Similarity Joins: Their Implementation and Interactions with Other Database Operators

Yasin N. Silva, Spencer S. Pearson, Jaime Chon, Ryan Roberts

Arizona State University, 4701 W. Thunderbird Road, Glendale, AZ 85306, USA

Abstract

Similarity Joins are extensively used in multiple application domains and are recognized among the most useful data processing and analysis operations. They retrieve all data pairs whose distances are smaller than a predefined threshold ε. While several standalone implementations have been proposed, very little work has addressed the implementation of Similarity Joins as physical database operators. In this paper, we focus on the study, design, implementation, and optimization of a Similarity Join database operator for metric spaces. We present *DBSimJoin*, a physical database operator that integrates techniques to: enable a non-blocking behavior, prioritize the early generation of results, and fully support the database iterator interface. The proposed operator can be used with multiple distance functions and data types. We describe the changes in each query engine module to implement DBSimJoin and provide details of our implementation in PostgreSQL. We also study ways in which DBSimJoin can be combined with other similarity and non-similarity operators to answer more complex queries, and how DBSimJoin can be used in query transformation rules to improve query performance. The extensive performance evaluation shows that DBSimJoin significantly outperforms alternative approaches and scales very well when important parameters like ε, data size, and number of dimensions increase.

Keywords: Similarity join, database operator, similarity queries, PostgreSQL, query processing and optimization

1. Introduction

It is widely recognized that the move from exact semantics of data and Boolean semantics of queries to imprecise and approximate semantics of data and queries is one of the key paradigm shifts in data management. This shift is fueled in part by the recognition that many application scenarios can significantly benefit from the identification of similarities in the data. One of the most useful similarity-aware data analysis operations is the Similarity Join (SJ), which retrieves all data pairs whose distances are smaller than a predefined threshold ε. Similarity Joins have been studied and extensively used in multiple application domains, e.g., record linkage, data cleaning, multimedia applications, sensor networks, marketing analysis, etc. Several Similarity Join algorithms and implementation techniques have been previously proposed. They range from out-of-database approaches for only in-memory or external memory data, to techniques that use standard database operators to answer Similarity Joins. Very little work, however, has addressed the implementation of Similarity Join as a first-class database operator. This type of implementation would enable interesting similarity queries that combine SJ with other operators. In this paper, we present a generic Similarity Join database operator for any dataset that lies in a metric space. The main contributions of this paper are:

- We present *DBSimJoin*, a physical Similarity Join database operator that is fully integrated into the database engine and incorporates techniques to: (1) enable a non-blocking behavior, (2) prioritize the early generation of results, and (3) fully support the database iterator interface and its functions *open*, *getNext*, and *close*.
- To the best knowledge of the authors, DBSimJoin is the first Similarity Join database operator that is general enough to be used with any dataset that lies in a metric space. The operator can be used with various distance functions and data types (vectors, text, etc.).
- We present multiple guidelines to implement DBSimJoin as an integrated component of a database system. We also provide details of our implementation in PostgreSQL [1], a popular open-source database system.
- We thoroughly evaluate the performance and scalability properties of DBSimJoin with synthetic and real-world data. Our evaluation uses multiple data types and distance functions. We show that DBSimJoin significantly outperforms alternative approaches and that it scales well when key parameters like ε and data size increase.
- We show that DBSimJoin can be combined with other operators in complex similarity queries and can be used in important query transformation rules that enable cost-based query optimization, e.g., pushing selection below join and associativity of SJ operators.

Email addresses: `ysilva@asu.edu` (Yasin N. Silva), `sspearso@asu.edu` (Spencer S. Pearson), `jchon@asu.edu` (Jaime Chon), `rwrobert@asu.edu` (Ryan Roberts)

The implementation of Similarity Joins as first-class database operators have the following key advantages: (1) SJ database operators can be interleaved with other regular and similarity-aware operators and their results pipelined for further processing; (2) important optimization techniques, e.g., pushing certain filtering operators to lower levels of the execution plan, pre-aggregation, and the use of materialized views can be extended to the new operators; and (3) the implementation of these operators can reuse and extend other operators and structures to handle large datasets, and use the cost-based query optimizer machinery to enhance query execution time.

The remaining part of this paper is organized as follows. Section 2 presents the related work. Section 3 describes in detail the DBSimJoin operator and the techniques to implement it. The performance evaluation is presented in Section 4. Section 5 presents the conclusions and future research directions.

2. Related Work

Significant work has been carried out on the study of Similarity Joins. This work introduced the semantics of several Similarity Join variants and proposed techniques to implement them primarily as standalone operations outside of a database system.

Several types of Similarity Join have been proposed in the literature, e.g., distance range join (retrieves all pairs whose distances are smaller than a predefined threshold ε) [2, 3, 4, 5, 6, 7], k-Distance join (retrieves the k most-similar pairs) [8], and kNN-join (retrieves, for each tuple in one table, the k nearest-neighbors in another table) [9, 10, 11]. The distance range join has been one of the most studied and useful types of Similarity Join. This type of join is commonly referred to simply as Similarity Join and is the focus of this paper. Among its most relevant implementation techniques, we find approaches that rely on the use of pre-built indices, e.g., eD-index [3], D-index [4], and List of Twin Clusters (LTC) [12]. These techniques strive to partition the data while clustering together the similar objects. While these indexing techniques support the SJ operation they also have some shortcomings: D-index and eD-index may require rebuilding the index to support queries with different ε, eD-index is applicable only to the case of self-joins, and LTC requires indexing each pair of input sets jointly. Several non-index-based techniques have also been proposed to solve the Similarity Join problem. EGO, GESS, and QuickJoin are three of the most relevant non-index-based algorithms. The Epsilon Grid Order (EGO) algorithm [5] imposes an epsilon-sized grid over the space and uses an efficient schedule of block reads to minimize I/O. The Generic External Space Sweep (GESS) algorithm [6] creates hypersquares centered on each data point with epsilon length sides, and joins these hypersquares using a spatial join on rectangles. The Quickjoin algorithm [7] recursively partitions the data until the subsets are small enough to be efficiently processed using a nested loop join. Quickjoin has been shown to outperform EGO and GESS [7]. DBSimJoin, the operator presented in this paper, builds on Quickjoin's approach to partition the data. However, the focus of our work is the design and implementation of an efficient database operator. The differences with the work in [7] are: (1) DBSimJoin uses a different partitioning sequence that prioritizes early generation of results and minimizes query response time, (2) DBSimJoin uses a non-blocking implementation approach that fully supports the database iterator interface, (3) DBSimJoin assumes a limited number of memory buffers, (4) our experimental section evaluates the effect on performance of key parameters not evaluated in [7], e.g., dimensionality and number of pivots, and (5) we study how DBSimJoin can be combined with other operators and used in query transformation rules.

Also, of importance is the work on Similarity Join techniques in the context of database systems. Some work has focused on the implementation of Similarity Joins using standard database operators [13, 14, 15]. These techniques are applicable only to string or set-based data. The general approach pre-processes the data and query, e.g., decomposes data and query strings into sets of grams (substrings of a string that are used as its signature), and stores the results of this stage on separate relational tables. Then, the result of the Similarity Join can be obtained using standard SQL queries. Indices on the pre-processed data are used to improve performance. DBSimJoin is experimentally compared with one such technique (SSJoin [13]) in Section 4.2. More recently, the work in [16, 17] proposed a SJ database operator for 1D numerical data based on a plane-sweep algorithm. This approach, however, cannot be easily extended to other data types. DBSimJoin is more generic and can be used with any dataset that lies in a metric space. DBSimJoin supports a variety of data types, e.g., numerical data, vector data, text, etc., and distance functions, e.g., Euclidean distance, Edit distance, etc. In a recent demonstration paper [18], we showed how DBSimJoin can be used to identify similar images (feature vectors), and similar publications in a bibliographic database.

Some recent work in the area of Similarity Joins has focused on: proposing a compact way to represent the output of this operation, i.e., reporting groups of nearby points instead of every join link [19], efficient algorithms for in-memory Similarity Join with edit distance constraints [20], algorithms for near duplicate detection that exploit the ordering of tokens in a record to reduce the number of distance computations [21], Similarity Join algorithms that exploit the capabilities of graphics processing units [22], and the implementation of highly distributed Similarity Join algorithms [23, 24, 25, 26, 27].

An earlier version of this paper appeared in [28]. This paper extends on [28] by integrating: (1) an extended experimental section including: (a) the evaluation of associativity of SJ operators, (b) the analysis of Lazy/Eager aggregation transformations with SJ and similarity grouping, (c) the evaluation of query performance while increasing ε and the number of dimensions, and (d) the analysis of increasing scale factor and ε with a real vector dataset; (2) the pseudo-code of several DBSimJoin algorithms including: *PartitionBasePart*, *PartitionWinPairPart*, *DBSimJoin_Open*, and *DBSimJoin_Close*; (3) a more detailed description of the algorithms and techniques to implement DBSimJoin as a database operator; and (4) additional diagrams and extended discussions throughout the paper.

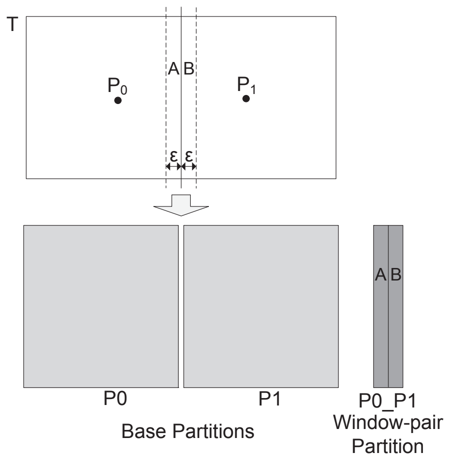


Figure 1: Repartitioning a base partition.

