Top-k String Similarity Joins

Shuyao Qi
Department of Computer Science
The University of Hong Kong
China PR
qisy@connect.hku.hk

Panagiotis Bouros
Institute of Computer Science
Johannes Gutenberg University Mainz
Germany
bouros@uni-mainz.de

Nikos Mamoulis
Department of Computer Science and Engineering
University of Ioannina
Greece
nikos@cs.uoi.gr

ABSTRACT

Top-k joins have been extensively studied in relational databases as ranking operations when every object has, among others, at least one ranking attribute. However, the focus has mostly been the case when the join attributes are of primitive data types (e.g., numerical values) and the join predicate is equality. In this work, we consider string objects assigned such ranking attributes or simply scores. Given two collection of string objects and a string similarity measure (e.g., the Edit distance), we introduce the top-k string similarity join (k-SSJoin) which returns the k sufficiently similar pairs of objects with respect to a similarity threshold ϵ, which have the highest combined score computed by a monotone aggregate function γ (e.g., SUM). Such a join operation finds application in data integration, data cleaning and de-duplication scenarios, and in emerging scientific fields such as bioinformatics. We investigate how existing top-k join methods can be adapted and optimized for k-SSJoin taking into account the semantics and the special characteristics of string similarity joins. We present techniques to avoid computing the entire string join and indexing that enables pruning candidates with respect to both the string join and the ranking component of the query. Our extensive experimental analysis demonstrates the efficiency of our methodology for k-SSJoin, comparing solutions that either prioritize the ranking/join component or are able to handle both components of the query at the same time.

KEYWORDS

Ranking, top-k queries, top-k joins, string joins, similarity queries

ACM Reference Format:
Shuyao Qi, Panagiotis Bouros, and Nikos Mamoulis. 2020. Top-k String Similarity Joins. In *SSDBM '20: 32nd International Conference on Scientific and Statistical Database Management, June 07–09, 2020, Vienna, Austria.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Consider two collections of objects R and S, each having (at least) one *scoring attribute* `score` and a *join attribute* `att`. Given a join *predicate* ϕ (e.g., equality =) on attribute `att` of two objects and a *monotone aggregation function* γ (e.g., SUM) on `score`, a top-k join retrieves a k-subset $J \subseteq R \times S$ such that for each object pair $(r, s) \in J$, $\phi(r, s)$ is satisfied and for all $(r', s') \in R \times S \setminus J$ satisfying $\phi(r', s')$, $\gamma(r, s) \geq \gamma(r', s')$ holds. As an example, assume a retail company needs to identify the top-10 sales that maximize the associated profit of the company; the profit is defined as the amount paid by a customer plus the discount offered by a supplier. With the Customers and Suppliers relations maintained by the company's sales database, we can write the following SQL top-k join query to retrieve the top-10 sales:

```
SELECT *
FROM Customers C, Suppliers S
WHERE C.productID = S.productID
ORDER BY (C.price + S.discount) DESC
LIMIT 10;
```

In the above query, `productID` acts as the join attribute `att`, the join predicate ϕ is equality, and `price`, `discount` are the scoring attributes for input relations C, S, respectively.

Top-k joins have been widely studied by the data management community as ranking operations [4, 16, 17, 25, 27, 32, 38]. The focus, however, has been mainly on relational data objects with join attributes of primitive data types such as numerical values, and for an equality join predicate. To our knowledge, only [14, 19, 20] consider the semantics and the efficient evaluation of top-k joins on complex (non-relational) attributes. In [19, 20], every object o has a location in space and a score; the top-k spatial distance join considers the distance of the object locations as join predicate ϕ. In [14], each object o (e.g., a biological cell) is assigned a set of probabilistic locations and a confidence p_o (e.g., for belonging to a specific cell class). The top-k join in this context considers the distance between the uncertain locations of the objects as predicate ϕ, while the aggregate function γ is defined based on the confidence probability of the objects.

In this work, we study the top-k join operation for string objects. We introduce the top-k *String Similarity Join* (k-SSJoin), where the join predicate ϕ qualifies object pairs (r, s) of sufficiently similar string attributes `att`, with respect to a string distance measure $dist(\cdot, \cdot)$ (e.g., the Edit distance) and a user-defined *distance threshold* ϵ, i.e., object pairs for which $dist(r, s) \leq \epsilon$ holds. The k-SSJoin operation finds application in tasks such as data integration where object ratings from different sources are combined, or data cleaning and de-duplication, and in emerging scientific fields such as bioinformatics where strings are used to model biological data such as DNA sequences, and string joins can identify similar sequences.

Motivation examples. Consider the scenario of data integration. A person looking for a good restaurant uses a meta-search engine

SSDBM '20, June 07–09, 2020, Vienna, Austria

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

id	restaurant name	rating
r_1	La Bella Napoli	5
r_2	New York Pancakes	3
r_3	Luigi's Pizza	3
r_4	x-treme Burgers	1

id	restaurant name	rating
s_1	La Bella Napoli	5
s_2	Louigi's Pizza	4
s_3	Extreme Burgers	3
s_4	New York's Pancakes	1

Figure 1: Motivation example in data integration.

that ranks available options by combining ratings from different websites. To this end, a string similarity join can be used to handle potential typographic errors or abbreviations in restaurant names. In addition, users are more often interested in the objects with the highest combined ratings; in fact, [10] showed that users tend to iterate over only the first one or two pages of (web) search results. Therefore, a k-SSJoin operation is more useful than computing the entire string similarity join of the sources. Figure 1 illustrates the restaurant collections R, S from two different websites which may store the same object under a slightly different name, e.g., r_3 with "Luigi's Pizza " and s_2 with "Louigi's Pizza". Assuming that qualifying pairs should have an Edit distance of at most $\epsilon = 2$ and aggregate function $\gamma = AVG$, the 2-SSJoin query returns pairs (r_1, s_1) with aggregate score 5 and (r_3, s_4) with aggregate score 3.5. Hence, the recommendation is restaurants "La Bella Napoli" and "Luigi's Pizza", despite the fact the latter is misspelled as "Louigi's Pizza" in the second website.

As another example, consider the emerging field of bioinformatics. In genome sequence assembly, the first step is to find all pairs of similar reads (modeled as strings) under the Edit distance measure. The third generation sequencing technology such as single molecule real time sequencing (SMRT) [24] generates reads with 12-18% sequencing errors; hence, a threshold-based similarity join is employed. At the same time, reads can also carry quality ratings based on the sequencing scores of their bases. Under this, a top-k join result is more useful than the complete string similarity join result in data exploration scenarios where different similarity thresholds and score aggregation need to be tested.

Contributions. In this work, we address the efficient computation of the k-SSJoin operation. Contrary to relational top-k joins [7], in non-relational top-k joins, such as top-k spatial joins [20], the computational cost dominates the cost of accessing the objects. Hence, we investigate the applicability of top-k join algorithms on complex join attributes [19, 20], for the case where the join attribute is of string data type. First, we redesign their core functions under the semantics and the challenges of the string similarity join. For this purpose, we built on the state-of-the-art string similarity join algorithm, Pass-Join from [13]. Note however that the objective of Pass-Join is to compute the complete string similarity join result set, while in k-SSJoin our goal is to report only the k objects pairs that qualify the join predicate and have the highest aggregate score. To this end, we also employ optimizations to early terminate the string similarity join computation when enough object pairs to answer the k-SSJoin query are already determined. Last, we present the *aggregate* inverted index which indexes both the join and the score attribute of the input objects and therefore allows for pruning in both dimensions of the data when answering k-SSJoin queries.

Outline. The rest of the paper is organized as follows. Section 2 provides a background on existing algorithms for the top-k join computation. Then, Section 3 presents our methodology for k-SSJoin. Section 4 reports our experimental analysis on the efficient evaluation of k-SSJoin queries. Last, Section 5 reviews related work and Section 6 concludes the paper.

2 BACKGROUND ON TOP-k JOINS

Similar to other top-k join operations, i.e.., based on relational equijoins [7, 8, 27] or spatial distance joins [19, 20], a k-SSJoin query combines a join with a top-k query. In particular, it returns pairs of objects with (i) similar string information and (ii) a high aggregate score with respect to a monotone aggregate function γ. Under this, we review top-k join evaluation methods proposed in the existing literature. The first two prioritize either of the two top-k join sub-queries/components; in brief, SFA primarily considers the scoring attributes of the objects and the ranking component of the query, while DFA prioritizes the join component. The third method, called BA [19, 20] acts as a hybrid which considers both components at the same time and therefore, alleviates the shortcomings of SFA and DFA. We discuss the algorithms as general frameworks that can work with any type of join attributes and predicates.

2.1 The Score-First Algorithm

The *Score-First Algorithm* (SFA) generalizes the binary HRJN* [7] / PBRJ$^*_c$ [27] method. The idea is to progressively access input collections R and S in order of their `score` attribute, and incrementally produce results. SFA joins the currently accessed object, e.g., from R, on their join attribute with the buffered (i.e., already accessed) objects from S, which are indexed by a dedicated data structure $\mathcal{I}_S$. In [7] and the relational top-k equijoins, a hash-table was used for this purpose while in [19, 20] and the top-k spatial distance joins, an aR-tree [18]. Join results are organized in a priority queue based on their aggregate score. Let ℓ_R, h_R (ℓ_S, h_S) be the lowest and highest scores seen in collection R (S) so far; all join results currently in the queue with aggregate score higher than *threshold* $T = \max\{\gamma(h_R, \ell_S), \gamma(\ell_R, h_S)\}$ are guaranteed to have higher aggregate score than every future join result and thus can incrementally be output as top-k join results.

