Database Similarity Join for Metric Spaces

Yasin N. Silva, Spencer S. Pearson, and Jason A. Cheney

Arizona State University,
4701 W. Thunderbird Road, Glendale, AZ 85306, USA
{ysilva,sspearso,jcheney1}@asu.edu

Abstract. Similarity Joins 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 Join as a physical database operator. In this paper, we focus on the study, design and implementation of a Similarity Join database operator for any dataset that lies in a metric space (DBSimJoin). We describe the changes in each query engine module to implement DBSimJoin and provide details of our implementation in PostgreSQL. The extensive performance evaluation shows that *DBSimJoin* significantly outperforms alternative approaches.

1 Introduction

Similarity Joins (SJs) have been studied and extensively used in multiple application domains, e.g., data cleaning and sensor networks. Several SJ algorithms have been previously proposed. Very little work, however, has addressed the implementation of SJ 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 SJ database operator for any dataset that lies in a metric space. The main contributions of this paper are:

- We present *DBSimJoin*, a SJ 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) support the database iterator interface (functions *open*, *getNext*, and *close*).
- To the best knowledge of the authors, DBSimJoin is the first SJ database operator that can be used with any dataset that lies in a metric space. The operator can be used with various distance functions and data types.
- We present multiple guidelines to implement DBSimJoin as an integrated component of a database system. We also provide details of our implementation in PostgreSQL, a popular open-source database system.
- We thoroughly evaluate the performance of DBSimJoin with multiple data types and show that it significantly outperforms alternative approaches.
- 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 Eager and Lazy aggregation transformations.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 describes DBSimJoin. The performance evaluation is presented in Section 4. Section 5 concludes the paper.

2 Related Work

Significant work has been carried out on the study of the SJ (retrieves all data pairs whose distances are smaller than a threshold ε). This work proposed techniques to implement them primarily as standalone algorithms outside of a database system. Some implementation techniques rely on the use of pre-built indices, e.g., eD-index [8], D-index [7], and List of Twin Clusters (LTC) [13]. They 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, e.g., EGO, GESS, and QuickJoin. The Epsilon Grid Order (EGO) algorithm [3] imposes an ε-sized grid over the space and uses a schedule of reads of blocks that minimizes 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 [12] 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 [12]. 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 [12] 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 [12], 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 SJ techniques in the context of database systems. Some work has focused on the implementation of SJs using standard database operators [4, 5, 10]. These techniques are applicable only to string or set-based data. The general approach decomposes data and query strings into sets of grams (substrings of a string that are used as its signature), and stores the results on separate tables. Then, the result of the SJ can be obtained using standard SQL queries. DBSimJoin is experimentally compared with one such technique (SSJoin [4]) in Section 4.1. More recently, the work in [15, 14] 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 multiple data types and distance functions. In a recent demo paper [17], we showed how DBSimJoin can be used to identify similar images (feature vectors), and similar publications in a bibliographic database.

3 The DBSimJoin Operator

The Similarity Join (SJ) operation between two datasets R and S is defined as: $R \ltimes_{\theta_\varepsilon(r,s)} S = \{\langle r,s\rangle | \theta_\varepsilon(r,s), r \in R, s \in S\}$, where $\theta_\varepsilon(r,s)$ represents the Similarity Join predicate, i.e., $dist(r,s) \leq \varepsilon$. 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 SJ 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 DBSimJoin Rounds

A 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*). Partitioning is performed using a set of K pivots, i.e., a random subset of the records to be partitioned. 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, it 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 [11]: $gen\_hyperpln\_dist(t,P_0,P_1) = (dist(t,P_0) - dist(t,P_1))/2$. This distance is replaced with the 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*

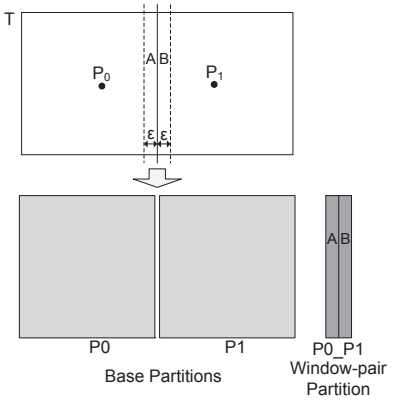

Fig. 1. Repartitioning a base partition.

