Balance-Aware Distributed String Similarity-Based Query Processing System

Ji Sun[†] Zeyuan Shang[†‡] Guoliang Li[†] Dong Deng[†‡] Zhifeng Bao[*]
[†]Tsinghua University, [‡]MIT, [*]RMIT University

sun-j16@mails.tsinghua.edu.cn, zeyuans@mit.edu, liguoliang@tsinghua.edu.cn, dengdong@csail.mit.edu, zhifeng.bao@rmit.edu.au

ABSTRACT

Data analysts spend more than 80% of time on data cleaning and integration in the whole process of data analytics due to data errors and inconsistencies. Similarity-based query processing is an important way to tolerate the errors and inconsistencies. However, similarity-based query processing is rather costly and traditional database cannot afford such expensive requirement. In this paper, we develop a distributed in-memory similarity-based query processing system called `Dima`. `Dima` supports four core similarity operations, i.e., similarity selection, similarity join, top-k selection and top-k join. `Dima` extends SQL for users to easily invoke these similarity-based operations in their data analysis tasks. To avoid expensive data transmission in a distributed environment, we propose *balance-aware signatures* where two records are similar if they share common signatures, and we can adaptively select the signatures to balance the workload. `Dima` builds signature-based global indexes and local indexes to support similarity operations. Since Spark is one of the widely adopted distributed in-memory computing systems, we have seamlessly integrated `Dima` into Spark and developed effective query optimization techniques in Spark. To the best of our knowledge, this is the first full-fledged distributed in-memory system that can support complex similarity-based query processing on large-scale datasets. We have conducted extensive experiments on four real-world datasets. Experimental results show that `Dima` outperforms state-of-the-art studies by 1-3 orders of magnitude and has good scalability.

PVLDB Reference Format:
Ji Sun, Zeyuan Shang, Guoliang Li, Dong Deng, Zhifeng Bao. Balance-Aware Distributed String Similarity-Based Query Processing System. *PVLDB*, 12(9): 961-974, 2019.
DOI: https://doi.org/10.14778/3329772.3329774

1. INTRODUCTION

In big data era, data are full of errors and inconsistencies and cause much trouble to data analysts. As reported in a New York Times article, 80% of a typical data science project is cleaning and preparing the data, while the remaining 20% is the actual data analysis. String similarity-based query processing is indispensable in data integration systems (e.g. the state-of-the-art data integration system relies on string similarity to find candidate pairs from datasets[28, 21, 7]). Therefore, it is demanding to have efficient and effective similarity-based query processing techniques to serve the large-scale data cleaning job. However, similarity-based query processing is very costly and traditional database cannot afford such expensive requirement [19, 16]. Some efficient serial algorithms [4, 44, 3, 40, 43, 39, 37, 31, 29] and parallel algorithms on Hadoop [24, 8, 12, 36, 2] have been proposed to improve the efficiency. They, however, suffer from several limitations. First, they are not full-fledged – they only support simple similarity operations but cannot support complex data analysis (e.g., SQL-based analysis), and have no effective optimization on queries involving multiple similarity operations. Second, the serial algorithms are not efficient to support large-scale data analysis. Third, the parallel algorithms still have the workload balance problem.

To address these limitations, we develop a distributed in-memory system `Dima` to support SQL-based similarity-based query processing. In particular, `Dima` focuses on supporting four core similarity-based operations, i.e., similarity selection, similarity join, top-k selection and top-k join. Similarity selection extends traditional exact selection by tolerating errors and finds similar results. Similarity join extends traditional exact join by tolerating errors between records and finds similar pairs of records. Top-k selection (top-k join) computes the k most similar records (similar pairs).

One big challenge in distributed computing is to avoid expensive data transmission. An effective way is to judiciously assign data into different partitions such that the results must be in the same partition (and avoid the Cartesian product over different partitions). To achieve this goal, we propose effective signatures where two records are similar if they share common signatures. On top of these signatures, we build global indexes and local indexes to support similarity operations, which can avoid unnecessary data transmission among irrelevant partitions.

Another challenge is to balance the workload among partitions. To this end, we propose the concept of *balance-aware signatures*, which are adaptively selectable based on the workload. Based on selected signatures and effective indexes, we devise efficient algorithms to support similarity-based query processing. In particular, for similarity selection, we propose dynamic-programming algorithms to select

Proceedings of the VLDB Endowment, Vol. 12, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3329772.3329774

[1]Guoliang Li is the corresponding author.

the optimal signatures; for similarity join, we prove that the optimal signature selection problem is NP-hard and propose a greedy algorithm to select high-quality signatures; for top-k selection and top-k join, we progressively select the signatures and avoid generating all signatures.

In summary, we make the following contributions.

(1) We develop a full-fledged distributed in-memory similarity-based query processing system `Dima`, which provides SQL-based programming interface and DataFrame API for further data analysis (Section 2). To the best of our knowledge, *this is the first distributed in-memory system that can support similarity-based query processing.*

(2) We devise selectable signatures and propose a *novel balance-aware signature selection* framework in order to balance the workload in distributed environments (Section 4).

(3) We propose global and local indexes, and design efficient algorithms to support similarity selection (Section 5), join (Section 6), and top-k selection and join (Section 7).

(4) We develop *cost-based query optimization* techniques to further enhance the performance (Section 8). As a result, complex SQL-based data analysis can be supported by `Dima`, beyond the standalone basic similarity operations.

(5) We have implemented `Dima` on top of Spark and conducted extensive experiments on real-world datasets (Section 9). The results show that `Dima` outperforms existing studies by *1-3 orders of magnitude*. Our source code is publicized at `https://github.com/TsinghuaDatabaseGroup/dima`.

2. SIMILARITY-BASED QUERY PROCESSING FRAMEWORK

We first define four core similarity-based query operations (Section 2.1) and then introduce our framework (Section 2.2). Finally we review related work (Section 3).

2.1 Similarity-Based Query Operations

Given two records r and s, we use a similarity function to compute their similarity. There are many choices, e.g., Jaccard, Cosine, Dice, and edit distance. Due to space limit, we focus on how to support Jaccard and the details for supporting other functions are in our technical report [34].

We first tokenize records as sets of tokens and the Jaccard similarity between r and s is $\text{JAC}(r,s) = \frac{|r \cap s|}{|r \cup s|}$, where $r \cap s$ and $r \cup s$ are the overlap and union of r and s respectively. Two records are similar w.r.t. Jaccard if their similarity is not smaller than a threshold τ. Next we define four core similarity-based operations.

DEFINITION 1 (SIMILARITY SELECTION). *Given a collection of records $\mathcal{R}$, a query s, a similarity function f and a threshold τ, the similarity selection problem aims to find all similar records from $\mathcal{R}$, i.e., $\{r \in \mathcal{R} | \text{JAC}(r,s) \geq \tau\}$.*

DEFINITION 2 (TOP-k SELECTION). *Given a collection of records $\mathcal{R}$, a query s, a similarity function f and an integer k, the top-k similarity selection problem aims to find k records from $\mathcal{R}$ with the largest Jaccard similarity.*

DEFINITION 3 (SIMILARITY JOIN). *Given two collections of records $\mathcal{R}$ and $\mathcal{S}$, a similarity function f and a threshold τ, the similarity join problem aims to find all similar record pairs, i.e., $\{(r \in \mathcal{R}, s \in \mathcal{S}) | \text{JAC}(r,s) \geq \tau\}$.*

DEFINITION 4 (TOP-k SIMILARITY JOIN). *Given two collections of records $\mathcal{R}$ and $\mathcal{S}$, a similarity function f and an integer k, the top-k similarity join problem aims to find k record pairs from the two sets with the largest similarity.*

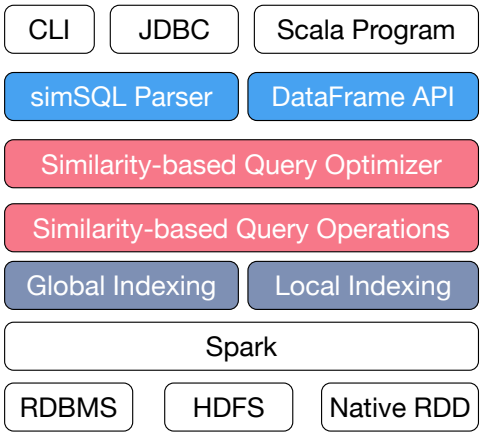


Figure 1: The Framework of `Dima`.

We extend SQL and define `simSQL` to support these four similarity operations.

(1) Similarity Selection. Users utilize the following `simSQL` query to find records in table T whose column $\mathcal{S}$ is similar to query s w.r.t. a similarity function f and threshold τ.

`SELECT * FROM` T `WHERE` $f(T.\mathcal{S}, s) \geq \tau$

(2) Top-k Similarity Selection. Users utilize the following `simSQL` query to find k records in table T whose column $\mathcal{S}$ has the largest similarity to query s w.r.t. a similarity function f and integer k.

`SELECT * FROM` T `WHERE KNN`$(f, T.\mathcal{S}, s, k)$

(3) Similarity Join. Users utilize the following `simSQL` query to find the records in tables T_1 and T_2 where T_1's column $\mathcal{S}$ is similar to T_2's column $\mathcal{R}$ w.r.t. a similarity function f and threshold τ.

`SELECT * FROM` T_1 `SIMJOIN` T_2 `ON` $f(T_1.\mathcal{S}, T_2.\mathcal{R}) \geq \tau$

(4) Top-k Similarity Join. Users utilize the following `simSQL` query to find k records in tables T_1 and T_2 with the largest similarity on table T_1's column $\mathcal{S}$ and table T_2's column $\mathcal{R}$ w.r.t. a similarity function f and integer k.

`SELECT * FROM` T_1 `SIMJOIN` T_2 `ON KNN`$(f, T_1.\mathcal{S}, T_2.\mathcal{R}, k)$

2.2 Our Framework

Our goal is to devise effective indexes and algorithms to support `simSQL` queries with single or multiple operations.

DataFrame. In addition to `simSQL`, users can perform similarity operations over DataFrame objects using a domain-specific language similar to data frames in R. We extend Spark DataFrame API to support similarity operations.

Index. Users can utilize the following `simSQL` query to create indexes (including global index and local index) on column $\mathcal{S}$ of table T using our signature based indexing scheme `SEGINDEX`, which will be introduced in Section 4.

`CREATE Index` SegIndex `ON` $T.\mathcal{S}$ `USE SEGINDEX`.

Similarity-Based Query Processing. For a selection query, we utilize the global index to prune irrelevant partitions and send the query request to relevant partitions. In each local partition, we utilize the local index to compute local answers. For a join query, we utilize the global index to make similar pairs be in the same partition to avoid expensive data transmission. In each partition, we utilize the local index to compute local join answers. For top-k selection and join, we progressively identify the top-k results. The details are discussed in Sections 5, 6, and 7.

Query Optimization. `Dima` extends the Catalyst optimizer of Spark SQL and introduces a cost-based optimiza-

tion (CBO) module to optimize the similarity-based queries. The CBO module leverages the (global and local) index to optimize complex `simSQL` queries. Query optimization in `Dima` is discussed in Section 8.

Workflow of `Dima`. Figure 1 shows the architecture of `Dima`. Its processing workflow is as follows. Given a `simSQL` query or a DataFrame object, `Dima` first constructs a tree model by the `simSQL` parser or a DataFrame object by the DataFrame API. Then `Dima` builds a logical plan using Catalyst rules. Next, the logical optimizer applies standard rule-based optimization to optimize the logical plan. Then `Dima` applies cost-based optimizations based on signature-based indexes and statistics to generate the most efficient physical execution plan.

