Exploiting Database Similarity Joins for Metric Spaces

Yasin N. Silva
Arizona State University
4701 W. Thunderbird Road
Glendale, AZ 85306, USA
ysilva@asu.edu

Spencer Pearson
Arizona State University
4701 W. Thunderbird Road
Glendale, AZ 85306, USA
sspearso@asu.edu

ABSTRACT

Similarity Joins are recognized among the most useful data processing and analysis operations and are extensively used in multiple application domains. They retrieve all data pairs whose distances are smaller than a predefined threshold ε. Multiple Similarity Join algorithms and implementation techniques have been proposed. They range from out-of-database approaches for only in-memory and external memory data to techniques that make use of standard database operators to answer similarity joins. Recent work has shown that this operation can be efficiently implemented as a physical database operator. However, the proposed operator only support 1D numeric data. This paper presents *DBSimJoin*, a physical Similarity Join database operator for datasets that lie in any metric space. DBSimJoin is a non-blocking operator that prioritizes the early generation of results. We implemented the proposed operator in PostgreSQL, an open source database system. We show how this operator can be used in multiple real-world data analysis scenarios with multiple data types and distance functions. Particularly, we show the use of DBSimJoin to identify similar images represented as feature vectors, and similar publications in a bibliographic database. We also show that DBSimJoin scales very well when important parameters, e.g., ε, data size, increase.

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, multimedia applications and marketing analysis. 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 [4, 7], to techniques that use standard database operators to answer Similarity Joins [3]. Very little work, however, has addressed the implementation of Similarity Join as a first-class database operator. The work in [9, 8] showed the feasibility of this approach but proposed operators that support only 1D numeric data.

This paper presents *DBSimJoin*, a physical Similarity Join database operator for datasets that lie in any metric space. DBSimJoin extends the previously proposed standalone (non integrated with database engine) QuickJoin algorithm [7] by adapting it to the database engine framework and integrating techniques to: (1) enable a non-blocking behavior, (2) prioritize the early generation of results, and (3) fully support the iterator interface and its functions *open*, *getNext*, and *close*. We implemented the proposed operator in PostgreSQL [1], an open source database system. We show how this operation can be used in multiple real-world data analysis scenarios with multiple data types and distance functions. Particularly, we show the use of DBSimJoin to identify: (1) similar images represented as feature vectors, and (2) publications in the DBLP bibliographic database with similar titles. We show that DBSimJoin scales very well when important parameters, e.g., ε, data size, and number of dimensions, increase.

The implementation of Similarity Join as a physical database operator has the following advantages: (1) SJ operators can be interleaved with regular and similarity operators and their results pipelined for further processing; (2) important optimization techniques, e.g., pushing selection under join and pre-aggregation, can be extended to the new operator [9]; and (3) the implementation of SJ can reuse and extend other operators and structures to handle large datasets.

This paper presents DBSimJoin and the guidelines to implement it as an integrated component of a database system, and describes the demonstration scenarios.

