-to-end computation. Figure 3.26 reports our results. Our implementation of both MJ and VJ recorded lower absolute runtimes compared to [DLH+14]. Lower runtimes are 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 and 0.85. The competing VJ shows runtimes of nearly one magnitude more in the original experiments, which are unclear us.

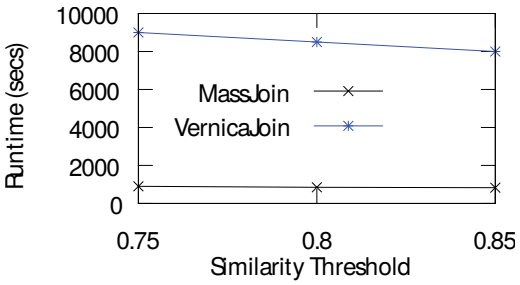


Figure 3.25: Original results.

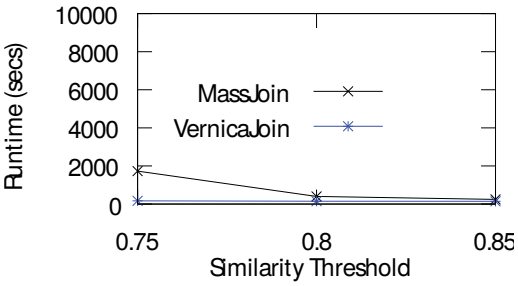


Figure 3.26: Our results.

Algorithm FS. Contrary to the original paper, our experiments show that VJ is faster than FS. The authors compared FS to VJ on ENRO varying the Jaccard similarity threshold [RLS+17]. The test ran on an 11 node cluster; each node had 15GBs of RAM while 3 reduce tasks were allowed per node. Figure 3.27 reports the results. To repeat this exper-

iment, 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. The input dataset contains 517k records, while the record length varies from 51 to 148k tokens [RLS+17]. 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 the authors in their publicly available source code. Yet, we were not able to reproduce exactly the same ENRO dataset used; this setup (named E2) contains records of 37-76k tokens. Figure 3.28 reports our results for setups E1 and E2. Regarding E1, the runtime of FS is similar to the original result (Figure 3.27), but VJ is at least one order of magnitude faster in our experiments. This result is expected as VJ's prefix filter benefits from the shorter records of E1. Regarding 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.

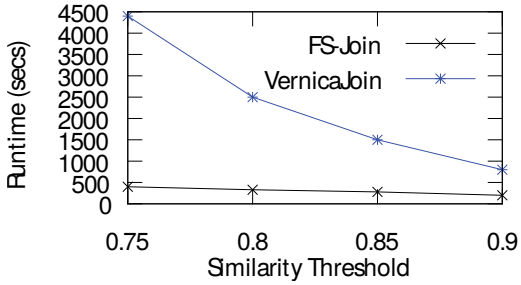


Figure 3.27: Original results.

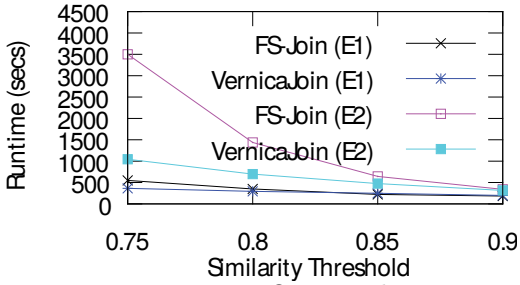


Figure 3.28: Our results.

3.4 Summary

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 result refutes experimental results of previous papers, which claim to outperform VJ. 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 FS algorithm.

The motivation to use distributed computing for the SSJ problem are large data volumes, which a single node cannot handle. None of the algorithms considered in this chapter 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. Specific characteristics

of the input data trigger this effect. Adding more nodes does not solve this problem. These shortcomings motivate us to utilize local resources more efficiently with a novel multicore-parallel SSJ we introduce in the following chapter. Subsequently, we propose a novel highly scalable distributed SSJ, which avoids intra-node replication and computes the SSJ on hundreds of GB of input data.

Chapter 4

Exploiting Multicore Parallelization

Modern hardware consists of multiple processors with many cores. Machines with dozens or even hundreds of CPUs are affordable and common. However, existing SSJ algorithms do not exploit this parallelization potential. In this chapter, we study how local parallelization can significantly speed up existing single-threaded approaches (as published in [FWZF20]). To the best of our knowledge there is no prior work on parallelizing the SSJ on multicores. We adapt existing sequential filter-and-verification SSJ approaches to run on one single multicore machine. Such an adaptation is not trivial. Modern hardware usually comprises multiple CPU sockets with non-uniform memory access (NUMA). Memory access speeds depend on thread placement on different CPUs. Furthermore, optimizations of single-threaded approaches, such as filters, do not necessarily have the same impact on the runtime in the multicore case.

In the following, we describe the NUMA architecture, our data-parallel adaptation of the sequential filter-and-verification SSJ approaches, technical details regarding the implementation, and our experimental evaluation.

4.1 Modern Multicore Systems

In this chapter, we target single-node shared memory systems with multiple processors consisting of a large number of cores. Figure 4.1 shows the architecture of such a system with four processors. All processors are connected through an interconnect. Each processor accesses its local memory through an integrated memory controller (cf. the four "memory" blocks in Figure 4.1). Local memory access is fast, while accessing memory attached to other processors on the same machine, comes with additional latency. The different access speeds are referred to as "NUMA effect".

Modern processors use multiple levels of caches for performance improvements. Processors implement cache coherence protocols to ensure that threads on different cores access consistent data. Another important technique for improving performance is prefetching. Processors probabilistically read data from main memory, which is likely to be used by a

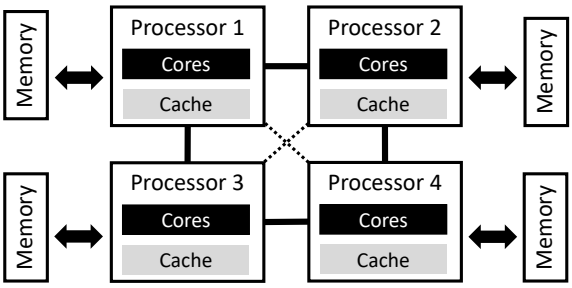


Figure 4.1: Architecture of a modern multi-socket machine. Each processor has local memory attached. All processors are interconnected.

running program subsequently. Prefetching can hide memory stalling, i.e., a core waiting for data to arrive from main memory. Prefetching is done automatically or explicitly as instructed by software.

Operating systems (together with the hardware) decide about thread placement, i.e., on which core a thread runs and migrate them to other cores to during execution. Regarding the runtime of one thread, moving it to a different core, especially if it is on a different processor, causes overhead. Even worse, if this thread shared cached data with other threads on the same processor, moving it to a different processor can significantly slow down execution, because this data has to be loaded there again. Modern processors usually provide hyperthreading [SMD+10]. This technique aims to better utilize a processor by allowing two processes to concurrently access different resources of one core, i.e., the arithmetic logic unit (ALU) or the floating point unit.

The properties of modern processors affect the runtime of multi-threaded programs in a non-trivial fashion [KLH+99]. We subsequently show how the runtimes of our SSJ implementations behave under varying conditions.

4.2 Related Work

In Chapter 2 we described the single-core SSJ algorithms AllPairs and PPJoin, which form the basis for our multicore SSJ. Besides the CPU-based algorithms, some recent work focuses on speeding up SSJ using different hardware platforms, notably GPUs. Quirino et al. proposed a standalone GPU algorithm that runs both candidate generation and verification within GPU using a block-based probing approach [QRRM17]. Bellas et al. proposed a different framework that uses GPUs for candidate verification, while keeping candidate generation a CPU task [BG19]: considering limited GPU memory, GPUs verify candidate pairs in chunks. The experimental result of Bellas et al. indicates that the CPU-GPU solution outperforms the standalone GPU algorithm of Quirino et al. [BG19, QRRM17]. It batches candidate pairs for verification when the number of candidates are large, which is the case for low similarity thresholds. Bellas et al. argue that the bottleneck in SSJ is often the candidate generation rather than candidate verification, thus the acceleration provided by GPUs is limited [BG19]. In comparison, our work

exclusively focuses on the parallelization potential of multicore hardware in combination with existing filtering approaches and implementation optimizations – it is orthogonal to recent work on GPU-based SSJ.

4.3 Parallelizing Filter-and-verification-based SSJ

We use the AllPairs algorithm as a basis, because it is fundamental for filter-and-verification algorithms as we discussed in Section 2.2. AllPairs uses the prefix and length filters. We furthermore consider the position filter from PPJoin as the single-threaded SSJ study of Mann et al. showed that this filter is efficient [MAB16]. We do not consider filters the study proved to be inefficient, i.e., the suffix filter.

Subsequently, we discuss the execution model (i.e., how to assign tasks to threads), discuss the design considerations in the context of multicores, and describe our algorithm. We show how different design decisions impact the runtimes in Section 4.4.

4.3.1 Execution Model

SSJ algorithms could be parallelized using *data parallelization* or *pipelining*. Figure 4.2 shows the basic idea of each design choice. In data parallelization, the input data is partitioned into disjoint batches consisting of a tunable number of records. Each thread then runs the AllPairs algorithm (or PPJoin when the position filter is activated) on a different batch. Multiple threads can proceed in parallel without conflicts. Practical implementations create a pool of threads upon system start. After a batch is processed, the thread continues with the next batch, avoiding the cost of creating and destroying threads at runtime.

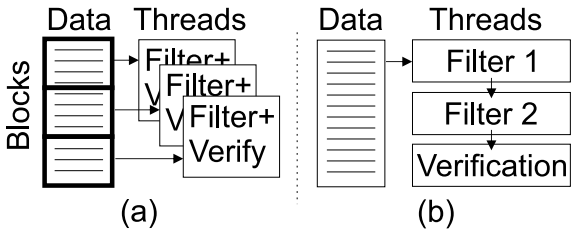


Figure 4.2: Data-parallel (a) vs. pipelined (b) execution models.

Apart from data parallelization we also considered *pipelined* execution (Figure 4.2b). We subdivided the SSJ task into sub-tasks, each of which is executed on a dedicated thread. The entire join algorithm finishes cooperatively by multiple threads, which communicate through message passing. Compared to data parallelization, this pipelined approach requires frequent inter-thread communication and synchronization using message queues. We experimentally verified that such overhead was too high to make our parallel SSJ approach efficient. Therefore, we focus on data parallelization in the rest of the chapter.

4.3.2 Design Considerations

In the following, we discuss design considerations of our parallel SSJ approach. Our discussion involves the algorithm and implementation. Algorithmically, we are interested in the efficiency of existing filters in the multi-threaded case. Implementation-wise, we explore if implementation optimizations achieve runtime benefits over straightforward implementations.

Filters. We are interested in how filter techniques used in single-core algorithms behave on multicores. As a base algorithm we use AllPairs, which contains the length and prefix filters. By default, our parallel SSJ algorithm ignores the entries in the inverted index, which cannot be similar due to length differences. This approach is comparable to the deletion filter of MPJoin, which deletes entries in the inverted index, which cannot pass the position filter for following probe records [RH11]. Since the approach to skip non-matching records when probing the inverted index has a significant positive impact in all our experimental cases, we decided to use this optimization by default. Furthermore, we explore the use of PPJoin's position filter (cf. 2.2.2). It shows differences compared to the single-core case as we show in the experiments.

Record Inlining. By inlining, we refer to a C/C++-specific implementation optimization. Records consist of a record ID (integer) and a variable number of integer tokens. A straightforward record implementation is to use a `struct`, which contains the id and a pointer to an array of integer tokens. To access the tokens, the pointer has to be dereferenced first, which often incurs expensive cache misses and CPU stalls as the processor waits for data to be fetched from memory to CPU caches. By inlining, we co-locate the tokens with the record ID without such extra layer of indirection. We expect inlining to be more efficient as it avoids pointer-chasing during record access. AllPairs reads the records including their tokens one-by-one in the filter phase, so we expect a positive effect on runtime. On the other hand, inlining introduces overhead when accessing records randomly due to the variable-length tokens. For random reads, we introduce a pointer array that maps token IDs to the location of the corresponding record in the record array. AllPairs accesses records randomly in the probe phase. As a result, in the probe phase, both variants (with and without inlining) do pointer chasing once per record.

Thread Affinity. We found that the operating system scheduler often migrates SSJ tasks amongst cores during the execution of our parallel SSJ. We assume that thread migrations can degrade runtime performance due to the NUMA effect in case a thread is migrated to a socket but the data it is accessing is on another socket. We implement two versions of the parallel SSJ. One allows thread migration and the other one pins the threads to cores at thread creation time.