3. RELATED WORK

Similarity Selection. There are many studies on similarity-based selection [11, 48, 23, 22, 17, 6, 46]. They utilized a count-based framework where the data records are similar to the query if they share enough common elements, e.g., tokens, q-grams (substrings of q-length), with the query [22, 48, 23, 17, 6, 11], using inverted lists to count the number.

Similarity Join. There are many studies on similarity join [4, 44, 3, 40, 43, 39, 12, 37, 31, 14, 45, 15, 10, 27, 38, 33, 41, 25]. Jiang et al. [19] conducted a comprehensive experimental study. Existing studies employed a signature-based framework, which generates some signatures for each record such that two records are similar if they share at least one common signature. There are two effective signatures, prefix filtering [4, 44, 40, 42] and segment-based filtering [26, 13]. The former sorts the elements and selects several infrequent elements as signatures such that if two records do not share a common signature, they cannot be similar. The latter partitions each record into different segments and takes the segments as signatures such that if two records are similar they must share a common signature. In addition, some studies [32, 18, 47, 5] focus on probabilistic techniques for set-based similarity join. However, they cannot find the exact answer and need to tune parameters which are tedious and less effective [4].

Different from existing works, we focus on distributed in-memory setting. We also extend them to support our setting and compare with them. Our system significantly outperforms them (see Section 9), because they involve huge amount of unnecessary data transmission and cannot address the data skew problem while we design novel balance-aware signatures, effective index, and efficient algorithms. Our segment-signature based method is much more efficient than the prefix-signature based method because the latter has the imbalance problem in parallel computing.

MapReduce-Based Similarity Join. There are some works on supporting similarity join using Map-Reduce framework [37, 30, 1, 9, 12]. Vernica et al. [37] utilized the prefix filtering to support set-based similarity functions. Metwally et al. [30] proposed a 2-stage algorithm for joining over sets, multisets and vectors. Afrati et al. [1] optimized the map, reduce and communication cost. Kim et al. [20] addressed the top-k similarity join problem using MapReduce. Deng et al [12] focused on supporting edit distance. However, they involve large disk IOs and data transmission in the cluster. Our system significantly outperforms them due to our effective balance-aware signatures, indexes and algorithms. Moreover, we offer more easy-to-use APIs, and plentiful functions for users. Different from our demo paper [35], we provide more technical details.

Spark. Spark is a fault-tolerant, distributed in-memory computing engine. Spark SQL enables Spark to support relational data query processing. However, Spark SQL does not support similarity operations, and we extend Spark SQL to support similarity-based query processing.

4. INDEXING

We propose a selectable signature that provides multiple signature options and we can judiciously select the signatures to reduce the transmission cost in query processing.

4.1 Selectable Signatures

4.1.1 Basic Idea

Segments. Given a data record r and a query record s, we split them into the same number of disjoint segments, say $\eta_{|r|}$ segments. $|r|$ is the number of tokens in r and we discuss how to set $\eta_{|r|}$ later. In order to assign the same token in different records to the same segment, we keep a hash function $\Gamma_{|r|}$ that maps a token t to the i-th segment, i.e., $\Gamma_{|r|}(t) = i$ where $1 \leq i \leq \eta_{|r|}$. Let $\texttt{iSig}^+_{r,i,|r|}$ and $\texttt{pSig}^+_{s,i,|r|}$ denote the i-th segment of r and s respectively. If $\texttt{iSig}^+_{r,i,|r|} \neq \texttt{pSig}^+_{s,i,|r|}$, we can deduce that r and s have at least 1 mismatched token.

Number of Segments $\eta_{|r|}$. Suppose s is similar to r, i.e., $\text{JAC}(r,s) \geq \tau$. We have $\frac{|r\cap s|}{|r\cup s|} \geq \tau$, $1-\frac{|r\cap s|}{|r\cup s|} \leq 1-\tau$, $\frac{|r\cup s|-|r\cap s|}{|r\cup s|} \leq 1-\tau$, $|r\cup s - r\cap s| \leq (1-\tau)|r\cup s| \leq (1-\tau)\frac{|r\cap s|}{\tau} \leq \frac{1-\tau}{\tau}|r|$. Since $|r\cup s - r\cap s|$ is the number of mismatch tokens between r and s, and $|r\cup s - r\cap s| \leq \frac{1-\tau}{\tau}|r|$, s has at most $\lfloor \frac{1-\tau}{\tau}|r| \rfloor$ mismatched tokens with r (see Lemma 1). If we split r and s into $\eta_{|r|} = \lfloor \frac{1-\tau}{\tau}|r| \rfloor + 1$ segments, s must share a common segment with r (otherwise s has more than $\lfloor \frac{1-\tau}{\tau}|r| \rfloor$ mismatched tokens with r).

LEMMA 1. *If s is similar to r, $|r\cup s - r\cap s| \leq \frac{1-\tau}{\tau}|r|$, i.e., s has at most $\lfloor \frac{1-\tau}{\tau}|r| \rfloor$ mismatched tokens with r.*

(1) Pruning Based on Segments. If $\texttt{iSig}^+_{r,i,|r|} \neq \texttt{pSig}^+_{s,i,|r|}$ for every $i \in [1, \eta_{|r|}]$, then r and s have at least $\eta_{|r|}$ mismatched tokens and they cannot be similar based on Lemma 1. We utilize this property to prune dissimilar pairs.

Figure 2 shows an example with a data record $r_1 = \{a,b,c,d,e\}$ and a query record $s_1 = \{a,b,c,d,e,f\}$. Suppose $\tau = 0.8$. We can get $\eta_{|r_1|} = \lfloor \frac{1-0.8}{0.8} * 5 \rfloor + 1 = 2$, so we split r_1 into two segments. The two segments of r_1 are $\{a,c,e\}$ and $\{b,d\}$, and the two segments of s_1 are $\{a,c,e\}$, $\{b,d,f\}$. Since they share a common segment, they could be similar. Consider another query record $s_2 = \{a,b,c,d,f,g\}$. The two segments of s_2 are $\{a,c,g\}$, $\{b,d,f\}$. As r_1 and s_2 have no common segment, they have at least two mismatch tokens and thus they cannot be similar based on Lemma 1.

Deletions on Segments. Let $\texttt{iSig}^-_{r,i,|r|}$ ($\texttt{pSig}^-_{s,i,|r|}$) denote the deletion set of sub-segments by removing a token from $\texttt{iSig}^+_{r,i,|r|}$ ($\texttt{pSig}^+_{s,i,|r|}$). The deletion set of $\{b,d,f\}$ is $\{\{b,d\},\{b,f\},\{d,f\}\}$.

(2) Pruning Based on Deletions. Actually, there may be more than one tuple for $\texttt{pSig}^+_{s,i,l}$ as figure 2, therefore we denote $\texttt{iSig}^+_{r,i,l}$ and $\texttt{pSig}^+_{s,i,l}$ as sets.

(*i*) If $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^+_{r,i,l} = \emptyset$, s and r have at least one mismatched token in the i-th segment.

Table 1: Important Notations.

Notations	Descriptions
τ	Threshold
η_l	#Segments for length l, $\eta_l = \lfloor \frac{1-\tau}{\tau} l \rfloor + 1$
$\theta_{\lvert s\rvert,\lvert r\rvert}$	Bound of #Mismatch Segments for r and s $\theta_{\lvert s\rvert,\lvert r\rvert} = \lfloor \frac{1-\tau}{1+\tau}(\lvert r\rvert + \lvert s\rvert) \rfloor + 1$
$\texttt{iSig}^+_{r,i,\lvert r\rvert}$	the i-th indexing segment signature of record r
$\texttt{iSig}^-_{r,i,\lvert r\rvert}$	the i-th indexing deletion signature of record r
$l^-_{\lvert s\rvert}/l^+_{\lvert s\rvert}$	Min/Max Length $l^-_{\lvert s\rvert} = \lceil \tau \lvert s\rvert \rceil$; $l^+_{\lvert s\rvert} = \lfloor \frac{\lvert s\rvert}{\tau} \rfloor$
$\texttt{pSig}^+_{s,i,l}$	i-th probing segment signature of s for length l
$\texttt{pSig}^-_{s,i,l}$	i-th probing deletion signature of s for length l
$\mathcal{F}^+/\mathcal{F}^-$	Frequency table for indexing signature
$\mathcal{L}^+/\mathcal{L}^-$	Inverted list for an indexing signature
$\mathcal{J}^+/\mathcal{J}^-$	Inverted list for a probing segment signature

(ii) If $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^+_{r,i,l}=\emptyset$ & $\texttt{iSig}^+_{r,i,l} \cap \texttt{pSig}^-_{s,i,l}=\emptyset$ & $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^-_{r,i,l}=\emptyset$, r and s have at least 2 mismatched tokens on the i-th segment. This is because if r and s have only one mismatched token, we have either $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^-_{r,i,l} \neq \emptyset$ or $\texttt{iSig}^+_{r,i,l} \cap \texttt{pSig}^-_{s,i,l} \neq \emptyset$. For example, consider $\texttt{iSig}^+_{r,i,|r|} = \{b,d,f\}$ and $\texttt{pSig}^+_{s,i,|r|} = \{b,e,g\}$. $\texttt{iSig}^-_{r,i,|r|} = \{\{b,d\},\{b,f\},\{d,f\}\}$ and $\texttt{pSig}^-_{s,i,|r|} = \{\{b,e\},\{b,g\},\{e,g\}\}$. As $\texttt{pSig}^-_{s,i,l} \cap \texttt{iSig}^+_{r,i,l}=\emptyset$ & $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^-_{r,i,l}=\emptyset$ & $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^+_{r,i,l}=\emptyset$, $\{b,d,f\}$ and $\{b,e,g\}$ have at least 2 mismatch tokens.

(3) Hybrid Pruning. Lemma 1 gives an upper bound of the number of mismatch tokens only based on $|r|$. Next, we give a tighter bound based on $|r|$ and $|s|$. As $\frac{|r\cap s|}{|r\cup s|} \geq \tau$, $\frac{|r\cap s|}{|r|+|s|-|r\cap s|} \geq \tau$, $|r\cap s| \geq \frac{\tau}{1+\tau}(|r|+|s|)$, $|r\cup s - r\cap s| = |r|+|s|-2|r\cap s| \leq |r|+|s|-2\frac{\tau}{1+\tau}(|r|+|s|) = \frac{1-\tau}{1+\tau}(|r|+|s|)$. Let $\theta_{|s|,|r|} = \lfloor \frac{1-\tau}{1+\tau}(|r|+|s|) \rfloor + 1$ denote the *dissimilar threshold bound*, i.e., if r and s have at least $\theta_{|s|,|r|}$ mismatched tokens, they cannot be similar (see Lemma 2).

LEMMA 2. *If s is similar to r, $|r\cup s - r\cap s| \leq \frac{1-\tau}{1+\tau}(|r|+|s|)$, i.e., s has at most $\frac{1-\tau}{1+\tau}(|r|+|s|)$ mismatched tokens with r.*

By selecting some segments and deletions of s, if we find s has at least $\theta_{|s|,|r|}$ mismatched tokens with r, we can prune (r,s). Next we discuss how to select some segments and deletions of s as its signatures.

4.1.2 Selectable Signatures Generation

Indexing Signatures. Given any record r, we generate two types of indexing signatures: *indexing segment signatures and indexing deletion signatures.*

Indexing Segment Signatures. We split a record r into $\eta_{|r|}$ disjoint segments $seg_1, seg_2, \cdots, seg_{\eta_{|r|}}$, and $\texttt{iSig}^+_{r,i,|r|} = (seg_i, i, |r|)$ is an indexing segment signature for $1 \leq i \leq \eta_{|r|}$.