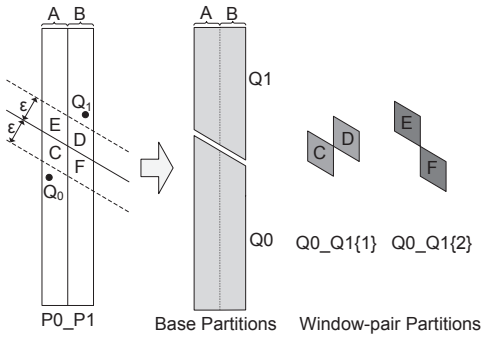

Fig. 2. Repartitioning a window-pair partition.

partition round, a round that repartitions a window-pair partition is referred to as a *window-pair partition round*.

Fig. 1 shows the repartitioning of a base partition using pivots P_0 and P_1. In this case, the result of the SJ 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 shows 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 ones in $Q1$.

Figures 3 and 4 represent the processing performed by DBSimJoin in round 0 and a generic round I, respectively. Round 0, shown in Fig. 3, 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. 4 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 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 3.3. 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.

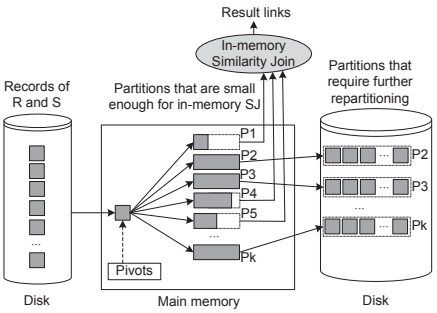

Fig. 3. Round 0.

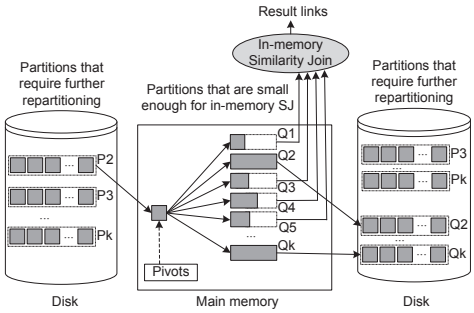

Fig. 4. Round I.

3.2 The Parser and Planner

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 SJ 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. To add support for the operator in the planner, a new plan node is created to represent the SJ 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 SJ predicates, they are processed one at a time, i.e., multiple SJ nodes are pipelined. It is important to observe that key transformation rules to optimize queries with SJ [15, 16], 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 4.3.

3.3 The Executor

DBSimJoin Executor Routine DBSimJoin's main routine is presented in Fig. 5. The routine first creates two lists to keep track of the base and window-pair partitions (line 1). Each partition is assigned a space in memory ($memT$) and if it needs to grow beyond this space, it is stored on disk and the memory space 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 will be executed while there is at least one base partition that needs to be processed (lines 3-16). In each iteration, the routine processes all the base partitions executing *InmemorySimJoin* (in-memory routine) to identify SJ links in small partitions (line 5) and hibernating larger partitions, i.e., transferring any in-memory data to disk (line 6). Then, the routine processes the window-pair partitions (and their sub-partitions) until all their SJ links have been produced (lines 7-13). When all the window-pair partitions have been fully processed, the routine gets the

```
DBSimJoin(R, S, eps, numPiv, memT)
Input: R and S (input data), eps, numPiv (No. of pivots), memT (memory threshold)
Output: the result of the Similarity Join between R and S
1.  create basePList and winPairPList
2.  PartitionBasePart(R ∪ S, basePList, winPairPList, eps, numPiv)
3.  while basePList.size > 0 do
4.    for each partition P of basePList do
5.       if P ≤ memT then InmemorySimJoin(P, eps)
6.       else HibernatePartition(P)
7.    while winPairPList.size > 0 do
8.       for each partition W of winPairPList do
9.         if W ≤ memT then InmemorySimJoinWin(W, eps)
10.        else HibernatePartition(W)
11.      if winPairPList.size > 0 then
12.        W ← winPairPList.getFirst()
13.        PartitionWinPairPart(W, winPairPList, eps, numPiv)
14.   if basePList.size > 0 then
15.      P ← basePList.getFirst()
16.      PartitionBasePart(P, basePList, winPairPList, eps, numPiv)
17. delete basePList and winPairPList
```

