The Similarity-aware Relational Database Set Operators

Wadha J. Al Marri and Qutaibah Malluhi[a], Mourad Ouzzani[b], Mingjie Tang and Walid G. Aref[c]

[a]*Qatar University, Qatar*
[b]*Qatar Computing Research Institute, Qatar*
[c]*Purdue University, USA*

Abstract

Identifying similarities in large datasets is an essential operation in several applications such as bioinformatics, pattern recognition, and data integration. To make a relational database management system similarity-aware, the core relational operators have to be extended. While similarity-awareness has been introduced in database engines for relational operators such as joins and group-by, little has been achieved for relational set operators, namely *Intersection*, *Difference*, and *Union*. In this paper, we propose to extend the semantics of relational set operators to take into account the similarity of values. We develop efficient query processing algorithms for evaluating them, and implement these operators inside an open-source database system, namely PostgreSQL. By extending several queries from the TPC-H benchmark to include predicates that involve similarity-based set operators, we perform extensive experiments that demonstrate up to three orders of magnitude speedup in performance over equivalent queries that only employ regular operators.

1. Introduction

Diverse applications, e.g., bioinformatics (Narayanan and Karp, 2004), data compression (Wang et al., 2011), data integration (Schallehn et al., 2004), and statistical classification (Mills, 2011), require similarity-awareness

Email addresses: `200450064@student.qu.edu.qa and qmalluhi@qu.edu.qa` (Wadha J. Al Marri and Qutaibah Malluhi), `mouzzani@qf.org.qa` (Mourad Ouzzani), `aref@cs.purdue.edu and tang49@purdue.edu` (Mingjie Tang and Walid G. Aref)

capabilities for identifying similar objects. Several similarity-aware relational operators that introduce similarity processing at the database engine level have been proposed in the past. These operators include similarity joins and similarity group-by's (Silva et al., 2010b), (Silva et al., 2009), (Silva et al., 2013). However, little attention has devoted to the class of relational set operations.

In standard SQL, relational set operations are based on exact matching. However, assume that we want to find common or different readings that are produced by two sensors. Assume further that the sensor readings are stored in two separate tables. The standard SQL set intersect or set difference (except) operators are not suitable for applying standard set intersection or set difference on these two sensor-data tables to get the common/different sensor readings. The reason is that sensor readings may be similar but not necessarily identical. Thus, it is desirable to perform similarity set operations on the two sensor-data tables to find similar or different readings. In this paper, we introduce similarity-aware set intersection, difference, and union as extended relational database operators.

This paper is a generalization of our previous work (Marri et al., 2014). In addition to the similarity set intersect operator that we present in (Marri et al., 2014), in this paper, we introduce the other similarity set operators, namely similarity set difference and union. We analyze their corresponding semantics and provide efficient algorithms for each operator. In addition, we realize these operators inside an open-source relational database management systems (DBMS) and provide an extensive experimental study of their performance.

The contributions of this paper are as follows.

- We introduce the similarity-aware relational set operators that extend the standard SQL relational set operators to produce results based on similarity rather than on equality (Section 3).
- We develop efficient algorithms for the proposed similarity-aware relational set operators (Section 4) and implement them inside PostgreSQL, an open-source relational database management system (PostgreSQL, 2015) (Section 5).
- We evaluate the performance and the scalability of the proposed algorithms using the TPC-H benchmark (TPC, 2015). We extend several

queries from the TPC-H benchmark by including predicates that involve similarity-based set operators. Performance results demonstrate up to three orders of magnitude enhancement in performance over equivalent queries that employ only regular relational operators (Section 5).

2. Related Work

Similarity-awareness in relational operators has been mainly addressed in the relational join and group by operators. There has been work in terms of devising efficient algorithms as well integrating these similarity-aware operators inside a database engine. Another line of related research deals with nearest neighbor search as one form of similarity. In this section, we go over the main contributions in these different facets.

A nearest neighbor (NN) search finds the closest object to a query focal point. There are mainly two variants, the k-NN (Seidl and Kriegel, 1998) and all-NN (Clarkson, 1997) operations. A k-NN operation identifies the k closest data objects to a query focal point, whereas the all-nearest-neighbor operation finds for each object in the outer table, its closest object(s) in the inner table. (Lian and Chen, 2008) propose an efficient similarity search algorithm by employing pruning techniques to find objects in selected subspaces instead of the full space. Other performance improvement mechanisms are achieved by exploiting indexing structures, e.g., the M-tree (Ciaccia et al., 1997) and the slim-tree (Traina Jr et al., 2002).