Indexing Deletion Signatures. For each segment signature $\texttt{iSig}^+_{r,i,|r|} = (seg_i, i, |r|)$, we generate an indexing deletion signature $\texttt{iSig}^-_{r,i,|r|,j} = (del^j_i, i, |r|)$ where del^j_i is a subset of seg_i by deleting the j-th token ($1 \leq j \leq |seg_i|$). Let $\texttt{iSig}^-_{r,i,|r|} = \cup_{j=1}^{|seg_i|}\{\texttt{iSig}^-_{r,i,|r|,j}\}$ denote the set of deletion signatures for the i-th segment.

For each indexing segment/deletion signature, we use an inverted list to keep records that contain the signature.

Probing Signatures. Given a record s, if it is similar to record r, the length difference between s and r should not be too large. In other words, s can only be similar to a record

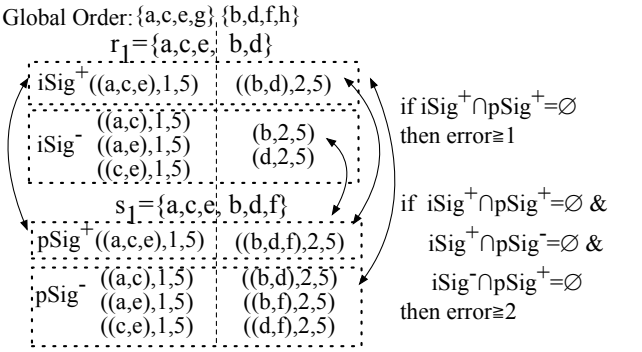


Figure 2: Signature Example.

r whose length $|r|$ ranges in $[l^-_{|s|}, l^+_{|s|}]$. Since $|r| \geq |r\cap s| \geq |r\cup s| \cdot \tau \geq |s| \cdot \tau$, we have $l^-_{|s|} = \lceil |s| \cdot \tau \rceil$. Similarly, since $|r| \leq |r\cup s| \leq \frac{|r\cap s|}{\tau} \leq \frac{|s|}{\tau}$, we have $l^+_{|s|} = \lfloor \frac{|s|}{\tau} \rfloor$. As records with different lengths have different segmentation strategies, we should consider every length $l \in [l^-_{|s|}, l^+_{|s|}]$ for s.

Probing Segment Signatures for Length l. As the record with length l is split into η_l segments, we also split s to η_l segments $seg_1, seg_2, \cdots, seg_{\eta_l}$ (using the same global order, e.g. the same hash function Γ_l). For $i \in [1, \eta_l]$, we generate a probing segment signature $\texttt{pSig}^+_{s,i,l} = (seg_i, i, l)$.

Probing Deletion Signatures for Length l. For each probing segment signature $\texttt{pSig}^+_{s,i,l} = (seg_i, i, l)$, we generate a deletion signature $\texttt{pSig}^-_{s,i,l,j} = (del^j_i, i, l)$ where del^j_i is a subset of sig_i by deleting the j-th token. Then we can get a probing deletion signature set $\texttt{pSig}^-_{s,i,l} = \cup_{j=1}^{\eta_l}\{\texttt{pSig}^-_{s,i,l,j}\}$.

Signature Selection. We select the deletion or segment signatures for s. Suppose we select x probing segment signatures and y probing deletion signatures of s such that $x + 2y \geq \theta_{|s|,|r|}$. The records on the inverted lists of these signatures are candidates of s; other records are pruned as they have at least $\theta_{|s|,|r|}$ mismatched tokens with s. We present how to select balance-aware signatures based on the workload in Sections 5 and 6.

EXAMPLE 1. *Figure 2 presents an example with a data record $r_1 = \{a,b,c,d,e\}$ and a query record $s_1 = \{a,b,c,d,e,f\}$. Suppose $\tau = 0.8$. We can get $\eta_{|r_1|} = \lfloor \frac{1-0.8}{0.8} * 5 \rfloor + 1 = 2$, so we split r_1 into two segments. The two indexing segment signatures of r_1 are $\texttt{iSig}^+_{r_1,1,5} = \{((a,c,e),1,5)\}$ and $\texttt{iSig}^+_{r_1,2,5} = \{((b,d),2,5)\}$. Their indexing deletion signatures are acquired by removing one token from their segment signatures, i.e., $\texttt{iSig}^-_{r_1,1,5}$=$\{((a,c),1,5),((a,e),1,5),((c,e),1,5)\}$ and $\texttt{iSig}^-_{r_1,2,5}$=$\{(b,2,5),(d,2,5)\}$.*

For record s_1, we need to generate its probing signatures. First, we need to get the max and min length of records that can match s_1, i.e. $l^+_{|s_1|} = \lfloor \frac{6}{0.8} \rfloor$=7 and $l^-_{|s_1|} = \lceil 6 * 0.8 \rceil$=5. Since $|r_1| = 5$, which is within $[5,7]$, r_1 can be a candidate similar to s_1, so l=5 and $\eta_5 = \lfloor \frac{1-0.8}{0.8} * 6 \rfloor + 1 = 2$. We split s_1 to two probing segment signatures: $\texttt{pSig}^+_{s_1,1,5} = \{((a,c,e),1,5)\}$, $\texttt{pSig}^+_{s_1,2,5} = \{((b,d,f),2,5)\}$. Their probing deletion signatures are $\texttt{pSig}^-_{s_1,1,5} = \{((a,c),1,5),((a,e),1,5),((c,e),1,5)\}$, $\texttt{pSig}^-_{s_1,2,5} = \{((b,d),2,5),(b,f),2,5),(d,f),2,5)\}$. $\theta_{|s_1|,|r_1|} = \lfloor \frac{1-0.8}{1+0.8}(5+6) \rfloor + 1 = 2$, which indicates that r_1 and s_1 have at most 2 mismatched tokens.

If we do not choose signatures from the second segment, we get that the second segments of s_1 and r_1 match. If we choose $\texttt{pSig}^+_{s_1,2,5}$ as the segment signature, we get that the

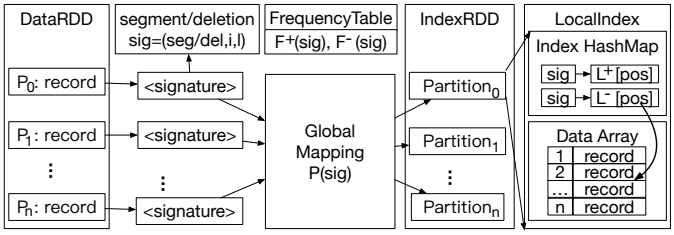


Figure 3: Indexing Structure.

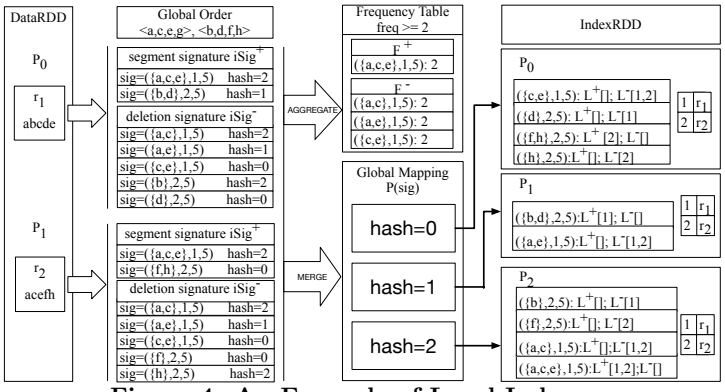


Figure 4: An Example of Local Index.

upper bound of the second segment mismatching is 1. If we choose $\texttt{pSig}^-_{s_1,2,5}$ *as the deletion signature, we get that the upper bound of the second segment mismatching is 2. Thus selecting different signatures has different pruning power.*

4.2 Distributed Indexing

Given a dataset $\mathcal{R}$, we build a global index and a local index. Figure 3 shows the index structure. How to utilize the indexes to process a query is presented in Section 5-7.

Indexing. Note that different queries may have different thresholds and we aim to support queries with any choice of threshold. To achieve this goal, we utilize a threshold bound to generate the index. For example, the threshold bound for Jaccard is the smallest threshold for all queries that the system can support, e.g., 0.6. Using this threshold bound, we can select the indexing segment/deletion signatures and build a *local index*. In addition, we also keep the *frequency table* of each signature to keep each signature's frequency and build a *global index* that keeps a mapping from the signature to the partitions that contain this signature.

Frequency Table. For each RDD $\mathcal{R}_i$, for each record $r \in \mathcal{R}_i$, we compute its indexing segment number $\eta_{|r|}$ using the threshold bound, then generate the indexing segment signature set and the indexing deletion signature set. For each segment signature g, we collect its global frequency $\mathcal{F}^+[g]$ and for each deletion signature g', we collect its global frequency $\mathcal{F}^-[g']$. We use the frequency to select signatures in Section 5. Note that the frequency table is very small and can be easily distributed to every node.

Local Index. Next we shuffle the indexing signatures such that (1) each signature and its inverted list of records that contain this signature are shuffled to one and only one partition, i.e., the same signature will be in the same partition and (2) the same partition may contain multiple signatures and their corresponding records. For each partition, we construct an IndexRDD $\mathcal{I}_i^{\mathcal{R}}$ for indexing signatures in this partition. Each IndexRDD $\mathcal{I}_i^{\mathcal{R}}$ contains several signatures and the corresponding records, which include two parts. The first part is a hash-map which keeps the mapping from a signature to two lists of records: $\mathcal{L}^+[g]$ keeps the records whose indexing segment signatures contain g and $\mathcal{L}^-[g]$ keeps the records whose indexing deletion signatures contain g. We use $\mathcal{L}[g]$ to denote $\mathcal{L}^+[g] \cup \mathcal{L}^-[g]$. The second part is all the records in this RDD, i.e., $\mathcal{D}_i = \cup_{g \in \mathcal{I}_i^{\mathcal{R}}} \mathcal{L}[g]$. Note that the records are stored in the data list $\mathcal{D}_i$ while $\mathcal{L}^+[g]$ and $\mathcal{L}^-[g]$ only keep a list of pointers to the data list $\mathcal{D}_i$. For example, Figure 4 shows the local index for two records.

Global Index. For each signature, we keep the mapping from the signature to the partitions that contain this signature. We only maintain a global function $\mathcal{P}$ that maps a signature g to a partition p, i.e., $\mathcal{P}(g) = p$. Thus the global index is rather small.

5. SIMILARITY SELECTION OPERATION

Algorithm Overview. Given a query s, Algorithm 1 shows how to utilize the proposed indexes to answer the query in three steps. (1) It first conducts a global search by utilizing the frequency table to select the probing signatures of s. Specifically, we propose an optimal signature selection method to achieve a balance-aware selection. (2) For each selected probing signature, it utilizes the global hash function to compute the partition that contains the signature and sends the search request to the corresponding partition (lines 3-13). (3) Each partition exploits a local search to retrieve the inverted lists of the probing signatures and verify the records on the inverted lists to get local answers (lines 14-19). Finally, it returns local answers to the master.

5.1 Probing Signature Selection

Given a query s we compute the maximal length l_{max} and minimal length l_{min} of records that are similar to s (see Table 1). Then for each length l in this range, we generate the probing signatures of s. We first compute the number of segments η_l for length l. For each segment at position $i \in [1, \eta_l]$, we generate a probing segment (deletion) signature $\texttt{pSig}^+_{s,i,l}$ ($\texttt{pSig}^-_{s,i,l}$). Then we compute the bound of mismatched tokens $\theta_{|s|,l}$ between s and a record r of length l, above which r could not be similar to s.