Fig. 5. DBSimJoin's main executor routine.

first base partition that needs further processing and repartitions it calling *PartitionBasePart* (lines 14-16). 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 links starting at the first round. This behavior enables the support of the iterator interface and its *getNext* function. The algorithm also prioritizes the processing of window-pair partitions before base partitions. This reduces the number of partitions that the routine needs to keep track of. Window-pair partitions are in general significantly smaller than base partitions. Thus, in general, it takes less time to reach the point where they can be processed in memory.

The main routine calls *PartitionBasePart* and *PartitionWinPairPart* to partition a base and a window-pair partition, respectively. *PartitionBasePart* randomly selects *numPiv* pivots and processes each tuple t adding it to the new base (*basePList*) and window-pair (*winPairPList*) partitions this tuple belongs to. This involves: (1) adding t to the base partition corresponding to its closest pivot p, and (2) adding t to all the window-pair partitions (corresponding to pivots p and i) where $gen\_hyperpln\_dist(t, p, i) \leq eps$. Fig. 6 shows an example of the partitions generated by *PartitionBasePart* with pivots P_0 and P_1. T is partitioned into $P0$, $P1$ and $P0\_P1$. Note that the tuples of the window-pair partition have an extra attribute that specifies their previous partition. This is used during the generation of window links and also to correctly repartition this partition. *PartitionWinPairPart* is similar to *PartitionBasePart* but distinguishes between the two window-pair partitions of any pair of pivots. Also, all the generated partitions are added to *winPairPList* since the links generated in a window-pair partition should always be window links (links between tuples of different previous partitions). Fig. 7 shows a partitioning generated by *PartitionWinPairPart*. $P0\_P1$ is partitioned into $Q0$, $Q1$, $Q0\_Q1\{1\}$, and $Q0\_Q1\{2\}$.

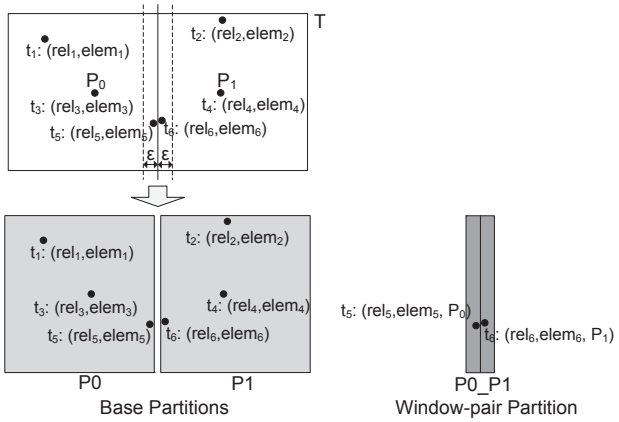

Fig. 6. Partitioning the tuples of a base partition.