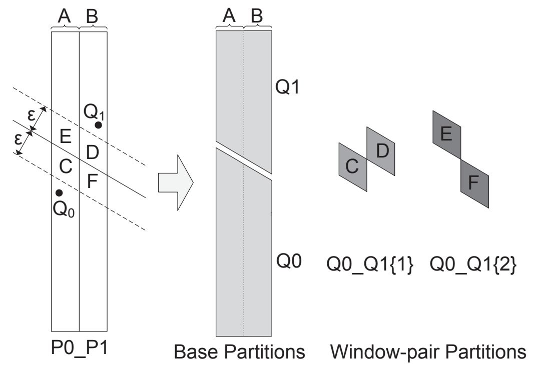


Figure 2: Repartitioning a window-pair partition.

3. The DBSimJoin Operator

The Similarity Join (SJ) operation between two datasets R and S is defined as follows:

$$R \bowtie_{\theta_\varepsilon(r,s)} S = \{\langle r, s\rangle | \theta_\varepsilon(r, s), r \in R, s \in S\},$$

where $\theta_\varepsilon(r, s)$ represents the Similarity Join predicate, i.e., $dist(r, s) \leq \varepsilon$.

The DBSimJoin operator presented in this section identifies all the pairs, i.e., links, that belong to the result of the Similarity Join operation. Furthermore, the operator can be used with any dataset that lies in a metric space. Even though the tuples of relations R and S are combined by DBSimJoin, each tuple is assumed to have an attribute that identifies its relation.

DBSimJoin iteratively partitions the input data into smaller partitions until each partition is small enough to be efficiently processed by an in-memory Similarity Join routine. The overall process is divided into a sequence of rounds. The initial round partitions the input data while any subsequent round partitions the data of a previously generated partition. Each round produces: (1) result pairs (links) for the small partitions that can be processed by an in-memory SJ routine, and (2) intermediate data for the partitions that will require further partitioning. Intermediate data is stored on disk (hibernated). The DBSimJoin operator executes the required rounds until all the input and intermediate data is processed. While rounds other than the first one can be processed in any order, DBSimJoin uses a partitioning sequence that favors the early generation of result links.

3.1. Partitioning in DBSimJoin

The core goal of the partitioning step in each round is to divide the round input data into a set of partitions such that all the result links in the input data are obtained by combining the links found in each partition independently. To accomplish this, the input data is partitioned into: (1) non-overlapping partitions (*base partitions*), and (2) partitions that contain the records in the boundary of each pair of base partitions (*window-pair partitions*). After the partitioning step, each generated partition can be processed independently.

Data partitioning is performed using a set of k pivots, i.e., a random subset of the data records to be partitioned. We use random selection since this method was found to be efficient in [7]. Each base partition contains all the records that are closer to a given pivot than to any other pivot. Each window-pair partition contains the records in the boundary between two base partitions. The window-pair records should be a superset of the records whose distance to the hyperplane that separates the base partitions is at most ε. This hyperplane does not always explicitly exist in a metric space. Instead, the hyperplane is implicit and known as a *generalized hyperplane*. Since the distance of a record t to the generalized hyperplane between two partitions with pivots P_0 and P_1 cannot always be computed exactly, a lower bound of the distance is used [29]:

$$gen\_hyperpln\_dist(t, P_0, P_1) = (dist(t, P_0) - dist(t, P_1))/2.$$

This distance is replaced with an exact distance if this can be computed, e.g., in Euclidean spaces.

Processing the window-pair partitions guarantees the identification of the links between records that belong to different base partitions. A round that repartitions a base partition or the initial input data is referred to as a *base partition round*, a round that repartitions a window-pair partition is referred to as a *window-pair partition round*.

Fig. 1 represents the repartitioning of a base partition using pivots P_0 and P_1. In this case, the result of the Similarity Join operation on the input dataset T is the union of the links in partitions $P0$ and $P1$, and the links in window-pair partition $P0\_P1$ where one element belongs to window A and the other one to window B. We refer to this last type of link as *window link*. Fig. 2 represents the repartitioning of the window-pair partition $P0\_P1$ of Fig. 1 using pivots Q_0 and Q_1. In this case, the set of window links in $P0\_P1$ is the union of the window links in $Q0$, $Q1$, $Q0\_Q1\{1\}$ and $Q0\_Q1\{2\}$. Note that windows C and F do not form a window-pair partition since their window links are a subset of the window links in $Q0$. Similarly, the window links between E and D are a subset of the window links in $Q1$.

3.2. DBSimJoin Rounds

Fig. 3 shows an example of the multiple rounds that are executed by the DBSimJoin operator. Each node in the tree with

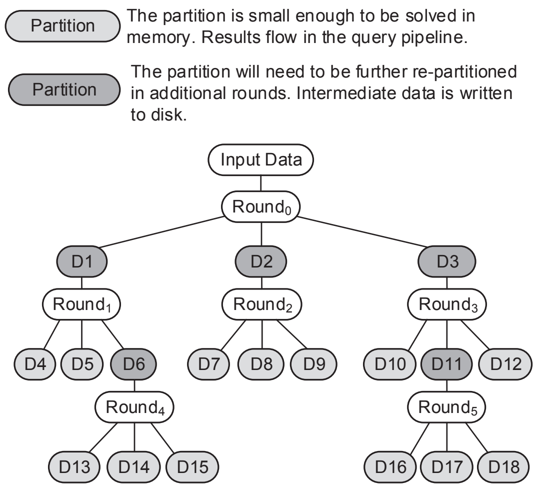

Partition
The partition is small enough to be solved in memory. Results flow in the query pipeline.
Partition
The partition will need to be further re-partitioned in additional rounds. Intermediate data is written to disk.
Input Data
$Round_0$
D1
D2
D3
$Round_1$
$Round_2$
$Round_3$
D4
D5
D6
D7
D8
D9
D10
D11
D12
$Round_4$
$Round_5$
D13
D14
D15
D16
D17
D18

Figure 3: Example of the rounds and partitions generated by DBSimJoin.

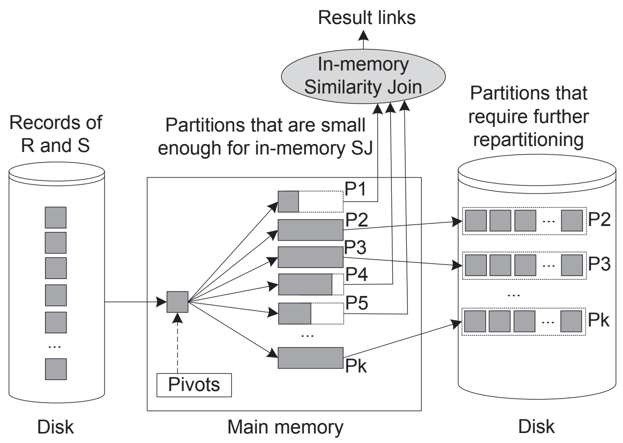

Result links
In-memory Similarity Join
Records of R and S
Partitions that are small enough for in-memory SJ
Partitions that require further repartitioning
P1
P2
P3
P4
P5
Pk
Pivots
Disk
Main memory
Disk

Figure 4: Round 0.

name $Round_N$ represents a round. This figure also shows the partitions generated by each round. Light gray partitions are small partitions that are processed running an in-memory Similarity Join routine. Dark gray partitions are partitions that require additional repartitioning. A sample sequence of rounds can be: $Round_0$, $Round_1$, $Round_2$, $Round_3$, $Round_4$ and $Round_5$. The original input data is always processed in the first round.

Figures 4 and 5 graphically represent the processing performed by DBSimJoin in round 0 and a generic round I, respectively. Round 0, shown in Fig. 4, partitions the original input data ($R \cup S$) into k partitions. Some generated partitions are small enough to be processed by the in-memory SJ routines, e.g., $P1$, $P4$, $P5$. Result links and window links are generated in these routines. The remaining partitions are stored on disk, e.g., $P2$, $P3$, Pk. Any other round further repartitions a previously generated partition. For instance the round represented in Fig. 5 repartitions partition $P2$. This round also generates some partitions that can be processed by the in-memory SJ routines, e.g., $Q1$, $Q3$, $Q4$, $Q5$, and partitions that need to be stored on disk for further processing, e.g., $Q2$, Qk.

While rounds other than the first one can be processed in any order, DBSimJoin uses a partitioning sequence that favors the early generation of result links. The algorithmic details of this approach are presented in Section 3.5.1.

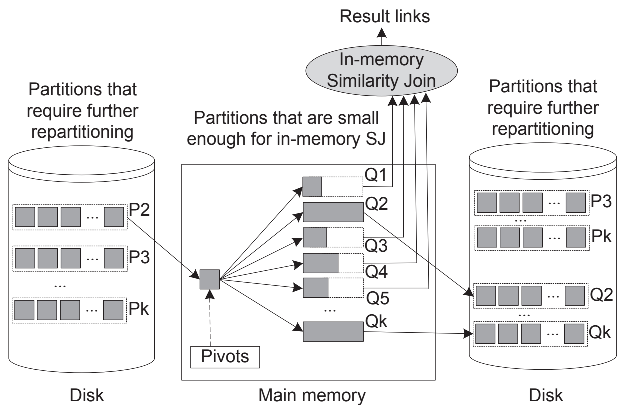

Result links
In-memory Similarity Join
Partitions that require further repartitioning
Partitions that are small enough for in-memory SJ
Partitions that require further repartitioning
P2
P3
Pk
Q1
Q2
Q3
Q4
Q5
Qk
Pivots
P3
Pk
Q2
Qk
Disk
Main memory
Disk

Figure 5: Round I.