Next we discuss how to select the probing signatures. Let Z denote a list where each element $Z[i] \in \{0, 1, 2\}$ for $1 \leq i \leq \eta_l$. $Z[i] = 0$ denotes that the probing signature on the i-th segment is not selected. $Z[i] = 1$ denotes that the probing *segment* signature on the i-th segment is selected. $Z[i] = 2$ denotes that the probing *deletion* signature on the i-th segment is selected.

If we select the probing segment signature in $\texttt{pSig}^+_{s,i,l}$, it can only match the indexing segment signature, and the candidate size is $\sum_{g \in \texttt{pSig}^+_{s,i,l}} \mathcal{F}^+[g]$, which is the total size of the inverted lists of segment signatures in $\texttt{pSig}^+_{s,i,l}$. If there is no matching signature, there exists at least 1 mismatched token. If we select the probing deletion signature in $\texttt{pSig}^-_{s,i,l}$, its probing deletion signature can match the indexing segment signature and its probing segment signature can match the indexing deletion/segment signature. Based on case (ii) in Section 4.1, the candidate size is $\sum_{g \in \texttt{pSig}^-_{s,i,l}} \mathcal{F}^+[g] + \sum_{g \in \texttt{pSig}^+_{s,i,l}} (\mathcal{F}^-[g] + \mathcal{F}^+[g])$. If there is no matching signature, there are at least 2 mismatched tokens. The verification cost on different records is nearly the same, and we utilize the candidate size to quantify efficiency. We select x segment signatures and y deletion signatures of s such that $x + 2y \geq \theta_{|s|,l}$. Intuitively, we select signatures

Algorithm 1: Dima-SimilaritySelection

Input: Dataset $\mathcal{R}$; Query s with threshold τ
Output: Answer set $\mathcal{A}$

Build index for $\mathcal{R}$ offline;
// Global Search
for $l \in [l^-_{|s|}, l^+_{|s|}]$ **do**
  Z=OptimalSignatureSelection(s, l);
  for *each* $Z_i \neq 0$ **do**
    if $Z_i = 1$ **then**
      for $g^+ \in \mathtt{pSig}^+_{q,i,l}$ **do**
        $\mathcal{P}(g^+)$.LocalSearch($s, \tau, \mathcal{L}^+, g^+$);
    if $Z_i = 2$ **then**
      for $g^- \in \mathtt{pSig}^-_{q,i,l}$ **do**
        $\mathcal{P}(g^-)$.LocalSearch($s, \tau, \mathcal{L}^+, g^-$);
      for $g^+ \in \mathtt{pSig}^+_{q,i,l}$ **do**
        $\mathcal{P}(g^+)$.LocalSearch($s, \tau, \mathcal{L}^+, g^+$);
        $\mathcal{P}(g^+)$.LocalSearch($s, \tau, \mathcal{L}^-, g^+$);
// LocalSearch Method
for g^+ *has a* $\mathcal{L}^+$ *request in local search* **do**
  for $r \in \mathcal{L}^+[g^+]$ **do**
    if *Verify*(r, s)=*true* **then** $\mathcal{A} = \mathcal{A} \cup \{r\}$;
if g^- *has a* $\mathcal{L}^-$ *request in local search* **then**
  for $r \in \mathcal{L}^-[g^-]$ **do**
    if *Verify*(r, s)=*true* **then** $\mathcal{A} = \mathcal{A} \cup \{r\}$;
return $\mathcal{A}$;

Function OptimalSignatureSelection

Input: Query s, length l, Frequency Tables $\mathcal{F}^-$,$\mathcal{F}^+$
Output: Selection Vector Z

for $i \in [0, \eta_l]$ **do** $M[i][0] = 0^{|P|}$;
for $j \in [1, \theta_{|s|,l}]$ **do** $M[0][j] = \infty^{|P|}$;
for $i \in [1, \eta_l]$ **do**
  for $j \in [1, \theta_{|s|,l}]$ **do** Compute $M[i][j]$; Set $Z[i]$;
return Z;

of s to minimize the number of candidates, i.e., minimizing

$$\sum_{i=1}^{\eta_l}(b_i \sum_{g\in \mathtt{pSig}^+_{s,l}} \mathcal{F}^+[g] + c_i(\sum_{g\in \mathtt{pSig}^-_{s,l}} \mathcal{F}^+[g] + \sum_{g\in \mathtt{pSig}^+_{s,l}} (\mathcal{F}^-[g] + \mathcal{F}^+[g])))$$

$$b_i = \begin{cases} 1 & Z[i] = 1 \\ 0 & Z[i] \neq 1 \end{cases} \quad c_i = \begin{cases} 1 & Z[i] = 2 \\ 0 & Z[i] \neq 2 \end{cases} \quad s.t. \sum_{i=1}^{\eta_l} Z[i] \geq \theta_{|s|,l}.$$

5.2 Balance-Aware Signature Selection

In order to balance the workload among different partitions, we propose a balance-aware probing signature selection method.

5.2.1 Problem Formulation

Given $|\mathcal{P}|$ partitions, let $\mathcal{W}_j$ denote the workload on the j-th partition, we want to minimize the maximal workload on every partition, i.e., *minimize* $\max(\mathcal{W}_1, \mathcal{W}_2, \cdots \mathcal{W}_{|\mathcal{P}|})$, where $\mathcal{W}_j$ is computed as below:

$$\mathcal{W}_j = \sum_{i=1}^{\eta_l}\Big(b_i \sum_{g\in \mathtt{pSig}^+_{s,i,l}\&\mathcal{P}(g)=j} \mathcal{F}^-[g] + c_i \sum_{g\in \mathtt{pSig}^-_{s,i,l}\&\mathcal{P}(g)=j} (\mathcal{F}^+[g] + \sum_{g\in \mathtt{pSig}^+_{s,i,l}\&\mathcal{P}(g)=j} \mathcal{F}^-[g] + \mathcal{F}^+[g])\Big)$$

$$b_i = \begin{cases} 1 & Z[i] = 1 \\ 0 & Z[i] \neq 1 \end{cases} \quad c_i = \begin{cases} 1 & Z[i] = 2 \\ 0 & Z[i] \neq 2 \end{cases} \quad s.t. \sum_{i=1}^{\eta_l} Z[i] \geq \theta_{|s|,l}.$$

where $\mathcal{W}_j$ can be computed by aggregating the size of the inverted list of each selected signature in the j-th partition. We can utilize the global search function $\mathcal{P}$ to efficiently check whether a signature is in the j-th partition.

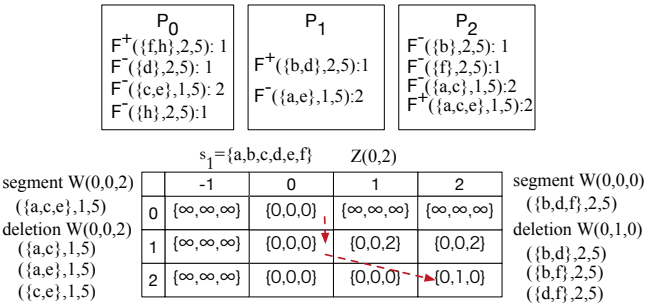


Figure 5: Example of Optimal Signature Selection.

A naive method enumerates every possible case and selects the case with a balanced workload. As each segment has 3 cases, this method has a time complexity of 3^{η_l}. To improve the performance, we propose a dynamic-programming algorithm to select the best probing signatures to minimize the maximal workload.

5.2.2 Optimal Signature Selection

To minimize the maximal workload of $|\mathcal{P}|$ partitions, we devise a dynamic programming algorithm to select the probing signatures. Let M denote a matrix with η_l columns and $\theta_{|s|,l}$ rows. Each cell $M[i][j]$ is a vector $\mathcal{W} = [\mathcal{W}_1, \mathcal{W}_2, \cdots, \mathcal{W}_{|\mathcal{P}|}]$ which denotes the optimal workload by selecting probing signatures in the first j segments with threshold i. Then we discuss how to compute $M[i][j]$ based on cells $M[i'][j']$ where $i' \leq i$ and $j' \leq j$. There are three cases.

Case 1: If we do not select any signature for the i-th segment, then $M[i][j] = M[i-1][j]$.

Case 2: If we select the probing segment signature for the i-th segment, then $M[i][j] = M[i-1][j-1] + \Delta_S[i]$, where Δ_S is the vector of the increased workload on each partition by selecting the probing segment signatures. $\Delta_S[i] = \sum_{g\in \mathtt{pSig}^+_{s,i,l}\&\mathcal{P}(g)=i} \mathcal{F}^+[g]$.

Case 3: If we select the probing deletion signature for the i-th segment, then $M[i][j] = M[i-1][j-2] + \Delta_D[i]$, where Δ_D is the vector of the increased workload on each partition by selecting the probing deletion signatures. $\Delta_D[i] = \sum_{g\in \mathtt{pSig}^-_{s,i,l}\&\mathcal{P}(g)=i} \mathcal{F}^+[g] + \sum_{g\in \mathtt{pSig}^+_{s,i,l}\&\mathcal{P}(g)=i} \mathcal{F}^-[g] + \mathcal{F}^+[g]$.

Among the three cases, we select the case whose maximal value in the vector is minimal, i.e.,

$$M[i][j] = \min \begin{cases} M[i-1][j] \\ M[i-1][j-1] + \Delta_S[i] \\ M[i-1][j-2] + \Delta_D[i] \end{cases}$$

Example 2. *Consider* $s_1 = \{a, b, c, d, e, f\}$ *of length* l *=5. We get* $\eta_5 = 2$ *and* $\theta_{|s_1|,5} = 2$. s_1 *has two segments and the mismatched bound is 2. The frequency in each partition is shown in Figure 5. The workload distribution is initialized as* $\{0, 0, 0\}$. $M[1][1] = \{0, 0, 2\}$ *which means, if we select segment signature of the first segment, the workload is* $(0, 0, 2)$. $M[2][1] = \{0, 0, 0\}$ *which means, if we select segment signature of the second segment, the workload is* $(0, 0, 0)$. $M[2][2] = \{0, 1, 0\}$ *which means, if we select deletion signature of the 2nd segment, the workload is* $(0, 1, 0)$.

Time Complexity. The time complexity of the dynamic programming algorithm is $\eta_l \cdot \theta_{|s|,l} \cdot |\mathcal{P}|$.

Consider the selected signature vector Z. If $Z[i] = 0$, we do not send a search request to any partition. If $Z[i] = 1$, for

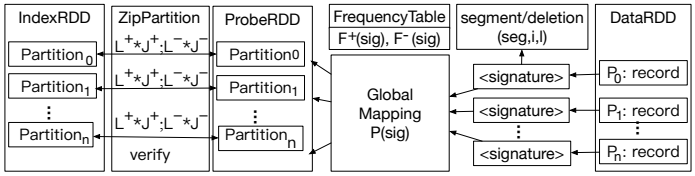


Figure 6: Similarity Join Workflow.

each signature $g^+ \in \texttt{iSig}^+_{q,i,l}$, we send a segment search request to partition $\mathcal{P}(g^+)$, which takes the records on $\mathcal{L}^+[g^+]$ as candidates and verifies them. If $Z[i] = 2$, for each signature $g^- \in \texttt{iSig}^-_{q,i,l}$, we send a segment search request to partition $\mathcal{P}(g^-)$, which takes the records on $\mathcal{L}^+[g^-]$ as candidates and verifies them; and for each signature $g^+ \in \texttt{iSig}^+_{q,i,l}$, we send a segment search request and a deletion search request to partition $\mathcal{P}(g^+)$, which takes the records on $\mathcal{L}^+[g^+]$ and $\mathcal{L}^-[g^+]$ as candidates and verifies them.