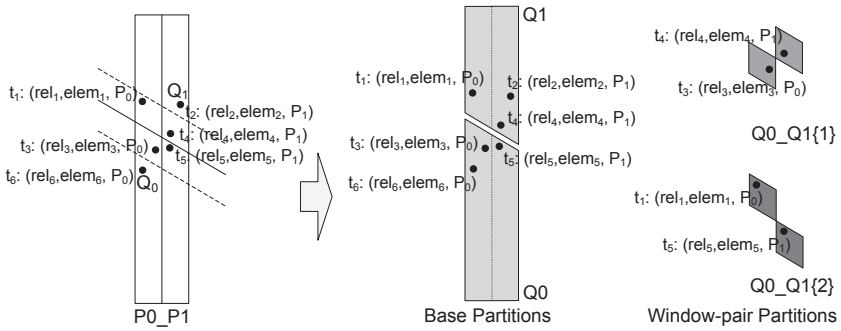

Fig. 7. Partitioning the tuples of a window-pair partition.

Implementation Using the Iterator Interface The DBSimJoin algorithms presented in the previous subsection are realized in a way that enables generating links one at a time, i.e., using the iterator interface and its function *getNext*. 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 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 call in turn calls the *getNext* function of its children nodes until it gets the required information to produce a result tuple. This process is propagated top-down. A non-blocking behavior reduces query response time. When *getNext* 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 *getNext* routine is implemented in the fashion of a state machine that uses the states and transitions presented in Fig. 8. When

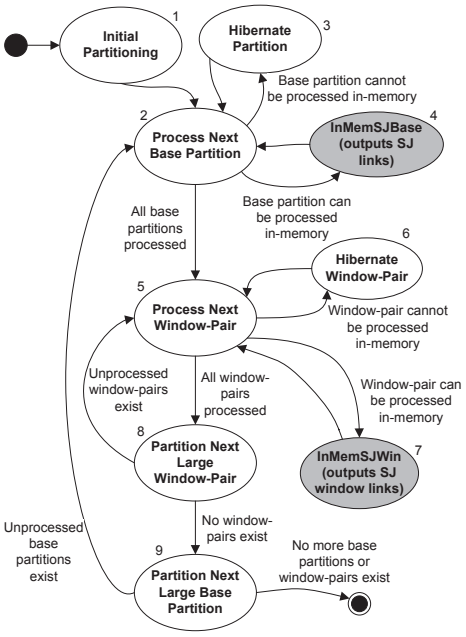

Fig. 8. DBSimJoin's GetNext.

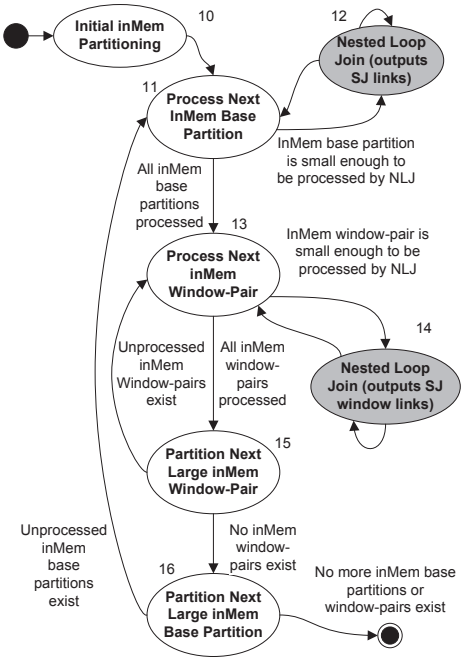

Fig. 9. Details of InMemSJBase.

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.

3.4 Analysis of I/O Cost and Number of Pivots

The work in [12] 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 ε of the partition boundary. DBSimJoin's analysis is similar to the one of Quickjoin but considers we have a limited number of buffer pages B to store the partitions. Moreover, each partition is assigned L buffer pages to store its data. If the partition grows beyond this space, it is 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 ε and close to $O(N^2)$ for large ε.