Algorithm 1 illustrates the pseudo-code of SFA which takes as input two object collections R and S, the predicate ϕ on their join attributes `att`, the monotone aggregate function γ on their scoring attributes `score`, and the number of requested results k. First, in Lines 1–2, the algorithm sorts (if needed) [1] inputs R, S and initializes indices $\mathcal{I}_R$, $\mathcal{I}_S$, min-heap C, bound θ and the lowest seen scores ℓ_R, ℓ_S. Next, in Lines 3–12, SFA incrementally accesses objects from collection R or S and evaluates the top-k join query. At each iteration, SFA first decides which collection should be accessed and consequently, which object will be examined. Following the pulling

[1] Input collections R and S need not to be sorted on their scoring attribute for example, if they stem from previous query operators which produce such interesting orders.

ALGORITHM 1: Score-First Algorithm (SFA)

Input : object collections R, S, join predicate ϕ, monotone aggregate function γ, number of results k
Output : result set C
Variables : indices $\mathcal{I}_R$, $\mathcal{I}_S$, bound θ, termination threshold T, the lowest seen scores ℓ_R, ℓ_S

1 **initialize** $C \leftarrow \emptyset$, $\theta \leftarrow -\infty$, $\mathcal{I}_R \leftarrow \emptyset$, $\mathcal{I}_S \leftarrow \emptyset$, $\ell_R \leftarrow \infty$, $\ell_S \leftarrow \infty$;
2 **sort** R, S in descending order of attribute score; ▷ If not already sorted
3 **while** *more objects exist in R and S* **do**
4 $i \leftarrow S$, **if** $\ell_S > \ell_R$; **otherwise** R ; ▷ Current input
5 $j \leftarrow R$, **if** $\ell_S > \ell_R$; **otherwise** S;
6 $o_i \leftarrow$ GetNextObject(i); ▷ Next object from current input
7 $\ell_i \leftarrow o_i$.score; ▷ Update the lowest seen score from current input
8 $T \leftarrow \max\{\gamma(h_R, \ell_S), \gamma(\ell_R, h_S)\}$; ▷ HRJN* termination threshold
9 $\langle \theta, C \rangle \leftarrow$ Probe($o_i, \mathcal{I}_j, T, \phi, \gamma, k, \theta, C$);
10 **if** $T \leq \theta$ **then** ▷ Result secured
11 **break;**
12 **insert** o_i to $\mathcal{I}_i$; ▷ Update index
13 **return** C;

strategy of HRJN*, SFA reads the next object from the collection with the higher last seen score, i.e., the higher between ℓ_R and ℓ_S.

With current object, e.g., r from input R (the case of s from S is symmetric), SFA first updates termination threshold $T = \max\{\gamma(h_R, \ell_S), \gamma(\ell_R, h_S)\}$ according to HRJN* in Line 8; h_R and h_S are the highest seen scores from R and S, respectively, i.e., they equal the score of the very first object in each collection. Then, it probes object r against the $\mathcal{I}_S$ index to retrieve objects $s \in S$, such that pair (r, s) qualifies predicate ϕ, and $\gamma(r, s) > \theta$ holds. To this end, SFA invokes the Probe procedure in Line 9. Procedure Probe employs $\mathcal{I}_S$ to identify every qualifying (r, s) pair and then updates C, θ as follows. If $|C| < k$, pair (r, s) is inserted into candidates set C regardless of its aggregate score. Otherwise, (r, s) is inserted into C only if $\gamma(r, s) > \theta$ and in this case, it *replaces* the k-th pair in C, such that set C always keeps the best k pairs found so far. Finally, θ is updated to the k-th aggregate score in C. The next step is to the check the termination condition in Line 10. Specifically, as soon as $T \leq \theta$, SFA terminates reporting C as the final result. Finally, SFA updates the $\mathcal{I}_R$ index on collection R inserting object r (Line 12) to be probed by objects of S in future iterations.

2.2 The Distance-First Algorithm

Due to prioritizing the join sub-query/component of top-k join, the *Distance-First Algorithm* (DFA) first joins the input collections R and S to produce all object pairs that qualify the predicate ϕ, and then identifies the k pairs among them with the highest aggregate score. Algorithm 2 illustrates the pseudo code of DFA. Different from SFA,

ALGORITHM 2: Distance-First Algorithm (DFA)

Input : object collections R, S, join predicate ϕ, monotone aggregate function γ, number of results k
Output : result set C
Variables : Indices $\mathcal{I}_R$, $\mathcal{I}_S$

1 **initialize** $C \leftarrow \emptyset$;
2 $\mathcal{I}_R \leftarrow$ CreateIndex(R);
3 $\mathcal{I}_S \leftarrow$ CreateIndex(S);
4 $C \leftarrow$ Join($\mathcal{I}_R, \mathcal{I}_S, \phi, \gamma, k$);
5 **return** C;

DFA makes no pre-assumptions about the order of the objects in collections R and S. DFA also employs a min-heap C of size k to produce the final results. In Lines 2–4, the algorithm computes the $R \bowtie_\phi S$ join invoking the Join procedure. The procedure passes each (r, s) result pair to heap C, which keeps track of the k pairs with the highest aggregate score. The type of the join attribute and the nature of the predicate ϕ dictate the strategy for computing the $R \bowtie_\phi S$ join. Without loss of generality, we assume that an index join of I_R and I_S (created for inputs R and S in Lines 2–3) is performed, but other approaches where one or none of the inputs are indexed can be employed instead.

The performance of DFA can be further enhanced by special indexing or sorting the inputs, to avoid computing the complete result of $R \bowtie_\phi S$. For this purpose, the Join procedure in relational top-k equijoins extends the HashJoin algorithm [28] with a min-heap that retains the k object pairs that satisfy the join predicate and have the highest aggregate score. In case of top-k spatial distance joins [19, 20], the input collections are indexed by aR-trees and Join adapts the classical algorithm of [2] to traverse the trees in a best-first, instead of a depth-first order, using a min-heap. In this manner, the tree entry pairs which have the maximum aggregate score are examined first during the join and this order guarantees that the object pairs qualifying the join predicate will be computed incrementally in decreasing order of their aggregate scores.

2.3 The Block-based Algorithm

The *Block-based Algorithm* (BA) can be seen as an adaptation of SFA and DFA at a block level. Similar to SFA, the algorithm examines the objects in decreasing order of their scores. However, instead of probing *each* accessed *object* against the buffered objects of the other collection seen so far, BA each time probes a *block* of accessed objects against the buffered *blocks* of objects of the other collection. Moreover, before probing a new block of objects, BA creates an index for this block, and therefore, the block-level probes correspond to instances of DFA. Different however to DFA, BA does not compute the entire $R \bowtie_\phi S$ join. For this purpose, BA associates each accessed block of objects b with a lower score bound b^ℓ and an upper score bound b^u. Since, the objects inside b are in decreasing order of their scoring attributes, b^u (b^ℓ) corresponds to the score of the first (last) object inside b.

Algorithm 3 illustrates the pseudo-code of BA. The algorithm receives the same inputs as the previous two. To compute the join,

ALGORITHM 3: Block-based Algorithm (BA)

Input : object collections R, S, join predicate ϕ, monotone aggregate function γ, number of results k
Output : result set C
Variables : Indices $\mathcal{I}_R$ and $\mathcal{I}_S$, bound θ, termination threshold T, the lowest seen scores ℓ_R and ℓ_S, block size λ

1 **initialize** $C \leftarrow \emptyset$, $\theta \leftarrow -\infty$, $\ell_R \leftarrow \infty$, $\ell_S \leftarrow \infty$;
2 **sort** R, S in descending order of the score attribute; ▷ if not already sorted
3 **determine** block size λ;
4 **while** *more blocks of objects exist in R and S* **do**
5 $i \leftarrow S$, **if** $\ell_S > \ell_R$; **otherwise** R ; ▷ Current input
6 $j \leftarrow R$, **if** $\ell_S > \ell_R$; **otherwise** S;
7 $b_i \leftarrow$ GetNextBlock(i, λ); ▷ Next objects block from current input
8 $\ell_i \leftarrow b_i^{\ell}$; ▷ Update the lowest seen score from current input
9 $\mathcal{I}_{b_i} \leftarrow$ CreateIndex(b_i); ▷ Index current block
10 **for** *each block* b_j *of* j **do**
11 **if** $\gamma({b_i}^u, {b_j}^u) > \theta$ **then**
12 $\langle \theta, C \rangle \leftarrow$ Join$(\mathcal{I}_{b_i}, \mathcal{I}_{b_j}, T, \phi, \gamma, k, \theta, C)$; ▷ Update current C and θ
13 $T \leftarrow \max\{\gamma(h_R, \ell_S), \gamma(\ell_R, h_S)\}$; ▷ Update termination threshold
14 **if** $T \leq \theta$ **then** ▷ Result secured
15 **break**
16 **return** C

it follows an approach similar to SFA, i.e., the object collections are sorted (Line 2), and accessed by their scoring attribute in decreasing order; also, at each iteration the last seen score and the termination threshold are defined similar to SFA (Lines 8 and 13). However, as discussed in the previous paragraph, BA operates on blocks of objects instead of single objects. In particular, the next block from current input, e.g., b_R from R, is accessed in Line 7. Then, BA constructs the $\mathcal{I}_{b_R}$ index (Line 9) and then joins b_R with every block b_S accessed (and buffered) so far from collection S. The b_S blocks are considered in decreasing order of their score ranges (i.e., first b_{S_1}, then b_{S_2} etc). A $b_R \bowtie_\phi b_S$ block join is computed in similar to DFA fashion, but with two key differences. First, BA decides to ignore the pair if at least k candidate result pairs are already found and $\gamma(b_R^u, b_S^u) \leq \theta$ (recall that θ is the k-th best aggregate score so far). In other words, BA computes only "promising" block-joins. Second, the Join procedure for BA updates both the min-heap C and bound θ, similar to SFA.