5.3 Local Search

For each selected probing signature, it locates the corresponding indexing signature and retrieves the inverted list. Each record on the inverted list is a candidate which needs to be verified. Since two candidates may have multiple matching signatures, this method may involve duplicate verification. To address this issue, for each candidate, we check whether this signature is the first matching. If yes, we verify it; otherwise, we ignore this pair. To check whether this signature is the first matching, we generate the indexing signature of r and the probing signature of s before this signature, and check whether there is a match between them. We can also add the probing signature before this signature and send it to local executor. So we can avoid on-the-fly generating signatures for candidates.

6. SIMILARITY-BASED JOIN OPERATION

For similarity join on two sets $\mathcal{R}$ and $\mathcal{S}$, a straightforward approach is to first build the index for a set, e.g., $\mathcal{R}$, then take each record $s \in \mathcal{S}$ as a query and invoke the similarity selection algorithm to compute its results. However, it is rather expensive for the driver, because it is costly to select signatures for a huge number of queries. To address this problem, we propose an effective join algorithm.

Algorithm Overview. Algorithm 2 shows the pseudo code. (1) It generates signatures and builds the indexRDD for one dataset, say $\mathcal{R}$ (line 1). (2) It selects probing signatures for each length l using a greedy algorithm (line 3-4). (3) For each selected signature it builds the probeRDD for the other dataset, say $\mathcal{S}$ (lines 5-14). Since the matched probing and indexing signatures are in the same executor, it avoids data transmission among partitions. (4) It computes the local results in each executor based on the indexRDD and probeRDD, and the master collects the results from local executors (lines 15-20). Figure 6 shows the workflow.

6.1 Indexing

Given two datasets $\mathcal{R}$ and $\mathcal{S}$, we first select a dataset to index based on the cost estimation techniques in Section 8.1. Without loss of generality, we select $\mathcal{R}$ to index using the method in Section 4. For the dataset $\mathcal{S}$, for each record $s \in \mathcal{S}$, we select its probing signatures based on the frequency tables $\mathcal{F}^+$ and $\mathcal{F}^-$. After generating the probing signatures, we build probeRDD. A straightforward method is to generate the probeRDD randomly. This will lead to the

Algorithm 2: Dima-SimilarityJoin

Input: Two datasets $\mathcal{R}$, $\mathcal{S}$, threshold τ
Output: Answer set $\mathcal{A}$
// Global Join
1 Build Index for $\mathcal{R}$;
2 **for** $s \in \mathcal{S}$ **do**
3 **for** $l \in [l^-_{|s|}, l^+_{|s|}]$ **do**
4 Z=GreedySignatureSelection(s, l);
5 **for** $Z_i \neq 0$ **do**
6 **if** $Z_i = 1$ **then**
7 **for** $g^+ \in \texttt{pSig}^+_{s,i,l}$ **do**
8 Shuffle g^+ and $\mathcal{J}^+[g^+] \leftarrow s$;
9 **if** $Z_i = 2$ **then**
10 **for** $g^- \in \texttt{pSig}^-_{s,i,l}$ **do**
11 Shuffle g^- and $\mathcal{J}^+[g^-] \leftarrow s$;
12 **for** $g^+ \in \texttt{pSig}^+_{s,i,l}$ **do**
13 Shuffle g^+ and $\mathcal{J}^+[g^+] \leftarrow s$;
14 Shuffle g^+ and $\mathcal{J}^-[g^+] \leftarrow s$;
// Local Join
15 **for** *g in each partition* **do**
16 **for** $(r, s) \in \mathcal{L}^+[g] \times \mathcal{J}^+[g]$ **do**
17 **if** *Verify*(r, s)=*true* **then** $\mathcal{A} = \mathcal{A} \cup \{(r, s)\}$;
18 **for** $(r, s) \in \mathcal{L}^-[g] \times \mathcal{J}^-[g]$ **do**
19 **if** *Verify*(r, s)=*true* **then** $\mathcal{A} = \mathcal{A} \cup \{(r, s)\}$;
20 **return** $\mathcal{A}$;

Function GreedySignatureSelection

Input: Query s, length l, Frequency Tables $\mathcal{F}^-$,$\mathcal{F}^+$
Output: Selection Vector Z
1 **for** $i \in [1, \eta_l]$ **do**
2 Compute $\mathcal{W}^{+i}$; Insert $(i, \max \mathcal{W}^{+i})$ into MinHeap M;
3 **for** $x \in [1, \eta_l]$ **do**
4 Pop min element $(i, \max \mathcal{W}^{+i})$ from M ;
5 **if** $Z[i] = 0$ **then** $Z[i]$=1; Insert$(i, \max \mathcal{W}^{-i})$ into M;
6 **else** $Z[i] = 2$;
7 **return** Z;

case that the matched probing signatures and indexing signatures may be in different partitions and thus involve huge data transmission cost. To alleviate this problem, we want to guarantee that the same probing/indexing signatures are always in the same executor. To achieve this goal, we utilize the same global hash function $\mathcal{P}$ as the indexRDD, and the same signature will be partitioned into the same partition. We can utilize the zip-partition to achieve this goal.

6.2 Balance-Aware Signature Selection

6.2.1 Problem Formulation

For a selection query, we can utilize the dynamic-programming algorithm (proposed in Section 5.2) to minimize the maximal workload. However for the join query, there are many records in the dataset, and balancing a record cannot guarantee workload balance for all records. Thus we want to select a balance-aware probing signature to minimize the overall maximal workload on every partition, i.e., *minmize* $\max(\mathcal{W}_1, \mathcal{W}_2, \cdots, \mathcal{W}_{|\mathcal{P}|})$, where $\mathcal{W}_j$ is computed as below:

$$\mathcal{W}_j = \sum_{s\in\mathcal{S}}\Big(\sum_{i=1}^{\eta_{|s|}}\big(b_i^s \sum_{g\in \texttt{pSig}^+_{s,i,|s|}\&\mathcal{P}(g)=j} \mathcal{F}^+[g] + c_i^s\big(\sum_{g\in \texttt{pSig}^-_{s,i,|s|}\&\mathcal{P}(g)=j} \mathcal{F}^+[g] + \sum_{g\in \texttt{pSig}^+_{s,i,|s|}\&\mathcal{P}(g)=j} (\mathcal{F}^+[g] + \mathcal{F}^-[g])\big)\big)\Big)$$

$$b_i^s = \begin{cases} 1 & Z^s[i] = 1 \\ 0 & Z^s[i] \neq 1 \end{cases} \quad c_i^s = \begin{cases} 1 & Z^s[i] = 2 \\ 0 & Z^s[i] \neq 2 \end{cases} \quad s.t. \sum_{i=1}^{\eta_l} Z^s[i] \geq \theta_{|s|,l}.$$

where $\mathcal{W}_j$ can be computed by aggregating the cost of signatures in the j-th partition. We can utilize the global function to efficiently check whether a signature is in the j-th partition. Here the parameters have a superscript as compared to those in the selection operation (in Section 5.2).

We prove that the balance-aware signature selection for the join query is NP-hard, because selecting a record's signatures can affect other records' workload. Even if we balance the workload for s, the balance can be broken by record s'.

THEOREM 1. *The balance-aware probing signature selection problem is NP-complete.*

6.2.2 A Greedy Algorithm for Workload Balancing

We propose a greedy algorithm to solve the balance-aware signature selection problem. We process each record s in $\mathcal{S}$ separately. For each record s, we can select its optimal probing signatures. However, the selection process is costly especially for a large number of records in $\mathcal{S}$. To avoid this issue, we propose a greedy algorithm.

Suppose the current workload is $\mathcal{W}$. For each of its i-th segment of record s, we compute the workload if we select the i-th probing segment signature, denoted by $\mathcal{W}^{+i}$, where

$$\mathcal{W}_j^{+i} = \mathcal{W}_j + \sum_{g \in \mathtt{pSig}^+_{s,i,l} \& \mathcal{P}(g)=j} \mathcal{F}^+[g]$$

Thus for each probing segment signature, we aim to select the i-th segment signature such that $i = \arg\min_i \max\{\mathcal{W}^{i+}\}$ for $i \in [1, \eta_l]$, where $\max\{\mathcal{W}^{+i}\} = \max\{\mathcal{W}_1^{+i}, \mathcal{W}_2^{+i}, \cdots, \mathcal{W}_{|\mathcal{P}|}^{+i}\}$.

If we select the i-th segment signature, we need to consider whether we replace the segment signature with the i-th deletion signature. Thus for each of its i-th segment, we compute the workload if we select the i-th probing deletion signature, $\mathcal{W}^{-i}$, where

$$\mathcal{W}_j^{-i} = \mathcal{W}_j + \sum_{g \in \mathtt{pSig}^+_{s,i,l} \& \mathcal{P}(g)=j} \mathcal{F}^-[g] + \sum_{g \in \mathtt{pSig}^-_{s,i,l} \& \mathcal{P}(g)=j} \mathcal{F}^+[g]$$

Here we do not add $\sum_{g \in \mathtt{pSig}^+_{s,i,l} \& \mathcal{P}(g)=j} \mathcal{F}^+[g]$, because we have added it when selecting the segment signature. Next we give the greedy signature selection algorithm. We maintain a min-heap. Initially, we compute $\{\mathcal{W}^{+i}\}$ and insert $\max\{\mathcal{W}^{+i}\}$ into the heap. Next we pop the element with the minimal value, e.g., the i-th segment. If $Z[i] = 0$, we set $Z[i] = 1$, i.e., select the i-th segment signature, and we compute $\mathcal{W}^{-i}$ and insert $\max\{\mathcal{W}^{-i}\}$ into the heap. If $Z[i] = 1$, we set $Z[i] = 2$, i.e., select the i-th deletion signature. After $\eta_{s,l}$ times, the algorithm terminates.

Time Complexity. The time complexity of the greedy algorithm is $\theta_{|s|,l} \cdot |\mathcal{P}| \cdot \log \eta_l$.

EXAMPLE 3. *Let us consider the workloads in Figure 7. We want to select the signatures of $s_1 = \{a, b, c, d, e, f\}$. We first compute the workload of each segment signature and get $\mathcal{W}^{+1} = (0, 0, 2)$ and $\mathcal{W}^{+2} = (0, 0, 0)$. We pop the minimal value $\mathcal{W}^{+2}$ and set $Z = (0, 1)$. Next we push $\mathcal{W}^{-2} = (0, 1, 0)$ into the min-heap. Then we pop the minimal value $\mathcal{W}^{-2} = (0, 1, 0)$ and get $Z = (0, 2)$. Thus we select the probing deletion signature of the second segment.*

6.3 Building ProbeRDD

After selecting the probing signatures, we shuffle them and build a probeRDD. For record s and length l, if $Z[i] = 1$, we select its segment signature and for each signature g, we insert s into local index $\mathcal{J}^+[g]$. If $Z[i] = 2$, we select its deletion signature and for each deletion signature g^-, we insert s into local index $\mathcal{J}^+[g^-]$ and for each segment signature g^+, we insert s into local index $\mathcal{J}^+[g^+]$ and $\mathcal{J}^-[g^+]$.