P (No. of partitions in a round) is related to the number of pivots K. Given K pivots, DBSimJoin generates K base partitions and $K^2 - K$ window-pair partitions, that is $P = K^2$. Since $P \leq \frac{B}{L}$, we have that $K \leq \lfloor\sqrt{\frac{B}{L}}\rfloor$. We 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.

4 Performance Evaluation

We implemented DBSimJoin in PostgreSQL 8.2.4. In this section we compare DBSimJoin with other approaches proposed for database systems. We do not compare DBSimJoin with standalone algorithms. These algorithms can outperform database implementations since they are not affected by database features like transaction processing, recovery, etc. The experiments used an Intel Core i5 2.27 GHz machine (4GB RAM, 500GB/5400RPM hard disk, Linux). 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 [9]. 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 [1]. 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.

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 in SF1. For vector data, the datasets for higher SF were obtained adding shifted copies of the SF1 dataset where the distance between copies were greater than the maximum value of ε. For string data, the datasets for higher SF were obtained adding a copy of the SF1 data where characters are shifted similarly to the process in [18]. The records of each dataset are equally divided between R and S. We used Euclidean and 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.

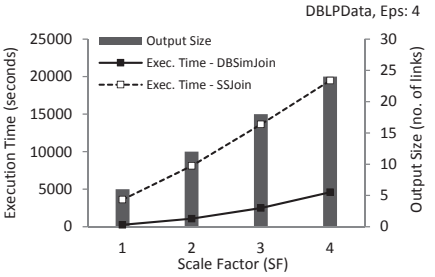

Fig. 10. Increasing SF - DBLPData.

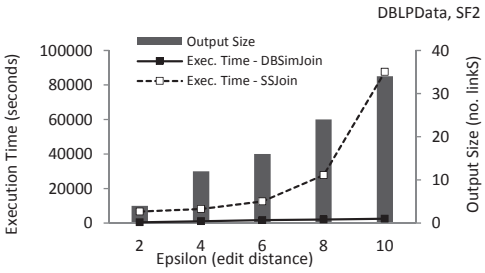

Fig. 11. Increasing Epsilon - DBLPData.

4.1 Performance Evaluation with DBLP String Data

We compare DBSimJoin with an implementation of SSJoin (q=3), the q-gram based approach proposed in [4] and explained in Section 2. SSJoin's execution times do not include q-gram generation. Sample queries are presented next.

```
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;
```

Increasing Scale Factor Fig. 10 shows the performance when data size increases. DBSimJoin's execution time is between 7% (SF1) and 24% (SF4) of the one of SSJoin. DBSimJoin also uses significantly less space than SSJoin. In our experiments, for SF1, SSJoin's q-gram tables have about 55 times the number of rows of the original tables. DBSimJoin uses only the original tables.

Increasing Epsilon Fig. 11 compares the performance 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 since larger values of ε generate larger window-pair partitions.

4.2 Performance Evaluation with Vector Data

We run all the tests using both vector datasets and found the same performance trends. We present the results using one dataset (specified in each figure) due to space constraints. We compare DBSimJoin with queries that produce the same results using only regular (non-similarity) database operators (RegDBOps). To

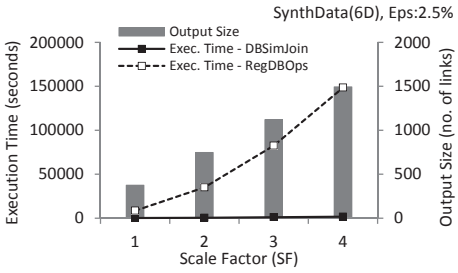

Fig. 12. Increasing SF - SynthData.