Setting block size λ. The analysis in [20] unveiled the trade-off between the response time of BA and its block size λ. Intuitively, small λ values incur a high indexing cost, and hence, BA benefits less from the block-wise join evaluation; while with a large λ, BA resembles an improved but still inefficient version of DFA, which computes a large part of the spatial distance join. Under this trade-off, [20] modelled the problem of automatically selecting the appropriate block size λ as an optimization problem. The optimal λ value minimizes the computational cost of BA, captured by the objective function:

$$C(\lambda) = |N_{index}(\lambda)| \cdot C_{index}(\lambda) + |N_{join}(\lambda)| \cdot C_{join}(\lambda)$$

where $N_{index}(\lambda)$ is the number of indexed blocks and $C_{index}(\lambda)$, the indexing cost per block, while $N_{join}(\lambda)$ denotes the number of block-joins and $C_{join}(\lambda)$, the cost of every block-join.

Costs $C_{index}(\lambda)$ and $C_{join}(\lambda)$ depend on the nature of the join attribute and predicate, and how a block-level join is implemented. In relational top-k equijoins, we can use MergeJoin to perform a block-level join. Thus, indexing simply involves sorting the contents of the blocks accessed from the input collections in the order of their join attribute, i.e., $C_{index}(\lambda) = \alpha_1 \cdot \lambda \cdot \log \lambda + \alpha_2$, and joining involves traversing the sorted blocks with a cost linear to the block size, i.e., $C_{join}(\lambda) = \alpha_3 \cdot \lambda + \alpha_4$. In case of top-$k$ spatial distance joins, the indexing cost is dominated by the cost of sorting a block to bulk-load its aR-tree, i.e., $C_{index}(\lambda) = \alpha_1 \cdot \lambda \cdot \log \lambda + \alpha_2$, while the joining cost for two aR-trees is quadratic to the block size λ, i.e., $C_{join}(\lambda) = \alpha_3 \cdot \lambda^2 + \alpha_4$. Constants α_1, α_2, α_3, α_4 can be derived via regression analysis. Last, $|N_{index}(\lambda)|$ and $|N_{join}(\lambda)|$ can be set by estimating the so-called any-k and top-k depths of the join, i.e., the number of objects required from each input to find k result pairs for the first time, and the total number of objects to be accessed from each input, to compute the final results, respectively.

3 EVALUATING k-SSJoin QUERIES

We next discuss the evaluation solutions for k-SSJoin. For this purpose, we first revisit in brief the Pass-Join algorithm; to the best of our knowledge, the algorithm still remains the most efficient method for *exact* string similarity joins as shown in [9, 29]. Then, we detail how SFA, DFA and BA can be adapted for k-SSJoin. The key is to carefully design their Probe and Join procedures in order to address the string similarity join, building on top of the Pass-Join algorithm. Naturally, the objective of Pass-Join is to report all similar string pairs. However, for the efficient evaluation of k-SSJoin queries, we need to avoid computing the complete result set of the string join, similar to the case of top-k joins in general. To this end, we also adjust the functionality of Pass-Join and employ advanced indexing which similar to the aR-tree in case of the top-k spatial distance joins, allows for pruning based both on the join predicate and the aggregate scores.

For the rest of the section, we use the collections of string objects $R = \{r_1, \ldots, r_8\}$ and $S = \{s_1, \ldots, s_8\}$ in Figure 2 as our running example. Also, for the rest of this paper, we use the Edit distance to measure the similarity of two strings. Hence, objects r and s are similar if the string attribute of r can be transformed to the string attribute of s by at most ϵ edit operations which include replacing an element in r, deleting an element from r, and inserting an element into r, i.e., join predicate $\phi = (dist(r, s) \leq \epsilon)$. For example, the Edit distance of r_3 to s_3 in Figure 2 is 2 as "*burgermeister*" can be transformed to "*burgermaster*" by replacing the second *e* with *a* and deleting *i*, i.e., minimum 2 operations.

id	string	score
r_1	"extreme_burgers"	1.0
r_2	"x−treme_burgers"	0.8
r_3	"burgermeister"	0.8
r_4	"dragon_snacks"	0.6
r_5	"the_cafe_drive"	0.6
r_6	"lougi's_pizza"	0.4
r_7	"golden_snacks"	0.3
r_8	"the_cake_place"	0.1

(a) Object collection R

id	string	score
s_1	"gourmet_food"	0.9
s_2	"luigi's_pizza"	0.9
s_3	"burgermaster"	0.8
s_4	"burger_meister"	0.7
s_5	"columbus_food"	0.7
s_6	"extreme_burgers"	0.4
s_7	"new_york_pancakes"	0.4
s_8	"the_cake_palace"	0.2

(b) Object collection S

Figure 2: Running example of collections R and S with 8 string objects each; for clarity purposes, we replace white space with the underscore character "_".

ALGORITHM 4: Pass-Join

Input : string collections R and S, string distance threshold ϵ
Output : result set $C = \{(r, s) \subseteq R \times S \mid dist(r, s) \leq \epsilon\}$

1 **sort** R and S first by string length and second in alphabetical order;
2 **initialize** inverted index $\mathcal{I}_R \leftarrow \emptyset$;
3 **for** *each* $r \in R$ **do**
4 **partition** r into $\epsilon + 1$ segments;
5 **add** segments to $\mathcal{I}_R$;
6 **for** *each* $s \in S$ **do**
7 **for** *each inverted list* $L_R^{l,i}$ *in* $\mathcal{I}_R$ *with* $|s| - \epsilon \leq l \leq |s| + \epsilon$ *and* $1 \leq i \leq \epsilon + 1$ **do**
8 $W \leftarrow$ SelectSubstrings$(s, L_R^{l,i})$;
9 **for** $w \in W$ **do**
10 **if** w *is in* $L_R^{l,i}$ **then**
11 $C \leftarrow$ Verify$(L_R^{l,i}[w], s, \epsilon, C)$; ▷ $L_R^{l,i}[w]$ is the entry for segment w in $L_R^{l,i}$
12 **return** C;

3.1 The Pass-Join Algorithm

Pass-Join [13] adopts a filter-and-refinement framework. During the filtering step, the algorithm follows a partition-based approach. Particularly, given two collections of string objects R and S and an Edit distance threshold ϵ, Pass-Join splits each object r in R into $\epsilon + 1$ disjoint segments. According to the pigeon hole principle, in order for a string object s in collection S to be similar to r w.r.t. threshold ϵ, object s must contain a substring which matches a segment of r; otherwise, candidate pair (r, s) can be safely pruned.

Algorithm 4 illustrates the pseudocode of Pass-Join. The algorithm takes as input two collections of string objects R and S, and an Edit distance threshold ϵ. The string objects in the collections are sorted first by their length and second lexicographically (Line 1). Next, an inverted index $\mathcal{I}_R$ is built on top of collection R (Lines 2–5) by dividing every object $r \in R$ into $\epsilon + 1$ segments. The inverted lists of $\mathcal{I}_R$ associate every distinct string segment with the objects that contain it. Particularly, inverted list $L_R^{l,i}$ in $\mathcal{I}_R$ indexes the i-th segment of every object with length l. Figure 3 shows inverted index $\mathcal{I}_R$ for collection R in Figure 2(a) with an Edit distance threshold $\epsilon = 3$. Consider for example object r_1.att = "extreme_burgers" of length 15. The object is partitioned into 4 segments {"ext", "reme", "_bur", "gers"} which are added to lists $L_R^{15,1}$, $L_R^{15,2}$, $L_R^{15,3}$, $L_R^{15,4}$, respectively. If more than one objects contain a segment w, a *bucket* entry is added to the corresponding inverted list. Observe bucket entries $\langle$"_bur", $\{r_1, r_2\}\rangle$ and $\langle$"gers", $\{r_1, r_2\}\rangle$ of lists $L_R^{15,3}$ and $L_R^{15,4}$, respectively in Figure 3, for objects r_1.att = "extreme_burgers" and r_2.att = "x−treme_burgers", which both contain segments "_bur" and "gers".

After indexing collection R, Pass-Join iterates over the objects in collection S and computes the join result C by probing the $\mathcal{I}_R$ index (Lines 6–11). According to the *length filtering* introduced in [6], every object s in S can be only joined with objects in R of length l, such that $l \geq |s| - \epsilon$ and $l \leq |s| + \epsilon$, where $|s|$ denotes the length of the string object s. To access such objects in R, Pass-Join traverses every inverted list $L_R^{l,i}$ with $|s| - \epsilon \leq l \leq |s| + \epsilon$ and $1 \leq i \leq \epsilon + 1$ (Line 7). The contents of every $L_R^{l,i}$ list are filtered keeping only the objects whose segments can match at least one of the substrings of s in set W (Lines 9–10). The substrings of s contained in set W are selected in a *multi-match-aware* manner employed by the SelectSubstrings procedure (Line 8). Finally, Pass-Join verifies candidate object pairs (r, s) for every object r in $L_R^{l,i}$ that contains a matched segment/substring w, using a length-based and an extension-based verification method (Line 11).

3.2 Aggregate Inverted Index

The inverted index employed by Pass-Join indexes only the string join attribute of the input collections and hence, no pruning on aggregate scores can be achieved. To address this issue, we propose an extension to the inverted index inspired by the aggregate score information maintained by an aR-tree [18] which was shown in [19, 20] to outperform traditional R-tree for top-k spatial distance joins. Such an *aggregate* inverted index introduces the following changes:

(i) every (bucket) entry of an inverted list is associated with the maximum score of the involved objects, and
(ii) every inverted list is augmented with the maximum score of all the contained entries.