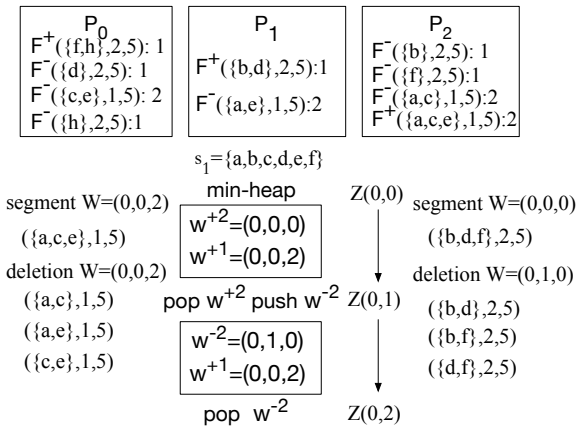


Figure 7: Running Example of Greedy Selection.

6.4 Local Join

In the same executor, for the same signature g, we have two lists of records $\mathcal{L}^+[g], \mathcal{L}^-[g]$ for $\mathcal{R}$ and two lists of records $\mathcal{J}^+[g], \mathcal{J}^-[g]$ for $\mathcal{S}$. Each pair $(r, s) \in \mathcal{L}^+[g] \times \mathcal{J}^+[g]$ and $(r, s) \in \mathcal{L}^-[g] \times \mathcal{J}^-[g]$ is a candidate, and we need to verify it. Since two candidates may have multiple matching signatures, we want to remove the duplicate verification. To address this issue, for each candidate pair, we check whether this signature is the first matching. If yes, we verify it; otherwise, we ignore the pair. If $\mathcal{L}^+[g] \times \mathcal{L}^-[g]$ is too large, we will partition $\mathcal{L}^-[g]$ into multiple parts $\mathcal{L}_1^-[g], \mathcal{L}_2^-[g], \cdots, \mathcal{L}_x^-[g]$ and distribute $\mathcal{L}^+[g] \times \mathcal{L}_i^-[g]$ to the node with less workload.

7. TOP-K SELECTION AND JOIN

7.1 Top-k Selection

Basic Idea. For similarity selection, we use a given threshold to generate segments and deletions as signatures. Top-k selection, however, has no threshold. To address this problem, we propose a progressive method to compute top-k results. We first generate a signature, use the signature to identify some candidates and put k best candidates in a priority queue $\mathcal{Q}$. We use τ_k to denote the minimal similarity among the candidates in $\mathcal{Q}$. Then we estimate an upper bound $\mathtt{ub}$ for other unseen records. If $\tau_k \geq \mathtt{ub}$, we can guarantee that the candidates in $\mathcal{Q}$ are the final results and the algorithm can terminate. If $\tau_k < \mathtt{ub}$, we generate next signatures, compute candidates and update the priority queue and τ_k. Next we discuss how to progressively identify candidates and how to estimate $\mathtt{ub}$.

Indexing. We aim to first identify the candidates with the largest similarity and add them into $\mathcal{Q}$ in order to get a larger τ_k. To achieve this goal, we first split record r into two segments, and take the first one as the first signature. Then we further split the second segment of r into two subsegments and take the first sub-segment as the second signature. Iteratively we can generate the signature of r. For example, for $r_1 = \{a, c, e, b, d\}$, its segments are $\{a, c, e\}$, $\{b\}$, and $\{d\}$. Note for different records, we need to use the same strategy to generate the signatures. Thus we first collect all the tokens and split them into two sets. If a token is in the first, we map it into the first signature. Then we split the second set into two sub-sets, and we terminate if

there is only one token. For a record with length l, it has $\log l$ segments. Similar to similarity selection and join, we build global and local indexes for the signatures.

Computing Candidates. Given a query s, we use the same method to generate its first signature g_1. We utilize the global index to get the relevant partitions. For each relevant partition, we use the local inverted index to get candidates $\mathcal{L}^+[g_1]$ and send top-k local candidates to the master. The master collects all the local candidates and puts them into the priority queue and compute τ_k.

Computing the Upper Bound ub. From the first segment, we can also estimate an upper bound of the similarities of other records to the query. Since other records do not share the same first signature with s, they have at least one mismatch token with s. If $|s| \geq |r|$, we have $|r \cap s| \leq |r|$ and $|r \cup s| \geq |r|+1$. We also have $|r \cap s| \leq |s|-1$ and $|r \cup s| \geq |s|$. Thus we have $\frac{|r \cap s|}{|r \cup s|} \leq \min(\frac{|s|-1}{|s|}, \frac{|r|}{|r|+1})$. If $|s| \leq |r|$, we have $\frac{|r \cap s|}{|r \cup s|} \leq \min(\frac{|r|-1}{|r|}, \frac{|s|}{|s|+1})$. So when we decide whether to access the i-th segment, we set ub for the second segment as $\mathtt{ub}_2 = \frac{|s|}{|s|+1}$. If $\tau_k \geq \mathtt{ub}_2$, we do not need to visit the second segment. Similarly, when we decide whether to access the i-th segment, $\mathtt{ub}_i = \frac{|s|}{|s|+i-1}$. If $\tau_k \geq \mathtt{ub}_i$, we do not need to visit the i-th segment and the algorithm terminates; otherwise, we access the i-th segment, retrieve the candidates and update the priority queue and τ_k.

Balance-Aware Method. We can also generate and index deletion signatures. When we select signatures to generate the candidates, we either select the segment signatures or the deletion signatures. We use the techniques in Section 5.2 to select the better signatures to balance the workload, and our above techniques still work for selectable signatures.

7.2 Top-k Join

It is rather expensive to generate all the signatures for all data. Instead, we only need to generate the first signatures, estimate a bound based on current results and utilize the bound to decide whether generating the next signatures. In other words, we progressively generate the signatures as follows. We first generate the first signatures of the two datasets and use zip-partition to shuffle the same signature into the same partition. In each partition, we compute the candidates. Then the master collects all the candidates, puts the candidates into the priority queue, and computes τ_k. Next for each record, we decide whether to generate its second signatures or not based on the upper bound ub. If $\tau_k \geq \mathtt{ub}$, we do not generate its signatures; otherwise, we generate its second signature. If we do not need to generate the next signatures for all records, the algorithm finds the top-k pairs and terminates.

8. COST-BASED QUERY OPTIMIZATIONS

In this section, we present the cost-based query optimizations. We first introduce the query estimation techniques (Section 8.1) and then discuss the parameter optimization techniques (Section 8.2).

8.1 Cost Estimation

A SQL query may contain multiple operations, so it is important to estimate the cost of each operation and thereby the query engine can utilize the cost to select a query plan, e.g., join order. Since Spark SQL has the cost model for exact selection and join, we focus on estimating the cost for similarity operations. If there are multiple join predicates, we also need to estimate the result size.

Cost/Size Estimation for Similarity Selection. Given a similarity selection operation, we first select its probing signatures and then we can estimate its candidate number by the sum of frequency of its probing signature, i.e., $\mathcal{C}^f = \sum_{g \in s} \mathcal{F}[g]$, where g is a selected signature. The cost of verification can also be estimated. The cost of verifying a pair is $\mathcal{C}^v = |s|$. Thus the estimation cost of a selection query is $\mathcal{C}^v\mathcal{C}^f$. Similarly, we can estimate its result size by $\mathcal{N} = \alpha \sum_{g \in s} \mathcal{F}[g]$, where α is the ratio of the result size to the candidate size, which we can get using a sample.

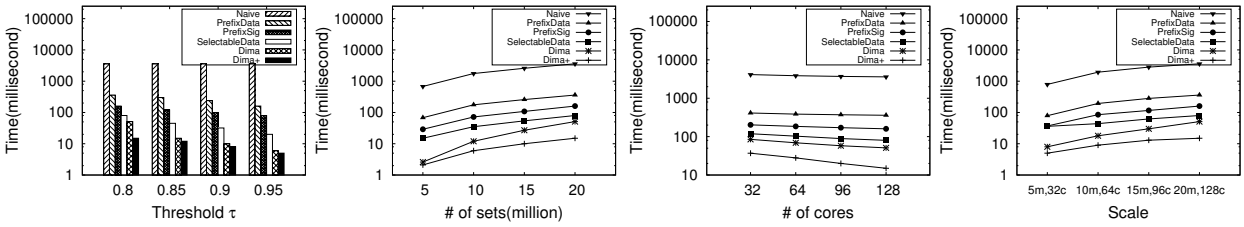
Cost/Size Estimation for Similarity Join. There are two methods to estimate the similarity join cost. First, if there is an index on a dataset, we can utilize the estimation method for similarity selection query to estimate the join cost. Specifically, we sample some records from another dataset, compute the cost of these samples, and then estimate the cost of the join query based on these selected samples. The cost is $\mathcal{C}^v\mathcal{C}^f\beta$ where β is the ratio of dataset size to the sample size. Similarly, we can estimate its result size by $\mathcal{N} = \alpha\beta \sum_{g \in q} \mathcal{F}[g]$. Second, if there is no index, we directly sample some records and estimate the cost and size.

8.2 Parameter Optimization

There are two parameters in our system. The first is the number of partitions and the second is the global order of elements (e.g., tokens) to make the same signature shuffle to the same partition.

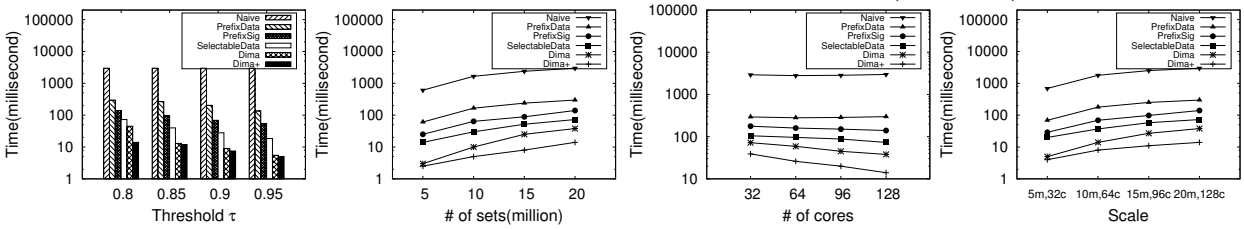
Number of Partitions. Increasing the number of partitions can increase the parallel scale but it also increases the signature selection time (and incurs system overhead for more partitions). We first consider the similarity selection. The time complexity of our dynamic-programming algorithm for signature selection is $\mathcal{O}(\theta_{|s|,l}\eta_l|\mathcal{P}|)$, which is approximately $\mathcal{O}(l^2|\mathcal{P}|)$. The total size of the inverted list is $\mathcal{O}(\sum |s|)$. Suppose the pruning power of the signature-based method is λ, i.e., $\mathcal{O}(\frac{\sum |s|}{\lambda})$ records will be taken as candidates. Then the expected selection cost in parallel is $\mathcal{O}(\frac{\sum |s|}{\lambda|\mathcal{P}|})$. To achieve the best performance for selection, we should set $|\mathcal{P}|$ as $\mathcal{O}(\sqrt{\frac{\sum |s|}{\lambda l^2}})$. Next we consider the similarity join. The time complexity of our greedy algorithm for signature selection is $\mathcal{O}(\theta_{|s|,l} \log \eta_l|\mathcal{P}|)$, which is approximately $\mathcal{O}(l \log |\mathcal{P}|)$. Suppose the pruning power of the signature-based method is λ^+, i.e., $\mathcal{O}(\frac{\sum |s|}{\lambda^+} \frac{\sum |r|}{\lambda^+})$ records will be taken as candidates. Then the expected selection cost in parallel is $\mathcal{O}(\frac{\sum |s|}{\lambda^+} \frac{\sum |r|}{\lambda^+|\mathcal{P}|})$. To achieve the best performance for join, we should set $|\mathcal{P}|$ as $\mathcal{O}(\sqrt{\frac{\sum |s| \sum |r|}{\lambda^{+3} l^2}})$.