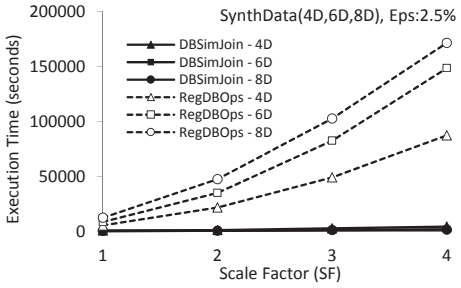

Fig. 13. Increasing SF and Number of Dimensions - SynthData.

the best of our knowledge, no previous work has proposed an alternative approach to support SJ over vectors in a DBMS. PostGIS, a spatial database extender for PostgreSQL [2], 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. Some sample queries are presented below.

```
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;
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;
```

Increasing Scale Factor Fig. 12 shows how 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. RegDBOps' execution time grows from being 33 times the one of DBSimJoin for SF1 to 87 times for SF5. RegDBOps' poor performance is due to a nested loop join between the joined relations.

Increasing SF and Number of Dimensions DBSimJoin perform much better than RegDBOps 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 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 (46,109-184,436) and the ones reported in 6D and 8D data ($< 1{,}500$).

Increasing Epsilon Fig. 15 shows that, for all the evaluated values of ε, DBSimJoin significantly outperforms RegDBOps. The execution time of DBSimJoin is only 0.42% of that of RegDBOps for ε=0.5% and 2.97% for ε=2.5%.

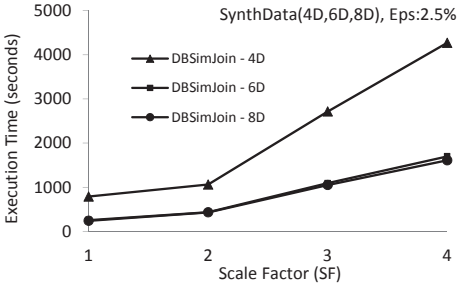

Fig. 14. Increasing SF and Number of Dimensions (DBSimJoin) - SynthData.

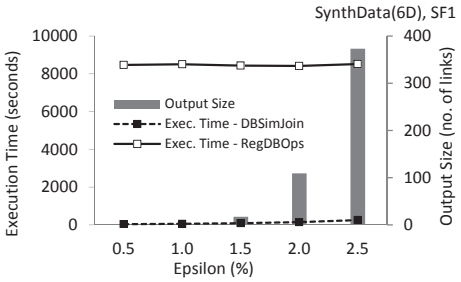

Fig. 15. Increasing Epsilon - SynthData.

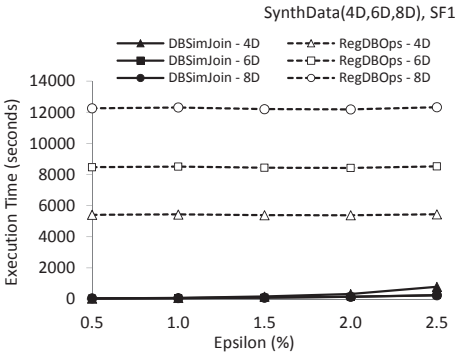

Fig. 16. Increasing Epsilon and Number of Dimensions - SynthData.

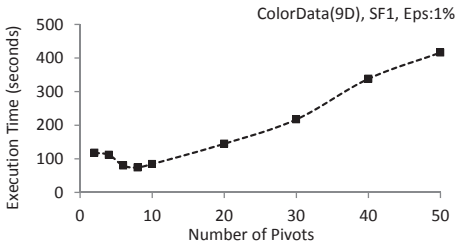

Fig. 17. Increasing No. of Pivots - ColorData.

RegDBOps' execution time remains almost constant 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.

Increasing Epsilon and Number of Dimensions DBSimJoin performs significantly better than RegDB-Ops 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 and only 1.9% for 8D.

Varying Number of Pivots Fig. 17 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 ($K = 8$), 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.