Figure 4 illustrates the inverted index of Figure 3 augmented with aggregate score information. Notice the $\langle$"the", $\{r_5, r_8\}\rangle$ bucket entry in list $L_R^{14,1}$ assigned a score of 0.6 which equals r_5.score, since r_5.score > r_8.score, and the $L_R^{13,4}$ inverted list which is associated

inverted list	entries
$L_R^{13,1}$	⟨"bur", {r_3}⟩⟨"dra", {r_4}⟩⟨"lou", {r_6}⟩, ⟨"gol", {r_7}⟩
$L_R^{13,2}$	⟨"ger", {r_3}⟩⟨"gon", {r_4}⟩⟨"gi'", {r_6}⟩, ⟨"den", {r_7}⟩
$L_R^{13,3}$	⟨"mei", {r_3}⟩⟨"_sn", {r_4, r_7}⟩⟨"s_p", {r_6}⟩
$L_R^{13,4}$	⟨"ster", {r_3}⟩⟨"acks", {r_4, r_7}⟩, ⟨"izza", {r_6}⟩
$L_R^{14,1}$	⟨"the", {r_5, r_8}⟩
$L_R^{14,2}$	⟨"_ca", {r_5, r_8}⟩
$L_R^{14,3}$	⟨"fe_d", {r_5}⟩, ⟨"ke_p", {r_8}⟩
$L_R^{14,4}$	⟨"rive", {r_5}⟩, ⟨"lace", {r_8}⟩
$L_R^{15,1}$	⟨"ext", {r_1}⟩, ⟨"x−t", {r_2}⟩
$L_R^{15,2}$	⟨"reme", {r_1, r_2}⟩
$L_R^{15,3}$	⟨"_bur", {r_1, r_2}⟩
$L_R^{15,4}$	⟨"gers", {r_1, r_2}⟩

Figure 3: Inverted index $\mathcal{I}_R$ on collection R of Figure 2 under an Edit distance threshold $\epsilon = 3$.

inverted list	score bound	entries
$L_R^{13,1}$	0.8	⟨"bur", {r_3}, 0.8⟩, ⟨"dra", {r_4}, 0.6⟩, ⟨"lou", {r_6}, 0.4⟩, ⟨"gol", {r_7}, 0.3⟩
$L_R^{13,2}$	0.8	⟨"ger", {r_3}, 0.8⟩, ⟨"gon", {r_4}, 0.6⟩, ⟨"gi'", {r_6}, 0.4⟩, ⟨"den", {r_7}, 0.3⟩
$L_R^{13,3}$	0.8	⟨"mei", {r_3}, 0.8⟩, ⟨"_sn", {r_4, r_7}, 0.6⟩, ⟨"s_p", {r_6}, 0.4⟩
$L_R^{13,4}$	0.8	⟨"ster", {r_3}, 0.8⟩, ⟨"acks", {r_4, r_7}, 0.6⟩, ⟨"izza", {r_6}, 0.4⟩
$L_R^{14,1}$	0.6	⟨"the", {r_5, r_8}, 0.6⟩
$L_R^{14,2}$	0.6	⟨"_ca", {r_5, r_8}, 0.6⟩
$L_R^{14,3}$	0.6	⟨"fe_d", {r_5}, 0.6⟩, ⟨"ke_p", {r_8}, 0.1⟩
$L_R^{14,4}$	0.6	⟨"rive", {r_5}, 0.6⟩, ⟨"lace", {r_8}, 0.1⟩
$L_R^{15,1}$	1.0	⟨"ext", {r_1}, 1.0⟩, ⟨"x−t", {r_2}, 0.8⟩
$L_R^{15,2}$	1.0	⟨"reme", {r_1, r_2}, 1.0⟩
$L_R^{15,3}$	1.0	⟨"_bur", {r_1, r_2}, 1.0⟩
$L_R^{15,4}$	1.0	⟨"gers", {r_1, r_2}, 1.0⟩

Figure 4: Aggregate inverted index $a\mathcal{I}_R$ on collection R of Figure 2 under an Edit distance threshold $\epsilon = 3$.

with a score of 0.8 equal to the maximum score of all its contained entries.

3.3 The Score-First Algorithm

As discussed in Section 2.1, SFA incrementally accesses the objects of the input collection R or S in decreasing order of their scoring attribute, and joins them with the objects already examined from S or R. Under this perspective, to employ Pass-Join for SFA, we need to make the following two adjustments:

(i) the objects in each collection are no longer sorted first by their length and second lexicographically, but according to their scoring attribute, and
(ii) instead of indexing *only* collection R to build a *traditional* index $\mathcal{I}_R$ *offline*, *two aggregate* inverted indices $a\mathcal{I}_R$ and $a\mathcal{I}_S$ are *incrementally* built *online* to buffer the objects examined from collections R and S, respectively.

PROCEDURE 1: Probe (for SFA)

Input : object o, aggregate inverted index $a\mathcal{I}$, termination threshold T, join predicate $dist(\cdot,\cdot) \le \epsilon$, monotone aggregate function γ, number of results k, k-th highest aggregate score θ, candidate set C
Output : updated θ, C

```
1 for each list L^{l,i} in aI with |o| − ε ≤ l ≤ |o| + ε, 1 ≤ i ≤ ε + 1
    and γ(o.score, L^{l,i}.score) > θ and while T > θ do
2     W ← SelectSubstrings(o, L^{l,i});
3     for w ∈ W do
4         if w is in L^{l,i} then
5             if γ(o.score, L^{l,i}[w].score) > θ then
6                 ⟨θ, C⟩ ← Verify(L^{l,i}[w], o, ε, θ, C);
7 return ⟨θ, C⟩;
```

Hence, the currently accessed object, e.g., r from R, is first probed against the $a\mathcal{I}_S$ aggregate inverted index to retrieve objects $s \in S$ such that pair (r, s) qualifies the join predicate $dist(r, s) \le \epsilon$ and $\gamma(r, s) > \theta$ holds, where θ equals the score of k-th candidate result pair found so far. Then, r is divided into $\epsilon + 1$ segments and indexed by $a\mathcal{I}_R$ according to the rationale of Pass-Join.

Algorithm 1 from Section 2.1 remains unchanged in case of SFA for k-SSJoin; the input collections are indexed by the aggregate inverted indices $a\mathcal{I}_R$, $a\mathcal{I}_S$ and predicate is defined according to Edit distance and threshold ϵ, $\phi = (dist(r, s) \le \epsilon)$. Procedure 1 illustrates the pseudocode of SFA's Probe for k-SSJoin. The functionality of Probe is reminiscent to the probing part of Algorithm 4 in Lines 6–11. The current object $o = r$ (assume without loss of generality that o is from R) is joined with already examined objects s in S of length l that is larger than or equal to $|r| - \epsilon$, and smaller than or equal to $|r| + \epsilon$, provided that the segments of s can match at least one of the substrings of r. However, there exist two important differences compared to the probing process of Pass-Join:

(i) An entire inverted list is ignored if the maximum aggregate score involving current object o and any entry in $L_S^{l,i}$ cannot exceed bound θ, i.e., if $\gamma(o.\texttt{score}, L_S^{l,i}.\texttt{score}) \le \theta$, where $L_S^{l,i}.score$ denotes the maximum score of all entries inside list $L_S^{l,i}$.
(ii) The objects s from S that contain a match substring w of r, i.e., the contents of the $L_S^{l,i}[w]$ entry, are further filtered using the $\gamma(o.\texttt{score}, L_S^{l,i}[w].\texttt{score}) > \theta$ condition, where $L_S^{l,i}[w].\texttt{score}$ equals the highest score of the objects in $L_S^{l,i}[w]$.

Finally, Probe employs the Verify procedure to verify the candidate object pairs similar to Pass-Join. Verify for SFA also updates C and the θ bound which equals the aggregate score of k-th candidate pair found so far.

PROCEDURE 2: Join (for DFA)

Input : aggregate inverted index $a\mathcal{I}_R$, object collection S, join predicate $dist(\cdot,\cdot) \leq \epsilon$, number of results k

Output : candidate set C

1 **initialize** $\theta \leftarrow -\infty$;
2 **sort** S in descending order of the `score` attribute;
3 **for** *each* s *in* S **do**
4 **if** $\gamma(r_{max}, s) \leq \theta$ **then**
5 **break**; ▷ r_{max} `is the object in` R `with the highest score.`
6 $\langle \theta, C \rangle \leftarrow$ Probe$(s, a\mathcal{I}_R, \infty, (dist(\cdot,\cdot) \leq \epsilon), \gamma, k, \theta, C)$; ▷ `Procedure 1`
7 **return** C;