Global Order of Elements. It is important to give a good global mapping function to make the signatures evenly distributed in each segment and avoid the unbalance problem. Intuitively, if two frequent tokens are in the same segment, then the inverted-list size of the segment will be rather long and lead to bad performance. To this end, we propose a round-robin method to generate a good global mapping function. We first get the frequency of each token, and then sort the tokens based on their frequency in a descending order. Suppose there are $|\mathcal{P}|$ partitions. For the first $|\mathcal{P}|$ tokens, we put the i-th token into the i-th segment. For the next $|\mathcal{P}|$ tokens, we put the i-th token into the $(|\mathcal{P}| - i)$-th segment. Iteratively, we get a good global mapping order.



(a) Varying τ: `Tweet` (b) Scalability: `Tweet` (c) Scale-up: `Tweet` (d) Scale-out: `Tweet`

Figure 8: Comparison with Baselines on `Tweet` (Selection)



(a) Varying τ: `Review` (b) Scalability: `Review` (c) Scale-up: `Review` (d) Scale-out: `Review`

Figure 9: Comparison with Baselines on `Review` (Selection)

Table 2: Datasets.

Datasets	Cardinality	AvgLen	MinLen	MaxLen	Size
`Tweet`	20,000,000	15.52	5	32	2.9GB
`Review`	20,000,000	10.3	1	399	1.5GB
`ReviewBig`	100,000,000	11.2	1	399	7.7GB

9. EXPERIMENT

9.1 Experimental Setup

Dataset. Table 2 shows the statistics of the datasets used: `Tweet`, `Review`, `ReviewBig`. `Tweet`[2] is a user-follower dataset. Each record is a set of followers of a user. `Review`[3] is a book review dataset on Amazon. Each record is a set of tokens. `ReviewBig` was too large to be supported by baselines, so we sampled 20% of the data to compare with existing methods.

Baselines. We tried the MapReduce-based method [37, 30, 1, 9, 12], but they were too slow due to huge data transmission cost. We extended their code to support Spark and compared with them. Here we showed the signature-partition method `PrefixSig` [37] that achieved the best result on Spark among these methods (note that [12] only supported edit distance and [30, 1, 9] were slow due to involving huge data transmission). We compared with the native Spark `Naive` without index, and two data-partition methods `PrefixData` and `SelectableData`. `PrefixData` and `SelectableData` directly partitioned the data into different partitions. To avoid Cartesian product, we partitioned the data based on the length. For records of each length l, we partitioned them into same partitions and the records with length between l_{min} and l were also in this partition. Then we performed join in each partition but the data should be replicated in different partitions to meet the length requirement. For local join, `PrefixData` and `SelectableData` used prefix filtering [37] and our method respectively to compute local results. We implemented two versions of our method: `Dima+` with balance-aware techniques while `Dima` without.

Parameters. We varied three parameters, similarity threshold τ, number of cores and data set size. When we varied a parameter, other parameters were set to default values (highlighted in bold in Table 3). Due to space limit, we omitted results for evaluating partition number, indexing, and other functions and please see [34] for more results.

[2] http://snap.stanford.edu/data/twitter7.html

[3] http://snap.stanford.edu/data/web-Amazon.html

Table 3: Parameters (Default value is highlighted).

Parameter	Value
Jaccard Threshold τ	**0.8**, 0.85, 0.9, 0.95
k for Top-k Selection	250, 500, 750, **1000**
k for Top-k Join	2500, 5000, 7500, **10000**
k for BigData Top-k Selection	1250, 2500, 3750, 5000
k for BigData Top-k Join	5000, 10000, 15000, 20000
$\#Cores$	32, 64, 96, **128**
$\#Size$	5M, 10M, 15M, **20M**

Machines. All experiments were conducted on a cluster consisting of 64 nodes with a 8-core Intel Xeon E5-2670 v3 2.30GHz and 48 GB RAM. Each node was connected to a Gigabit Ethernet switch and ran Ubuntu 14.04.2 LTS with Hadoop 2.6.0 and Spark 1.5.0. The Spark cluster was deployed in standalone mode.

9.2 Comparison with Baselines

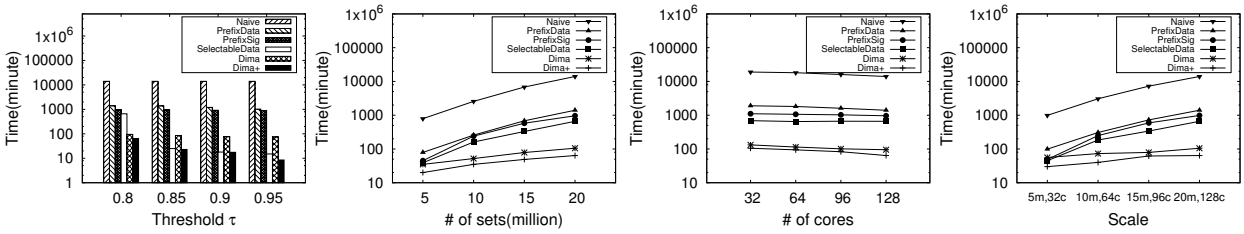
9.2.1 Similarity Selection

We first evaluated different methods for similarity selection. For each dataset, we randomly sampled 10,000 queries and reported the average time in Figures 8 and 9.

Varying Thresholds (Figures 8(a), 9(a)). We had five observations. (1) With the threshold increasing, the performance of all methods improved because a larger threshold can result in a smaller number of results. (2) Our methods significantly outperformed baseline approaches, by 1-3 orders of magnitude. This is attributed to our indexing framework and our selectable signatures. The two data partition based methods had low performance because i) they only partitioned the data but cannot guarantee the workload balance and ii) the prefix filter had lower pruning power than our signature based method. `PrefixSig` had lower performance because it had lower pruning power than our method. (3) `Dima+` outperformed `Dima` because `Dima+` used the balance-aware signature to make workload much more balanced. (4) `Naive` was rather slow as it had no index for similarity queries. (5) Our methods achieved high performance and answered a query within 10 ms.

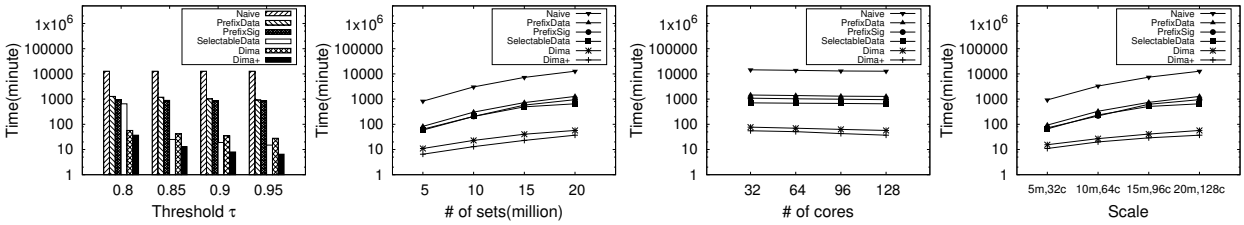
Scalability (Figures 8(b),9(b)). We evaluated scalability and made two observations. (1) The running time of all methods increased but our methods were better. (2) Our methods scaled better and outperformed competitors due to our effective signatures and workload-balance techniques.

Scale-up (Figures 8(c),9(c)). We varied the number of cores and had the following observations. (1) With more cores provided, the performance of all methods increased



(a) Varying τ: `Tweet` (b) Scalability: `Tweet` (c) Scale-up: `Tweet` (d) Scale-out: `Tweet`

Figure 10: Comparison with Baselines on `Tweet` (Join)



(a) Varying τ: `Review` (b) Scalability: `Review` (c) Scale-up: `Review` (d) Scale-out: `Review`

Figure 11: Comparison with Baselines on `Review` (Join)

as we could utilize more cores to compute answers. (2) Our methods still outperformed the baselines by 1-3 orders of magnitude (3) With more cores, our superiority against baselines were more signifiant as we had a large opportunity to keep balanced workloads on more cores.

Scale-out (Figures 8(d),9(d)). We varied the number of cores and the dataset size. In the figures "5m, 32c" means 5 million sets and 32 cores. We had three observations. (1) `Dima+` scaled-out well on both datasets, especially when we increased from "15m,96c" to "20m,128c". This matched the results of our previous experiments on data size and cores standalone. (2) Our methods outperformed baselines by 1-2 orders of magnitude. (3) The balance-aware techniques could improve the performance.

9.2.2 Similarity Join

Varying Thresholds (Figures 10(a),11(a)). First, we found that `Dima+` significantly outperformed `PrefixData`, `PrefixSig`, `SelectableData` and `Dima` in term of efficiency, especially when the threshold was small. For example, for $\tau = 0.8$ on the `Review` dataset, `PrefixData` took 1280 minutes, `PrefixSig` took 950 minutes, `SelectableData` took 651 minutes while `Dima` took 57 minutes and `Dima+` took 37 minutes. The main reasons were three-fold. (1) Prefix-based signatures generally generated more candidate pairs than our selectable signatures. (2) We employed global indexing to decrease communication between nodes and local indexing to prune candidate pairs. (3) Our balancing mechanism also played an important role in improving the performance as the stragglers in the distributed environment would cause great delay. Second, we found that `SelectableData` and `Dima+` were sensitive to the similarity threshold settings and `SelectableData` was faster than `Dima` when the threshold was higher than 0.8, but still slower than `Dima+`. `SelectableData` was less affected by slow tasks compared with `Dima` because it did not send signatures over the network while `Dima` suffered from this issue. `Dima+` resolved this problem by incorporating the balancing mechanism and it outperformed `SelectableData`. The main reason was that `Dima+` utilized global indexing to avoid redundant communication between cores over the network, which was considered to be a major performance killer in distributed computing.

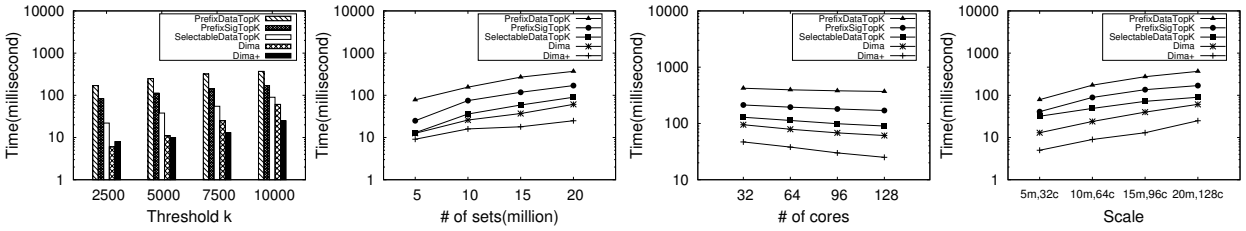
Scalability (Figures 10(b),11(b)). We varied the dataset sizes and tested the scalability. We had the following observations. (1) The superiority of our method over baselines was signifiant when the dataset was large, because larger dataset made the problem harder and involved more data transmission to compute the join results. (2) `Dima+` had better scalability because `Dima+` utilized selectable signatures to balance the workload. (3) `Dima+` performed comparably better on selection than join, as the loads on all machines were closer with each other and the data transmission over the network was rather little.