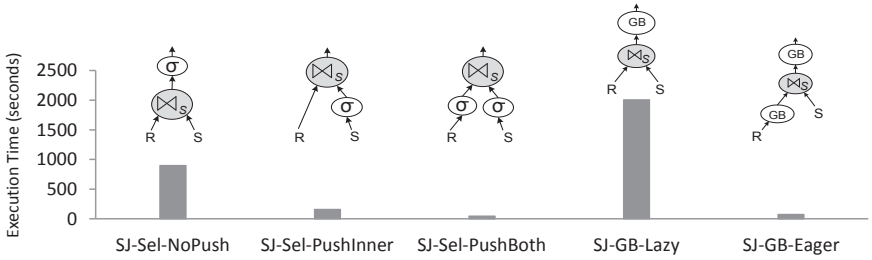

Fig. 18. Combining DBSimJoin with other Operators.

4.3 Combining DBSimJoin with other Database Operators

This section shows that DBSimJoin can be combined with other database operators and used in important transformation rules. We use the following queries.

```
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;
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];
```

This section uses ColorData (SF1) with the following changes. Table S in SJ-GB has 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 duplicate tuples, table R in SJ-GB is generated taking 20% of the original dataset and duplicating each tuple 5 times. The thresholds 0.28 and 1.38 correspond to ε=1.0% and ε=5.0%, respectively. SJ-Sel-NoPush, SJ-Sel-PushInner and SJ-Sel-PushBoth in Fig. 18 are different ways to execute SJ-Sel. SJ-Sel-NoPush executes the SJ first and then the selection operator. In SJ-Sel-PushInner, the selection operator ($\sigma_{EucDist(S,Const)\leq 1.38}$) is pushed to 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). SJ-Sel-PushBoth's execution time is only 5% of the one of SJ-Sel-NoPush. SJ-GB-Lazy and SJ-GB-Eager correspond to the eager and lazy aggregation plans of SJ-GB. SJ-GB-Lazy executes SJ first and then group-by. Grouping 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.

5 Conclusions and Future Work

This paper presents DBSimJoin, an efficient and non-blocking SJ database operator. DBSimJoin supports the iterator interface and uses a sequence of rounds

that prioritizes the quick generation of results. DBSimJoin can be used with multiple data types and distance functions. We present the implementation details of DBSimJoin and extensively evaluate its performance showing that DBSimJoin outperforms alternative approaches. 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 operations as database operators, (2) indexing techniques to improve the efficiency of similarity queries, and (3) database queries with multiple similarity operators.

References

1. Dblp bibliography. `http://www.informatik.uni-trier.de/~ley/db/`.
2. PostGIS. `http://postgis.net/documentation`.
3. C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In SIGMOD '01: 379–388.
4. S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In ICDE '06: 5–.
5. S. Chaudhuri, V. Ganti, and R. Kaushik. Data debugger: An operator-centric approach for data quality solutions. *IEEE Data Eng. Bull.*, 29(2):60–66, 2006.
6. J.-P. Dittrich and B. Seeger. Gess: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In KDD '01: 47–56.
7. V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. Similarity join in metric spaces. In ECIR '03: 452–467.
8. V. Dohnal, C. Gennaro, and P. Zezula. Similarity join in metric spaces using ed-index. In DEXA '03: 484–493.
9. A. Frank and A. Asuncion. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2010.
10. L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In VLDB '01: 491–500.
11. G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, 2003.
12. E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2):7:1–7:38, 2008.
13. R. Paredes and N. Reyes. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *J. of Discrete Algorithms*, 7(1):18–35, 2009.
14. Y. N. Silva, A. M. Aly, W. G. Aref, and P. -A. Larson. SimDB: a similarity-aware database system. In SIGMOD '10: 1243–1246.
15. Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In ICDE '10: 892–903.
16. Y. N. Silva, W. G. Aref, P. -A. Larson, S. Pearson, and M. H. Ali. Similarity queries: their conceptual evaluation, transformations, and processing. *VLDB Journal*, 22(3):395–420, 2013.
17. Y. N. Silva and S. Pearson. Exploiting database similarity joins for metric spaces. *Proc. VLDB Endow.*, 5(12):1922–1925, 2012.
18. R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In SIGMOD '10: 495–506.