Similarity join retrieves objects from the two relations that overlap based on a predefined threshold. Many types of similarity join have been proposed, e.g., (Yu et al., 2007; Hjaltason and Samet, 1998; Silva et al., 2010b; Arasu et al., 2006; Böhm and Krebs, 2004). k-nearest neighbor join (kNN join) is a similarity join that combines each element in a dataset, say R, with the k-nearest elements in another dataset, say S. Böhm and Krebs (Böhm and Krebs, 2004) compute the kNN join using the multipage index (MuX). MuX is an R-tree-based method to solve the optimization conflicts between the CPU cost and the I/O cost. MuX uses large-sized pages for the input data to reduce the I/O cost. Then, a secondary structure, namely buckets, with a much smaller size within pages, is used to optimize the CPU time. Recent approaches have investigated employing MapReduce to perform kNN join (Lu et al., 2012) and hamming distance based similarity join (Tang et al., 2015). The Quickjoin (Jacox and Samet, 2008) is a metric-space algorithm that

works by processing a nested-loops join on smaller subsets that are obtained after partitioning the dataset recursively. Join-Around (Silva et al., 2010b) uses some properties of the distance and kNN joins. In addition to having each object from the first set join with its closest object in the second set, only the pairs within a pre-specified distance or radius are reported. There are several similarity join algorithms that are based on the application of grids to multidimensional datasets, e.g., Epsilon Grid Order (EGO) (Böhm et al., 2001) and the Generic External Space Sweep (GESS) (Dittrich and Seeger, 2001). EGO is designed to process the similarity join on massive datasets. This solution is based on obtaining a sort order of the data objects by setting an equi-distant grid with cell length ϵ over the data space and comparing the grid cells lexicographically. GESS associates with each point an ϵ-length hypercube and then executes an intersection join on these hypercubes.

The Trie-Join (Wang et al., 2010), Fast-Join (Wang et al., 2011), ED-Join (Xiao et al., 2008), Part-Enum (Arasu et al., 2006), and SSjoin (Chaudhuri et al., 2006) are methods for string similarity joins. In the Trie-Join approach, a trie-based structure indexes the strings and a sub-trie pruning technique is used to efficiently perform the similarity join. SSjoin, denoting set similarity join, presents strings as sets of q-grams. Based on the string sets, SSjoin applies an overlapping function to exclude the non-matching string pairs. Then, distance computation is performed on the pairs that satisfy the overlapping condition. The other string similarity joins, namely, Part-Enum, Fast-join, and Ed-Join, employ a filter-and-refine framework. In the filter step, they produce candidate pairs by using string signatures. In the refine step, the candidate pairs are tested to see whether they are part of the final result or not.

The similarity group-by operator assigns every object to a group based on the similarity condition. A similarity-based group-by is useful in data mining applications, e.g., clustering, and duplicate detection and elimination. Group-by-Context and Group-by-Similarity are presented in (Schallehn et al., 2002, 2001, 2004; Schallehn and Sattler, 2003; Tang et al., 2014). Group-by-Context provides a mechanism for applying user-defined functions for grouping purposes. In contrast, Group-by-Similarity is a special case of context-aware grouping that provides the possibility to describe the similarity among tuples and grouping strategies in a descriptive way. In (Silva et al., 2009), the authors extend the standard database group-by operation to form groups of similar tuples. They implement three instances of the similarity grouping operator. Unsupervised Similarity Group-by (U-SGB)

produces similarity groups based only on the specification of group properties (compactness and size). Supervised Similarity Group Around (SGB-A) forms the groups around certain central points of interest and restricts their extent based on group properties. Supervised SGB using Delimiters (SGB-D) identifies groups based on a set of delimiting points.

In order to enable similarity queries into an RDBMS, an extension to SQL to support nearest-neighbor queries is studied in (Gao et al., 2004). This extension offers the ability to express the nearest neighbor queries in the RDBMS through a user-defined predicate termed NN-UDP. Another work (Barioni et al., 2005, 2006) allows expressing similarity queries in SQL and executing them via a similarity retrieval engine, called SIREN (SImilarity Retrieval ENgine). SIREN is a service implemented between an RDBMS and the application programs. It processes and answers every similarity-based SQL command sent from the application. The regular SQL commands are forwarded to the RDBMS and the answers are sent back from the RDBMS to the application program. In (Silva et al., 2010a, 2012), extensions to SQL make the similarity operators first-class database operators by implementing the operators inside the database engine.

Parallel to the presented work to support similarity-aware operators, we propose new similarity set operators that extend the standard relational set operators and that are evaluated using similarity predicates. In contrast to realizing these operators as UDFs, we realize them inside the database engine. In addition to enhancement in performance, integrating these operators into a database engine allows for the interleaving of these similarity operators with other database operators.