The remaining part of this section presents the guidelines to implement the DBSimJoin operator inside the query engine of standard DBMSs. Although the presentation is intended to be applicable to any DBMS, some specific details refer to our implementation in PostgreSQL [1]. One of the goals of the implementation is to reuse and extend already available routines and structures to minimize the effort needed to realize the operator.

3.3. The Parser

To add support for Similarity Joins in the parser, the raw-parsing grammar rules, e.g., *yacc* rules in the case of PostgreSQL, are extended to recognize the syntax of the new similarity join predicate. The parse-tree and query-tree data structures are extended to include the information of the new operator, i.e., type of join, value of ε and distance function. The routines in charge of transforming the parse tree into the query tree are updated accordingly to process the new fields in the parse tree. In our implementation, we support the following Similarity Join syntax.

```
SELECT R.r, S.s FROM R, S
   WHERE R.r WITHIN <epsilon> OF S.s
   USING <dist_function>
```

The values of $R.r$ and $S.s$ can be, in general, of any data type, e.g., strings, numbers, vectors, etc. *dist_function* specifies the distance function to be used by the SJ operation, e.g., Euclidean distance, Edit distance, etc.

3.4. The Planner

To add support for the operator in the planner, a new plan node is created to represent the Similarity Join operator. This node is similar to the regular join node but also stores information about ε and the distance function. If a query has multiple Similarity Join predicates, they are processed one at a time, i.e., multiple Similarity Join nodes are pipelined. The routines that find the similarity links at every Similarity Join node are presented in Section 3.5. It is important to observe that key

Algorithm 1 *DBSimJoin*(R, S, *eps*, *numPiv*, *memT*)

```
Input: R and S (input datasets), eps (epsilon), numPiv (number of pivots), memT (memory threshold)
Output: all the results of the Similarity Join operation $R \bowtie_{\theta_\varepsilon(r,s)} S$
 1. create basePList and winPairPList
 2. PartitionBasePart($R \cup S$, basePList, winPairPList, eps, numPiv)
 3. while basePList.size> 0 do
 4.   for each partition P of basePList do
 5.     if $P \leq memT$ then
 6.       $InmemorySimJoin(P, eps)$
 7.     else
 8.       $HibernatePartition(P)$
 9.     end if
10.   end for
11.   while winPairPList.size> 0 do
12.     for each partition W of winPairPList do
13.       if $W \leq memT$ then
14.         $InmemorySimJoinWin(W, eps)$
15.       else
16.         $HibernatePartition(W)$
17.       end if
18.     end for
19.     if winPairPList.size> 0 then
20.       W ← winPairPList.getFirst()
21.       PartitionWinPairPart(W, winPairPList, eps, numPiv)
22.     end if
23.   end while
24.   if basePList.size> 0 then
25.     P ←basePList.getFirst()
26.     PartitionBasePart(P, basePList, winPairPList, eps, numPiv)
27.   end if
28. end while
29. delete basePList and winPairPList
```

Algorithm 2 *PartitionBasePart*(*basePart*, *basePList*, *winPairPList*, *eps*, *numPiv*)

```
Input: basePart (data to be partitioned), basePList (list of base partitions), winPairPList (list of window-pair partitions), eps (epsilon), numPiv (number of pivots)
Output: partitions basePart and updates basePList and winPairPList accordingly
1. Pick numPiv tuples from basePart as pivots
2. Add new empty partitions to basePList and winPairPList
3. for each tuple t in basePart do
4.   Add t to proper new partitions of basePList and winPairPList
5. end for
6. Remove basePart from basePList
```

Algorithm 3 *PartitionWinPairPart*(*winPairPart*, *winPairPList*, *eps*, *numPiv*)

```
Input: winPairPart (data to be partitioned), winPairPList (list of window-pair partitions), eps (epsilon), numPiv (number of pivots)
Output: partitions winPairPart and updates winPairPList accordingly
1. Pick numPiv tuples from winPairPart as pivots
2. Add new empty partitions to winPairPList
3. for each tuple t in winPairPart do
4.   Add t to proper new partitions of winPairPList
5. end for
6. Remove winPairPart from winPairPList
```

transformation rules to optimize queries with Similarity Joins [16, 30], e.g., associativity of SJ operators and pushing selection below SJ, can be applied to plans with DBSimJoin operators. We evaluate the use of several transformation rules with DBSimJoin in Section 4.5.

3.5. The Executor

This section presents the general executor algorithm of DBSimJoin as well as specific details of its implementation using the iterator interface.

3.5.1. DBSimJoin Executor Routine

The main executor routine of the DBSimJoin operator is presented in Algorithm 1. The routine first creates two lists that will keep track of the base and window-pair partitions (line 1). Each partition is assigned a certain space in memory (*memT*). If a partition needs to grow beyond the assigned space, the partition is stored on disk and the memory space assigned to this partition is used as a buffer. The routine partitions the initial input data ($R \cup S$) into base and window-pair partitions (line 2). The main loop in the algorithm will be executed while there is at least one base partition that needs to be processed (lines 3 to 28). In each iteration, the routine processes all the base partitions executing *InmemorySimJoin* to identify SJ links in small partitions (line 6) and hibernating larger partitions, i.e., transferring any in-memory data to disk (line 8). Then, the routine processes the window-pair partitions (and their sub-partitions) until all their SJ links have been produced (lines 11 to 23). The routine iteratively (1) processes all the current window-pair partitions executing *InmemorySimJoinWin* in the case of small partitions (line 14) and hibernating larger partitions (line 16), and (2) gets the first window-pair partition that needs further processing and repartitions it calling *PartitionWinPairPart* (lines 19 to 22). When all the window-pair partitions have been fully processed, the routine gets the first base partition that needs further processing and repartitions it calling *PartitionBasePart* (lines 24 to 27). After this step, the main while loop iterates again.

The main DBSimJoin routine prioritizes the early generation of links. After any partitioning step, the algorithm will process first all the partitions that can be solved in-memory. The routine has the potential to produce result links starting at the first round. This behavior enables the support of the iterator interface and its *getNext* function. The algorithm also prioritizes

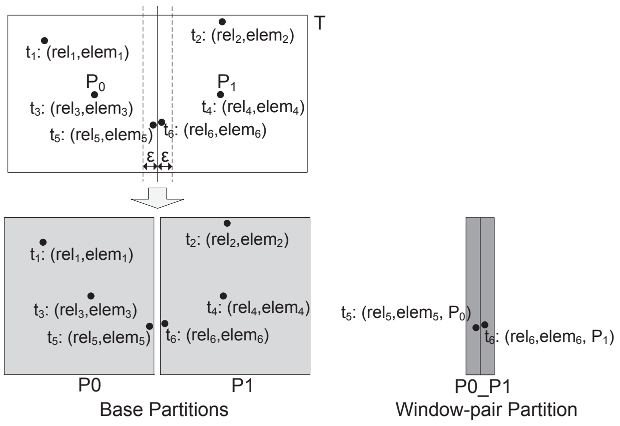


Figure 6: Partitioning the tuples of a base partition.

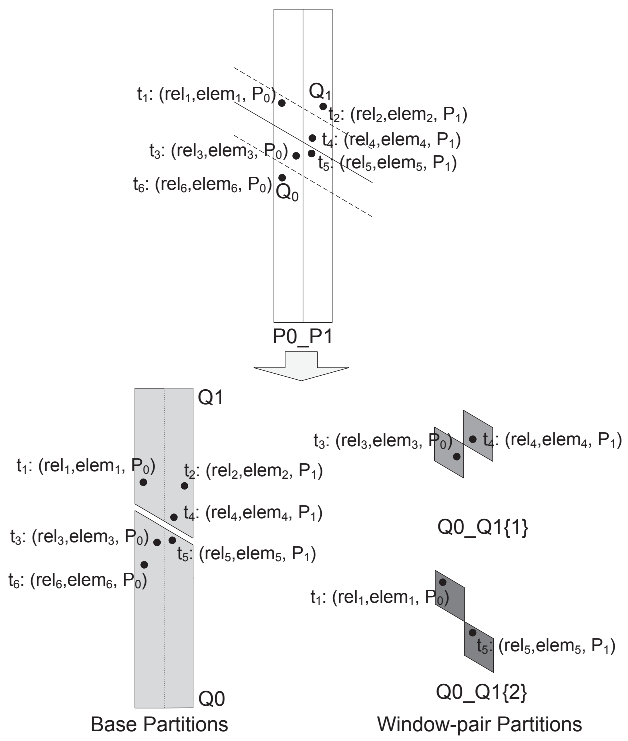


Figure 7: Partitioning the tuples of a window-pair partition.

the processing of window-pair partitions before base partitions. This is done to reduce the number of partitions that the routine needs to keep track of. Window-pair partitions are in general significantly smaller than base partitions. Consequently, in general, it takes less time to reach the point where they can be processed in memory.

InmemorySimJoin and *InmemorySimJoinWin* are in-memory routines to find the links and window links, respectively. These algorithms iteratively partition the data in memory until the partitions are small enough to be quickly solved using a nested loop join (NLJ). As explained in Section 3.5.2, both routines are implemented using a non-blocking approach to minimize the time to produce the next result link. The nested loop join is applied only over very small partitions (up to 20 tuples in our experiments). NLJ can also be replaced by adaptations of other algorithms to identify SJ links on small datasets, e.g., the techniques recently proposed in [31]. One of these techniques proposes the use of indexing structures with subquadratic construction cost. The adaptation and integration of these techniques in PostgreSQL is a task for future work.