Scale-up (Figures 10(c),11(c)). We varied the number of cores. We had the following observations. (1) With the increase of the number of cores, the performance of all methods increased because we could utilize more cores to compute answers. (2) With different number of cores, our methods significantly outperformed existing approaches by 1-3 orders of magnitude. For example, `Dima+` took 83 minutes for 96 cores and 64 minutes for 128 cores on the `Tweet` dataset. `Dima+` performed better than other methods with increasing number of cores as only it utilized balancing mechanism to distribute loads as equally as possible among different nodes. (3) Our superiority against baselines became more signifiant for more cores, as we had a large opportunity to do balance on more cores.

Scale-out (Figures 10(d),11(d)). We varied both number of cores and the dataset size. We had the following observations. (1) Most methods scaled out well: with the increase of the number of cores and dataset sizes, their performance slightly increased. (2) Our methods still outperformed the baselines by 1-2 orders of magnitude. (3) The balance-aware techniques could improve the performance. For example, `Dima+` took 35 minutes for 10 million sets and took 64 minutes for 20 million sets on the `Tweet` dataset. Our method achieved nearly linear scalability. This was attributed to our efficient filtering methods to prune dissimilar candidate pairs, signature-based partition with balancing mechanism which was better than rough data-based partition as it was more accurate and fine-grained to calculate the loads on different machines, and multi-level indexing to avoid unnecessary network IO.

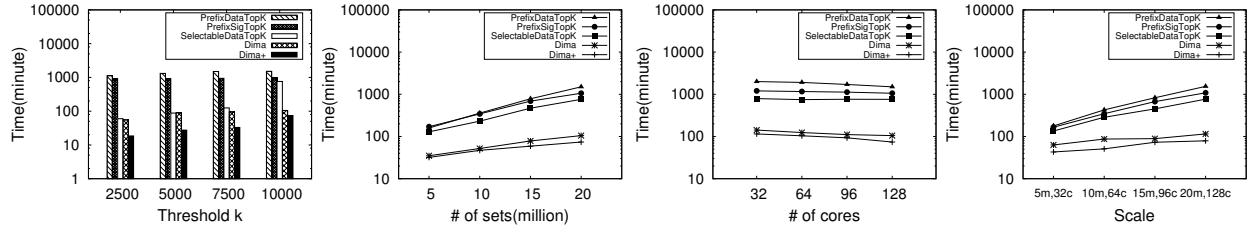
9.2.3 Top-K Similarity Selection

We compared with three baselines, the signature partition based method `PrefixSigTopK` [20] and two data partition based methods `PrefixDataTopK` and `SelectableDataTopK`. We used the threshold-based algorithms to compute the results with threshold 0.95. If there were k results, the algo-



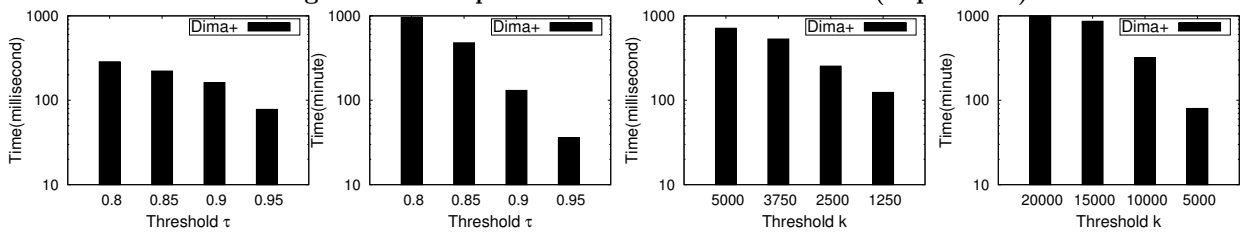
(a) Varying k: `Tweet` (b) Scalability: `Tweet` (c) Scale-up: `Tweet` (d) Scale-out: `Tweet`

Figure 12: Comparison with Baselines on `Tweet` (Top-k Selection)



(a) Varying k: `Tweet` (b) Scalability: `Tweet` (c) Scale-up: `Tweet` (d) Scale-out: `Tweet`

Figure 13: Comparison with Baselines on `Tweet` (Top-k Join)



(a) Selection (b) Join (c) Top-k Selection (d) Top-k Join

Figure 14: Evaluation on `ReviewBig` (100M)

rithms terminated; otherwise, we decreased the threshold to 0.9 until we found k results. Due to space limit, we only showed results on `Tweet`. From Figure 12, we had three observations. (1) With the increase of k, the runtime of all methods increased, because a larger k led to a larger number of results. (2) Our methods significantly outperformed baselines due to our indexing framework and progressive signatures. (3) `Dima+` was better than `Dima`, because `Dima+` used the signatures with decreasing length to make workload more balanced among iterations, and `Dima+` used selectable signatures to accelerate convergence and balance workload.

9.2.4 Top-K Similarity Join

From Figure 13, we found `Dima+` significantly outperformed `Dima`, `PrefixDataTopK`, `PrefixSigTopK`, `SelectableDataTopK` in efficiency. The reasons were three-fold: (1) Prefix-based signatures generally generated many more candidate pairs than our selectable signatures, especially during the early iteration round. (2) Splitting records with decreasing segment length could reduce operations on the queue. (3) Our balancing mechanism, also played an important role in improving the performance of computing the most time-consuming partitions. (4) Our methods could make full use of cores of the cluster even when the degree of parallelism was high.

9.3 Evaluation on Big Dataset

We evaluated our framework on a big data set `ReviewBig` to study the scalability w.r.t. a varying threshold. Since other methods could not support big data, we only showed the results for `Dima+`. Figure 14 showed the results. We had the following observations. (1) Our method still kept high performance for selection, join and top-k. (2) Our method could answer a selection query in milliseconds. For example, when the threshold was 0.8, `Dima+` was able to answer each query within 280ms in average. This was attributed to our efficient filtering methods to prune as many dissimilar pairs as possible and our balancing mechanism to prevent from data skewing and stragglers. (3) Even for the join operation, our method still achieved very high performance. (4) For top-k queries, `Dima+` still achieved very high performance. This was attributed to our progressive framework, the balance-aware signature selection methods and efficient signature generation methods.

Summary. Our methods significantly outperformed the baselines by 1-3 orders of magnitude for similarity selection, join and top-k selection, join. Our balance-aware signature selection could address the skewed workload problem. Our method scaled very well and could support big data.

10. CONCLUSION

We have developed a distributed in-memory similarity-based query processing system called `Dima` which supported four core similarity query operations: similarity-based selection and join, top-k similarity selection and join. We designed various selectable signatures as well as global & local index to facilitate efficient processing of similarity operators. We developed balance-aware signature selection algorithms to balance the workload for each involved partition in distributed environment. Extensive experiments on real-world datasets demonstrated the efficiency and scalability of `Dima`, and verified the effectiveness of our balance-aware signatures, indexes, and algorithms.

Acknowledgement. This work was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61521002, 61661166012), Huawei, and TAL education.

11. REFERENCES

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, pages 498–509, 2012.

[2] M. Alam, K. S. Perumalla, and P. Sanders. Novel parallel algorithms for fast multi-gpu-based generation of massive scale-free networks. *Data Science and Engineering*, pages 1–15, 2019.

[3] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[4] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[5] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *ACM STOC*, pages 327–336, 1998.

[6] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD Conference*, pages 313–324, 2003.

[7] S. Das, P. S. G. C., A. Doan, J. F. Naughton, G. Krishnan, R. Deep, E. Arcaute, V. Raghavendra, and Y. Park. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD Conference*, pages 1431–1446. ACM, 2017.

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

[9] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *PVLDB*, 6(13):1606–1617, 2013.

[10] D. Deng, G. Li, and J. Feng. A pivotal prefix based filtering algorithm for string similarity search. In *SIGMOD*, pages 673–684, 2014.

[11] D. Deng, G. Li, J. Feng, and W.-S. Li. Top-k string similarity search with edit-distance constraints. In *ICDE*, pages 925–936, 2013.

[12] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351, 2014.

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

[14] D. Deng, Y. Tao, and G. Li. Overlap set similarity joins with theoretical guarantees. In *SIGMOD*, pages 905–920, 2018.

[15] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

[16] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[17] M. Hadjieleftheriou, N. Koudas, and D. Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *SIGMOD Conference*, pages 429–440, 2009.

[18] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *ACM STOC*, pages 604–613, 1998.

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

[20] Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *ICDE*, pages 510–521, 2012.

[21] P. Konda, S. Das, P. S. G. C., A. Doan, A. Ardalan, J. R. Ballard, H. Li, F. Panahi, H. Zhang, J. F. Naughton, S. Prasad, G. Krishnan, R. Deep, and V. Raghavendra. Magellan: Toward building entity matching management systems. *PVLDB*, 9(12):1197–1208, 2016.

[22] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266, 2008.

[23] C. Li, B. Wang, and X. Yang. Vgram: Improving performance of approximate queries on string collections using variable-length grams. In *VLDB*, pages 303–314, 2007.

[24] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 46(3):31:1–31:42, 2014.

[25] G. Li, D. Deng, and J. Feng. A partition-based method for string similarity joins with edit-distance constraints. *ACM Trans. Database Syst.*, 38(2):9:1–9:33, 2013.

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

[27] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD*, pages 1137–1151, 2015.

[28] H. Li, P. Konda, P. S. G. C., A. Doan, B. Snyder, Y. Park, G. Krishnan, R. Deep, and V. Raghavendra. Matchcatcher: A debugger for blocking in entity matching. In *EDBT*, pages 193–204. OpenProceedings.org, 2018.

[29] K. Li and G. Li. Approximate query processing: What is new and where to go? *Data Science and Engineering*, 3(4):379–397, 2018.

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

[31] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD Conference*, pages 743–754, 2004.

[32] V. Satuluri and S. Parthasarathy. Bayesian locality sensitive hashing for fast similarity search. *PVLDB*, 5(5):430–441, 2012.

[33] Z. Shang, Y. Liu, G. Li, and J. Feng. K-join: Knowledge-aware similarity join. *IEEE Trans. Knowl. Data Eng.*, 28(12):3293–3308, 2016.

[34] J. Sun. Dima: A distributed in-memory similarity-based query processing system. *Tsinghua Technical Report. http://dbgroup.cs.tsinghua.edu.cn/ligl/dima.pdf.*

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

[36] G. Vargas-Solar, J.-L. Zechinelli-Martini, and J.-A. Espinosa-Oviedo. Big data management: What to keep from the past to face future challenges? *Data Science and Engineering*, 2(4):328–345, 2017.

[37] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[38] J. Wang, G. Li, D. Deng, Y. Zhang, and J. Feng. Two birds with one stone: An efficient hierarchical framework for top-k and threshold-based string similarity search. In *ICDE*, pages 519–530, 2015.

[39] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[40] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[41] J. Wang, G. Li, and J. Feng. Extending string similarity join to tolerant fuzzy token matching. *ACM Trans. Database Syst.*, 39(1):7:1–7:45, 2014.

[42] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[43] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[44] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[45] M. Yu, G. Li, D. Deng, and J. Feng. String similarity search and join: a survey. *Frontiers Comput. Sci.*, 10(3):399–417, 2016.

[46] M. Yu, J. Wang, G. Li, Y. Zhang, D. Deng, and J. Feng. A unified framework for string similarity search with edit-distance constraint. *VLDB J.*, 26(2):249–274, 2017.

[47] J. Zhai, Y. Lou, and J. Gehrke. ATLAS: a probabilistic algorithm for high dimensional similarity search. In *ACM SIGMOD*, pages 997–1008, 2011.

[48] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava. Bed-tree: an all-purpose index structure for string similarity search based on edit distance. In *SIGMOD*, 2010.