2. THE DBSimJoin OPERATOR

The Similarity Join (SJ) operation between two datasets R and S is defined as $R \bowtie_{\theta_\varepsilon(r,s)} S = \{\langle r,s \rangle | \theta_\varepsilon(r,s), r \in R, s \in S\}$, where $\theta_\varepsilon(r,s)$ represents the Similarity Join predicate, i.e., $dist(r,s) \leq \varepsilon$. The result pairs $\langle r,s \rangle$ are referenced as *links*. Even though the tuples of relations R and S are combined by DBSimJoin, each tuple is assumed to have an attribute that identifies which relations the tuple belongs to.
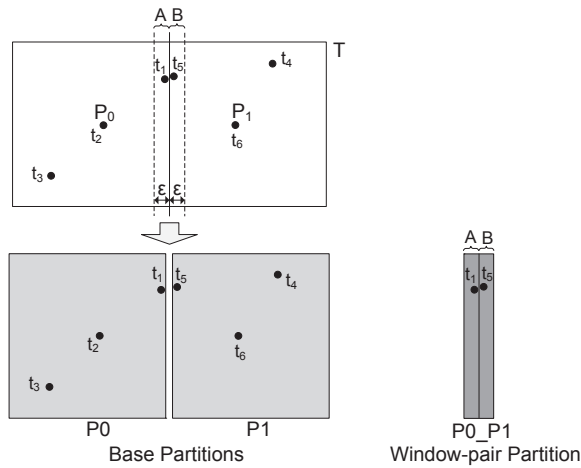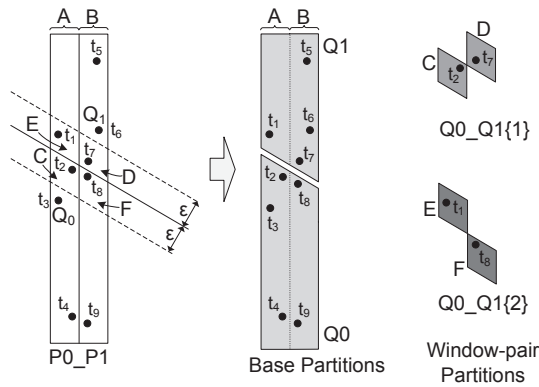
Figure 1: Repartitioning a base partition.

Figure 2: Repartitioning a window-pair partition.

DBSimJoin partitions the data until the partitions are small enough to be 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. Data partitioning is performed using a set of K pivots (subset of the tuples to be partitioned). The process generates two types of partitions. A *base partition* contains all the records that are closer to a given pivot than to any other pivot. A *window-pair partition* contains the records in the boundary between two base partitions. In general, the window-pair records are a superset of the records whose distance to the hyperplane that separates the base partitions is at most ε. Such hyperplane does not always explicitly exist in a metric space. Instead, the hyperplane is implicit and known as *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 [6]:

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

Processing the window-pair partitions guarantees the identification of the links between records that belong to different base partitions. Figure 1 represents the repartitioning of a base partition using two pivots. The SJ links in T is the union of the links in $P0$ and $P1$, and the links in $P0\_P1$

where one element belongs to window A and the other one to window B. We refer to this last type of links as *window links*. Figure 2 represents the repartitioning of a window-pair partition $P0\_P1$. 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 because 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 $Q1$. Rounds that identify all the links in the input data are referred to as base rounds. Rounds that identify only the window links, i.e., links between records that correspond to different previous partitions, are referred to as window-pair rounds.

While DBSimJoin's partitioning is conceptually similar to the one of QuickJoin [7], DBSimJoin is a non-blocking operator, uses a sequence of rounds that prioritizes the early generation of results and fully supports the iterator interface. The remaining part of this section presents guidelines to implement DBSimJoin inside the query engine of DBMSs.

2.1 The Parser and the Planner

To add support for Similarity Joins in the parser, the raw-parsing grammar rules, e.g., *yacc* rules in PostgreSQL, are extended to recognize the syntax of the new SJ predicates. The parse 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. Our implementation support the following SJ syntax.

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

In the planner, a new plan node is created for the DBSimJoin operator. This node is similar to the regular join node but stores also 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.

2.2 The Executor

The main DBSimJoin's executor routine 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 part of 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 3-28). 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 19-22). When all the window-pair

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. $W \leftarrow winPairPList.getFirst()$
21. $PartitionWinPairPart(W, winPairPList, eps, numPiv)$
22.   **end if**
23.   **end while**
24.   **if** $basePList.size > 0$ **then**
25. $P \leftarrow basePList.getFirst()$
26. $PartitionBasePart(P, basePList, winPairPList, eps, numPiv)$
27.   **end if**
28. **end while**
29. delete $basePList$ and $winPairPList$

---

partitions have been fully processed, the routine gets the first base partition that needs further processing and repartitions it calling *PartitionBasePart* (lines 24-27). After this step, the main while loop iterates again. *PartitionBasePart* and *PartitionWinPairPart* update the partition lists as follows: base partitions generated in a base round are added to *basePList*, all other partitions are added to *winPairPList*.