Batching. In the data-parallel execution model, each thread runs the SSJ algorithm on batches (partitions). The batch size controls the number of records joined on one thread without synchronizing with other threads. Thus, we expect the batch size to influence the runtime.

4.3.3 Algorithm

Our algorithm consists of a non-parallel main function and a worker function, which computes the SSJ in parallel. The main function reads the input data R, creates the inverted index, and initializes the variable *nextRid* with 0. *nextRid* is synchronized such that multiple threads can concurrently access it. The synchronized property allows reading and updating the variable only as a single atomic operation. The atomicity assures that two threads do not recompute the same part of the join. Furthermore, the main function spawns the user-configured number of threads, starts the worker function on each of the threads, and waits until all threads are finished. Algorithm 2 provides pseudocode for the worker function.

Lines 1–3 implement the batching mechanism. Line 1 checks if the next record ID to be processed is within the limits of the input dataset R. If not, there is no record left to join and the function terminates. If the condition of Line 1 is true, we copy *nextRid* into *startRid* and increment *nextRid* with the batch size.

Line 4 iterates over all records within the current batch. Lines 5–12 represent the AllPairs algorithm we described in Section 2.2.3. We added the position filter in Line 9. The user can configure if it is active or not. In the previous section we discussed that we ignore inverted list entries, which cannot be similar due to length differences. For brevity, we did not include this index position optimization in Algorithm 2. The optimization works as follows. Each worker function initializes a local nulled array for each token contained in the input dataset (before Line 1). Before each index probe (Line 8), we check if the token is contained in the array and if so, we access the corresponding postings list entries only starting from the start value in the array. Within the probe iteration, we check if a probed record is larger than r and if so, we update the index starting positions.

Algorithm 2: Worker function.

Data: R, $invertedIndex$, θ, $nextRid$, $batchSize$
Result: $\{(r,s)|(r,s) \in R \times R, r \neq s, sim(r,s) \geq \theta\}$

```
1  while nextRid < maxRid(R) do
2      startRid ← nextRid
3      nextRid ← nextRid + batchSize
4      for rid = startRid; rid < startRid + batchSize; rid++ do
5          r ← R[rid]
6          candidates ← {}
7          foreach token ∈ GetPrefix(r, θ) do
8              foreach s ∈ GetList(invertedIndex, token) do
9                  if PositionFilter(r, s) then /* if pos. filter active */
10                     candidates ← candidates ∪ {s}
11         foreach s ∈ candidates do
12             Verify(r, s, θ)
```

4.4 Experiments

In this section, we empirically quantify the impact of the design considerations discussed in the previous section.

4.4.1 Setup

Our implementation[1] uses C++11 and allows tuning of various parameters as described in the previous section. We compile the code with gcc using optimization level 3 (-O3). Lower optimization levels lead to significantly higher runtimes. For worker threads, we use C++11 `std::thread`. We run our experiments on Linux, so std::thread acts as a lightweight wrapper for POSIX threads. POSIX threads allow to have the core affinity set so that the OS should not interfere with the physical thread placement. We run experiments on a server with two Intel(R) Xeon(R) CPU E5-2620 processors clocked at 2.0GHz and 32 GB of DRAM. Each CPU has six cores (12 hyperthreads) and 32KB/256KB/15MB of L1/L2/L3 caches. We use the machine exclusively for the experiments. Since system processes and hardware events (network etc.) can influence the measurements, we repeat each experiment three times and report the average to even out such effects.

Inverted index. We implement the inverted index using a hash table mapping token IDs to an array of pointers. Each index entry contains a record pointer and token position (for the position filter). A C++11 unordered map implements the hash table and a C++ vector implements the array. The index is read-only and accessed randomly. We optimize the index access such that the SSJ only considers entries of records with a potentially matching length for the probe. This optimization requires the input to be ordered by ascending record lengths and also the index holding the record IDs in the order of their lengths. In each iteration of the candidate generation we keep the starting position for each token ID in a variable. When we probe a token and find a record ID in the index not fulfilling the length filter, we can disregard this record ID for all subsequent (equal length or larger) records.

Candidate generation. Candidate generation involves the counting of overlaps of records. We save overlap information in a hash table (C++ unordered map) mapping a record pointer to an integer indicating the overlap count. Every iteration updates the overlaps. Note that we use only integer record IDs and do not pass the entire records including their tokens. During verification, we access the record list stored in main memory. Sharing data is an advantage over MapReduce-based approaches (cf. Chapter 3) where the complete records have to be copied for the verification, even if the two verifying processes are on the same physical node and access the same records.

Datasets. We use ten real-world and two synthetic datasets as described in Section 2.3. We also use artificially increased versions of the datasets up to the factor 10.

Number of threads. We set the number of threads to 1, 2, 4, 8, 12, 16, 24, 32, and 64. Besides the powers of two we chose 12 and 24, because our hardware has 12 physical CPUs, which support 2 physical threads concurrently. For a thread number of

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

1 we execute a non-parallelized implementation of AllPairs, which does not spawn any additional threads besides the main thread.

Metrics. We vary the Jaccard similarity threshold among 0.6, 0.75, and 0.9. We assume these are sensible values for many SSJ applications. Thresholds below 0.6 reveal a high number of result pairs on our datasets, which we assume is not intended. On the contrary, thresholds above 0.9 lead to few results. We measure the runtime within our program from including the index build until the join computation is completed. We do not store the join result itself, only its size. We run our code with all combinations of variables described in the previous section and report on the results in the following. In each run, we also profile the execution using `perf` to gather metrics, such as cache misses. Using `perf` adds a small and constant amount of overhead. However, it does not affect relative runtimes, which are important in our discussion.

4.4.2 Speedups and Scalability

We first investigate how the number of threads affects runtime.

Speedup over Single-Core Execution. Using multithreading is beneficial for the SSJ runtime under all combinations of our input datasets and thresholds. We observed speedups of roughly 2-10 times on our hardware. For detailed results, please refer to Table A.1 in the appendix. The absolute runtimes of the multi-threaded version vary between roughly 0.2 and 262 seconds for all datasets and thresholds. For each combination of input and threshold, we evaluated the parameter combination of number of threads, core affinity, position filter, inlining, and batch size leading to the lowest runtime. Overall, the best runtimes were achieved by using 24 or 32 threads. 70% of the best runtimes were achieved for a batch size of 125 or 250. The position filter is effective for most (70%) of the cases. However, we did not find an optimal parameter configuration for all the combinations of dataset and threshold. In the following, we discuss the influence and interdependencies of and between the variables and draw conclusions, under which conditions which variable values are favorable.

Scalability. Figure 4.3 shows the speedup of our experiments relative to the number of threads. Without loss of generality, we consider only results with the following parameters: no inlining, batch size 500, no CPU affinity, and no position filter. Other parameter combinations show a similar behavior, so we omit them here. For the majority of results, the speedup increases linearly up to 12 threads (number of physical cores). Starting with 16 threads, we record decreasing speedups. The optimal runtime is achieved at 24 threads for all datasets except ORKU and LIVE (0.75 and 0.9 similarity thresholds), and ENRO (0.9 similarity threshold). Since the machine has 12 physical cores, this result shows that our data-parallel SSJ algorithm benefits from hyperthreading, which can hide memory access latency caused by cache misses. The result is not trivial, as hyperthreading is benefitial only in a limited range of cases [Dre07].

The results show that the scalability varies depending on the input dataset, the threshold, and the number of threads. Note that SPOT is an exception showing a hard limit at a speedup of 2, independent of the threshold and the number of threads. We found that the scability behavior is related to index lengths (the number of record IDs for each token).

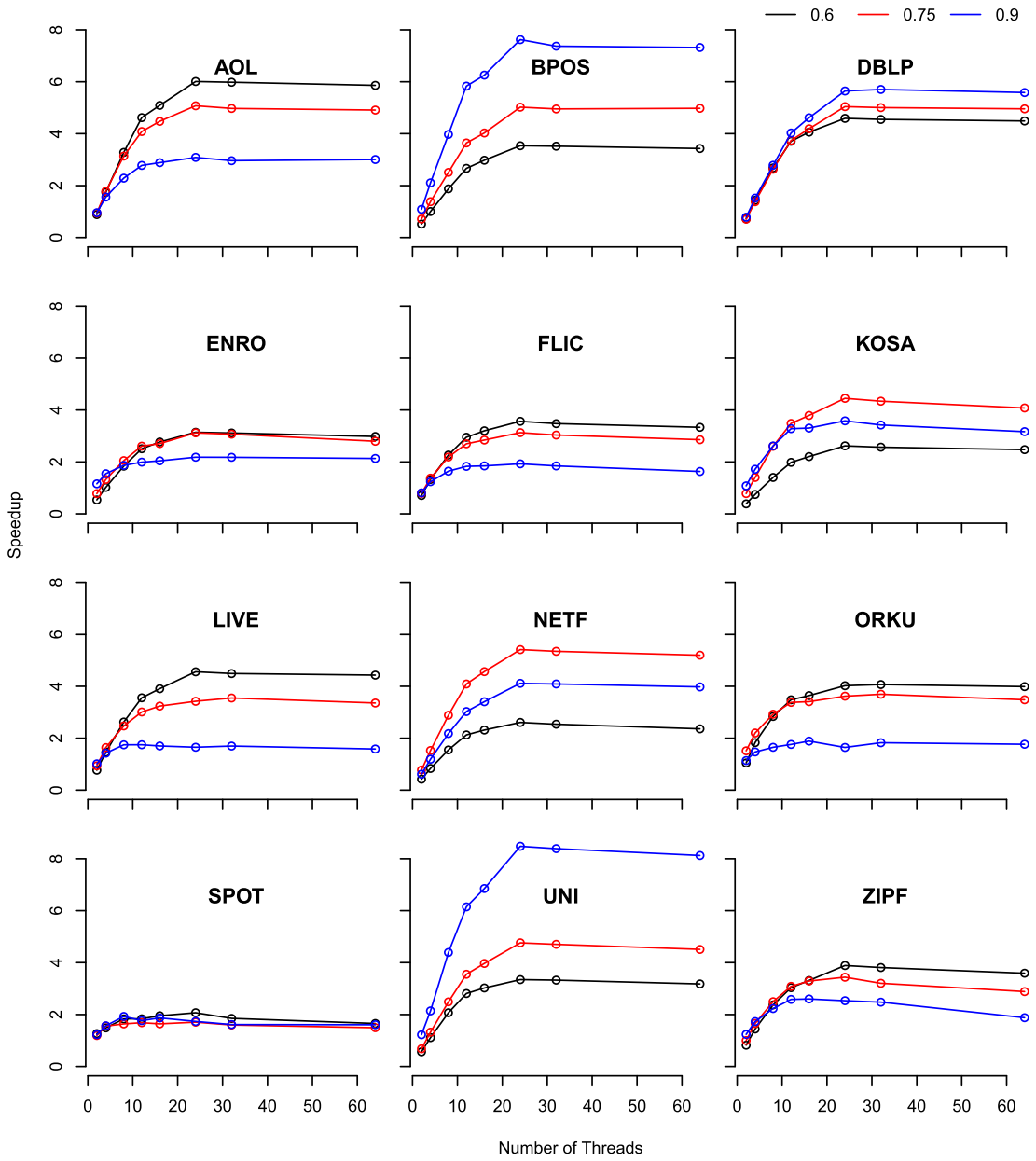


Figure 4.3: Speedup under various datasets and similarity thresholds.

Table 4.1: Average number of index entries per input dataset and threshold. Low numbers marked bold lead to a low scalability.

input	0.6	0.75	0.9
AOL	3.54	2.74	2.60
BPOS	570.48	360.56	213.48
DBLP	310.18	180.46	71.81
ENRO	7.59	4.39	**1.69**
FLIC	4.79	3.08	**1.74**
KOSA	52.84	32.96	19.41
LIVE	3.98	2.36	**1.02**
NETF	1430.47	823.21	311.87
ORKU	9.56	5.53	**2.15**
SPOT	**2.19**	**1.41**	**0.82**
UNI	1494.23	959.54	481.51
ZIPF	12.89	7.58	3.09

Table 4.1 shows the average index lengths for each input and threshold. For SPOT, the average index length varies between 2.19 and 0.82 (for thresholds 0.6 and 0.9, respectively). The index lengths of other datasets and thresholds vary between roughly 1500 (for UNI and threshold 0.6) and 2.6 (AOL, threshold 0.9). Only ENRO, FLIC, LIVE and ORKU reveal comparably short index lengths for a threshold of 0.9. The low speedup of SPOT can be explained by data access patterns, which can improve or prevent prefetching. Our implementation probes the inverted index for each prefix token in each record. If there are sufficiently many entries in the postings list, the CPU can guess that they are needed subsequently. If there is only a small number contained, prefetching does not apply and wait can occur. Longer postings lists in the inverted index give better scaleups as we show in the following subsection.