Example 3.1. Consider collections R and S of Figure 2 and a k-SSJoin query with $k = 1$, $\epsilon = 3$, and $\gamma = SUM$. SFA first accesses r_1 from R. As aggregate inverted index $a\mathcal{I}_S$ is currently empty, r_1 is only split into $\epsilon + 1 = 4$ segments which are inserted to $a\mathcal{I}_R$: $L_R^{15,1} = \{1.0, \langle \text{“}ext\text{”}, \{r_1\}, 1.0\rangle\}$, $L_R^{15,2} = \{1.0, \langle \text{“}reme\text{”}, \{r_1\}, 1.0\rangle\}$, $L_R^{15,3} = \{1.0, \langle \text{“}\_bur\text{”}, \{r_1\}, 1.0\rangle\}$, $L_R^{15,4} = \{1.0, \langle \text{“}gers\text{”}, \{r_1\}, 1.0\rangle\}$. Next, s_1 is accessed from S and probed against $a\mathcal{I}_R$ without producing any join results as no substring of s_1 matches the existing segments in the $a\mathcal{I}_R$ index. Since $\ell_R = 1.0 > \ell_S = 0.9$, r_2 is the next object to be accessed and joined (unsuccessfully) with $a\mathcal{I}_S$. Similarly, s_2 and s_3 are accessed in turn, still without producing any string join results. When r_3 is accessed and probed against $a\mathcal{I}_S$ (now containing entries for s_1, s_2 and s_3), the substring “bur” of r_3 is matched with entry $L_S^{12,1}$[“bur”] that contains s_3. Verification confirms that $dist(r_3, s_3) = 2 < \epsilon$, and as bound θ is not yet defined, SFA inserts to C the first join result (r_3, s_3) and sets $\theta = \gamma(r_3, s_3) = 1.6$. Currently, $T = \max\{\gamma(1.0, 0.8), \gamma(0.8, 0.9)\} = 1.8 > \theta$, which means that a possibly better object pair can be found and SFA cannot terminate yet. The next accessed object is r_4. At this point, all inverted lists in $a\mathcal{I}_S$ (containing entries for s_1, s_2 and s_3) have a score bound of 0.9 (i.e., due to an entry for a segment of either s_1 or s_2), and as $\gamma(r_4, 0.9) = 1.5 < \theta$, no object pair involving r_4 can be better than current C. Hence, SFA ignores r_4 without probing against any inverted lists. Then, s_4 is accessed and unsuccessfully probed only against $L_R^{15,1}$–$L_R^{15,4}$ lists of $a\mathcal{I}_R$ with a score bound of 1.0; lists $L_R^{13,1}$–$L_R^{13,4}$ are ignored as their score bound is 0.8 and $\gamma(0.8, s_4) = 1.5 < \theta$. Next, s_5 gives no new join pairs. Finally, s_6 is retrieved without traversing any inverted lists, similar to r_4. At this points, since the T termination threshold is now set to 1.5, i.e., lower than $\theta = 1.6$, SFA terminates reporting $C = \{(r_3, s_3)\}$ as the final result.

3.4 The Distance-First Algorithm

To compute k-SSJoin following the rationale of DFA, we directly employ the Pass-Join algorithm discussed in Section 3.1 in order to identify the object pairs that qualify the string distance predicate $dist(r, s) \leq \epsilon$. Compared to Algorithm 2 in Section 2.2 where DFA performs an index join, for k-SSJoin we index only R using aggregate index $a\mathcal{I}_R$ and probe every object in S against $a\mathcal{I}_R$, according to Pass-Join. For probing $a\mathcal{I}_R$, we employ the Probe procedure introduced in the previous section for SFA by setting termination threshold $T = \infty$; in other words, the probing process of Join cannot terminated before all inverted lists in $a\mathcal{I}_R$ are considered.

Contrary to Pass-Join however, the Join procedure of DFA needs score bounds to avoid computing the entire string join between the input collections, accelerating therefore the computation of k-SSJoin. For this purpose, we sort the objects of collection S in decreasing order of their `score` attribute; essentially, we replace Line 3 in Algorithm 2 with a sort command for collection S. When DFA accesses an object s, the procedure checks whether s can contribute to a join result of aggregate score higher than the θ threshold, i.e., the score of the k-th object pair found so far. If not, DFA terminates, because all objects from S not examined yet have score equal to or lower than current object s. Procedure 2 illustrates the pseudocode of the Join procedure for DFA. For the termination condition in Line 4, we consider the object r_{max} with the highest score in R, and therefore, $\gamma(r_{max}, s)$ is an upper bound of the aggregate score a join pair that includes current object s could have.

Example 3.2. Consider the k-SSJoin query of Example 3.1. As a first step, DFA partitions the string attribute of the objects in R into $\epsilon + 1 = 4$ segments and builds the $a\mathcal{I}_R$ aggregate inverted index of Figure 3. It also sorts the objects of S in descending order of their scores. Next, DFA performs the string join by probing first object s_1 against $a\mathcal{I}_R$ which produces no join result. Then, s_2 is successfully joined with r_6 as $dist(r_6, s_2) = 1$; thus, the result pair (r_6, s_2) is inserted into C and $\theta = \gamma(r_6, s_2) = 1.3$. When s_3 is examined, DFA considers the best possible result pair (r_1, s_3) combined with $r_1 = r_{max}$ from collection R. Since, $\gamma(r_1, s_3) = 1.8 > \theta$, DFA cannot terminate; thus, s_3 is probed against $a\mathcal{I}_R$. At this point, the candidate join pair (r_3, s_3) which has higher aggregate score than current θ is identified and hence, (r_3, s_3) replaces (r_6, s_2) in C and $\theta = \gamma(r_3, s_3) = 1.6$. In the following, objects s_4 and s_5 are examined without however producing any better results than (r_3, s_3). Specifically, both objects are probed against lists $L_R^{15,1}$–$L_R^{15,4}$ with a score bound of 1.0 but not against $L_R^{13,1}$–$L_R^{13,4}$ with score bound 0.8 as $\gamma(0.8, 0.7) = 1.5 < \theta$. Finally, when s_6 is accessed and the best possible result pair (r_1, s_6) is considered, $\gamma(r_1, s_6) = 1.4 < \theta$ holds; thus, DFA terminates ignoring the rest of the objects in S and reporting $C = \{(r_3, s_3)\}$ as the final result.

3.5 The Block-based Algorithm

As discussed in Section 2.3, BA operates as an adaptation of both SFA and DFA at the block level. To compute k-SSJoin, the objects of input collections R and S, sorted in decreasing order of their `score` attribute similar to SFA, are accessed in blocks of size λ. BA accesses one block of objects at a time from either of the collections. Following the rationale of Pass-Join, the current block is indexed only if it originates from R, i.e., b_R, while in both cases the current block b_R or b_S is joined against the blocks already accessed from collection S or R, respectively. [2] Recall at this point that BA utilizes the score bounds retained for each block to avoid computing every

[2] We also experimented with a version of BA that uses two aggregate inverted indices, which however was always less efficient.

PROCEDURE 3: Join (for BA)

Input : aggregate inverted index $a\mathcal{I}_{b_R}$, block b_S, termination threshold T, join predicate $dist(\cdot,\cdot) \leq \epsilon$, monotone aggregate function γ, number of results k, k-th highest aggregate score θ, candidate set C

Output : updated θ, C

1 **for** *each object s in block* b_S **do**
2 **if** $\gamma(b_R^u, s.\texttt{score}) \leq \theta$ **then**
3 **break**;
4 $\langle \theta, C \rangle \leftarrow$ Probe$(s, a\mathcal{I}_{b_R}, T, (dist(\cdot,\cdot) \leq \epsilon), \gamma, k, \theta, C)$;
 ▷ Procedure 1
5 **return** $\langle \theta, C \rangle$;

possible block-level join, and consequently, the entire string join (Line 10 in Algorithm 3). The process of joining two blocks b_R and b_S is similar to the one employed by DFA, i.e., every object s in b_S is probed against aggregate inverted index $a\mathcal{I}_{b_R}$. However, different from DFA and its Join procedure, objects in b_S are already sorted in decreasing order of their `score` attribute.

Similar to SFA, the pseudocode of BA for k-SSJoin is identical to Algorithm 3; input collections are indexed by aggregate inverted indices $a\mathcal{I}_R$, $a\mathcal{I}_S$ and predicate $\phi = (dist(r,s) \leq \epsilon)$. Procedure 3 illustrates the pseudocode of BA's Join for k-SSJoin. Similar to DFA, Join for BA employs a breaking condition to terminate a block-level join between b_R and b_S if for current object s in b_S, $\gamma(b_R^u, s.\texttt{score}) \leq \theta$ holds, i.e., s cannot contribute to a join result with an aggregate score higher than the score of the k-th object pair found so far; recall that b_R^u is the highest score of the objects contained in block b_R. Join for BA uses the Probe procedure of SFA and it also utilizes the termination threshold T to avoid considering all inverted lists in $a\mathcal{I}_{b_R}$, i.e., unlike DFA's Join, threshold T is not set to ∞.

Example 3.3. Consider once again collections R and S in Figure 2 and the k-SSJoin query with $k = 1$, $\epsilon = 3$, $\gamma = SUM$. For the sake of this example, the collections are partitioned into blocks of 2 objects each, as pictured in Figure 5. Initially, block b_{R_1} is read and aggregate inverted index $a\mathcal{I}_{b_{R_1}}$ is built by dividing the string attribute of the contained objects into $\epsilon + 1 = 4$ segments; we have $L^{15,1}_{b_{R_1}} = \{\langle$"$ext$", $\{r_1\}, 1.0\rangle, \langle$"$x{-}t$", $\{r_2\}, 0.8\rangle\}$, $L^{15,2}_{b_{R_1}} = \{\langle$"$reme$", $\{r_1, r_2\}, 1.0\rangle\}$, $L^{15,3}_{b_{R_1}} = \{\langle$"$\_bur$", $\{r_1, r_2\}, 1.0\rangle\}$, and $L^{15,4}_{b_{R_1}} = \{\langle$"$gers$", $\{r_1, r_2\}, 1.0\rangle\}$. Then, b_{S_1} is accessed and objects s_1 and s_2 are probed against $a\mathcal{I}_{b_{R_1}}$ producing no join results. Similarly, b_{S_2} is accessed and joined (unsuccessfully) with b_{R_1}. After reading b_{R_2}, the block is joined with b_{S_1} and b_{S_2} (in this order). When joining with b_{S_2}, the substring "bur" in s_3 is matched with the $L^{13,1}_{b_{R_2}}$["bur"] entry that contains object r_3. Further verification confirms that $dist(r_3, s_3) = 2 < \epsilon$ and thus, the (r_3, s_3) pair is inserted into C and $\theta = \gamma(r_3, s_3) = 1.6$. In addition, index $a\mathcal{I}_{b_{R_2}}$ for current block b_{R_2} is built. The next block b_{S_3} is joined with b_{R_1} but not with b_{R_2}, because $\gamma(b^u_{R_2}, b^u_{S_3}) = \gamma(0.8, 0.7) = 1.5 < \theta = 1.6$. Specifically for b_{S_3}, probing s_5 against $a\mathcal{I}_{b_{R_1}}$ does not improve the current k-SSJoin result while s_6 is not probed against $a\mathcal{I}_{b_{R_1}}$ at all

block	id	string	score
b_{R_1}	r_1	"extreme_burgers"	1.0
	r_2	"x-treme_burgers"	0.8
b_{R_2}	r_3	"burgermeister"	0.8
	r_4	"dragon_snacks"	0.6
b_{R_3}	r_5	"the_cafe_drive"	0.6
	r_6	"lougi's_pizza"	0.4
b_{R_4}	r_7	"golden_snacks"	0.3
	r_8	"the_cake_place"	0.1