Unlike QuickJoin, the main routine prioritizes the early generation of links. After any partitioning step, the algorithm processes 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 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 smaller than base partitions. Consequently, in general, it takes less time to reach the point where they can be processed in memory.

Both *InmemorySimJoin* and *InmemorySimJoinWin* are in-memory routines that find SJ links and window links, respectively. They are implemented using a variation of the Quickjoin algorithm [7] that iteratively partitions the data in memory until the partitions are small enough to solve
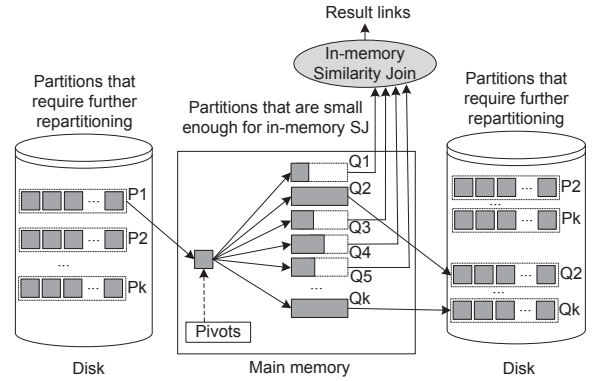

Figure 3: Round I.

the SJ using a nested loop join. These routines are also implemented using a non-blocking approach to minimize the time to produce the next result link.

Figure 3 represent the processing performed by the main routine in a generic round I. The round repartitions $P1$. Some generated partitions are small enough to be processed by the in-memory SJ routines , e.g., $Q1$, $Q3$, $Q4$ and $Q5$, the remaining ones are stored on disk, e.g., $Q2$ and Qk.

DBSimJoin's routines 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. The *open* routine initializes data structures and computes the value of the number of pivots based on the available memory. The *close* routine deletes all the temporary data structures. The *getNext* routine is implemented in the fashion of a state machine that uses the states and transitions presented in Figure 4. 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 (3 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 these routines are similar to the ones of *getNext* but use nested loop join instead of InMemSJBase and InMemSJWin, and do not hibernate data.

3. DEMONSTRATION SCENARIOS

The demonstration of DBSimJoin will be performed using our implementation in PostgreSQL 8.2.4. The demo user will be able to interactively run arbitrary SJ queries using the syntax presented in Section 2.1. The user will also be able to configure and run SJ queries in two particular real-world data analysis scenarios using multiple data types and distance functions. In each demonstration scenario, the execution time of DBSimJoin will be compared to the ones of alternative approaches. Furthermore, we will show how the different approaches scale when important parameters, e.g., ε, data size and number of dimensions, increase. To facilitate the understanding of the inner workings of SJ queries,
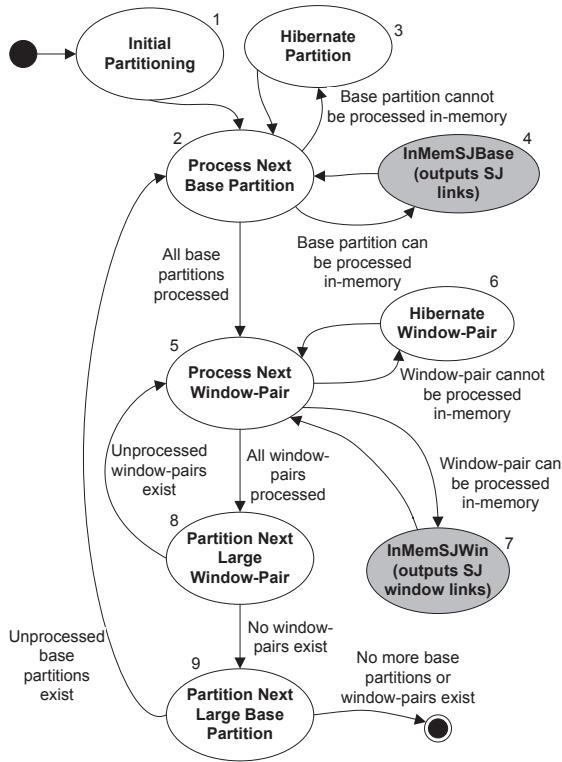
Figure 4: DBSimJoin's GetNext.