4.4.3 Impact of Dataset Size

We increase our datasets synthetically by factors 5 and 10 as described in Section 2.3. Figure 4.4 shows the best runtimes for all increased datasets and all thresholds in relation to the best runtime of the corresponding non-increased dataset. For detailed results please refer to Table A.2 in the appendix. With 5× larger data, the runtimes increase between 3.6× and 44×; the numbers for 10× larger data are 6.1× and 182×. In most cases, the runtime does not increase linearly with respect to the data size. The non-linear increase of runtimes is expected, because the SSJ is a quadratic operation. The filter-and-verification framework optimizes the operation only depending on favorable data characteristics.

Only ENRO, FLIC, LIVE, ORKU, SPOT, and ZIPF show roughly a linear runtime increase for a threshold of 0.9; for SPOT we observe linear runtime increases under thresholds 0.75 and 0.6. As we have shown in the previous section with the original datasets, SPOT was not well parallelizable. The relative runtime increase for 5/10× larger data is

below 5/10× for all thresholds, hence the scalability is better for the enlarged datasets. With larger datasets, the postings list lengths in the inverted index also increase. We suspect that the larger SPOT datasets enable prefetching.

4.4.4 Impact of Inlining

Inlining has a positive impact on runtime only for a minority of our experiments, except for the AOL dataset. The boxplots[2] in Figures 4.5-4.6 show the runtime gain of AOL compared to the non-inlined version relative to the parameters *method* (single-threaded [allp], multi-threaded [allph], multi-threaded with CPU affinity [allps]) and *threshold*. There are no clear interdependencies to the other parameters *number of threads*, *batching*, and *position filter*. The figures show that the largest runtime benefit occurs at a threshold of 0.6. Furthermore, only the multicore implementations profit from inlining. BPOS behaves similar to AOL. DBLP shows only small positive effects using inlining. The largest runtime gain occurs for a threshold of 0.9. For KOSA, there is only a positive effect at 0.6. SPOT shows a positive effect only for a threshold of 0.9. Inlining has a neutral or negative effect on the runtimes of ENRO, FLIC, LIVE, NETF, ORKU, UNI, and ZIPF.

We expected inlining to have a positive effect on the filter phase, because it saves pointer chasing to obtain the prefix tokens. It helps the CPUs to perform prefetching. However, if prefixes are much shorter than the complete records, many tokens must be skipped to read the next record. As shown in Table 2.1, AOL has the smallest average record size of 3. For such short lengths, the prefix is usually not much shorter than the record. Thus, prefetching might increase the runtime if there are many short records in the input dataset.

4.4.5 Impact of Batching

We grouped the experimental results by all variables except the batch size and computed the percentaged difference between the lowest and the highest runtime. It varies between 0.7% and 1%. Thus, we consider the impact of batching on the runtime as rather low. Our runtime experiments suggest that the batch size of 125 is the best in most cases (23 times) and 250 is the second best (10 times). We could not find a pattern under which conditions which batch size is optimal. We suspect a complex relation with other variables and with the data characteristics.

4.4.6 Impact of Position Filter

Using position filter is benefitial in most cases with thresholds of 0.6 and 0.75. Figure 4.7 shows the relative runtime gains using the position filter grouped by threshold. For a threshold of 0.6 the runtime gain varies between 20-50% except for SPOT, where the median is close above zero. The position filter has a small impact only on SPOT for

[2]The solid middle line is the median (50% quantile), the box boundaries mark the 25% and the 75% quantile, and the whisker length is 1.5 times the difference between the 75% and the 25% quantile. If the minimum (or maximum) value is larger (or smaller), the whisker ends at that point instead. Points above and below the whiskers show outliers.

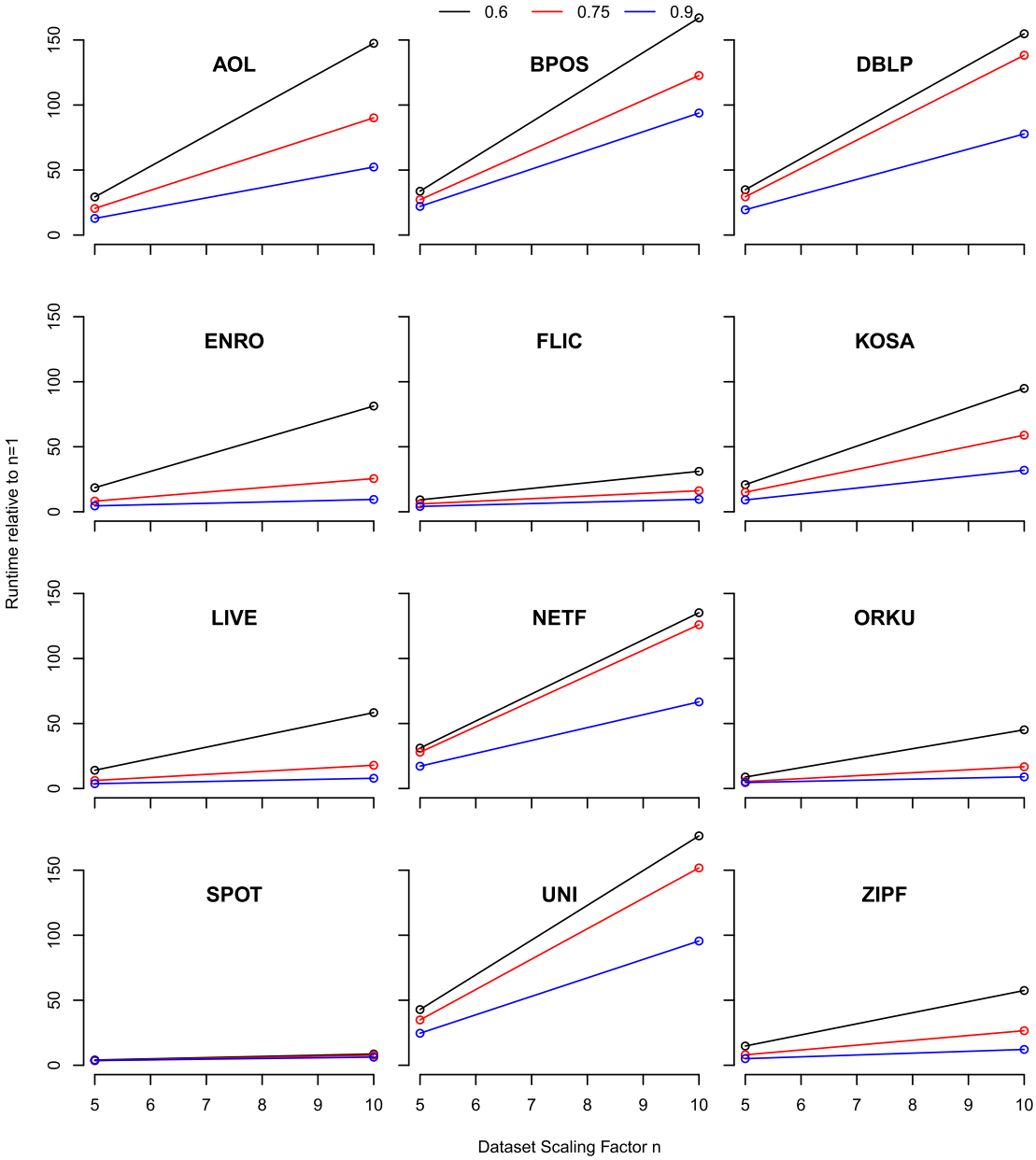


Figure 4.4: Runtimes for increased datasets $n \in \{5, 10\}$ relative to the best runtime for $n = 1$.

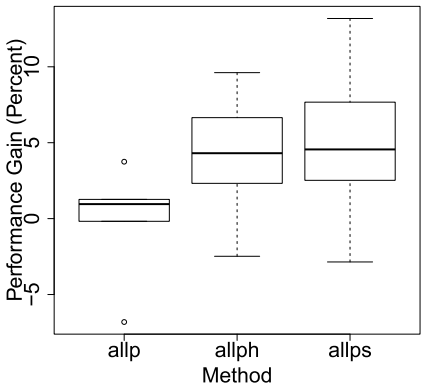


Figure 4.5: AOL: Runtime gain of inlining relative to single-threaded (allp), multi-threaded without (allph), and with CPU affinity (allps).

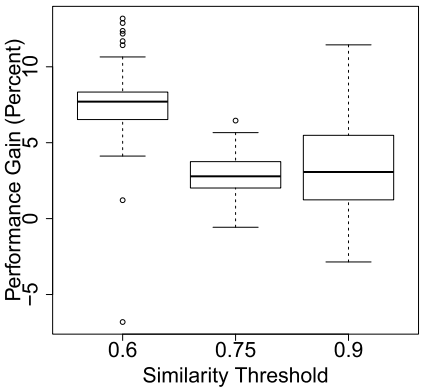


Figure 4.6: AOL: Runtime gain of inlining relative to thresholds.

all thresholds, which can be explained by the number of candidates (cf. Table A.3 in the appendix). For SPOT and the threshold 0.6 the position filter saves roughly 8% of candidates, or in absolute numbers 50 000. The verification of this number of additional candidates is cheaper than to filter them out before. For other datasets this filter saves 28% of candidates on average. Only for AOL, the savings with the position filter are equally low with 7%. However, the absolute number of saved candidates is orders of magnitude higher with 86 600 000, so the position filter pays off for AOL. The `maxFreq` of tokens (cf. Table 2.1) gives a hint on the effectiveness of the prefix filter. The most frequent token in SPOT occurs roughly 9 700 times, which is low compared to all other datasets. A low `maxFreq` implies that the prefix filter generates few candidates. The position filter only pays off if the prefix filter is less effective, which is the case for all other datasets and thresholds below 0.9. For a threshold of 0.75, the gain varies between 5-50% for all datasets except for SPOT (as discussed before) and AOL. For AOL the number of saved candidates relative to the number of candidates without position filter is 0.3% and thus comparably low. For a threshold of 0.9, all gains are close to zero except for DBLP and NETF where there is still a gain of 40% to 50%. One explanation is also the number of saved candidates.

Table 4.2 shows the effect of the position filter between the single-threaded SSJ vs. the multi-threaded (using average runtimes). The effect is the same for the majority of cases. However, for AOL 0.75, FLIC 0.9, KOSA 0.9, ORKU 0.9, and SPOT 0.6 and 0.75 the position filter has a positive effect in the multicore case, while it does not have a positive effect in the single-core case. This observation suggests that the overhead of the position filter pays off more often in the multicore case.

There is no obvious relationship between the runtimes using the position filter and the remaining parameters inlining, batching, the number of threads, and CPU affinity.

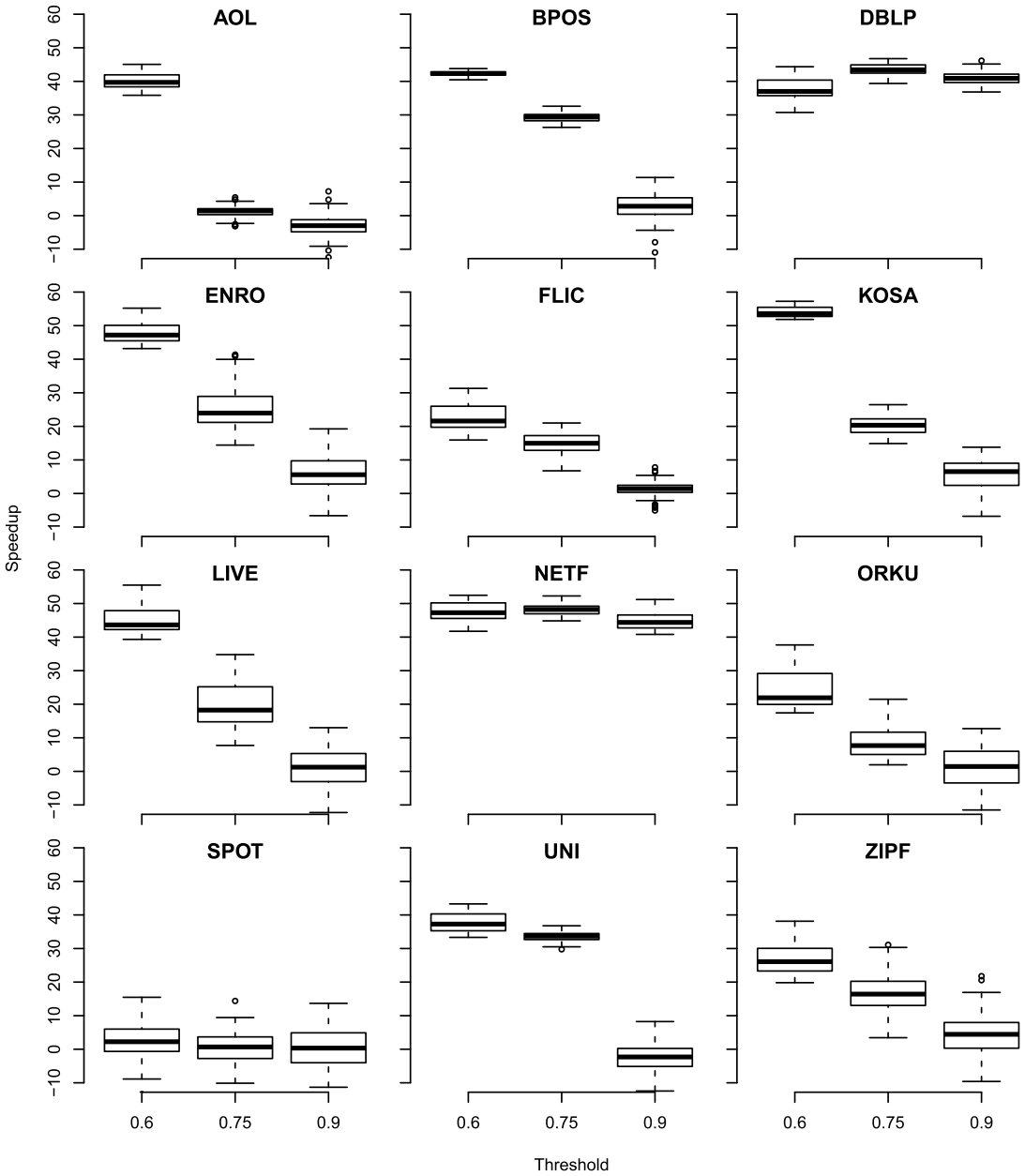


Figure 4.7: Runtime gain/loss of using the position filter grouped by similarity threshold.