The main DBSimJoin routine calls *PartitionBasePart* and *PartitionWinPairPart* to partition a base and a window-pair partition, respectively.

The *PartitionBasePart* routine is presented in Algorithm 2. This routine randomly selects *numPiv* pivots that will be used to partition *basePart* and adds the partitions to the lists of base and window-pair partitions (line 1 to 2). Then, the routine processes each tuple *t* of *basePart* and adds it to the new base (*basePList*) and window-pair (*windowPairPList*) partitions this tuple belongs to (line 4). This task involves two steps: (1) *t* is added to the base partition corresponding to its closest pivot *p*, and (2) *t* is also added to all the window-pair partitions (corresponding to pivots *p* and *i*) where $gen\_hyperpln\_dist(t, p, i) \leq eps$. Finally, the routine removes *basePart* from the list of base partitions (line 6).

Fig. 6 shows an example of the partitions generated by *PartitionBasePart* using pivots P_0 and P_1. Region T contains all the tuples of the dataset to be repartitioned. The figure shows the three generated partitions, i.e., $P0$, $P1$ and $P0\_P1$. Each input tuple belongs to its associated base partition ($P0$ or $P1$). Additionally, each tuple in the windows between the two base partitions, e.g., t_5 and t_6, belongs to the window-pair partition $P0\_P1$. The base partitions, i.e., $P0$, $P1$, are added to *basePList* and the window-pair partition is added to *winPairPList*. Note that the tuples of the window-pair partition are extended with an additional attribute that specifies their previous partition. This attribute is used during the generation of window links and also to correctly repartition this partition if needed.

Algorithm 3 presents the *PartitionWinPairPart* routine. This routine is similar to *PartitionBasePart*. The main difference is in the way tuples are added to the different new partitions (line 4). Each tuple *t* is added to its base partition and all the window-pair partitions it belongs to. The routine, however, distinguishes between the two window-pair partitions of any pair of pivots. The correct identification of the window that a tuple belongs to can be obtained using *gen_hyperpln_dist* and the previous partition attribute of the tuple.

Additionally, in *PartitionWinPairPart*, all the generated partitions (new base and window-pair partitions) are added to *winPairPList*. This is the case because the input partition is a window-pair partition. All the links generated in a window-pair partition or in any of its generated subpartitions should always be window links, i.e., links between tuples of different previous partitions.

Fig. 7 shows an example of the partitions generated by *PartitionWinPairPart*. In this example, the input partition $P0\_P1$ is partitioned into four partitions, i.e., $Q0$, $Q1$, $Q0\_Q1\{1\}$, $Q0\_Q1\{2\}$. The *PartitionWinPairPart* routine will add all these partitions to *winPairPList*.

Algorithm 4 *DBSimJoin_Open*()

```
1. create basePList and winPairPList
2. numPiv ←CalcNumPivots(availableMem, memT)
3. initialize other conventional DB structures
```

Algorithm 5 *DBSimJoin_Close*()

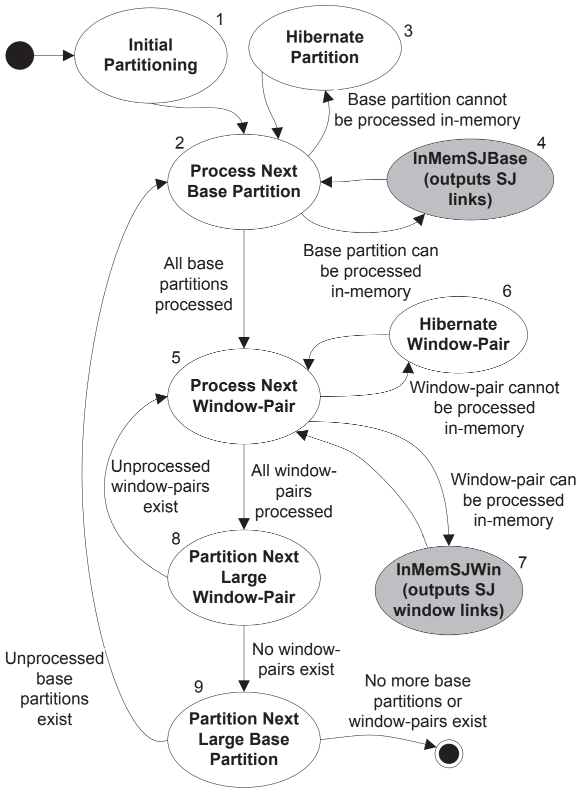
```
1. delete basePList
2. delete winPairPList
3. delete other conventional DB structures
```

3.5.2. Implementation Using the Iterator Interface

The DBSimJoin algorithms presented in section 3.5.1 are realized in a way that allows generating links one at a time, i.e., using the iterator interface and its functions *open*, *getNext*, and *close*. Furthermore, DBSimJoin is a non-blocking operator. That is, it does not require the full generation of results before it can start reporting results. Database queries are commonly processed using a query pipeline. This pipeline is composed of a tree of operators where tuples flow bottom-up. The process is initiated by calls to the *getNext* function at the root operator. Each time *getNext* is called, the operator will call the *getNext* function of its children nodes one or multiple times until it obtains all the required information to produce a result tuple. This process is propagated top-down. A non-blocking behavior is a very desirable property since it enables tuples to flow quickly in the pipeline and reduces query response time. When the *getNext* function is called in a DBSimJoin node, the operator executes the described process only until the next result link is found. Since small partitions that can be solved using the in-memory routines can be generated starting at the first round, DBSimJoin will quickly find the next link.

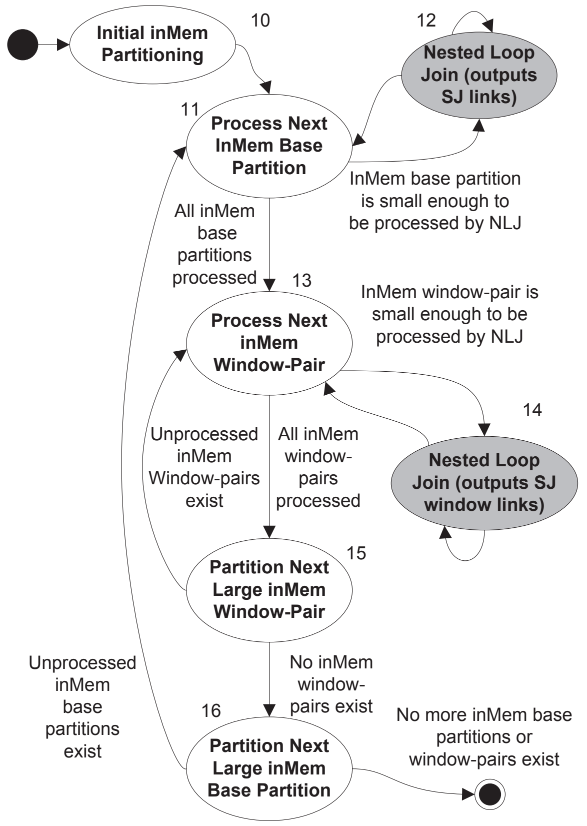
The *open* and *close* routines are presented in Algorithms 4 and 5, respectively. The *open* routine is called once at the beginning of DBSimJoin's execution. This routine initializes data structures and computes the value of the number of pivots (see Section 3.6.2 for details). The *close* routine is called once after all the SJ links have been reported. This routine deletes all the temporary data structures used by the operator.

The *getNext* routine is implemented in the fashion of a state machine that uses the states and transitions presented in Fig. 8. States that produce results are marked in gray. When *getNext* is called in the DBSimJoin operator, the routine transitions over the states until it produces the next tuple. The system keeps track of the current state and other required information to resume execution when the next *getNext* is invoked. The states InMemSJBase and InMemSJWin (4 and 7) represent the in-memory SJ routines. These two routines are also implemented using a state machine approach to further reduce the time to produce the next link. The states and transitions of InMemSJBase are presented in Fig. 9. Observe that states 12 and 14 produce the links using a Nested Loop Join approach. The states and transitions of InMemSJWin are very similar to the ones of InMemSJBase with the difference that InMemSJWin only produces window links (links between tuples of different previous partitions).



1
Initial Partitioning
3
Hibernate Partition
Base partition cannot be processed in-memory
2
Process Next Base Partition
4
InMemSJBase (outputs SJ links)
Base partition can be processed in-memory
All base partitions processed
6
Hibernate Window-Pair
5
Process Next Window-Pair
Window-pair cannot be processed in-memory
Unprocessed window-pairs exist
All window-pairs processed
Window-pair can be processed in-memory
8
Partition Next Large Window-Pair
7
InMemSJWin (outputs SJ window links)
Unprocessed base partitions exist
No window-pairs exist
9
Partition Next Large Base Partition
No more base partitions or window-pairs exist

Figure 8: DBSimJoin's GetNext.



Initial inMem Partitioning
10
12
Nested Loop Join (outputs SJ links)
11
Process Next InMem Base Partition
InMem base partition is small enough to be processed by NLJ
All inMem base partitions processed
13
Process Next inMem Window-Pair
InMem window-pair is small enough to be processed by NLJ
14
Nested Loop Join (outputs SJ window links)
Unprocessed inMem Window-pairs exist
All inMem window-pairs processed
15
Partition Next Large inMem Window-Pair
Unprocessed inMem base partitions exist
No inMem window-pairs exist
16
Partition Next Large inMem Base Partition
No more inMem base partitions or window-pairs exist

Figure 9: Details of InMemSJBase.