3. Semantics of Similarity-based Relational Set Operators

3.1. Distance and Similarity

Let Q (resp. P) be a relation with k attributes denoted by $a_1, a_2, \ldots, a_k$ (resp. b) and n (resp. m) tuples $A_1, A_2, \ldots, A_n$ (resp. B), where the schemas of P and Q are union compatible as required by standard relational set operations. To express the similarity between two tuples, one may use several possible functions to describe the distance between each pair of corresponding attribute values, e.g., edit distance, p-norm, or Jaccard distance. Let $D = \{dis_1, dis_2, \ldots, dis_r\}$ be r distance functions. For any $dis_t \in D$, let $dis_t(A_i.a_t, B_j.a_t)$ be the distance corresponding to attribute a_t between the tuples A_i and B_j using the distance function dis_t.

In this paper, we adopt the following similarity predicate: Given r thresholds $\epsilon_1, \epsilon_2, ..., \epsilon_r$ that correspond to each of the attributes $a_1, a_2, \ldots, a_r$, respectively, where $r \leq k$, we say that two tuples A_i and B_j match iff: $pred(A_i, B_j) = \quad dis_1(A_i.a_1, B_j.a_1) \leq \epsilon_1$ AND $dis_2(A_i.a_2, B_j.a_2) \leq \epsilon_2 \ldots$ AND $dis_r(A_i.a_r, B_j.a_r) \leq \epsilon_r$.If $r < k$, the set of thresholds $\epsilon_{r+1}, \ldots, \epsilon_k$ are assumed to have the value zero. An ϵ_i of value zero has to be assigned explicitly if at least one later attribute is assigned an $\epsilon >0$. Furthermore, an ϵ_i can be assigned an infinity value.

3.2. Similarity-aware Set Intersection

Similarity-aware Set Intersection takes the tuples of two tables as input and returns only those tuple pairs that are similar within a threshold from both tables. More formally, given two tables, say P and Q, that have union compatible schemas, and a similarity predicate $pred(A, B)$, the similarity-aware set intersection operation is defined as follows.

$$\begin{aligned} Q \mathbin{\widetilde{\cap}} P = \{A \mid A \in Q,\ \exists\ B \in P : pred(A, B)\} \\ \cup \\ \{B \mid B \in P,\ \exists\ A \in Q : pred(A, B)\} \end{aligned} \tag{1}$$

$Q \mathbin{\widetilde{\cap}} P$ has the same schema as Q and P.

Example: Consider the following two union compatible tables Q and P; each with one single attribute. If an attribute value x from Q has a similar attribute value in P, then that value is denoted as $\widetilde{x}$. $Q = \{a, b, c, d, e, f, g, z\}$ and $P = \{\widetilde{a}, \widetilde{b}, \widetilde{c}, h, i, j, k, l, z\}$ For all calculated $pred(t_1, t_2)$ such that $t_1 \in P$ and $t_2 \in Q$, only $pred(a, \widetilde{a}), pred(b, \widetilde{b})$, $pred(c, \widetilde{c})$, and $pred(z, z)$ evaluate to true. Thus, $P \mathbin{\widetilde{\cap}} Q = \{a, b, c, \widetilde{a}, \widetilde{b}, \widetilde{c}, z\}$.

Three-way similarity-aware set intersection, denoted by $\widetilde{\cap}$, is defined as follows. Let Q, P, and R be three tables such that $\widetilde{\cap}(Q, P, R) = U$. Each tuple in U exists in at least one table and has two similar tuples in the two other tables such that these two tuples are also similar to each other. This can easily be extended to more than three tables. We skip the formal definition of the three-way and multi-way similarity intersect operators for brevity.

Example: In addition to the tables P and Q, given in the previous example, let $R = \{\widetilde{\widetilde{a}}, \widetilde{\widetilde{b}}, v, y\}$. Assume further that $pred(a, \widetilde{\widetilde{a}})$, $pred(\widetilde{a}, \widetilde{\widetilde{a}})$, and $pred(b, \widetilde{\widetilde{b}})$ hold. Thus, applying the three-way similarity set intersect operator produces:

$\widetilde{\cap}(P,Q,R)$=$\{a,\widetilde{a},\widetilde{\widetilde{a}}\}$. Notice that because $pred(\widetilde{b},\widetilde{\widetilde{b}})$ does not hold, $b,\widetilde{b},\widetilde{\widetilde{b}}$ are not part of the answer.