(a) collection R

block	id	string	score
b_{S_1}	s_1	"gourmet_food"	0.9
	s_2	"luigi's_pizza"	0.9
b_{S_2}	s_3	"burgermaster"	0.8
	s_4	"burger_meister"	0.7
b_{S_3}	s_5	"columbus_food"	0.7
	s_6	"extreme_burgers"	0.4
b_{S_4}	s_7	"new_york_pancakes"	0.4
	s_8	"the_cake_palace"	0.2

(b) collection S

Figure 5: Example of BA with block size $\lambda = 2$ on the collections in Figure 2.

because $\gamma(b^u_{R_1}, s_6.score) = \gamma(r_1, s_6) = 1.4 < \theta$, i.e., s_6 is not able to produce join results of aggregate score higher than θ. At this stage, BA terminates as $T = 1.5$ is not higher than $\theta = 1.6$, and $C = \{(r_3, s_3)\}$ is returned as the final result.

Setting block size λ. We finally discuss the automatic tuning of block size λ for k-SSJoin. For this purpose, it suffices to define the $C_{index}(\lambda)$ and $C_{join}(\lambda)$ costs of the objective cost function $C(\lambda)$. Specifically, the cost of partitioning the string attribute of the objects inside a block and building an aggregate inverted index is linear to λ, i.e., $C_{index}(\lambda) = \alpha_1 \cdot \lambda + \alpha_2$, while, the joining cost for two blocks is dominated by substring selection and verification; according to [13], $C_{join}(\lambda) = \alpha_3 \cdot \lambda + \alpha_4$. Again, constants α_1, α_2, α_3, α_4 can be derived via regression analysis, while $|N_{index}(\lambda)|$ and $|N_{join}(\lambda)|$ can be estimated by applying the model for any-k and top-k depths proposed in [20].

4 EXPERIMENTAL EVALUATION

We finally present our experimental analysis for k-SSJoin. We implemented all k-SSJoin methods in C++ and run the tests on a 2.3 GHz Intel Core i7 CPU with 8GB of RAM. All data (input collections and inverted indices) reside in main memory. First, Section 4.1 details the setup of our analysis. Then, Section 4.2 demonstrates the effectiveness of adapting the [20] model for selecting BA's block size in case of k-SSJoin. Last, Section 4.3 conducts an extensive comparison of SFA, JFA and BA.

4.1 Setup

Datasets. Our analysis involves both real-world and synthetic datasets. First, we used a collection of 645K hotels from Booking.com denoted by BHOTELS, and a collection of 255K hotels from

Table 1: Experimental parameters (default values in bold).

description	parameter	values
Join selectivity	ϵ	READS: 0, 4, **8**, 12, 16 CITIES: 0, 1, **2**, 3, 4 BHOTELS-THOTELS: 0, 1, **2**, 3, 4
Number of results	k	1, 5, **10**, 50, 100
Number of seeds	$\|\Sigma\|$	CORR: 10, **20**, 50, 100
Number of objects ($\times 1,000,000$)	$\|R\|+\|S\|$	READS: 0.625, 1.25, **2.5**, 5
Cardinality ratio	$\|R\|/\|S\|$	**1**, 2, 3, 4, 5

TripAdvisor.com denoted by THOTELS. The scoring attribute is a user-generated rating, while the joining attribute stores the name of a hotel. Second, we used collections of string attributes from [29] with synthetic scores. Specifically, CITIES is a collection of 1M geographical names taken from World Gazetteer with a 200 symbols dictionary and a non-uniform distribution of string length (5-64), while READS is a collection of 5M reads obtained from a human genome with a 5-symbol dictionary and a uniform length distribution (around 100 symbols per string). To generate scores, we employed a similar strategy to [19] which produces two types of scores, named IND and CORR. For IND, score values are normally distributed inside the $[0, 1]$ interval and independent to the values of the string join attribute. In contrast, for CORR, we first randomly generate $|\Sigma|$ seeds, and assign to each of them a score uniformly distributed inside $[0, 0.8]$. The generated objects are divided into $|\Sigma|$ clusters based on their distance to the seeds and the score of each object equals the score of its closest seed plus a noise normally distributed inside $[0, 0.2]$.

Tests. To assess the performance of the evaluation methods, we measure their response time for $\gamma = SUM$ [3], including any indexing and/or sorting costs, while varying: (i) the join selectivity, captured by distance threshold ϵ, (ii) the number of results k, and (iii) the number of seeds $|\Sigma|$ for synthetic collections of CORR scores. We also perform scalability and cardinality tests over subsets of the synthetic collections varying parameters $|R|+|S|$ and $|R|/|S|$. Table 1 summarizes all parameters involved in our study. On each test, we vary one parameter; the rest are set to their default value. Note that as the value of $|\Sigma|$ increases the score generator produces more independent and less correlated scores; for $|\Sigma| = 100$, the generated scores are uniformly distributed.

4.2 Setting Block Size λ

To automatically set block size λ for BA, we adapt our model from [20] for k-SSJoin, as discussed in Section 3.5. We then investigate the accuracy of the model by running BA while varying the value of λ. Figure 6 reports the algorithm's response time excluding the sorting costs. For clarity purposes, we only show the time around the optimal value λ_{opt} of the block size and mark λ_{est} estimated by the model. The tests reveal the anticipated trade-off between BA's response time and the value of λ. Recall that for $\lambda = 1$ BA operates similar to SFA but as the block size increases towards λ_{opt} the algorithm increasingly benefits from the block-wise evaluation. However, when λ increases beyond optimal value λ_{opt}, BA becomes less efficient as it resembles an improved version of DFA which computes an increasing larger part of the $R \bowtie_\epsilon S$ string similarity join. Although the model is not able to find the exact λ_{opt}, the figure shows that BA's execution time for IND and CORR score types increases only 3% and 2%, respectively when λ_{est} is used; for the real datasets (plot omitted due to lack of space) the time increases only by 1%. Note that this estimation procedure is very fast; our tests show that the time spend to compute λ_{est} corresponds to only the 3.3% of the total response time of BA, on average.

[3] Our analysis can be directly extended to any monotone aggregate function.

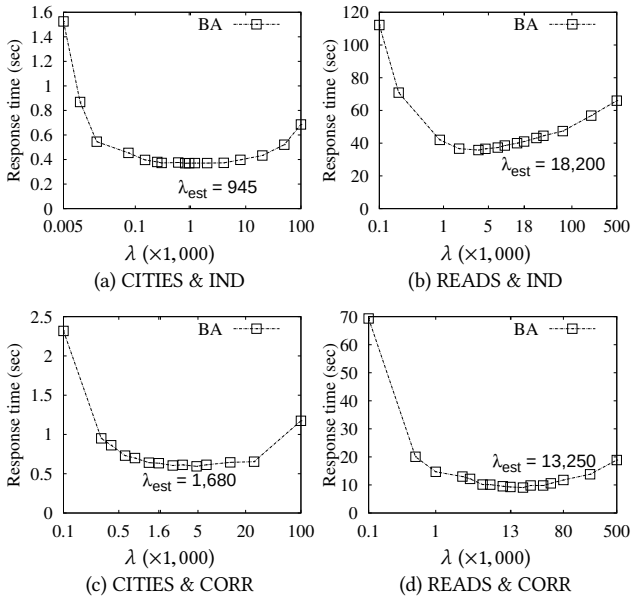

Figure 6: Response time of BA (excluding sorting cost), varying block size λ.

4.3 Comparison of Evaluation Algorithms

We next present our tests on comparing SFA, DFA and BA. Figure 7 reports the response time of the methods for the synthetic datasets in case of IND scoring attributes while reducing the selectivity of the join (i.e., increasing the string distance threshold ϵ) and while varying the number of requested results k. Similarly, Figure 8 reports the times in case of CORR scoring attributes while also varying the number of seeds $|\Sigma|$. We observe that BA is the most efficient method for k-SSJoin queries; its response is lower than both SFA's and DFA's in all cases. These findings are completely aligned with the case of the top-k spatial distance join, we studied in [19, 20], which proves the merit of BA and the advantage of the block-based evaluation for top-k joins. Regarding, the second fastest method, SFA has the advantage in most of the cases.