3.6. Analysis

3.6.1. I/O Analysis

The work in [7] showed that the average I/O cost of the external Quickjoin algorithm is $O(N(1+w)^{\lceil log(N/M) \rceil})$. N and M are the number of blocks of the input data and the number of tuples that fit in internal memory, respectively. w is the fraction of tuples that lie within epsilon of the partition boundary (these tuples form the window-pair partitions). The I/O analysis of DBSimJoin is similar to the one of Quickjoin. However, the analysis of DBSimJoin differs in the following aspect. In DBSimJoin, we assume we have a limited number of buffer pages B that can be used to store the data of partitions during a partitioning round. Furthermore, each partition is assigned L buffer pages to store its data. If the partition grows beyond the allocated space, the partition will be stored on disk. The maximum number of new partitions generated by the algorithm in a round will be limited by $P_{max} = \frac{B}{L}$. Also, the value of M in our case is the number of tuples that fit in L buffer pages. That is $M = T \times L$, where T is the number of tuples that fit in a single page. Using these properties, we have that the average I/O cost of DBSimJoin is:

$$O(N(1+w)^{\lceil log(\frac{N \times P_{max}}{B \times T}) \rceil}).$$

This cost will be close to $O(N)$ for small values of ε (small values of w) and will be close to $O(N^2)$ for large values of ε.

Note that B does not include the space for the structures used to keep track of the current partitions or the space required by the in-memory SJ routine.

3.6.2. Number of Pivots

The value of P (number of partitions in a round) is directly related to the number of pivots K. Given K pivots, DBSimJoin generates K base partitions and $K^2 - K$ window-pair partitions. Note that in a window-pair partition round, there are 2 window-pair partitions for each pair of pivots. The total number of partitions that is generated with K pivots is K^2, that is $P = K^2$. Since $P \leq \frac{B}{L}$, we have that $K \leq \lfloor \sqrt{\frac{B}{L}} \rfloor$. In practical scenarios, we can use $K = \lfloor \sqrt{\frac{B}{L}} \rfloor$ as an initial value of K. Also note that the number of partitions P is not affected by the number of dimensions. An experimental evaluation of the effect of the number of pivots in execution time is presented in Section 4.4.3.

4. Performance Evaluation

We implemented DBSimJoin in PostgreSQL 8.2.4. In this section we evaluate its performance with synthetic and real-world data. We consider both vector and string data and study the performance of the implemented operator when important parameters, e.g., data size, ε, and number of dimensions, increase. While we compare DBSimJoin with a standalone algorithm (D-Index) in Section 4.6, the focus of this section is the comparison of DBSimJoin with other approaches proposed for database systems. Standalone algorithms can outperform database implementations since they are not affected by database features like transaction processing, recovery, etc.

4.1. Test Configuration

All the experiments are performed on an Intel Core i5 2.27 GHz machine with 4GB RAM running Linux (OpenSUSE 11.3 32-bit) as the operating system. The machine has a 5400 RPM hard disk with a capacity of 500 GB. We use the following datasets:

- **SynthData** This is a synthetic vector dataset. A version of this dataset was created for each evaluated number of dimensions, i.e., 4D, 6D and 8D. The components of each vector are randomly generated numbers in the range [0 - 100]. The dataset for scale factor 1 (SF1) contains 80,000 records.

- **ColorData** This dataset contains feature vectors extracted from a Corel image collection [32]. Each record is a 9D vector with components in the range [-4.8 - 4.4]. The SF1 dataset contains 68,040 records.

- **DBLPData** This dataset is a subset of the DBLP bibliographic dataset [33]. Each extracted record contains a unique identifier and the title. The SF1 dataset contains 2,500 records. The minimum, maximum and average lengths of the title attribute are 33, 281, and 57, respectively.

In all cases, the datasets for SF greater than 1 were generated in such a way that the number of links of any SJ operation in SFN is N times the number of links of the operation in SF1. For vector data (*SynthData* and *ColorData*), the datasets for higher SF were obtained adding shifted copies of the SF1 dataset such that the the distance between copies were greater than the maximum value of ε used in our tests. For string data (*DBLPData*), the datasets for higher SF were obtained adding a copy of the SF1 data where characters are shifted similarly to the process in [23]. The records of each dataset are equally divided between R and S. We used the Euclidean and the Levenshtein distance functions for vector and string data, respectively. The number of pivots (*numPiv*) in the experiments was 30 for SynthData and ColorData and 50 for DBLPData, the threshold to switch to in-memory SJ was 4KB and the threshold to switch to nested loop join in the in-memory SJ routines was 20 tuples.

4.2. Performance Evaluation with DBLP String Data

This section evaluates the performance of DBSimJoin with string data (DBLP titles). We compare DBSimJoin with an implementation of SSJoin (q=3), the q-gram based approach proposed in [13]. As explained in Section 2, this approach decomposes each input string s into sets of q-grams (substrings of length q that are used as the signature of s) and stores the results on separate relational tables. Then, the result of Similarity Join can be obtained using a standard SQL query. We did not include the q-gram preparation time in the reported SSJoin execution times. The preparation time was a very small fraction of the query execution times (smaller than 1% in most cases). Examples of the SSJoin and DBSimJoin queries are presented next.

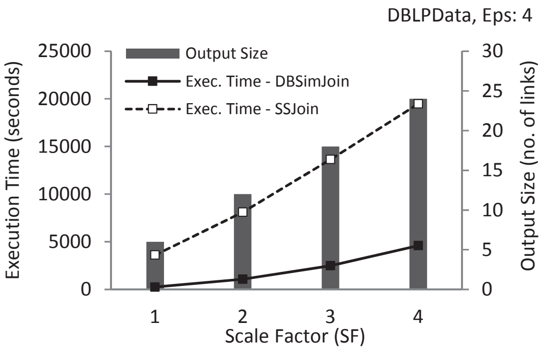


Figure 10: Increasing SF - DBLPData.

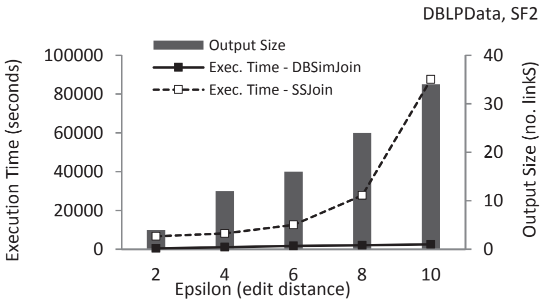


Figure 11: Increasing Epsilon - DBLPData.

```
SSJoin query:
SELECT R.pka, R.origstringa, S.pkb, S.origstringb
FROM qgramsR R, qgramsS S WHERE R.qgrama = S.qgramb
GROUP BY R.pka, R.origstringa, S.pkb, S.origstringb
HAVING count(*) >=
   (char_length(R.origstringa) - 3 + 1 - 3 * 2) AND
   editdist(R.origstringa, S.origstringb) <= 2;

DBSimJoin query:
SELECT R.pka, R.origstringa, S.pkb, S.origstringb
FROM R, S
WHERE R.origstringa WITHIN 2 OF S.origstringb
   USING EditDistance;
```

4.2.1. *Increasing Scale Factor*

Fig. 10 shows the execution time of DBSimJoin and SSJoin for several values of scale factor. The execution time of DBSimJoin is consistently smaller than that of SSJoin. Specifically, the execution time of DBSimJoin is between 7% (SF1) and 24% (SF4) of the one of SSJoin.

4.2.2. *Increasing Epsilon*

Fig. 11 compares the performance of DBSimJoin and SSJoin when ε increases. DBSimJoin's execution time is 13% of that of SSJoin for ε=2, and only 3% for ε=10. While for very low values of ε the number of tuples returned by the join used in SSJoin is relatively small, this number grows quickly when ε increases affecting negatively its execution time. DBSimJoin's execution time increases moderately when ε increases because larger values of ε generate larger window-pair partitions.

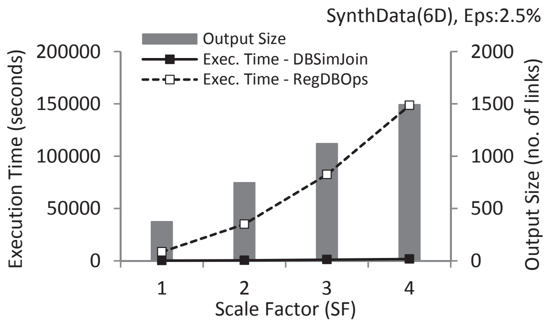


Figure 12: Increasing SF - SynthData.

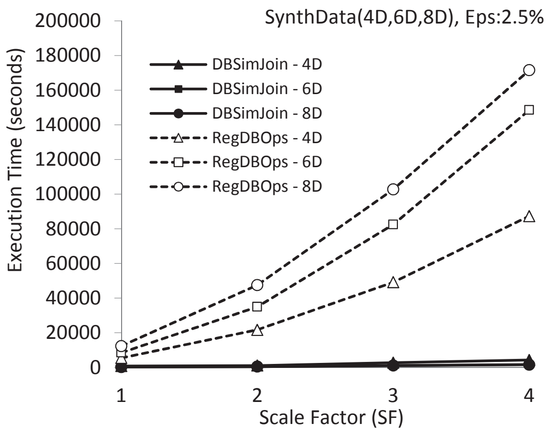


Figure 13: Increasing SF and Number of Dimensions - SynthData.