3.3. Similarity-aware Set Difference

Similarity-aware Set Difference is an operator that returns the tuples from one table such that these returned tuples do not have similar tuples satisfying the similarity predicate in the other table. A Similarity-aware Set Difference operation of P and Q is denoted by $Q \stackrel{\sim}{-} P$. Given two tables Q and P having identical (or compatible) schemas and the similarity predicate $pred(A,B)$, the Similarity-aware Set Difference operator can be defined as follows:

$$Q \stackrel{\sim}{-} P = \{A \mid A \in Q \ \wedge \ !\exists B \in P : pred(A,B)\} \tag{2}$$

Example: For the tables from the previous example, the result of the Similarity-aware Set Difference of Tables Q and P is: $Q \stackrel{\sim}{-} P = \{e,f,d,g\}$

3.4. Similarity-aware Set Union

In set theory, the intersection (or difference) of sets is a subset of the union of these sets. Thus, the similarity union should be defined such that it includes the results of the similarity set intersection as well as the similarity set difference of the input sets. Notice that the similarity-aware set union works in the same way as the standard union. Therefore, it is not discussed in subsequent sections. Another option that is beyond the scope of this paper is to leverage (Pola et al., 2013); the duplicate elimination function is extended to consider the very similar tuples as duplicates; then the similarity union can apply this similarity operator after combining the tuples from the input tables.

It can be easily verified that the distributive law applies for similarity interesection over (similarity) union; i.e., $Q \,\widetilde{\cap}\, (P \cup R) = (Q \,\widetilde{\cap}\, P) \cup (Q \,\widetilde{\cap}\, R)$. On the other hand, union cannot be distributed over similarity intersection; i.e., $Q \cup (P \,\widetilde{\cap}\, R) \neq (Q \cup P) \,\widetilde{\cap}\, (Q \cup R)$. This can be seen by considering a tuple in P which does not have a similar tuple in R but has a similar tuple in Q. This tuple will be part of the output for $Q \cup (P \,\widetilde{\cap}\, R)$ but will be part of the output for $(Q \cup P) \,\widetilde{\cap}\, (Q \cup R)$.

3.5. Extended SQL Syntax for Similarity-aware Set Operations

We extend SQL to introduce the similarity-aware set operators in the following way.

(**SELECT** $a_1, a_2, ...$ **FROM** *table*$_1$
[**SETOP**]
SELECT $a_1, a_2, ...$ **FROM** *table*$_2$
[**SETOP**]
...
SELECT $a_1, a_2, ...$ **FROM** *table*$_n$
) **WITHIN VALUES** (ϵ_1,ϵ_2,...)

Where SETOP is either INTERSECT, EXCEPT, or UNION. For example, multi-way similarity-aware set intersection is expressed by multiple INTERSECT keywords between the sub-queries with the same parentheses. The phrase **WITHIN VALUES** provides the similarity thresholds for each of the attributes participating in the similarity set operation.

3.6. Implementation-Independent Similarity-set Operator

It is important for the semantics of the proposed similarity-set operators to be implementation-independent, that is, regardless of the way each similarity-set operator is implemented in a database engine, it should return the same results. To illustrate, assume to the contrary, that the similarity-aware set intersection is defined as: For each element in one set, say e, report e if it has a similar element in the other set, and from the other set report only the first matching element. One implementation of this definition may sort the input sets and start a sequential reading over the other data set looking for matching tuples. Another implementation may start without sorting. Typically, these two different implementations will produce different results. It is important to define similarity set operators in a way that is implementation independent.

To demonstrate that the semantics of our similarity set operators are implementation-independent, we express them using standard relational operators that are already known to be implementation-independent. We use the standard Join ($\bowtie$), Project (Π), and Union ($\cup$) to realize the similarity intersection set operator and hence demonstrate the implementation-independence of the latter. The similarity intersect operator on two relations Q and P can be expressed using standard relational operators as illustrated in Figure 1.

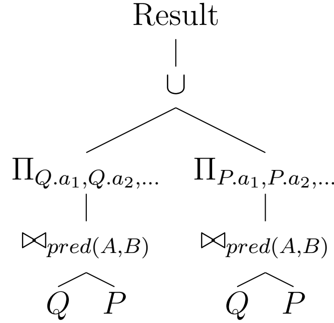


Figure 1: Expressing Similarity Set Intersection Using Relational Operators.

This tree applies a theta-join between the two relations using the similarity predicate on all attributes involved in the similarity-aware intersection. Then, a projection operator is performed twice to separate the joined tuples into two tables, one corresponding to Q's tuples and the other to P's tuples. Finally, a union operator is applied to combine the sub-results.