Table 4.2: Comparing the effects of position filter on runtime for single-thread and multiple-thread SSJ execution. + stands for a positive (less runtime), - for a negative effect (more runtime).

	0.6		0.75		0.9	
input	single	multi	single	multi	single	multi
AOL	+	+	-	+	-	-
BPOS	+	+	+	+	+	+
DBLP	+	+	+	+	+	+
ENRO	+	+	+	+	+	+
FLIC	+	+	+	+	-	+
KOSA	+	+	+	+	-	+
LIVE	+	+	+	+	+	+
NETF	+	+	+	+	+	+
ORKU	+	+	+	+	-	+
SPOT	-	+	-	+	+	+
UNI	+	+	+	+	-	-
ZIPF	+	+	+	+	+	+

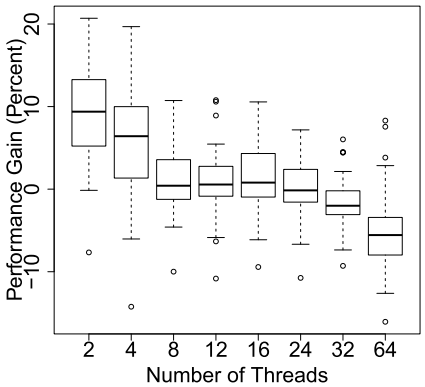


Figure 4.8: Runtime gain/loss of using CPU affinity on the ENRO dataset.

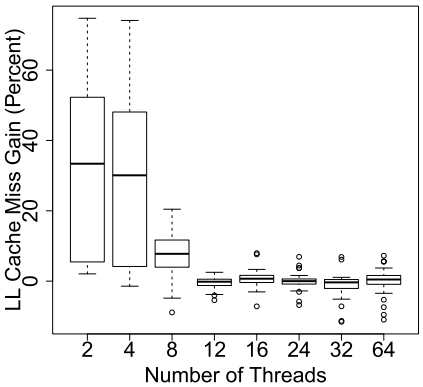


Figure 4.9: Reduction of LLC misses using CPU affinity on the ENRO dataset.

4.4.7 Impact of Thread Placement

By statically assigning the CPU affinity we expected a more optimal use of the cores and prevent thread migrations. However, our experiments reveal that statically assigning the CPU affinity is benefitial for the runtime only in a minority of cases. Figure 4.8 shows the performance gain using CPU affinity for ENRO exemplarily. There is a performance gain from 2 to 4 threads. From there, the gain decreases until reaching 12 threads, stays nearly the same up to 24 threads, and decreases for more threads. This behavior can be explained by the saved cache misses. Figure 4.9 shows the percentage of saved last-level cache (LLC) misses with CPU affinity. The runtime is generally the best from a number of threads starting from 24. Our results suggest that manually setting CPU affinity is not helpful for optimizing SSJ algorithms.

4.5 Summary

Multi-threading has not been systematically considered for filter-and-verification-based SSJ algorithms so far, leaving much computing capability provided by multicore processors unused. In this chapter, we fill the gap to explore the potential of parallelizing SSJ on multicores. We propose a data-parallelization execution model along with various design considerations, including the use of filters, CPU affinity, record inlining and batching. Experiments using real-world datasets revealed several important insights. Using multi-threading improves SSJ runtime by 2–10× on a 12-core machine; the optimal number of threads is often the number of hardware threads (hyperthreads). Surprisingly, unlike in other workloads, using hand-crafted data structures (e. g., inlining) or CPU affinity do not always lead to significantly lower runtimes. We also find that the position filter is more effective than in the single-core scenario and should generally be used for parallel SSJ.

By running experiments on enlarged datasets up to 25 GB we showed the limitations of parallelizing the SSJ on multicores. Albeit using sophisticated parallelization, it inherently remains a quadratic operation with high runtimes on (arguably) large datasets. To push the limit further and allow for even larger datasets, we propose a distributed SSJ framework in the following chapter.

Chapter 5

Exploiting Distributed Parallelization

Existing parallel SSJ approaches are limited regarding the amount of input data they can process. In Chapter 3, we show that common MapReduce-based approaches exhibit high runtimes on only moderately enlarged input datasets. The main reason for the high runtimes is data skew leading to straggling reducers. The maximum amount of input data we were able to join with the MapReduce approaches was 12.5GB (ORKU×5). Adding more nodes does not resolve this issue. In Chapter 4, we scale up existing single-core SSJ approaches using multicores. On our hardware, this approach is able to compute the join on up to 25GB of input data (ORKU×10). Multicore parallelization improves the SSJ runtime. However, the number of cores in one computer is limited. We cannot scale the computation to larger input datasets or decrease the computation runtime by adding more cores.

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. 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 replication* since replication is the main bottleneck of the MapReduce approaches we analyzed previously. Our approach assures that each record is present only once in the main memory of each node. Furthermore, we spawn only one single multi-threaded SSJ instance per node to be able to efficiently share commonly used data structures, 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.

As a distributed execution environment our approach solely requires a standard shared nothing architecture. Our approach is generic so that 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 uses techniques to keep the number of candidates low and to distribute the remaining compute load evenly among nodes. However, depending on the dataset size, token distribution, and similarity threshold, the demand for compute nodes can still be high. Modern cloud computing allows to obtain a high number of compute nodes for a limited timeframe. Thus, we can safely assume that it is realistic today to have hundreds or even thousands of compute nodes available for just one operation. Furthermore, we regard only input datasets without exact duplicates (cf. Section 1.1) as we consider finding exact duplicates an orthogonal problem.

Our approach uses a cost-based data-dependent heuristic to break down the SSJ computation into pieces to be computed independently in parallel. The approach does not compare records with non-matching lengths relative to the similarity measure and threshold. Additionally, we provide a data-independent scaling mechanism that allows to subdivide each piece if necessary. Users can reduce the join runtime to an acceptable one by adding more compute nodes.

The main contributions of this chapter are:

- We introduce a novel distributed-parallel SSJ approach, which is capable of computing the SSJ on hundreds of gigabytes of input data;
- The approach is highly scalable. It uses a cost-based heuristic in conjunction with a data-independent mechanism to split the compute load among nodes;
- The approach is generic and thus potentially applicable to common distributed systems.

In the following, we give an overview on related work. Subsequently, we introduce our approach in detail and experimentally show its behavior on large datasets and high compute node counts. Finally, we conclude the chapter.

5.1 Related Work

In our previous paper we outline our initial approach of a robust cost-based SSJ framework [Fie17]. This initial approach advances existing MapReduce SSJ algorithms by balancing the compute load evenly among the nodes using a cost function and data statistics. The distributed SSJ approach in this chapter is based on our paper. However, in contrast to our initial approach, we removed the assumption of data replication during the execution, because it turned out to be practically infeasible. Furthermore, we removed the system dependency of MapReduce/Hadoop to create a generic approach.

Using statistics for load balancing in distributed systems has been discussed in the literature before. The TopCluster algorithm is an online approach, which includes cardinality estimations at runtime [GARK12]. Our approach on the other hand needs exact data statistics before the join execution. Our approach is comparable to the one by Kolb et al., which also involves a preprocessing MapReduce job to collect data statistics and

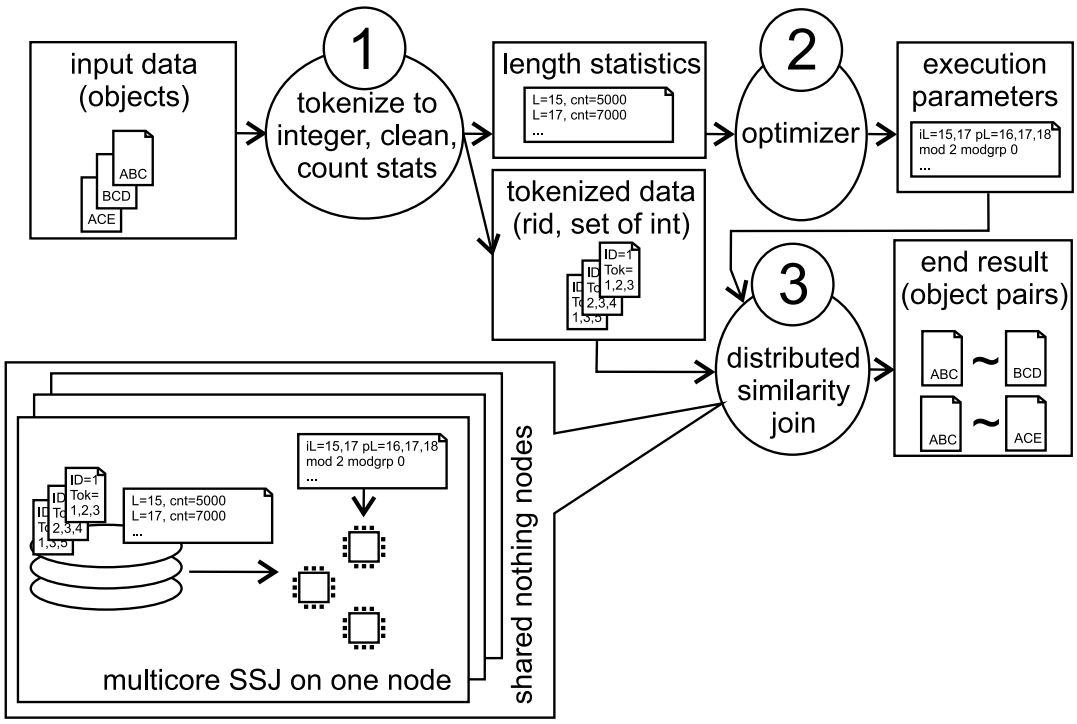


Figure 5.1: Schematic dataflow of our distributed-parallel SSJ approach.

a join job, which uses the statistics for an optimal data grouping [KTR12]. However, in contrast to the solution there, we refrain from data replication.

The authors of DIMA use Apache Spark to compute the SSJ in a distributed fashion [SSL+19]. DIMA creates signatures for each record, such that the SSJ compute load is balanced among the existing nodes. To find such signatures, it partitions the tokens comparable to FS-Join [RLS+17] and MRGroupJoin [DLWF15] (cf. Section 3.2.1), but additionally uses dynamic programming to group them to achieve the balance property. The approach uses distributed and local indexes to speed up the computation. This work is orthogonal to our work by using the execution environment of a specific distributed system, especially its data structures.

5.2 Distributing Filter-and-verification-based SSJ

Figure 5.1 gives 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 length statistic. 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), which we refer to as optimizer, contains the major part of our distributed SSJ approach. It generates parameters for each node to distribute the compute workload. Step (3) is the actual distributed join. We require the tokenized input data and the length statistics to be available on

every compute node. The join is an extension of the multicore SSJ from Chapter 4. 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 approach 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 avoid the unnecessary replication of data, such as inverted indexes, which can be shared by all threads. 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 pieces. 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 the cost-based heuristic.