Figure 5: Finding similar images in *ColorMoments*.

we will show execution statistics, query plans and animations of the state machine-based DBSimJoin algorithm.

3.1 Identifying Similar Images

In this scenario, DBSimJoin is used to identify similar images in the Corel image collection [5] as shown in Figure 5. The scenario uses two different datasets: *ColorMoments* and *CoOccurrenceTexture*. Both dataset have 68,040 records each (scale factor 1). Each record of *ColorMoments* is a 9D feature vector with components in the range [-4.8 - 4.4]. Each vector contains the following values: the mean, standard deviation, and skewness for each of H, S and V in the HSV color space. Each record of *CoOccurrenceTexture* is a 16D feature vector. *CoOccurrenceTexture* was generated converting the images to 16 gray-scale images. Each vector contains the following values: the Second Angular Moment, Contrast, Inverse Difference Moment, and Entropy in 4 directions (horizontal, vertical, and two diagonal directions). We use the Euclidean distance function to measure the similarity between images. The queries in this scenario show that DBSimJoin significantly outperforms queries that get similar results using only regular, i.e., non-similarity, operators.

3.2 Identifying Similar Publications

In this scenario, DBSimJoin queries are used to identify publications with similar titles in the DBLP bibliographic dataset [2]. The scale factor 1 dataset has 10,000 records. The title of each publication record is a string of 7 to 342 characters. The Levenshtein distance function is used to measure the similarity between publication titles.
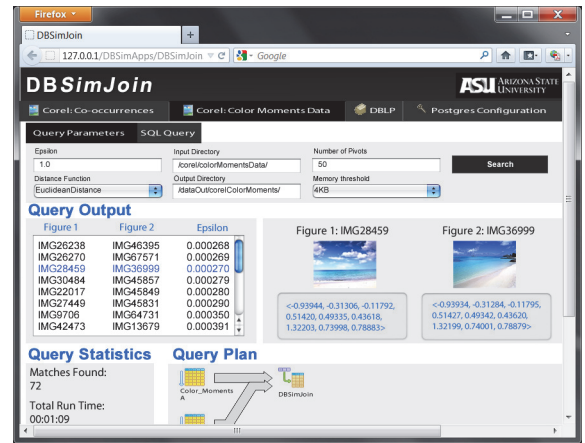
The queries in this scenario show that DBSimJoin performs significantly better than q-gram based techniques [3].

4. CONCLUSIONS

Similarity Join is recognized as one of the most useful data analysis operations. While multiple implementation techniques have been proposed for this operation, very little work has addressed its study as a physical database operator. This paper presents *DBSimJoin*, a physical Similarity Join database operator for any dataset that lies in a metric space. We implemented the proposed operator in PostgreSQL and show how this operator can be used in multiple real-world data analysis scenarios with multiple data types and distance functions. Particularly, we show the use of DB-SimJoin to identify similar images and similar publications. We show that DBSimJoin scales very well when important parameters increase.

5. REFERENCES

[1] PostgreSQL. http://www.postgresql.org/.
[2] DBLP Bibliography.
    http://www.informatik.uni-trier.de/~ley/db/.
[3] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–, 2006.
[4] V. Dohnal, C. Gennaro, and P. Zezula. Similarity join in metric spaces using ed-index. In *Database and Expert Systems Applications*, volume 2736 of *Lecture Notes in Computer Science*, pages 484–493. 2003.
[5] A. Frank and A. Asuncion. UCI machine learning repository. http://archive.ics.uci.edu/ml, 2010.
[6] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28(4):517–580, 2003.
[7] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2):7:1–7:38, 2008.
[8] Y. N. Silva, A. M. Aly, W. G. Aref, and P.-A. Larson. Simdb: a similarity-aware database system. In *ACM SIGMOD*, pages 1243–1246, 2010.
[9] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *ICDE*, pages 892–903, 2010.