To express the similarity set difference, we use another standard relational operator, namely Except ($-$), that excludes the tuples from one table that appear in the other table. The similarity-aware set difference is expressed in Figure 2. The leaf level of the tree is a join to combine each tuple in one table to its matching tuple(s) in the other table based on the similarity predicate. Then, a projection is performed to get the matching tuples from only the left relation. Finally, an except operator excludes these matching tuples from the left relation.

From the above, the two similarity operators can be expressed using standard relational operators and hence the proposed semantics are necessarily implementation-independent.

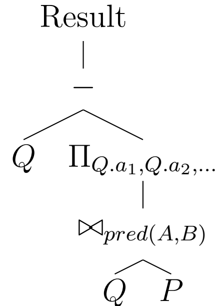


Figure 2: Expressing Similarity Set Difference Using Relational Operators.

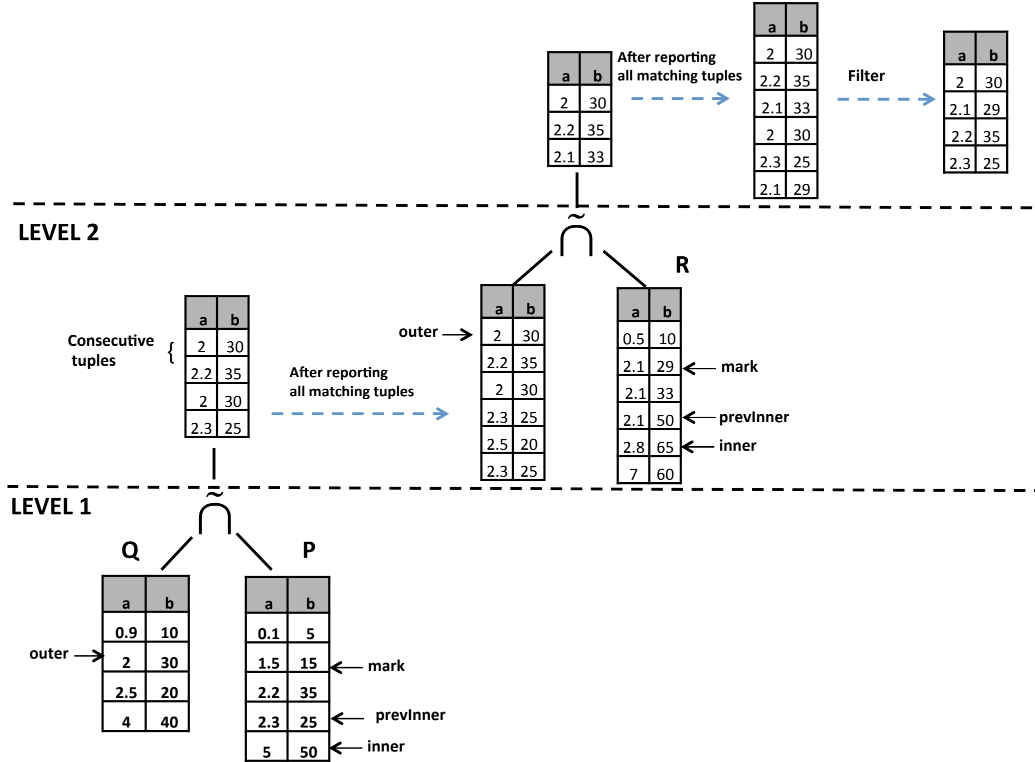


Figure 3: Sample execution: Sim-Intersect. Threshold list={0.5,5}

4. Algorithms for Realizing the Similarity-aware Set Operators

In this section, we realize the proposed similarity-aware set operators. The query processing algorithm for a similarity set operator is an extension of the sort-merge join algorithm. The first step of the algorithm sorts both input tables unless they are already sorted. We thus assume that our algorithms work for totally ordered attributes only. In high-level terms, the similarity set operator compares tuples based on a Mark/Restore mechanism that avoids the $O(n^2)$ complexity that would result from a nested-loops implementation. To find matching/non-matching tuples between two tables (named the outer and inner tables), the Mark/Restore mechanism marks the position of a tuple that may need to be restored later if some condition is satisfied as explained next.

4.1. Similarity-aware Set Intersection Algorithm

The semantics of the similarity intersect operator is implementation-independent. Therefore, while it cannot impact the result, the order in which

relations are processed may impact performance. Thus, the query optimizer has to make ordering decision based on available statistics. This optimization is outside the scope of this paper. Our implementation uses left associativity to process the relations. Since the binary and multi-way similarity set intersection operators work in the same way, we develop one algorithm for both.