5.2.1 Data-Dependent Cost-Based Heuristic

One aim of the cost-based heuristic is to avoid the cross product by only regarding record pairs with *matching* lengths, i.e., which survive the AllPairs length filter (cf. Section 2.2.2). This filter is effective on datasets with varying lengths and cheap to apply by using the length statistic computed beforehand. As discussed in Section 1.1, we focus on the Jaccard similarity function and the self-join.

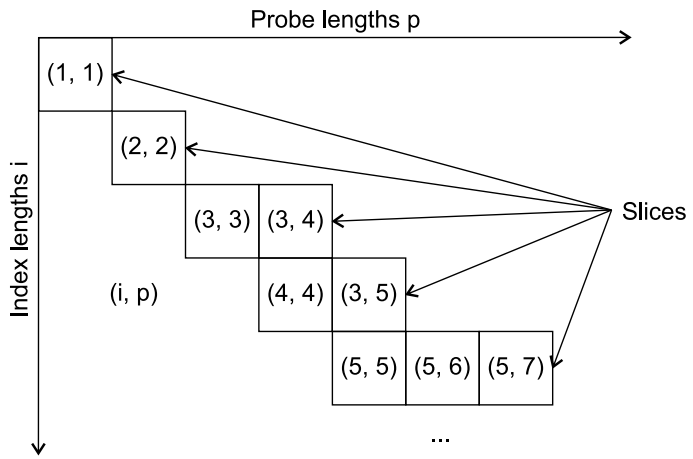


Figure 5.2: Example join matrix for $\theta = 0.7$. Squares with the same index length compose one slice.

Note that records with differing lengths can be similar. For Jaccard, matching lengths for a record r are 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 5.2 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, 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:

$$\mathcal{C}(i) = \mathcal{P}(i) * |R_i| * \sum_{p=i}^{\lfloor\frac{i}{\theta}\rfloor} \mathcal{P}(p) * |R_p| \tag{5.1}$$

Table 5.1 serves as a symbol reference for the symbols we use in the equation and throughout this chapter. The basic assumption of the cost estimation is that each probe of the inverted index causes a cost of the length of the postings list. We do not know the concrete lengths 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 5.1: Symbol reference.

R	input dataset
θ	similarity threshold
$\|r\|$	number of tokens in r
$\|R_l\|$	number of records with length l
$\mathcal{P}(l)$	prefix length of length l: $\mathcal{P}(l) = l - \lceil\theta * l\rceil + 1$
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

Table 5.2: 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 5.1. Table 5.2 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 5.1 highlights that slices can exhibit uneven costs. Thus, we assign *sets* of slices to compute nodes with the intention to distribute the costs evenly. To achieve an even cost distribution, we use a greedy heuristic. 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. We then assign each slice to each node round robin. So 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 5.2. Consider again Table 5.2 and $n = 2$. The highest cost appears for $i = 4$, so we assign this slice to the first node. The next highest cost appears for $i = 5$, so 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 concrete lengths of the postings lists. It assumes all records with matching lengths to be present in it, which is only the worst case. In this regard, it is 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 this approach. Next, we introduce the second part of the optimizer, the data-independent scaling mechanism.

5.2.2 Data-Independent Scaling Mechanism

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

$$isRecordInProbeSubset(rid, m, modgroup) = rid\%m \stackrel{?}{=} modgroup \tag{5.2}$$

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 5.3. 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 $modgroup = 1$. Recall that we ordered the records in our input datasets by ascending record lengths (cf. Section 2.3). 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.

5.2.3 RAM Demand

Our heuristic and the scaling mechanism do not assure 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 5.2.1, 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 the multicore SSJ from Chapter 4 on each compute node. 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 the main memory. Without loss of generality, we estimate the three demands 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 $\mathcal{P}(l) * |R_l|$ for each index length l. We can estimate the size of the inverted index (in Bytes) as follows:

$$indexRamDemand(indexLengths) = \sum_{l \in indexLengths} \mathcal{P}(l) * |R_l| * 12 \tag{5.3}$$

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:

$$probeRamDemand(probeLengths, m) = \sum_{l \in probeLengths} \frac{|R_l| * (60 + l * 4)}{m} \tag{5.4}$$

Lastly, we focus on the candidate size. Our AllPairs-based SSJ approach 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:

$$\begin{aligned} candidateRamDemand(indexLengths, numberThreads, candFact) = \\ \sum_{l \in indexLengths} |R_l| * 12 * numberThreads * candFact \end{aligned} \tag{5.5}$$

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 4GB on the system we run our experiments on and thus consider this value in the following.

Example 5.4. 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 21GB, 2GB for the probe records, and up to 10GB for candidates. We estimate the total demand including the static demand to be 37GB. 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 24GB. 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 join runtime.

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.

5.2.4 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 5.5 illustrates and motivates the need for an appropriate parameter choice.

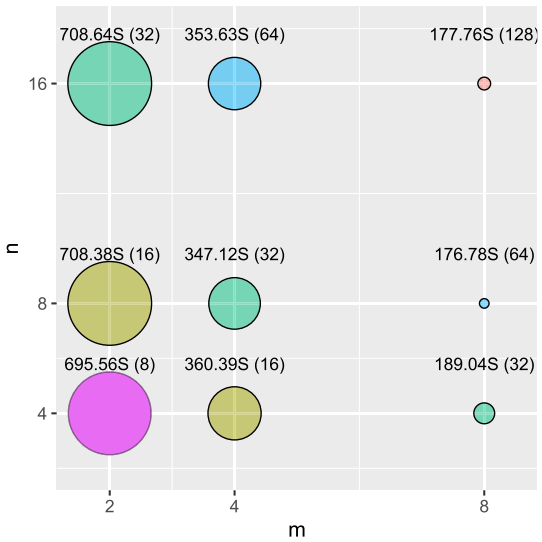


Figure 5.3: AOL×10 runtimes.

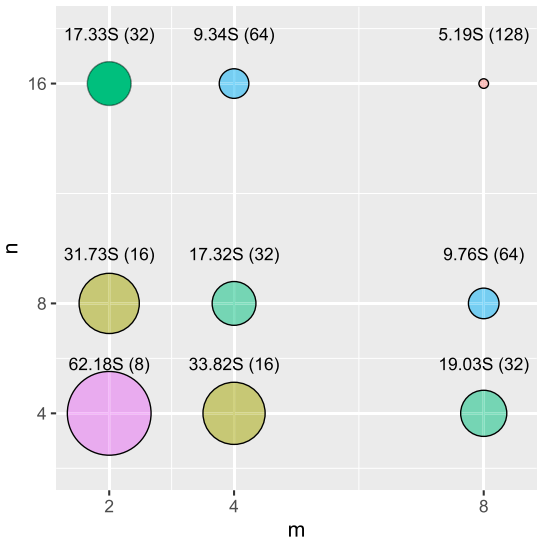


Figure 5.4: ENRO×10 runtimes.

Example 5.5. Figures 5.3 and 5.4 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.

In Example 5.5, the length distributions of the datasets are essential for the efficiency of parameter n. Figure 2.1 shows the length frequencies of the datasets with scaling factor 1. The enlarged datasets exhibit a corresponding distribution with frequencies roughly times the increase factor. 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 5.3 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.

Table 5.3: 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 $\lvert R_i \rvert$	ENRO $\lvert R_i \rvert$
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

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:

$$\min_{n \in valueRange} = \left\{ \max_{i \in [0;n-1]} \{\mathcal{C}(i)\} \div \min_{j \in [0;n-1]} \{\mathcal{C}(j)\} \right\} \tag{5.6}$$

Example 5.6. Consider AOL×10, $\theta = 0.6$, and $n \in \{4, 8, 16, 32\}$. Using Equation 5.6, $n = 4$ 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.

5.2.5 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 5.2.3) and the minimum and maximum cost over all slices (cf. Section 5.2.4). 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 $t = t * 2$ 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.

5.3 Experiments

This subsection 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 the multicore C++ SSJ implementation from Chapter 4 running it on each compute node by extending the multicore SSJ with the parameters *indexLengths*, *probeLengths*, m, and *modgroup*[1]. By default, we run the multicore SSJ with the optimal parameters identified in the previous chapter. 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), 24GBs of RAM, and two 1TB hard disks. Whenever we report runtimes, we refer to the maximum runtime over all slices since the maximum runtime determines the overall runtime. To keep the experimental results comparable to the ones from the previous chapter, we exclusively report the runtimes of the index build and the join (filter and verification).

As input datasets, we continue to use the 10 real-world and two synthetic datasets we described in Section 2.3. 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.

5.3.1 Impact of Cost-based Heuristic

In this experiment, we show how the runtimes develop varying parameter n. We do not set parameter m. Thus, the probe records per slice remain complete with regard to the *probeLengths* computed with the heuristic from Section 5.2.1. Figure 5.5 shows the maximum runtimes over all slices for all datasets scaled by factor 10 and $\theta \in \{0.6, 0.75, 0.9\}$. We choose $n \in \{4, 8, 16, 32\}$ and compare it to the non-distributed multicore SSJ, which is represented by $n = 1$ in Figure 5.5.

For all datasets and all thresholds, $n = 4$ significantly reduces the runtimes compared to $n = 1$. The speedups vary between 1.8 (AOL×10, $\theta = 0.75$) and 13.9 (ORKU×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 (cf. Figure 2.1) and the similarity threshold determine the largest and potentially slowest slice, which cannot be further partitioned with the heuristic. AOL×10 is exemplary for this circumstance. As we discussed in Section 5.2.4, AOL has roughly 80 percent of its records within the length range 1 to 4. n values

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

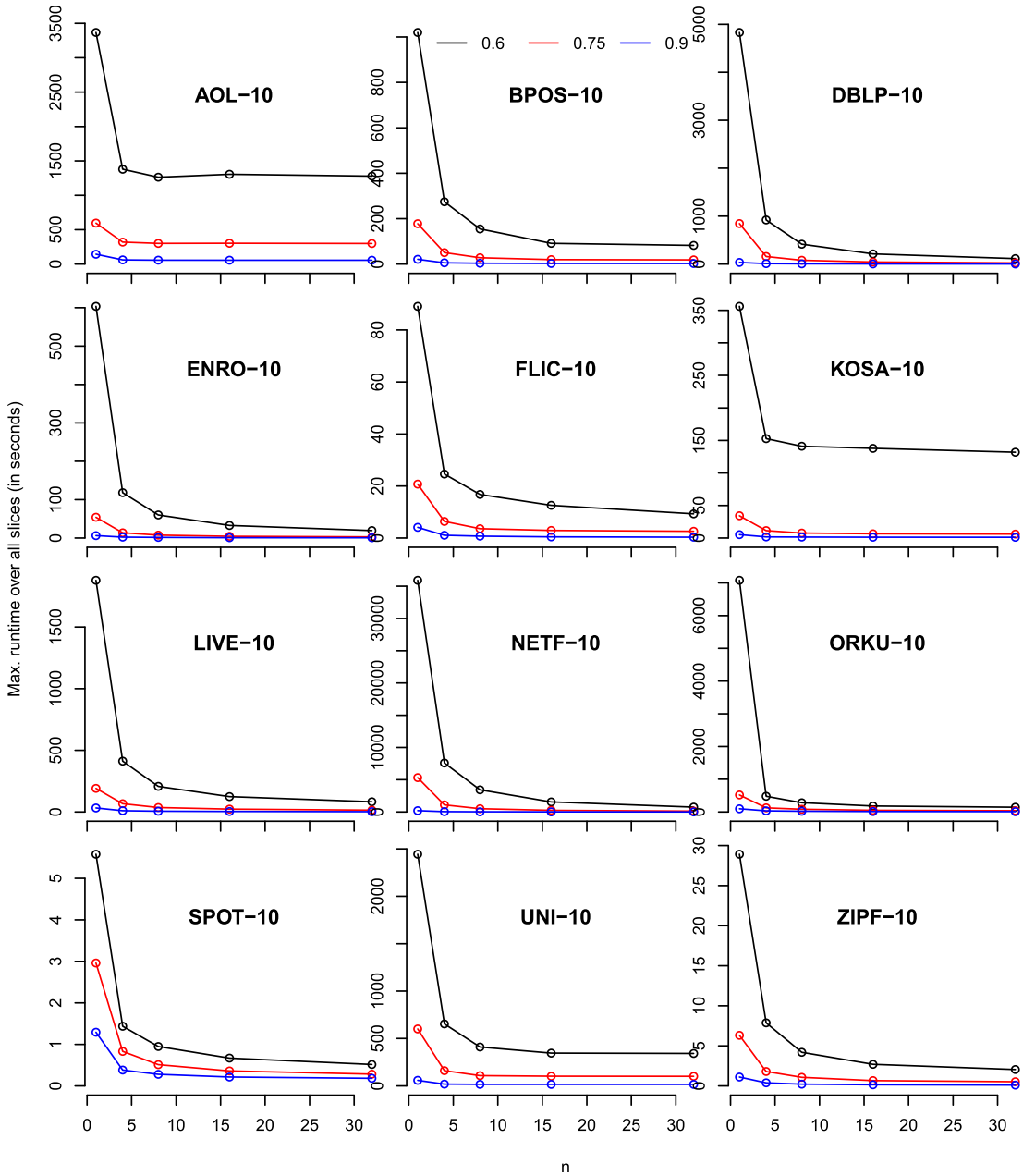


Figure 5.5: Maximum runtimes over all slices for $n \in \{4, 8, 16, 32\}$. $n = 1$ represents the multicore SSJ without distributed parallelization. Thresholds $\theta \in \{0.6, 0.75, 0.9\}$.