4.2.3. *Space Usage*

Besides having a better performance, DBSimJoin also uses significantly less space on disk than the SSJoin approach. SSJoin requires the creation of tables that store the generated q-grams, the size of these tables can be significantly large since they will commonly store many more rows than the original tables. In our experiments, for SF1, the q-gram tables have about 55 times the number of rows of the original tables. DBSimJoin, on the other hand, uses only the original tables.

4.3. *Performance Evaluation with Synthetic Vector Data*

This section and the next one compare DBSimJoin with queries that produce the same results using only regular (non-similarity) database operators (RegDBOps). To the best knowledge of the authors, no previous work has proposed an alternative approach to support Similarity Joins over multidimensional vectors in a relational database system. PostGIS, a spatial database extender for PostgreSQL [34], is not considered since it only supports 2D/3D data. Also, PostGIS' spatial distance function (ST-Distance) does not use indexes and thus SJ will perform like RegDBOps. Examples of RegDBOps and DBSimJoin queries are presented next.

```
RegDBOps query:
SELECT R.r1, R.r2, S.s1, S.s2
FROM R, S
WHERE sqrt((R.r1-S.s1)^2 + (R.r2-S.s2)^2) <= 0.5;
```

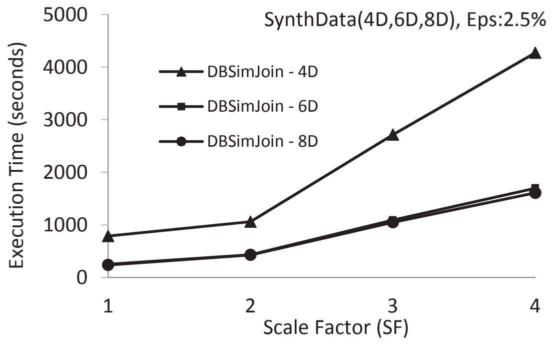


Figure 14: Increasing SF and Number of Dimensions (DBSimJoin) - SynthData.

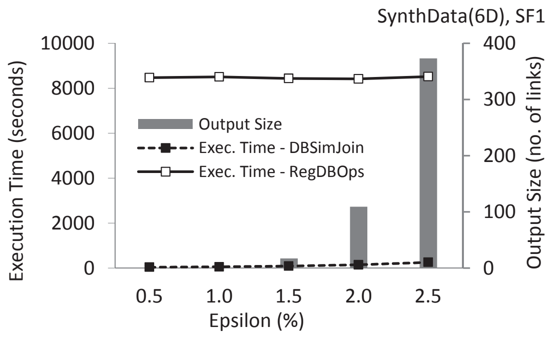


Figure 15: Increasing Epsilon - SynthData.

```
DBSimJoin query:
SELECT R.r1, R.r2, S.s1, S.s2
FROM R, S
WHERE [R.r1, R.r2] WITHIN 0.5 OF [S.s1, S.s2]
   USING EuclideanDistance;
```

4.3.1. *Increasing Scale Factor*

Fig. 12 shows the way DBSimJoin and RegDBOps scale when the data size increases. This experiment uses 6D vectors and a value of ε of 2.5% of the maximum possible distance. DBSimJoin performs significantly better than RegDBOps for all the values of SF. The execution time of DBSimJoin is always less than 3% of the execution time of RegDBOps. Furthermore, the execution time of RegDBOps grows from being 34 times the one of DBSimJoin for SF1 to 88 times for SF4. In general, the poor execution time of RegDBOps is due to a nested loop join between the joined relations.

4.3.2. *Increasing SF and Number of Dimensions*

DBSimJoin queries perform much better than the RegDBOps queries also for different number of dimensions as shown in Fig. 13. In all cases, the execution time of DBSimJoin is a small fraction of that of RegDBOps. Moreover, when the number of dimensions increases, DBSimJoin takes a smaller fraction of the execution time of RegDBOps. Specifically, for 4D data the execution time of DBSimJoin is at most 15% of that of RegDBOps. The percentage is 3% for 6D data and only 2% for 8D data. Fig. 14 allows a better visualization of the DBSimJoin execution times for different number of dimensions. The figure

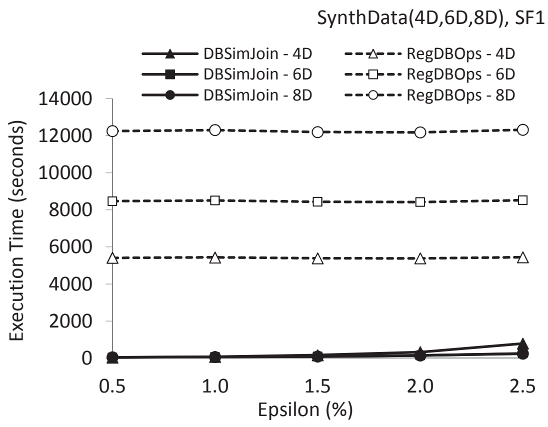


Figure 16: Increasing Epsilon and Number of Dimensions - SynthData.

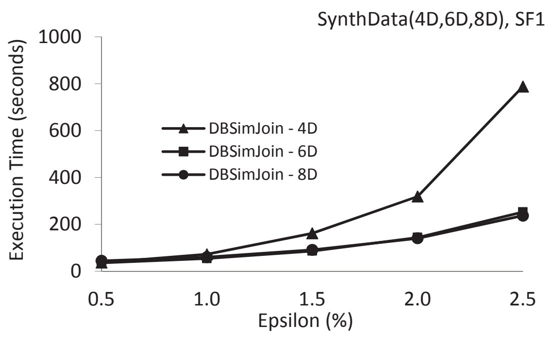


Figure 17: Increasing Epsilon and Number of Dimensions - SynthData.

shows that the execution time of DBSimJoin decreases when the number of dimensions increases. This is mainly due to the large difference between the links reported in 4D data and the ones reported in 6D and 8D data. The tests with 4D data report between 46,109 to 184,436 links, the tests with higher number of dimensions report always less than 1,500 links.

4.3.3. *Increasing Epsilon*

Fig. 15 shows how DBSimJoin and RegDBOps scale when ε increases. The figure shows that, for all the evaluated values of ε, the execution time of DBSimJoin is significantly smaller than that of RegDBOps. The execution time of DBSimJoin is only 0.42% of the execution time of RegDBOps for ε=0.5% and 2.97% for ε=2.5%. The execution time of RegDBOps remains almost constant when the value of ε increases because it always executes a nested loop join checking the SJ predicate. DBSimJoin's execution time increases slightly with larger values of ε since they generate larger window-pair partitions.

4.3.4. *Increasing Epsilon and Number of Dimensions*

DBSimJoin performs significantly better than the RegDB-Ops queries also for different numbers of dimensions as shown in Fig. 16. For 4D data the execution time of DBSimJoin is at most 14.5% of that of RegDBOps. The percentage is 3% for 6D data and only 1.9% for 8D data. Fig. 17 shows that the execution time of DBSimJoin decreases when the number of dimensions increases. The reason is also the large difference on

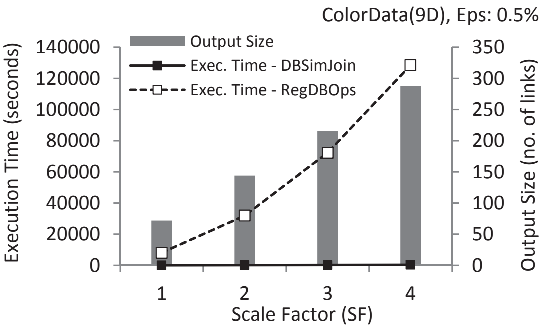


Figure 18: Increasing SF - ColorData.

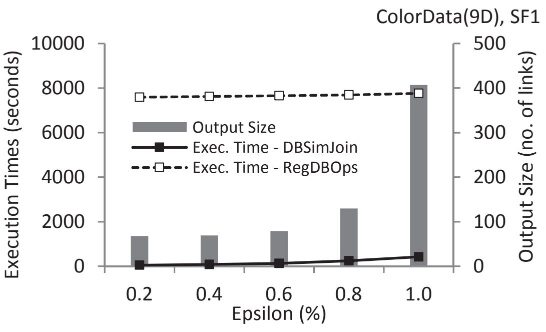


Figure 19: Increasing Epsilon - ColorData.

the number of links reported: 46,109 for 4D and less than 400 for 6D and 8D.

4.4. *Performance Evaluation with Color Vector Data*

4.4.1. *Increasing Scale Factor*

Fig. 18 presents the execution time of DBSimJoin and RegDBOps for different values of SF. The results with the real vector data follow the same trends identified with synthetic vectors. Particularly, the execution time of DBSimJoin is 1.4% of that of RegDBOps for SF1 and 0.2% for SF4.

4.4.2. *Increasing Epsilon*

Fig. 19 compares the execution time of DBSimJoin and RegDBOps for different values of ε. The results in this case also follow the trends identified for the counterpart tests using *SynthData*. The execution time of DBSimJoin is 0.6% of that of RegDBOps for ε=0.2% and 9.8% for ε=1.0%.

4.4.3. *Varying Number of Pivots*

Fig. 20 shows DBSimJoin's execution time when the number of pivots (K) increases from 2 to 50. As K increases, the execution time decreases at first due to fewer rounds required to reach the point where all partitions can be processed in memory. When K increases past the optimal value, the execution time grows because the extra data duplication and I/O costs of the window-pair partitions outweigh the effect of decreased partition size and number of rounds. In this experiment, the optimal execution time is obtained at $K = 8$ being only 17.8% of the execution time at $K = 50$.