The result of a multi-way similarity intersect is constructed in stages, where each stage has a binary operator that produces an intermediate result that is then sent to the next stage. In the first stage (first level), the intermediate result is constructed so that each similar outer and inner tuples are consecutive, i.e., are next to each other in the order of emission. Similarly, results of the second stage are constructed such that the three similar tuples from the three input relations of the multi-way similarity intersect are produced in consecutive order similar to the order of the relations. In other words, the first tuple is from the first relation, the second tuple is from the second relation, and so on.

Algorithm 2 realizes the similarity-aware set intersection operator. Lines 1 and 2 initialize the outer and inner tuples. Both input relations are assumed to be sorted. Lines 4-11 advance the current inner and outer tuple(s) until a match based on the first attribute is found, i.e., when $dist(outer[0], inner[0]) \leq \epsilon_1$, where 0 refers to the index of the first attribute. Once a match is found, Line 12 marks the inner tuple position. Marking a tuple allows re-positioning the inner cursor to the marked tuple later in the process.

Algorithm 1 Advance Outer

1: **function** ADVANCEOUTER(*outer*,*level*)
2:     **while** $level \neq 0$ **do**
3:         advance outer
4:         $level \leftarrow level - 1$
5:     **end while**
6: **end function**

Algorithm 2 SimIntersect(*inner*, *outer*, *nodeLevel*)

Input: outer relation, inner relation and the level of the similarity set intersection.
Output: similarity set intersection result.

```
1: get initial outer tuple
2: get initial inner tuple
3: do forever {
4: while outer[0]! ~ inner[0] do
5:     if outer[0] < inner[0] then
6:         level ← nodeLevel
7:         advanceOuter(outer,level)
8:     else
9:         advance inner
10:    end if
11: end while
12: mark inner position
13: do forever {
14: do{
15:     count ← compare(outer,inner,nodeLevel)
16:     level ← nodeLevel
17:     if count = level then
18:         ReportMatchingTuples(inner,outer,level)
19:     end if
20:     prevInner ← inner
21:     advance inner
22:     }
23: while inner[0] ~ outer[0]
24: level ← nodeLevel
25: advanceOuter(outer,level)
26: if outer[0] ~ prevInner[0] then
27:     restore inner position to mark
28: end if
29: break
30: }
31: }
```

The above procedure is demonstrated in Figure 3 that illustrates the similarity intersection of the three tables P, Q, and R. *Level* 1 performs the similarity intersect between Q and P, and the result is intersected with R in *Level* 2. The threshold is usually determined by the application require-

ments. For this example, we select a threshold around 10% of the attribute range of values, i.e., list={0.5,5}. Initially, the outer points to Tuple (0.9,10) and the inner points to Tuple (0.1,5). Based on the value of the first attribute, the outer and the inner are advanced until the outer reaches (2,30) and the inner reaches (1.5,15). Then, the inner position is marked because both tuples match on their first attribute values. Lines 14-23 are executed to report only the matching tuples while advancing the inner because the first attribute's value is within the outer's corresponding value and assign to *prevInner* a copy of the current inner location before advancing the inner cursor. Notice that the matching tuples are reported consecutively, i.e., tuple(s) from the outer then tuples from the inner. The reason is that in the next level, the consecutive tuples will be reported if a tuple of the next relation is similar to these consecutive similar tuples. This loop finishes when the inner reaches (5,50) as $dist(2,5) > 0.5$. Then, the outer is advanced and is compared to the previous inner, and if both match on the first attribute, the inner cursor is restored to the marked position (as in Lines 25-28). In the running example, this happens when the outer is advanced to tuple (2.5,20) and is compared to the prevInner's tuple (2.3,25). The inner is restored to the marked tuple because $dist(2.5, 2.3) \leq 0.5$. The process repeats the search for other matching tuples.

The functions ADVANCEOUTER, COMPARE, and REPORTMATCHINGTUPLES, as presented in Algorithms 1, 3 and 4, respectively, work based on the level of the similarity intersection operator. In $Level1$, the outer is advanced once to perform any processing, while in $Level2$, the outer is advanced twice and so on for next levels. When comparing the inner tuple to the outer, if the process is in $Level1$, the inner is only compared to the current outer whereas if the process is in $Level2$ the inner is compared to the current and the next outer tuples (i.e., the consecutive similar tuples). Referring to our example, the inner tuple (2.1,33) is similar to the outer consecutive tuples (2,30) and (2.2,35) in $Level2$. Then, REPORTMATCHINGTUPLES reports them by first reporting the two consecutive outer tuples (2,30) and (2.2,35), because both are in $Level2$, this function will report the current two consecutive tuples, then it will report the current matching inner tuple, which is (2.1,33).