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×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. Figure 5.8 visualizes the runtime distributions over all slices of each n. Outliers are marked as points outside the whiskers. Note that there are outliers for $n \in \{8, 16, 32\}$. The highest outlier runtime is close to the highest runtime for $n = 4$. For reference, Figure 5.6 shows the corresponding runtimes for DBLP×10, where such outliers do not occur. The reason for the outlier slices in KOSA×10 are their high number of candidates compared to all other slices. Figures 5.7 and 5.9 show the respective candidate distributions, which explain the runtimes. The runtimes of KOSA×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.

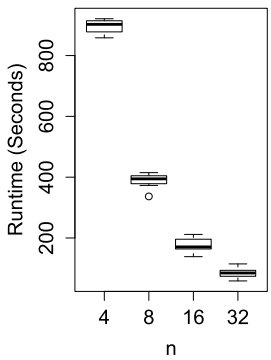


Figure 5.6: DBLP×10 Runtimes.

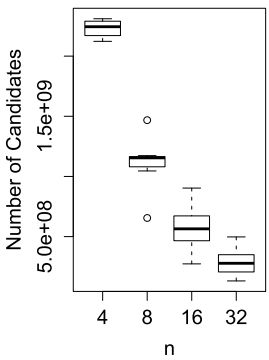


Figure 5.7: DBLP×10 Candidates.

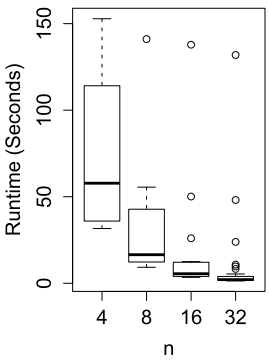


Figure 5.8: KOSA×10 Runtimes.

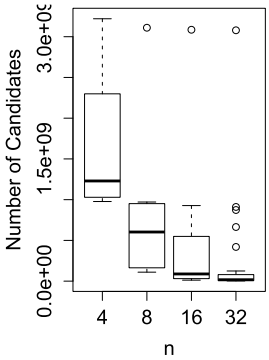


Figure 5.9: KOSA×10 Candidates.

5.3.2 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 $m \in \{2, 4, 8\}$; results for $m = 1$ represent the runtimes from the previous experiment with scaling factor 1. Figure 5.10 shows the maximum runtimes over all slices for each m. The results indicate that $m \geq 2$ is beneficial to achieve a lower runtime for all datasets and thresholds, including AOL×10 and KOSA×10, which showed scalability boundaries for $n \geq 4$ in the previous experiment.

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. Table 5.4 shows the minimum, maximum, and average speedups for $m \in \{2, 4, 8\}$ in relation to $m = 1$, grouped by θ. For each threshold group, there is a maximum

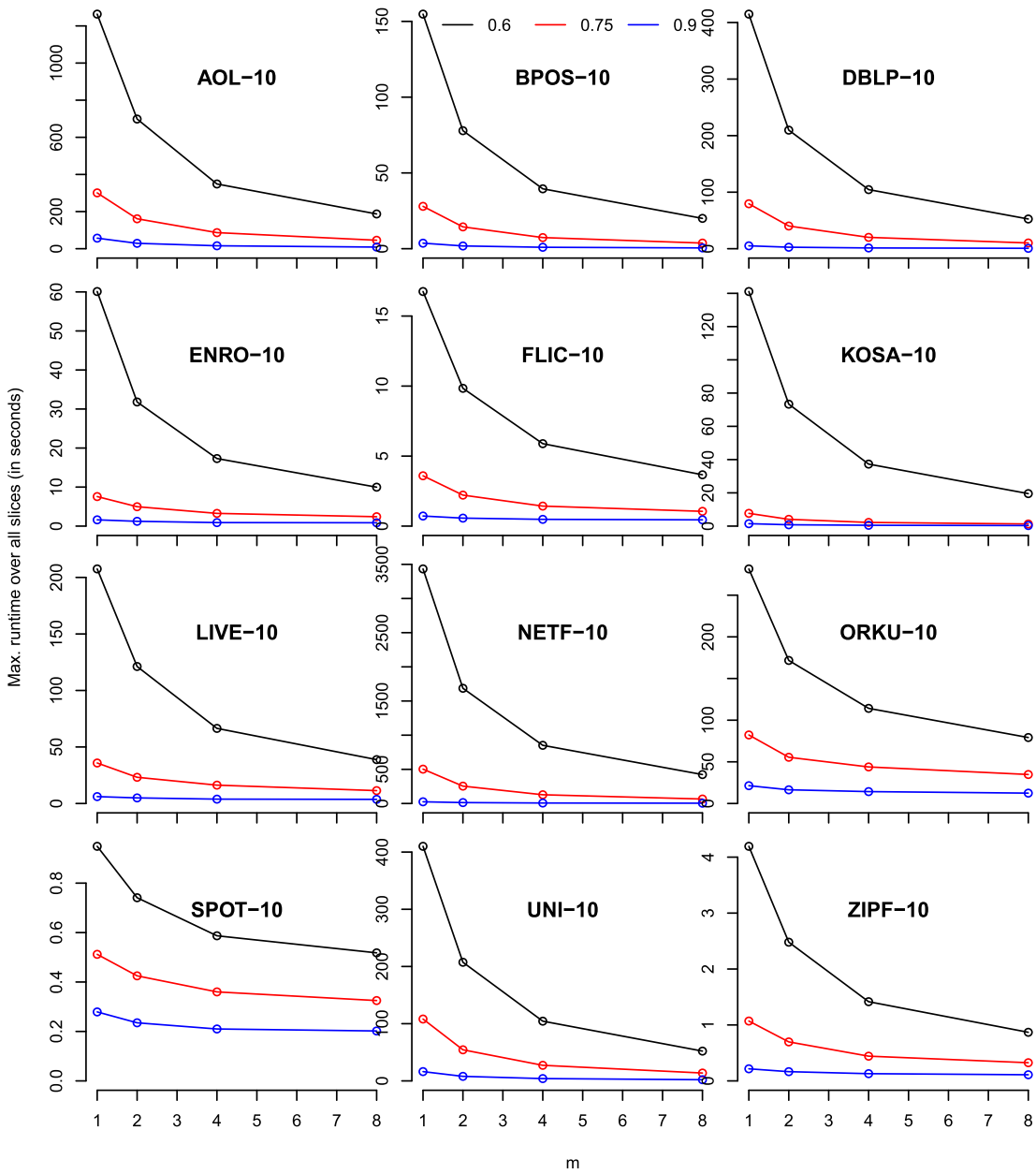


Figure 5.10: Maximum runtimes over all slices for $n = 8$, $\theta \in \{0.6, 0.75, 0.9\}$, $m \in \{2, 4, 8\}$. $m = 1$ indicates runtimes without the scaling mechanism.

speedup close to the optimum m. The averages over all thresholds for $m = 2$ are close to the optimum 2. The average speedups for larger values for m decrease.

We made three observations related to dataset/threshold combinations leading to a suboptimal scalability with respect to m. First, the scalability is better for lower values of θ. The highest scalability values occur for $\theta = 0.6$ with BPOS-10, DBLP×10, NETF×10, and UNI×10. On the opposite, the lowest scalability values occur for $\theta = 0.9$ with FLIC×10, LIVE×10, ORKU×10, SPOT×10, and ZIPF×10. Second, the scalability is better for datasets, which exhibit a more uniform token distribution rather than a Zipfian one. BPOS×10, DBLP×10, NETF×10, and UNI×10 show a roughly uniform distribution (cf. Figure 2.2) and are well scalable. On the other hand, FLIC×10, ORKU×10, SPOT×10, LIVE×10, and ZIPF×10 show a more Zipfian distribution and are less scalable. Lastly, if the runtimes are already low (below one second) as for SPOT×10 under all θ values and FLIC×10 with $\theta = 0.9$, the scalability towards m is suboptimal, which can be explained by static overhead.

Table 5.4: Aggregated speedups relative to $m = 1$ over all datasets grouped by threshold.

θ	$m = 2$			$m = 4$			$m = 8$		
	min	max	avg	min	max	avg	min	max	avg
0.6	1.28	2.04	1.80	1.62	4.03	3.31	1.83	8.11	5.97
0.75	1.20	1.99	1.71	1.42	3.97	2.95	1.58	7.86	5.04
0.9	1.19	2.04	1.60	1.33	4.03	2.58	1.38	7.94	4.10

5.3.3 Impact of Dataset Size

In this subsection, we investigate how the runtimes evolve from increasing the dataset size by scaling factors $s \in \{10, 25, 50, 100\}$. We statically set $n = 8$ and $m = 64$. Figure 5.11 shows the maximum runtimes per slice for $s \in \{25, 50, 100\}$ relative to maximum runtime for $s = 10$. For detailed runtime results, we refer to Table A.4 in the appendix.

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 5.2.3, 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.

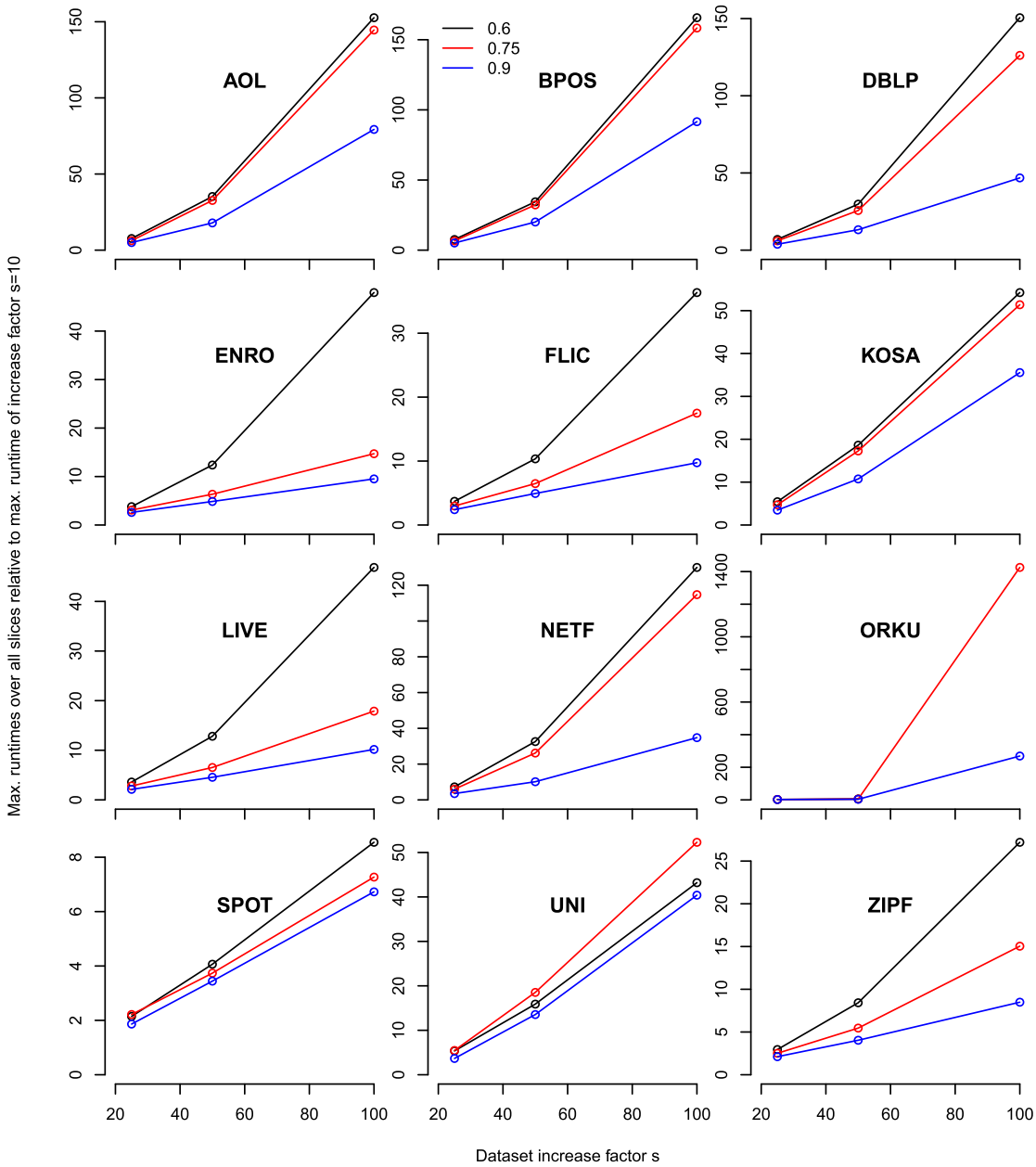


Figure 5.11: Maximum runtimes per slice for $n = 8$ and $m = 64$, varying the dataset increase factor $s \in \{25, 50, 100\}$ relative to the max. runtimes per slice for $s = 10$.