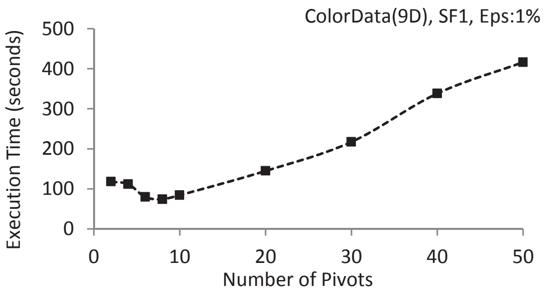


Figure 20: Increasing No. of Pivots - ColorData.

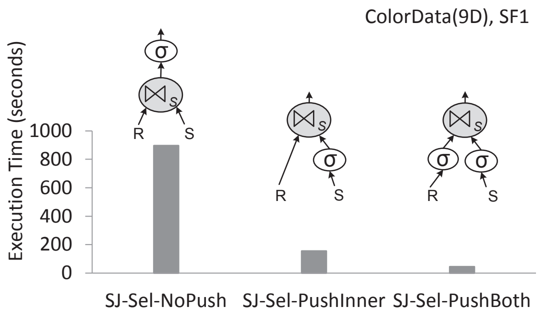


Figure 21: Pushing Selection below DBSimJoin.

4.5. *Combining DBSimJoin with other Database Operators*

This section shows that DBSimJoin can be combined with other similarity and non-similarity database operators, e.g., Selection, Group-by, and Similarity Group-by, to build more complex and useful queries. Moreover, we show that DBSimJoin can be used in important transformation rules that can significantly reduce the execution time of similarity queries.

4.5.1. *Pushing Selection below DBSimJoin*

We use the following query where the distance thresholds of 0.28 and 1.38 correspond to ε=1.0% and ε=5.0%, respectively.

```
Combining SJ and Selection (SJ-Sel):
SELECT * FROM R, S
WHERE
   [R.r1 ... R.r9] WITHIN 0.28 OF [S.s1 ... S.s9]
   USING EuclideanDistance AND
   EucDist([S.s1 ... S.s9],[0.02 ... 0.02])<=1.38;
```

SJ-Sel-NoPush, SJ-Sel-PushInner and SJ-Sel-PushBoth in Fig. 21 are different ways to execute query SJ-Sel. SJ-Sel-NoPush executes the SJ first and then the selection operation. In SJ-Sel-PushInner, the selection operator ($\sigma_{EucDist(S,Const)\leq 1.38}$) is pushed to table S. SJ-Sel-PushInner's execution time is 17% of that of SJ-Sel-NoPush. In SJ-Sel-PushBoth the filtering benefit is further improved by pushing selection operations on both inputs of the join ($\sigma_{EucDist(S,Const)\leq 1.38}$ on S and $\sigma_{EucDist(R,Const)\leq(1.38+0.28)}$ on R). The execution time of SJ-Sel-PushBoth is only 5% of the one of SJ-Sel-NoPush.

4.5.2. *Associativity of DBSimJoin Operators*

For this experiment, the initial SF1 dataset is divided into three relations: relation R has 1/2 of the dataset, relation S has 1/3, and relation T has 1/6. We use the following query.

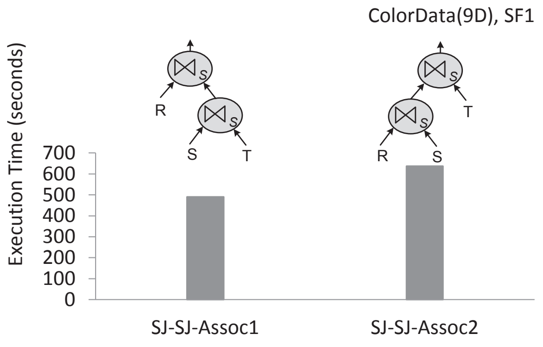


Figure 22: Associativity of DBSimJoin Operators.

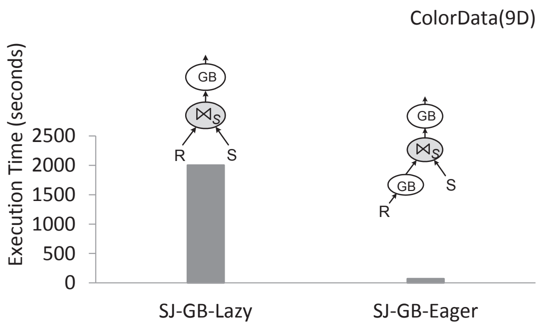


Figure 23: Lazy/Eager Aggregation with DBSimJoin and Group-by.

```
Combining Two SJ Operators (SJ-SJ):
SELECT * FROM R, S, T
WHERE
   [R.r1 ... R.r9] WITHIN 0.28 OF [S.s1 ... S.s9]
   USING EuclideanDistance AND
   [S.s1 ... S.s9] WITHIN 0.28 OF [T.t1 ... T.t9]
   USING EuclideanDistance;
```

SJ-SJ-Assoc1 and SJ-SJ-Assoc2 in Fig. 22 represent two plans to execute query SJ-SJ. SJ-SJ-Assoc1 executes first the join between S and T and then joins the result with R. SJ-SJ-Assoc2, on the other hand, joins R and S first and then joins the result with T. The execution time of SJ-SJ-Assoc1 is 77% of that of SJ-SJ-Assoc2. SJ-SJ-Assoc1 outperforms SJ-SJ-Assoc2 because it joins the two smaller tables (S and T) first generating a significantly smaller number of intermediate records.

4.5.3. *Lazy/Eager Aggregation with DBSimJoin and Group-by*

The experiments in this subsection use ColorData (SF1) with the following changes. Table S contains 1,000 randomly selected 9D vectors that are used as reference points around which the points of R are grouped. Also, in order to generate some duplicate tuples that can be pre-aggregated, table R is generated taking 20% of the original dataset and duplicating each tuple 5 times. We use the following query.

```
Combining SJ and Group-by (SJ-GB):
SELECT count(*), S.s1 ... S.s9
FROM R, S
WHERE
   [R.r1 ... R.r9] WITHIN 0.28 OF [S.s1 ... S.s9]
   USING EuclideanDistance
GROUP BY [S.s1 ... S.s9];
```

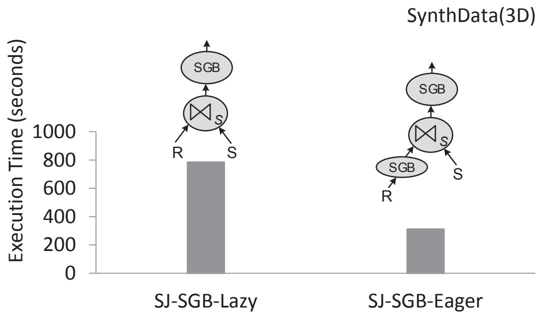


Figure 24: Lazy/Eager Aggregation with DBSimJoin and Similarity Group-by.

SJ-GB-Lazy and SJ-GB-Eager in Fig. 23 correspond to the eager and lazy aggregation plans of query SJ-GB. SJ-GB-Lazy executes SJ first and then group-by. The group-by operator of the lazy approach is split into two parts in SJ-GB-Eager. The first part groups on R.r and calculates the count before the SJ. The second part groups on S.s and uses the intermediate data to calculate the final results (sum of the intermediate counts) after the SJ. The execution time of SJ-GB-Eager is only 4% of that of SJ-GB-Lazy.

4.5.4. *Lazy/Eager Aggregation with DBSimJoin and Similarity Group-by*

The experiments in this subsection use a SynthData dataset of 50,000 3D vectors. The records of the dataset are equally divided between R and S. *RefPoints* is a relation that contains reference points (central points) around which the query clusters the records of R. We use the following query.

```
Combining SJ and Similarity Group-by (SJ-SGB):
SELECT count(*) AS count_tuples, a1 as around_a1
FROM R, S
WHERE (R.r2,R.r3) WITHIN 1.41 OF (S.s2,S.s3)
   USING EuclideanDistance
GROUP BY R.r1 AROUND (SELECT g FROM RefPoints);
```

The eager and lazy aggregation plans can also be applied to DBSimJoin and the Similarity Group-by operator *Group Around* as represented in Fig. 24. The Group Around operator clusters the data around a set of reference points (RefPoints) such that each records is associated to the group of its closest reference point [35, 36]. SJ-SGB-Lazy and SJ-SGB-Eager correspond to the eager and lazy aggregation plans of query SJ-SGB. As in the previous subsection, the eager plan significantly outperforms the lazy one. In this case, SJ-SGB-Eager's execution time is 40% of that of SJ-SGB-Lazy.

4.6. *Comparison with D-Index approach*

This section compares the performance of DBSimJoin, RegDBOps, and a standalone approach based on the D-Index [4], an indexing structure that supports Similarity Search and Similarity Join operations. Specifically, we use the Range

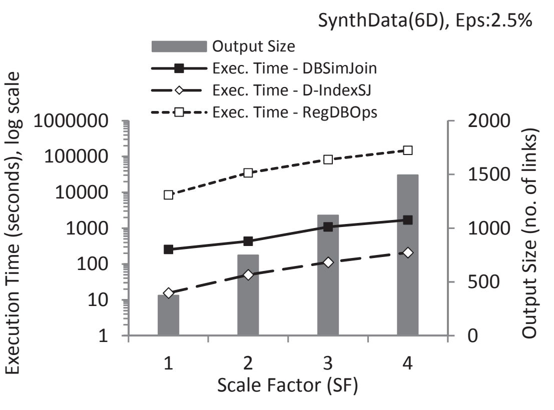


Figure 25: Comparison with D-IndexSJ.