The key in understanding the behaviour of the algorithms when varying the join selectivity is to recall that k-SSJoin comes as a hybrid of a join and a top-k query, which introduces an interesting trade-off. Specifically, while increasing ϵ, the join component of a k-SSJoin query becomes less selective and therefore, more expensive. However, as more object pairs qualify the join predicate, the best k results can be now identified faster, sometimes even among the highly ranked objects. In other words, the top-k component of

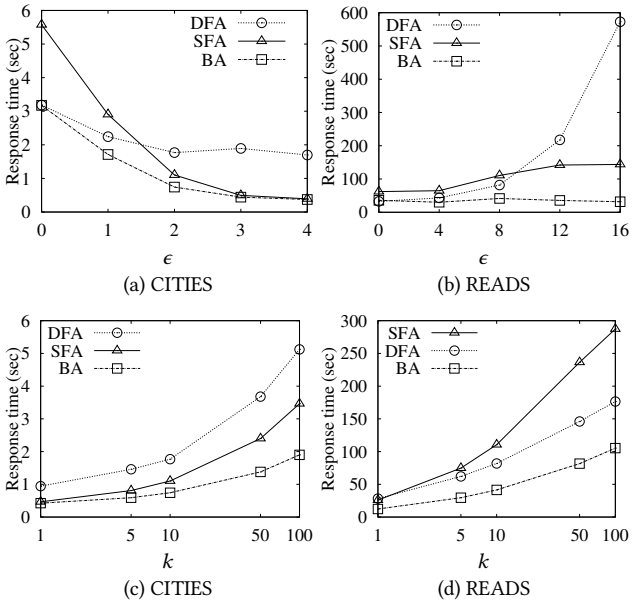

Figure 7: Evaluating k-SSJoin queries: scoring attributes of type IND.

the query becomes cheaper. As a result, DFA is competitive to SFA and BA only when the string join component of the k-SSJoin query is very selective and hence cheap, i.e., for ϵ equal to 0, 1 for CITIES and 0, 2, 4 for READS. In all other cases, pruning based on the aggregate score is more effective and therefore, BA and SFA have the benefit. We also observe a difference in the way ϵ impacts each dataset. Specifically, while ϵ increases the response time of SFA, BA rises for READS but drops for CITIES. As READS collections contain more in number and longer in length strings drawn from a much smaller dictionary compared to CITIES, the join component of the query is far more expensive than the top-k selection and becomes even more expensive as ϵ increases.

On the other hand, when increasing the number of results, all algorithms are negatively affected as they need to examine and compute the aggregate score for increasingly more objects, and hence compute a larger part of the $R \bowtie_\epsilon S$ string join. In contrast, the algorithms are positively affected by the increase of $|\Sigma|$ for CORR scores. This is because the scoring and the join attribute become less correlated, similar to the collections of IND scores.

We also compare the methods for our real-world datasets while increasing the number of requested results and varying the join selectivity via ϵ; Figure 9 reports the response times. Similar to the synthetic collections, BA outperforms both SFA and DFA. However, we also observe that DFA is now the second fastest method, instead of SFA. Due to their characteristics (primarily, fewer objects), the string similarity join for the real collections is cheaper compared to the synthetic datasets which benefits DFA that prioritizes the join component of the k-SSJoin queries.

Last, we conduct scalability and cardinality tests varying parameters $|R| + |S|$, $|R|/|S|$ for READS with both IND and CORR scores. Figure 10 and 11 report the response time of the evaluation algorithms. We observe that (i) BA outperforms SFA and DFA in all cases, and (ii) BA scales always better than DFA and in most of setups also than SFA.

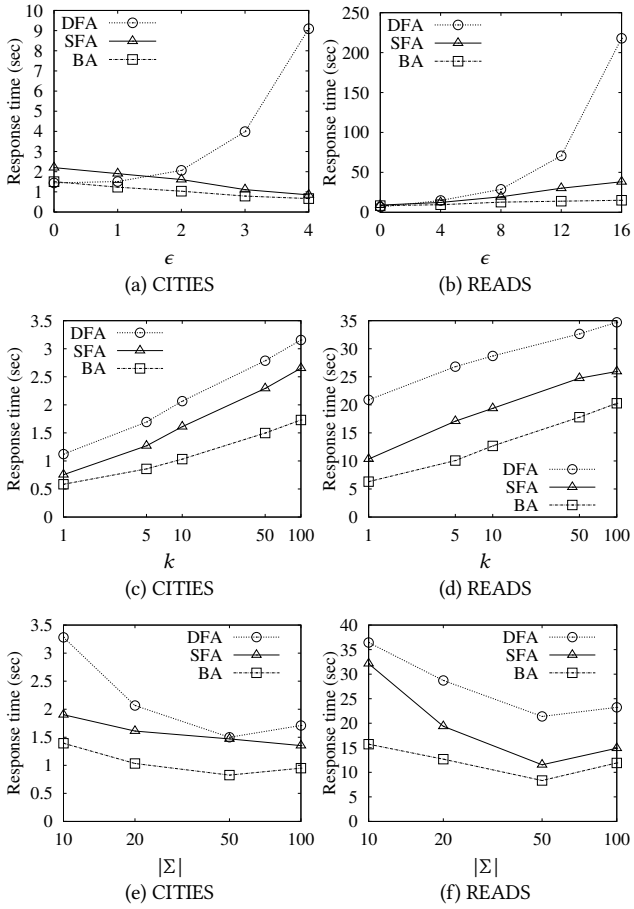

Figure 8: Evaluating k-SSJoin queries: scoring attributes of type CORR.

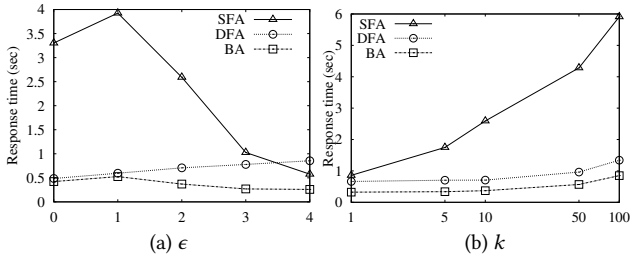

Figure 9: Evaluating k-SSJoin queries: real object collections.

5 RELATED WORK

Finally, we review previous work on ranking queries, i.e., top-k queries and top-k joins (besides [7, 19, 20]), and also briefly discuss previous work on string similarity joins (besides [13]).

Top-k Queries. Fagin et al. [5] present an analytical study of various methods for top-k aggregation of ranked inputs by monotone aggregate functions. Consider a collection of objects (e.g., restaurants) which have scores (i.e., rankings) at two or more different

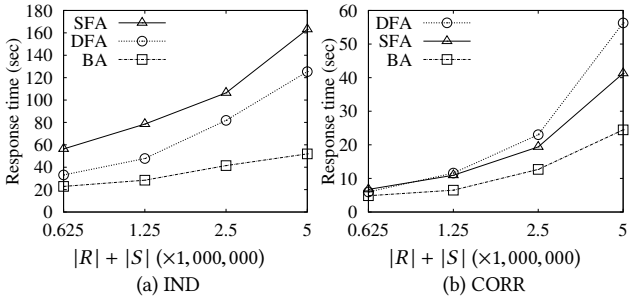

Figure 10: Evaluating k-SSJoin queries: scalability tests on READS.

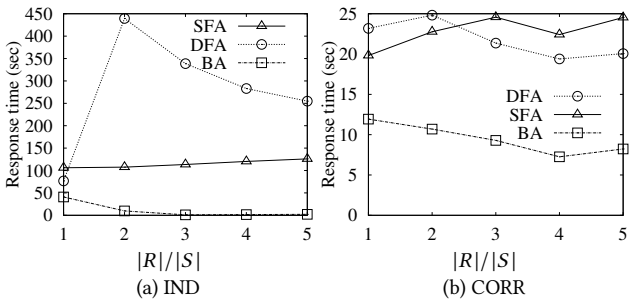

Figure 11: Evaluating k-SSJoin queries: cardinality tests on READS.

sources (e.g., different ranking websites). Given an aggregate function γ (e.g., SUM) the top-k query returns the k restaurants with the highest aggregated scores (from the different sources). Each source is assumed to provide a sorted list of the objects according to their atomic scores there; requests for random accesses of scores based on object identifiers may be also possible. For the case where both sorted and random accesses are possible, the *Threshold Algorithm* (TA) retrieves objects from the ranked inputs (e.g., in a round-robin fashion) and a priority queue is used to organize the best k objects seen so far. Let ℓ_i be the lowest seen score in source S_i; $T = \gamma(\ell_1, ..., \ell_m)$ defines a lower bound for the aggregate score of objects never seen in any S_i yet. If the k-th highest aggregate score found so far is no less than T, the algorithm is guaranteed to have found the top-k objects and terminates. For the case where only sorted accesses are possible, [15] presents an optimized implementation of the *No-Random accesses Algorithm* (NRA), originally proposed also in [5]. The top-k results are incrementally fetched based on their aggregate scores. [34] presents a framework for top-k queries on top of relations having multi-attribute indices. An *index-merge* paradigm is proposed to merge multiple index nodes progressively and selectively. Recently, there has also been work for multiple attributes in top-k ranking criteria. For example, [21] studies the semantic based spatial keyword querying, which finds the k objects most similar to the query, subject to their spatial, textual and semantic meaning properties. To this end, the authors propose hierarchical indexing structures to integrate all types of involved information and devise appropriate pruning techniques.

Top-k Joins. Natsev et al. [16] first studied the top-k join evaluation proposing multi-way join operator J*. Objects are accessed incrementally from the input streams (e.g., in round-robin) sorted by their scores. Partial join results are computed, and at each step, the top partial combination is completed by filling the missing values from the streams. J* incrementally outputs the top combinations in the heap if they are complete join results. As a follow-up to [7], Li et al. [11] applied HRJN* on multiple inputs with one or more scoring attributes each. Rank joins with multiple inputs and more than one scoring attributes were also covered by PBRJ in [27]. Further, [4, 25, 32] targeted top-k joins on inputs from different physical locations. Wu et al. [32] model this as a graph problem solved by a branch-and-bound algorithm that minimizes the number of network accesses. In contrast, [4, 25] determined the number of objects to be accessed from each network input, through the depth estimation procedure. Last, Ntarmos et al [17] studied top-k joins in NoSQL databases using statistical structures (similar to 2-dimensional histograms) to reduce object accesses in a distributed environment. In contrast, our focus is on the centralized scenario.

String Similarity Joins. A *string similarity* join finds all pairs of similar string objects based on a similarity measure, e.g., the Edit distance, and a threshold ϵ. Most of the existing solutions [1, 6, 12, 13, 22, 26, 30, 31, 33, 36, 37] employ a filter-and-refinement evaluation framework. In addition, some of these methods [13, 22, 31, 37] follow a string partition framework and so, the performance of the algorithm is largely determined by the number of partitions generated for each string, and the number of index probes to search for similar strings. Recent surveys on the string similarity join methods can be found in [9, 29, 35].