5.3.4 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×100 and $\theta = 0.6$, because the runtime becomes large exceeding 12 hours. In Section 5.2.3, we discussed the same example and concluded that swapping occurs. We first show that our parameter finding strategy from Section 5.2.5 can avoid this worst case. When we apply the parameter strategy to an 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 5.2.4, we discussed that for AOL×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 $n = 4$.

5.4 Summary

In this chapter, 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 datasets much larger than the ones which could be computed with existing parallel methods so far.

Our approach requires parameters. 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.

For the future, it would be interesting to combine our SSJ approach with machine learning techniques. Given a target runtime by the user, such techniques could find matching parameters to compute the SSJ within the desired runtime. Another interesting direction of research would be data representation. With increasing dataset sizes, the efficient storage and retrieval to the RAM of the compute nodes becomes crucial. It might be beneficial for the SSJ runtime efficiency if there would be ways to compress the data losslessly such that potentially matching records (or groups of them) can still be filtered. Also the integration into a concrete Big Data system with dedicated data structures and concurring processes would be interesting.

Chapter 6

Conclusions

In this thesis, we investigated the parallelization of algorithms for computing the set similarity join (SSJ) on multicores and shared-nothing compute nodes:

- We analytically and experimentally compared existing MapReduce-based SSJ algorithms. We evaluated the algorithms and discussed their strengths and limits;
- We proposed a novel multicore-parallel filter-and-verification-based SSJ approach. We experimentally evaluated the runtime and scalability of this approach;
- We proposed a novel highly scalable distributed-parallel SSJ approach. We evaluated the approach using enlarged datasets.

This thesis makes an important contribution to the SSJ research with a detailed analysis on the unexpected scalability limits of existing MapReduce approaches. It significantly improves the scalability of SSJ algorithms and their implementation in a multicore and multi-node execution environment. With the novel multicore SSJ we utilize modern hardware to achieve significantly better runtimes than single core SSJ approaches. Our novel distributed SSJ approach is independent from specific Big Data systems and proposes a generic way to partition the SSJ. We introduce cost models and resource estimations to evenly distribute compute load and avoid swapping. Our highly scalable approach pushes the input dataset size limits of existing parallel approaches significantly.

6.1 Future Work

Interesting directions of future work are:

- Using GPU in conjunction with CPU and distributed parallelization for the SSJ. It could be investigated under which conditions and how GPUs can be efficiently used to enhance the scalability of the SSJ;
- Applying machine learning techniques to optimize the parameters for the execution of our distributed SSJ approach. It would be especially beneficial to find a way to estimate candidates a priori;

- Adapt the distributed SSJ approach to a concrete Big Data system. Especially the efficient usage of existing data structures might be challenging and interesting.

The overall goal of future work should be to keep it comparable to previous work and benchmark against it. Also robustness against data characteristics was important throughout this thesis and should be regarded for future work on SSJs.

Appendix A

Appendix

A.1 Tables

Table A.1: Multicore: Best runtimes for each input dataset and threshold comparing multicore and single-core executions. Parameters: number of threads *thr.*, CPU affinity *aff.*, use of position filter (*posf.*), use of inlining (*inline*), and *batch* size. *rm* are the best average multicore runtimes, *r1* the best average single-core runtimes (both in seconds).

input	θ	thr.	aff.	posf.	inline	batch	rm (S)	r1 (S)	speedup
AOL	0.60	24	0	1	1	500	22.83	235.48	10.31
AOL	0.75	24	1	0	1	250	6.57	35.02	5.33
AOL	0.90	24	0	0	0	250	2.71	9.21	3.40
BPOS	0.60	32	1	1	1	250	6.06	39.22	6.47
BPOS	0.75	32	1	1	1	250	1.44	10.71	7.44
BPOS	0.90	32	0	0	1	250	0.22	1.78	8.20
DBLP	0.60	24	0	1	1	125	29.80	227.42	7.63
DBLP	0.75	24	1	1	1	125	5.88	56.95	9.69
DBLP	0.90	24	1	1	1	125	0.43	4.57	10.53
ENRO	0.60	24	0	1	1	125	7.13	43.52	6.10
ENRO	0.75	24	1	1	1	125	2.07	8.70	4.19
ENRO	0.90	24	0	1	1	1000	0.66	1.62	2.45
FLIC	0.60	24	1	1	0	125	2.55	12.87	5.05
FLIC	0.75	24	0	1	0	125	1.16	4.68	4.01
FLIC	0.90	24	1	0	1	125	0.42	0.85	2.02
KOSA	0.60	24	1	1	1	125	3.75	21.62	5.77
KOSA	0.75	24	1	1	1	125	0.57	3.24	5.70
KOSA	0.90	24	1	1	1	125	0.17	0.64	3.85
LIVE	0.60	32	1	1	0	250	32.46	259.23	7.99
LIVE	0.75	24	1	1	0	250	10.57	46.11	4.36
LIVE	0.90	24	1	1	1	125	4.08	8.16	2.00
NETF	0.60	24	1	1	0	125	262.46	1265.62	4.82

input	θ	thr.	aff.	posf.	inline	batch	rm (S)	r1 (S)	speedup
NETF	0.75	24	0	1	1	125	41.49	438.55	10.57
NETF	0.90	24	1	1	1	250	2.90	21.52	7.42
ORKU	0.60	32	0	1	0	125	65.63	342.40	5.22
ORKU	0.75	24	1	1	0	250	29.94	121.86	4.07
ORKU	0.90	24	1	1	1	250	10.72	20.33	1.90
SPOT	0.60	24	1	1	1	125	0.58	1.25	2.16
SPOT	0.75	24	0	1	1	125	0.39	0.71	1.82
SPOT	0.90	24	0	1	1	500	0.21	0.42	2.00
UNI	0.60	24	0	1	0	125	13.42	78.51	5.85
UNI	0.75	24	0	1	0	125	3.82	28.48	7.46
UNI	0.90	24	1	0	1	125	0.55	5.25	9.47
ZIPF	0.60	24	0	1	1	125	0.50	2.78	5.60
ZIPF	0.75	32	1	1	1	125	0.23	0.97	4.32
ZIPF	0.90	24	1	1	1	125	0.09	0.26	2.80

Table A.2: Multicore: Best average absolute runtimes r_n on increased datasets with increase factor n in seconds and relative runtimes $rr_5 = \frac{r_5}{r_1}$ and $rr_{10} = \frac{r_{10}}{r_1}$.

input	θ	r_1	r_5	r_{10}	rr_5	rr_{10}
AOL	0.60	22.83	668.17	3367.58	29.26	147.47
AOL	0.75	6.57	135.28	595.89	20.59	90.68
AOL	0.90	2.71	34.97	143.28	12.90	52.86
BPOS	0.60	6.06	205.78	1019.95	33.97	168.35
BPOS	0.75	1.44	39.39	177.83	27.36	123.52
BPOS	0.90	0.22	4.94	21.05	22.75	97.00
DBLP	0.60	29.80	1087.32	4831.31	36.49	162.12
DBLP	0.75	5.88	179.74	845.07	30.59	143.84
DBLP	0.90	0.43	8.63	34.52	19.90	79.61
ENRO	0.60	7.13	137.30	603.54	19.25	84.60
ENRO	0.75	2.07	17.34	54.08	8.36	26.07
ENRO	0.90	0.66	3.00	6.28	4.53	9.48
FLIC	0.60	2.55	26.27	89.07	10.30	34.94
FLIC	0.75	1.16	7.75	20.73	6.65	17.80
FLIC	0.90	0.42	1.76	4.11	4.19	9.77
KOSA	0.60	3.75	78.26	356.09	20.89	95.03
KOSA	0.75	0.57	8.73	34.24	15.33	60.18
KOSA	0.90	0.17	1.48	5.21	8.99	31.58
LIVE	0.60	32.46	452.08	1878.60	13.93	57.87
LIVE	0.75	10.57	66.21	191.27	6.26	18.09
LIVE	0.90	4.08	15.25	32.22	3.74	7.90
NETF	0.60	265.67	8256.49	35918.50	31.08	135.20
NETF	0.75	42.27	1179.56	5325.16	27.90	125.97

input	θ	r_1	r_5	r_{10}	rr_5	rr_{10}
NETF	0.90	2.99	51.22	199.02	17.15	66.64
ORKU	0.60	65.63	593.31	3039.43	9.04	46.31
ORKU	0.75	29.94	165.48	519.44	5.53	17.35
ORKU	0.90	10.72	49.70	96.53	4.64	9.00
SPOT	0.60	0.58	2.42	5.20	4.18	8.98
SPOT	0.75	0.39	1.39	2.96	3.56	7.62
SPOT	0.90	0.21	0.76	1.29	3.59	6.14
UNI	0.60	13.42	593.10	2443.45	44.20	182.10
UNI	0.75	3.82	138.01	600.85	36.15	157.39
UNI	0.90	0.55	15.06	58.49	27.18	105.58
ZIPF	0.60	0.50	7.44	28.92	14.96	58.20
ZIPF	0.75	0.23	1.91	6.32	8.48	28.03
ZIPF	0.90	0.09	0.46	1.11	4.93	11.85

Table A.3: Multicore: Number of candidates without position filter and the number of saved candidates using the position filter.

	0.6		0.75		0.9	
input	# cand	# saved	# cand	# saved	# cand	# saved
AOL	1 308 195 395	86 604 209	477 737 983	1 462 862	118 099 782	1 425
BPOS	224 417 691	43 406 692	77 689 193	6 636 410	13 719 100	215 952
DBLP	156 143 630	51 875 687	41 437 407	14 854 689	4 716 630	1 476 104
ENRO	22 809 972	10 237 840	5 312 895	1 986 620	723 727	280 488
FLIC	72 361 084	14 893 299	28 601 771	2 613 289	5 386 489	179 509
KOSA	146 607 084	27 617 867	24 508 507	1 157 877	3 889 750	40 059
LIVE	358 213 341	107 366 064	76 240 199	11 666 792	9 137 126	160 393
NETF	506 267 551	288 259 041	98 457 108	55 148 405	7 354 527	3 640 376
ORKU	26 475 950	11 385 557	4 595 978	1 660 877	476 480	78 329
SPOT	625 364	50 032	197 035	6 433	102 457	123
UNI	452 910 564	110 484 321	183 463 661	28 514 181	36 770 329	469
ZIPF	2 050 369	638 168	429 949	127 164	71 315	15 419

Table A.4: Distributed: Max. runtimes r_s on increased datasets with increase factor s in seconds, $n = 8$ and $m = 64$. Relative runtimes $rr_{25} = \frac{r_{25}}{r_{10}}$, $rr_{50} = \frac{r_{50}}{r_{10}}$, and $rr_{100} = \frac{r_{100}}{r_{10}}$.