Query Similarity Join algorithm proposed in [4] (D-IndexSJ). We extended this algorithm to the case of R-S similarity joins since the original algorithm only considered the case of similarity self joins. The algorithm applies successive similarity search operations over the indexed dataset R, using all elements of the dataset S as the targets of the similarity searches. For each object s in S, the output is the collection of all objects in R that are within ε of s. The D-Index was constructed using 6 levels and 10 reference objects. The parameter ρ, which controls the size of the exclusion set, was 2.5% of the maximum possible distance. The experiments in this subsection use SynthData (6D) and ε=2.5% of the maximum possible distance.

Fig. 25 shows the way DBSimJoin, RegDBOps, and D-IndexSJ scale when the data size increases. As expected, the standalone approach performs better than the database approaches. Specifically, DBSimJoin's execution time is about 16 times the one of D-IndexSJ for SF1 and about 8 times for SF4. Observe that the relative advantage of D-IndexSJ over DBSimJoin decreases rapidly when the dataset size increases. Furthermore, given the value of ρ (2.5%), D-IndexSJ's performance will be less efficient for queries with ε >2.5%. An important reason for the difference in performance is that DBSimJoin is a database operator while D-IndexSJ isn't. DBSimJoin's execution time is affected by the overhead associated with important database features, e.g., transaction processing (atomicity, consistency, isolation, and durability), recovery, SQL parsing, etc. Even though D-IndexSJ is not affected by database features, it also does not provide the benefits of a first-class database operator. Moreover, D-IndexSJ requires building an index (15s in our experiments, not included as part of query execution time) and may require rebuilding the index to efficiently support queries with different values of ε. RegDBOps, the approach that uses regular database operators to answer SJ queries, performs significantly worse than DBSimJoin and D-IndexSJ. RegDBOps's execution time is 34 times the one of DBSimJoin for SF1 and 88 times for SF4.

5. Conclusions and Future Work

The Similarity Join is recognized as one of the most useful data analysis operations and has been used in many application scenarios. While multiple implementation techniques have been proposed for the Similarity Join, very little work has addressed the study of Similarity Joins as first-class database operators.

This paper presents DBSimJoin, an efficient and non-blocking SJ database operator. DBSimJoin fully supports the database iterator interface (*open*, *getNext*, and *close*) and uses a sequence of rounds that prioritizes the quick generation of results. The proposed algorithm can be used with any dataset that lies in a metric space. Thus, it can be used with multiple data types and distance functions. We present the implementation details of DBSimJoin and extensively evaluate its performance using vector and string data as well as synthetic and real-world datasets. We show that DBSimJoin performs significantly better than alternative approaches (q-gram based approach for string data and queries with conventional operators for vector data). DBSimJoin scales well when important parameters like epsilon, data size, and number of dimensions increase. We also present queries that combine DBSimJoin with other database operators and show that important transformation rules can be effectively applied to queries with DBSimJoin.

Our paths for future work include the study of: (1) other similarity-aware operations, e.g., kNN Join and kDistance Join, as physical database operators, (2) indexing techniques to improve the efficiency of similarity queries, and (3) database queries with multiple similarity-based operators.

References

[1] Postgresql, `http://www.postgresql.org/`, 2013.

[2] C. Böhm, H.-P. Kriegel, A cost model and index architecture for the similarity join, in: Proceedings of the 17th International Conference on Data Engineering, ICDE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 411 –420.

[3] V. Dohnal, C. Gennaro, P. Zezula, Similarity join in metric spaces using ed-index, in: Proceedings of the 25th European Conference on IR Research, ECIR'03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 452–467.

[4] V. Dohnal, C. Gennaro, P. Savino, P. Zezula, Similarity join in metric spaces, in: Proceedings of the 25th European Conference on IR Research, ECIR '03, Springer-Verlag, Berlin, Heidelberg, 2003, pp. 452–467.

[5] C. Böhm, B. Braunmüller, F. Krebs, H.-P. Kriegel, Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data, in: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, SIGMOD '01, ACM, New York, NY, USA, 2001, pp. 379–388.

[6] J.-P. Dittrich, B. Seeger, Gess: A scalable similarity-join algorithm for mining large data sets in high dimensional spaces, in: Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '01, ACM, New York, NY, USA, 2001, pp. 47–56.

[7] E. H. Jacox, H. Samet, Metric space similarity joins, ACM Trans. Database Syst. 33 (2008) 7:1–7:38.

[8] G. R. Hjaltason, H. Samet, Incremental distance join algorithms for spatial databases, in: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, SIGMOD '98, ACM, New York, NY, USA, 1998, pp. 237–248.

[9] C. Böhm, F. Krebs, The k-nearest neighbour join: Turbo charging the kdd process, Knowledge and Information Systems 6 (2004) 728–749.

[10] C. Yu, B. Cui, S. Wang, J. Su, Efficient index-based knn join processing for high-dimensional data, Inf. Softw. Technol. 49 (2007) 332–344.

[11] C. Xia, H. Lu, B. C. Ooi, J. Hu, Gorder: An efficient method for knn join processing, in: Proceedings of the 30th International Conference on Very

large Data Bases - Volume 30, VLDB '04, VLDB Endowment, 2004, pp. 756–767.

[12] R. Paredes, N. Reyes, Solving similarity joins and range queries in metric spaces with the list of twin clusters, J. of Discrete Algorithms 7 (2009) 18–35.

[13] S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in: Proceedings of the 22nd International Conference on Data Engineering, ICDE '06, IEEE Computer Society, Washington, DC, USA, 2006, pp. 5–.

[14] S. Chaudhuri, V. Ganti, R. Kaushik, Data debugger: An operator-centric approach for data quality solutions, IEEE Data Eng. Bull. 29 (2006) 60–66.

[15] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, Approximate string joins in a database (almost) for free, in: Proceedings of the 27th International Conference on Very Large Data Bases, VLDB '01, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001, pp. 491–500.

[16] Y. N. Silva, W. G. Aref, M. H. Ali, The similarity join database operator, in: Proceedings of the 2010 IEEE International Conference on Data Engineering, ICDE '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 892 –903.

[17] Y. N. Silva, A. M. Aly, W. G. Aref, P.-A. Larson, Simdb: A similarity-aware database system, in: Proceedings of the 2010 International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 1243–1246.

[18] Y. N. Silva, S. Pearson, Exploiting database similarity joins for metric spaces, Proc. VLDB Endow. 5 (2012) 1922–1925.

[19] B. Bryan, F. Eberhardt, C. Faloutsos, Compact similarity joins, in: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering, ICDE '08, IEEE Computer Society, Washington, DC, USA, 2008, pp. 346 –355.

[20] C. Xiao, W. Wang, X. Lin, Ed-join: An efficient algorithm for similarity joins with edit distance constraints, Proc. VLDB Endow. 1 (2008) 933–944.

[21] C. Xiao, W. Wang, X. Lin, J. X. Yu, G. Wang, Efficient similarity joins for near-duplicate detection, ACM Trans. Database Syst. 36 (2011) 15:1–15:41.

[22] M. D. Lieberman, J. Sankaranarayanan, H. Samet, A fast similarity join algorithm using graphics processing units, in: Proceeding of the 17th International Conference on World Wide Web, WWW '08, ACM, New York, NY, USA, 2008, pp. 131–140.

[23] R. Vernica, M. J. Carey, C. Li, Efficient parallel set-similarity joins using mapreduce, in: Proceedings of the 2010 International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 495–506.

[24] Y. N. Silva, J. M. Reed, L. M. Tsosie, Mapreduce-based similarity join for metric spaces, in: Proceedings of the 1st International Workshop on Cloud Intelligence, Cloud-I '12, ACM, New York, NY, USA, 2012, pp. 3:1–3:8.

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

[26] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, J. D. Ullman, Fuzzy joins using mapreduce, in: Proceedings of the 2012 IEEE 28th International Conference on Data Engineering, ICDE '12, IEEE Computer Society, Washington, DC, USA, 2012, pp. 498–509.

[27] Y. N. Silva, J. M. Reed, Exploiting mapreduce-based similarity joins, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, ACM, New York, NY, USA, 2012, pp. 693–696.

[28] Y. N. Silva, S. S. Pearson, J. A. Cheney, Database similarity join for metric spaces, in: Proceedings of the 6th International Conference on Similarity Search and Applications, SISAP '13, Springer Berlin Heidelberg, 2013, pp. 266–279.

[29] G. R. Hjaltason, H. Samet, Index-driven similarity search in metric spaces (survey article), ACM Trans. Database Syst. 28 (2003) 517–580.

[30] Y. N. Silva, W. G. Aref, P.-A. Larson, S. S. Pearson, M. H. Ali, Similarity queries: Their conceptual evaluation, transformations, and processing, The VLDB Journal 22 (2013) 395–420.

[31] K. Fredriksson, B. Braithwaite, Quicker similarity joins in metric spaces, in: Proceedings of the 6th International Conference on Similarity Search and Applications, SISAP '13, Springer Berlin Heidelberg, 2013, pp. 127–140.

[32] A. Frank, A. Asuncion, UCI machine learning repository, `http://archive.ics.uci.edu/ml`, 2010.

[33] Dblp bibliography, `http://www.informatik.uni-trier.de/~ley/db/`, 2013.

[34] Postgis, `http://postgis.net/documentation`, 2013.

[35] Y. N. Silva, W. G. Aref, M. H. Ali, Similarity group-by, in: Proceedings of the 2009 IEEE International Conference on Data Engineering, ICDE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 904–915.

[36] Y. N. Silva, M. U. Arshad, W. G. Aref, Exploiting similarity-aware grouping in decision support systems, in: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09, ACM, New York, NY, USA, 2009, pp. 1144–1147.