Algorithm 3 Compare Tuples

```
 1: function COMPARE(inner,outer,level)
 2:     mark outer position
 3:     count ← 0
 4:     while level ≠ 0 do
 5:         if outer ∼ inner then
 6:             count ← count + 1
 7:             level ← level − 1
 8:             advance outer
 9:         else
10:             break
11:         end if
12:     end while
13:     restore outer
14:     return count
15: end function
```

Algorithm 4 Report Matching Tuples

```
1: function ReportMatchingTuples(inner,outer,level)
2:     while level ≠ 0 do
3:         report outer
4:         advance outer
5:         level ← level − 1
6:     end while
7:     report inner
8:     restore outer
9: end function
```

4.2. Similarity-aware Set Difference Algorithm

Similar to the mechanism for the similarity-aware set intersection, Algorithm 5 applies a Mark/Restore method and uses the first attribute value as a filter that indicates if there is a possibility of tuples to match. The input is assumed to be two sorted relations, i.e., the outer and inner relations. Lines 1-2 initialize the outer and inner tuples. The comparison between the inner and outer tuples starts at Line 4. If initially, the first attribute of the outer tuple has a greater value than that of the inner's first attribute, then the inner is repeatedly advanced until this inner's first attribute has a

similar or greater value. If the inner has a greater value, then the outer is reported since the relation is sorted and definitely there are no similar next inner tuples as indicated in Lines 19-21.

The other case is presented in Lines 7-17, when the values of the outer first attribute and the inner first attribute are similar. In this case, the algorithm marks the inner because if this inner is advanced, there may exist a next outer tuple that matches the marked inner tuple or next skipped inner tuple. Therefore, no outer tuple is going to be mistakenly reported. After marking the inner position, the outer and current inner tuples are compared; if they do not match on all attributes then the inner is advanced. As the inner's first attribute value is similar to that of the outer's and as they do not fully match, this procedure will be repeated. As the inner cursor is advanced, it has to eventually reach one of two cases (1) having the outer matches the inner; the outer is advanced (outer is skipped since it has a similar tuple) and the algorithm breaks out of the loop; or (2) having the inner's first attribute with a greater value; the outer tuple is reported because no next inner tuple is going to match. When the loop finishes, Line 22 tests if the algorithm passes the previous loop (advancing inner and keeping a copy of the previous inner) by testing the previous inner value. If the algorithm passes this loop (previous inner is not NULL), then it executes Lines 23-30. First, it has to check the outer with the current inner as presented in Line 23, if they match, the outer is advanced. Otherwise, if the outer matches the previous inner, then the outer gets also advanced. If both cases are not true, the current outer's first attribute is compared to the previous inner's first attribute and if they are within the assigned threshold, the inner is restored (there is a possibility that a skipped inner matches the current outer). The process is repeated looking for other non-matching tuples.

4.3. Complexity Analysis

As mentioned in the previous section, the proposed algorithms assume sorted inputs, and are based on a Mark/Restore mechanism that may lead to having a nested loop in the worst case. The complexity is computed as follows:

- Sorting the input relations: Assume that the outer and inner relations have n tuples, then the complexity is $O(nlogn)$.
- Processing the similarity set operator: Assume that the n outer tuples each iterates on average over c tuples of the inner relation, then

Algorithm 5 SimDiff(*outer*,*inner*)

Input: outer and inner relations.
Output: similarity set difference result.

```
 1: get initial outer tuple
 2: get initial inner tuple
 3: do forever {
 4: while outer[0] > inner[0] do
 5:     advance inner
 6: end while
 7: if outer[0] ~ inner[0] then
 8:     mark inner position
 9:     while outer[0] ~ inner[0] do
10:         if outer ~ inner then
11:             advance outer
12:             break
13:         else
14:             prevInner ← inner
15:             advance inner
16:         end if
17:     end while
18: else
19:     report outer
20:     advance outer
21: end if
22: if prevInner !=NULL then
23:     if outer ~ inner || outer ~ prevInner then
24:         advance outer
25:     else if outer[0] ~ prevInner[0] then
26:         restore inner position
27:     end if
28:     prevInner ← NULL
29: end if
30: }
```

the complexity is $O(n * c)$. The best-case scenario happens if $c = 1$, the average case is achieved when c is small with respect to the number of the inner tuples, and the worst case occurs when $c = n$. The worst-case scenario typically happens when applying a large similarity threshold (e.g., a big fraction of the domain range). In our algorithms,