dataset	θ	r_{10}	r_{25}	r_{50}	r_{100}	rr_{25}	rr_{50}	rr_{100}
AOL	0.60	26.67	203.91	939.59	4071.52	7.65	35.23	152.66
AOL	0.75	7.65	47.88	249.86	1105.99	6.26	32.66	144.57
AOL	0.90	2.51	12.57	44.91	198.99	5.01	17.89	79.28
BPOS	0.60	2.91	22.07	100.44	482.86	7.58	34.47	165.70
BPOS	0.75	0.57	3.76	18.43	90.42	6.59	32.27	158.35
BPOS	0.90	0.13	0.69	2.69	12.27	5.11	20.07	91.57

dataset	θ	r_{10}	r_{25}	r_{50}	r_{100}	rr_{25}	rr_{50}	rr_{100}
DBLP	0.60	7.09	49.28	211.34	1067.66	6.95	29.80	150.57
DBLP	0.75	1.57	9.48	40.36	198.20	6.03	25.69	126.16
DBLP	0.90	0.19	0.73	2.54	8.98	3.82	13.23	46.79
ENRO	0.60	3.04	11.52	37.61	145.76	3.79	12.37	47.95
ENRO	0.75	1.32	4.09	8.42	19.49	3.09	6.35	14.71
ENRO	0.90	0.56	1.44	2.69	5.28	2.60	4.86	9.51
FLIC	0.60	1.33	4.92	13.75	48.31	3.71	10.35	36.35
FLIC	0.75	0.63	1.88	4.10	11.07	2.97	6.48	17.49
FLIC	0.90	0.31	0.75	1.54	3.05	2.40	4.93	9.74
KOSA	0.60	3.49	18.94	64.96	189.06	5.43	18.63	54.22
KOSA	0.75	0.33	1.55	5.67	16.85	4.71	17.28	51.38
KOSA	0.90	0.15	0.53	1.64	5.44	3.47	10.75	35.56
LIVE	0.60	14.81	53.27	189.97	694.14	3.60	12.82	46.85
LIVE	0.75	7.01	19.62	45.76	125.35	2.80	6.53	17.89
LIVE	0.90	3.04	6.45	13.85	30.91	2.12	4.56	10.17
NETF	0.60	56.34	409.64	1836.40	7321.80	7.27	32.60	129.96
NETF	0.75	10.39	62.39	272.06	1192.07	6.01	26.19	114.75
NETF	0.90	1.50	5.28	15.21	52.11	3.52	10.15	34.79
ORKU	0.60	61.48	147.92	338.83	inf	2.41	5.51	inf
ORKU	0.75	26.69	69.41	141.71	38050.00	2.60	5.31	1425.47
ORKU	0.90	10.98	22.51	41.31	2953.43	2.05	3.76	268.88
SPOT	0.60	0.41	0.89	1.68	3.53	2.15	4.07	8.55
SPOT	0.75	0.27	0.61	1.02	1.99	2.22	3.74	7.27
SPOT	0.90	0.18	0.33	0.61	1.18	1.86	3.44	6.73
UNI	0.60	7.37	39.49	117.05	318.30	5.36	15.89	43.21
UNI	0.75	1.89	10.31	34.96	98.68	5.46	18.53	52.30
UNI	0.90	0.43	1.56	5.77	17.21	3.66	13.54	40.40
ZIPF	0.60	0.40	1.17	3.33	10.77	2.95	8.42	27.20
ZIPF	0.75	0.19	0.49	1.06	2.92	2.51	5.46	15.04
ZIPF	0.90	0.09	0.19	0.35	0.75	2.11	4.03	8.49

Bibliography

[AB13] Nikolaus Augsten and Michael Böhlen. Similarity joins in relational database systems. *Synthesis Lectures on Data Management*, 2013.

[ABG10] Nikolaus Augsten, Michael Böhlen, and Johann Gamper. The pq-gram distance between ordered labeled trees. *ACM Transactions on Database Systems (TODS)*, 2010.

[AGK06] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2006.

[Amd67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *Proceedings of the Joint Computer Conference*, 1967.

[AMNK14] Nikolaus Augsten, Armando Miraglia, Thomas Neumann, and Alfons Kemper. On-the-fly token similarity joins in relational databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2014.

[Aol] AOL Dataset. `http://www.cim.mcgill.ca/~dudek/206/Logs/AOL-user-ct-collection/`. [Online; accessed January 2021].

[ASM+12] Foto N Afrati, Anish Das Sarma, David Menestrina, Aditya Parameswaran, and Jeffrey D Ullman. Fuzzy joins using MapReduce. *International Conference on Data Engineering (ICDE)*, 2012.

[BDFML10] Ranieri Baraglia, Gianmarco De Francisci Morales, and Claudio Lucchese. Document similarity self-join with MapReduce. *International Conference on Data Mining (ICDM)*, 2010.

[BG19] Christos Bellas and Anastasios Gounaris. Exact set similarity joins for large datasets in the GPGPU paradigm. *Proceedings of the International Workshop on Data Management on New Hardware*, 2019.

[BGM12] Panagiotis Bouros, Shen Ge, and Nikos Mamoulis. Spatio-textual similarity joins. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2012.

[BMGT16] Panagiotis Bouros, Nikos Mamoulis, Shen Ge, and Manolis Terrovitis. Set containment join revisited. *Knowledge and Information Systems*, 2016.

[BMS07] Roberto J Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. *Proceedings of the International Conference on World Wide Web*, 2007.

[Bpo] BPOS Dataset. `https://www.kdd.org/kdd-cup/view/kdd-cup-2000/Data`. [Online; accessed January 2021].

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. *International Conference on Data Engineering (ICDE)*, 2006.

[Chr07] Peter Christen. Performance and scalability of fast blocking techniques for deduplication and data linkage. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2007.

[Dbl] DBLP Dataset. `https://dblp.uni-trier.de/`. [Online; accessed February 2014].

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Operating Systems Design and Implementation (OSDI)*, 2004.

[DLH+14] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. MassJoin: A MapReduce-based method for scalable string similarity joins. *International Conference on Data Engineering (ICDE)*, 2014.

[DLWF15] Dong Deng, Guoliang Li, He Wen, and Jianhua Feng. An efficient partition based method for exact set similarity joins. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2015.

[Dre07] Ulrich Drepper. What every programmer should know about memory. 2007.

[ELO08] Tamer Elsayed, Jimmy Lin, and Douglas W Oard. Pairwise document similarity in large collections with MapReduce. *Proceedings of the Annual Meeting of the Association for Computational Linguistics on Human Language Technologies*, 2008.

[Enr] ENRO Dataset. `https://www2.cs.sfu.ca/~jnwang/projects/adapt/`. [Online; accessed January 2021].

[FAB+18] Fabian Fier, Nikolaus Augsten, Panagiotis Bouros, Ulf Leser, and Johann-Christoph Freytag. Set similarity joins on MapReduce: an experimental survey. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2018.

[Fie17] Fabian Fier. Distributed similarity joins on big textual data: toward a robust cost-based framework. *Proceedings of the VLDB PhD Workshop co-located with the International Conference on Very Large Databases*, 2017.

[FWZF20] Fabian Fier, Tianzheng Wang, Erkang Zhu, and Johann-Christoph Freytag. Parallelizing filter-verification based exact set similarity joins on multicores. *Proceedings of the International Conference on Similarity Search and Applications (SISAP)*, 2020.

[GARK12] Benjamin Gufler, Nikolaus Augsten, Angelika Reiser, and Alfons Kemper. Load balancing in MapReduce based on scalable cardinality estimates. *International Conference on Data Engineering (ICDE)*, 2012.

[GIJ+01] Luis Gravano, Panagiotis G Ipeirotis, Hosagrahar Visvesvaraya Jagadish, Nick Koudas, Shanmugauelayut Muthukrishnan, Divesh Srivastava, et al. Approximate string joins in a database (almost) for free. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2001.

[Hen06] Monika Henzinger. Finding near-duplicate web pages: a large-scale evaluation of algorithms. *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2006.

[JLFL14] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String similarity joins: an experimental evaluation. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2014.

[JP05] Ravindranath Jampani and Vikram Pudi. Using prefix-trees for efficiently computing set joins. *International Conference on Database Systems for Advanced Applications*, 2005.

[JS08] Edwin H Jacox and Hanan Samet. Metric space similarity joins. *ACM Transactions on Database Systems (TODS)*, 2008.

[KLH+99] Hantak Kwak, Ben Lee, Ali R Hurson, Suk-Han Yoon, and Woo-Jong Hahn. Effects of multithreading on cache performance. *IEEE Transactions on Computers*, 1999.

[Kos] KOSA Dataset. `http://fimi.uantwerpen.be/data/`. [Online; accessed January 2021].

[KRS+16] Anja Kunkel, Astrid Rheinländer, Christopher Schiefer, Sven Helmer, Panagiotis Bouros, and Ulf Leser. PIEJoin: towards parallel set containment joins. *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM)*, 2016.

[KTR12] Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for MapReduce-based entity resolution. *International Conference on Data Engineering (ICDE)*, 2012.

[LDWF11] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: a partition-based method for similarity joins. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2011.

[Liv] LIVE and ORKU Datasets. `http://socialnetworks.mpi-sws.org/data-imc2007.html`. [Online; accessed January 2021].

[LLL08] Chen Li, Jiaheng Lu, and Yiming Lu. Efficient merging and filtering algorithms for approximate string searches. *International Conference on Data Engineering (ICDE)*, 2008.

[LRU20] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive data sets*. Cambridge university press, 2020.

[LTMN12] Wuman Luo, Haoyu Tan, Huajian Mao, and Lionel M. Ni. Efficient similarity joins on massive high-dimensional datasets using MapReduce. *IEEE International Conference on Mobile Data Management (MDM)*, 2012.

[MA14] Willi Mann and Nikolaus Augsten. PEL: position-enhanced length filter for set similarity joins. In *Grundlagen von Datenbanken*. 2014.

[MAB16] Willi Mann, Nikolaus Augsten, and Panagiotis Bouros. An empirical evaluation of set similarity join techniques. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2016.

[MAEA07] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Detectives: detecting coalition hit inflation attacks in advertising networks streams. *Proceedings of the International Conference on World Wide Web*, 2007.

[MF12] Ahmed Metwally and Christos Faloutsos. V-SMART-Join: a scalable MapReduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2012.

[MJZ17] Youzhong Ma, Shijie Jia, and Yongxin Zhang. A novel approach for high-dimensional vector similarity join query. *Concurrency and Computation: Practice and Experience*, 2017.

[MMW16] Youzhong Ma, Xiaofeng Meng, and Shaoya Wang. Parallel similarity joins on massive high-dimensional data using MapReduce. *Concurrency and Computation: Practice and Experience*, 2016.

[Net] NETF Dataset. `https://www.cs.uic.edu/~liub/Netflix-KDD-Cup-2007.html`. [Online; accessed January 2021].

[OR11] Alper Okcan and Mirek Riedewald. Processing theta-joins using MapReduce. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2011.

[QRRM17] Rafael David Quirino, Sidney Ribeiro-Júnior, Leonardo Andrade Ribeiro, and Wellington Santos Martins. Efficient filter-based algorithms for exact set similarity join on GPUs. In *International Conference on Enterprise Information Systems (ICEIS)*, 2017.

[RH11] Leonardo Andrade Ribeiro and Theo Härder. Generalizing prefix filtering to improve set similarity joins. *Information Systems*, 2011.

[RLS+17] Chuitian Rong, Chunbin Lin, Yasin N Silva, Jianguo Wang, Wei Lu, and Xiaoyong Du. Fast and scalable distributed set similarity joins for big data analytics. *International Conference on Data Engineering (ICDE)*, 2017.

[RLW+13] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony KH Tung. Efficient and scalable processing of string similarity join. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 2013.

[SHC14] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. ClusterJoin: a similarity joins framework using Map-Reduce. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2014.

[SMD+10] Angela C Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh. Parallelism via multithreaded and multicore CPUs. *Computer*, 2010.

[Spo] SPOT Dataset. `https://dbis-informatik.uibk.ac.at/context-aware-music-recommendation`. [Online; accessed January 2021].

[SR12] Yasin N Silva and Jason M Reed. Exploiting MapReduce-based similarity joins. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[SRB+16] Yasin N Silva, Jason Reed, Kyle Brown, Adelbert Wadsworth, and Chuitian Rong. An experimental survey of MapReduce-based similarity joins. *Proceedings of the International Conference on Similarity Search and Applications (SISAP)*, 2016.

[SSL+17] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. Dima: a distributed in-memory similarity-based query processing system. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2017.

[SSL+19] Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, and Zhifeng Bao. Balance-aware distributed string similarity-based query processing system. *Proceedings of the International Conference on Very Large Data Bases (PVLDB)*, 2019.

[TSP08] Martin Theobald, Jonathan Siddharth, and Andreas Paepcke. Spotsigs: robust and efficient near duplicate detection in large web collections. *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2008.

[VCL10] Rares Vernica, Michael J Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2010.

[Whi12] Tom White. *Hadoop: The definitive guide.* O'Reilly Media, Inc., 2012.

[WLF12] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering? An adaptive framework for similarity join and search. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

[WMP13] Ye Wang, Ahmed Metwally, and Srinivasan Parthasarathy. Scalable all-pairs similarity search in metric spaces. In *Proceedings of SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2013.

[WXQ+16] Pei Wang, Chuan Xiao, Jianbin Qin, Wei Wang, Xiaoyang Zhang, and Yoshiharu Ishikawa. Local similarity search for unstructured text. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2016.

[XWL+11] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 2011.

[XWLS09] Chuan Xiao, Wei Wang, Xuemin Lin, and Haichuan Shang. Top-k set similarity joins. *International Conference on Data Engineering (ICDE)*, 2009.

[ZDNM19] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J Miller. JOSIE: overlap set similarity search for finding joinable tables in data lakes. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2019.