Gravano et al. [6] proposed string join techniques on top of commercial databases. By matching *q-grams* and taking into account both positions and total number of matches, several techniques were proposed to prune string pairs not within the desired Edit distance. Xiao et al. proposed ED-Join [33], which is also a q-gram-based method but enhances the filter process using Edit distance lower bounds derived from the location-based and content-based q-grams *mismatching*. Qin et al. [22] also used q-grams. QChunk first obtains a global q-grams order and then partitions each string into a set of chunks; the first $\epsilon + 1$ out of which (according to the global order) are stored in a hash table, where ϵ is the Edit distance join threshold. For each string s the algorithm probes the hash table using a subset of s q-grams according to the global order. Trie-Join [30] is a trie index based framework which employs the prefix filtering [3] to generate similar string pairs without the need for a refinement step. However, Trie-Join is only efficient for short strings. PETER [23] is a prefix tree based indexing algorithm for approximate search and joins. It combines an efficient implementation of compressed prefix trees with advanced pre-filtering techniques that exclude many candidate strings early. Wang et al. [31] proposed VChunk which partitions each string into at least $2 \cdot \epsilon + 1$ chunks of possibly different lengths, determined by a chunk boundary dictionary (CBD). Computing the optimal CBD is an NP-hard problem and so the authors proposed a greedy algorithm. Zhang and Zhang [36] proposed EmbedJoin which first embeds each string from the Edit distance metric space to the Hamming distance metric space. Then, it uses Locality Sensitive Hashing to compute (approximate) similarity joins in the Hamming space. Last, [37] proposed MinJoin, a randomized and approximate algorithm

which partitions each string into a set of substrings, and then uses hash join on these substrings to find all pairs of strings that share at least one common substring.

In this work, we built on Pass-Join [13] as experimental studies [9, 29] showed that it is the most efficient *exact* algorithm for both short and long strings.

6 CONCLUSIONS

In this paper, we extended the top-k join ranking operation in case of string objects, introducing the top-k string similarity join (k-SSJoin). For its efficient evaluation, we investigated the merit of existing top-k join solutions for non-relational complex data types. We considered SFA which prioritizes the ranking component of a k-SSJoin query, DFA which prioritizes the join component and BA which acts a hybrid that combines the two methods under a block-based instead of an object-based accessing and evaluation paradigm. We redesigned their core procedure to determine similar string objects building on top of the state-of-the-art string similarity join algorithm Pass-Join. At the same time, we proposed optimizations and indexing to avoid computing the complete result of the string join. Our extensive experimental analysis on both real-world and synthetic data showed the effectiveness of our methodologies and proved that similar to the case of top-k on spatial objects, the block-based evaluation of BA clearly outperforms both SFA and DFA.

REFERENCES

[1] Thomas Bocek, Ela Hunt, and Burkhard Stiller. 2007. *Fast Similarity Search in Large Dictionaries.* Technical Report. University of Zurich, Department of Informatics.

[2] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. 1993. Efficient Processing of Spatial Joins Using R-Trees. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, DC, USA, May 26-28, 1993.* 237–246.

[3] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA.* 5.

[4] Christos Doulkeridis, Akrivi Vlachou, Yannis Kotidis, and Neoklis Polyzotis. 2012. Processing of Rank Joins in Highly Distributed Systems. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012.* 606–617.

[5] Ronald Fagin, Amnon Lotem, and Moni Naor. 2001. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 21-23, 2001, Santa Barbara, California, USA.* 102–113.

[6] Luis Gravano, Panagiotis G. Ipeirotis, H. V. Jagadish, Nick Koudas, S. Muthukrishnan, and Divesh Srivastava. 2001. Approximate String Joins in a Database (Almost) for Free. In *VLDB.* 491–500.

[7] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. 2003. Supporting Top-k Join Queries in Relational Databases. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases, September 9-12, 2003, Berlin, Germany.* 754–765.

[8] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.* 40, 4 (2008).

[9] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String Similarity Joins: An Experimental Evaluation. *PVLDB* 7, 8 (2014), 625–636.

[10] Thorsten Joachims, Laura A. Granka, Bing Pan, Helene Hembrooke, Filip Radlinski, and Geri Gay. 2007. Evaluating the accuracy of implicit feedback from clicks and query reformulations in Web search. *ACM TOIS* 25, 2 (2007).

[11] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. 2005. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005.* 131–142.

[12] Chen Li, Jiaheng Lu, and Yiming Lu. 2008. Efficient Merging and Filtering Algorithms for Approximate String Searches. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico.* 257–266.

[13] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. 2011. Pass-join: a partition-based method for similarity joins. *PVLDB* 5, 3 (2011), 253–264.

[14] Vebjorn Ljosa and Ambuj K. Singh. 2008. Top-k Spatial Joins of Probabilistic Objects. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico.* 566–575.

[15] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W. Cheung. 2007. Efficient top-k aggregation of ranked inputs. *ACM TODS* 32, 3 (2007).

[16] Apostol Natsev, Yuan Chi Chang, John R. Smith, Chung Sheng Li, and Jeffrey Scott Vitter. 2001. Supporting incremental join queries on ranked inputs. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy.* 281–290.

[17] Nikos Ntarmos, Ioannis Patlakas, and Peter Triantafillou. 2014. Rank Join Queries in NoSQL Databases. *PVLDB* 7, 7 (2014), 493–504.

[18] Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao. 2001. Efficient OLAP Operations in Spatial Data Warehouses. In *Advances in Spatial and Temporal Databases, 7th International Symposium, SSTD 2001, Redondo Beach, CA, USA, July 12-15, 2001, Proceedings.* 443–459.

[19] Shuyao Qi, Panagiotis Bouros, and Nikos Mamoulis. 2013. Efficient Top-k Spatial Distance Joins. In *Advances in Spatial and Temporal Databases - 13th International Symposium, SSTD 2013, Munich, Germany, August 21-23, 2013. Proceedings.* 1–18.

[20] Shuyao Qi, Panagiotis Bouros, and Nikos Mamoulis. 2020. Top-k Spatial Distance Joins. *GeoInformatica* (2020).

[21] Zhihu Qian, Jiajie Xu, Kai Zheng, Pengpeng Zhao, and Xiaofang Zhou. 2018. Semantic-aware Top-k Spatial Keyword Queries. *World Wide Web* 21, 3 (May 2018), 573–594.

[22] Jianbin Qin, Wei Wang, Yifei Lu, Chuan Xiao, and Xuemin Lin. 2011. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD.* 1033–1044.

[23] Astrid Rheinländer, Martin Knobloch, Nicky Hochmuth, and Ulf Leser. 2010. Prefix Tree Indexing for Similarity Search and Similarity Joins on Genomic Data. In *Scientific and Statistical Database Management, 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30 - July 2, 2010. Proceedings.* 519–536.

[24] Richard J. Roberts, Mauricio O. Carneiro, and Michael C. Schatz. 2013. The advantages of SMRT sequencing. *Genome Biology* 14, 6 (2013), 405.

[25] Mei Saouk, Christos Doulkeridis, Akrivi Vlachou, and Kjetil Nørvåg. 2016. Efficient processing of top-k joins in MapReduce. In *2016 IEEE International Conference on Big Data, BigData 2016, Washington DC, USA, December 5-8, 2016.* 570–577.

[26] Sunita Sarawagi and Alok Kirpal. 2004. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004.* 743–754.

[27] Karl Schnaitter and Neoklis Polyzotis. 2010. Optimal algorithms for evaluating rank joins in database systems. *ACM TODS* 35, 1 (2010).

[28] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. 2005. *Database System Concepts, 5th Edition.* McGraw-Hill Book Company. I–XXVI, 1–1142 pages.

[29] Sebastian Wandelt, Dong Deng, Stefan Gerdjikov, Shashwat Mishra, Petar Mitankin, Manish Patil, Enrico Siragusa, Alexander Tiskin, Wei Wang, Jiaying Wang, and Ulf Leser. 2014. State-of-the-art in string similarity search and join. *SIGMOD Record* 43, 1 (2014), 64–76.

[30] Jiannan Wang, Jianhua Feng, and Guoliang Li. 2010. Trie-join: efficient trie-based string similarity joins with edit-distance constraints. *PVLDB* 3, 1-2 (2010), 1219–1230.

[31] Wei Wang, Jianbin Qin, Chuan Xiao, Xuemin Lin, and Heng Tao Shen. 2013. VChunkJoin: An Efficient Algorithm for Edit Similarity Joins. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1916–1929.

[32] Minji Wu, Laure Berti-Équille, Amélie Marian, Cecilia M. Procopiuc, and Divesh Srivastava. 2010. Processing Top-k Join Queries. *PVLDB* 3, 1-2 (2010), 860–870.

[33] Chuan Xiao, Wei Wang, and Xuemin Lin. 2008. Ed-Join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB* 1, 1 (2008), 933–944.

[34] Dong Xin, Jiawei Han, and Kevin C. Chang. 2007. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007.* 103–114.

[35] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers Comput. Sci.* 10, 3 (2016), 399–417.

[36] Haoyu Zhang and Qin Zhang. 2017. EmbedJoin: Efficient Edit Similarity Joins via Embeddings. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Halifax, NS, Canada, August 13 - 17, 2017.* 585–594.

[37] Haoyu Zhang and Qin Zhang. 2019. MinJoin: Efficient Edit Similarity Joins via Local Hash Minima. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019.* 1093–1103.

[38] Keping Zhao, Shuigeng Zhou, Kian-Lee Tan, and Aoying Zhou. 2005. Supporting Ranked Join in Peer-to-Peer Networks. In *16th International Workshop on Database and Expert Systems Applications (DEXA'05).* 796–800.