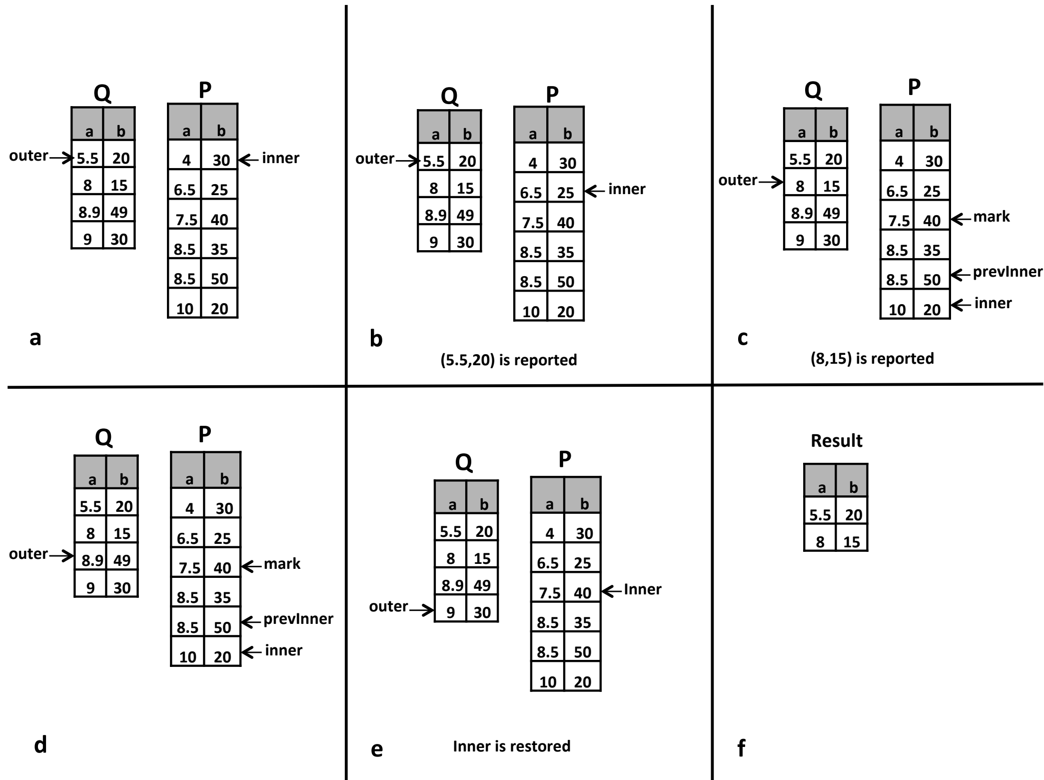


Figure 4: Example2: Sim-Diff. Threshold list={0.5,5}

the threshold assigned to the first attribute is the one influencing the performance the most.

- Filtering the output: Filtering is performed by sorting the output from the previous step, namely the similarity-aware set operation, and then grouping the duplicates. Assuming that there are k output tuples, then the complexity is $O(klogk + k)$ or equivalently $O(klogk)$.

Adding the time complexities of the three steps shows that the average case complexity is $O(nlogn)$, while the worst case complexity is $O(n^2)$. Typically, a threshold value is expected to be small compared to the domain size. Therefore, the complexity of our algorithms is closer to the average case. In our experiments, we evaluate the performance on average-case scenarios by using reasonably small threshold values.

5. Implementation and Experiments

5.1. Implementation in PostgreSQL

We have modified PostgresSQL to support the similarity set operators. The changes we made are in the Parser, Optimizer, and Executor modules of PostgreSQL so they can consume our proposed operators and their corresponding algorithms discussed earlier.

5.1.1. The Parser

We extend the grammar rules to include the defined similarity predicate. The parse-tree and query-tree data structures are extended to include the type and parameters, e.g., *setOpType*, *thresholdList*, and *level*. Accordingly, the routines that are responsible for transforming the parse tree into the query tree are updated to process the new fields in the parse tree.

5.1.2. The Optimizer

The plan for a tree of similarity set operator is given in Figure 5. The set operation plan node, *SetOp* node, is modified to have a parameter named *type* that indicates if the set operation node is a standard or similarity-based node, and to accept right and left sub-plans. As the planner finds that the query node is a similarity-aware set operator node, it determines its position by checking the operator type (*SetOpStartAndEnd*, *SetOpStart*, *SetOpIn* or *SetOpEnd*). Based on the type, the planner decides how the sub-nodes or upper-nodes should be planned. If the current processed query tree consists only of one set operator (type=*SetOpStartAndEnd*), the input relations have to be sorted. Therefore, the planner plans this part of the query by first adding a sort node on top of each sub-plan, which are actually sequential plans that scan the input relations.

Notice that the applied sort node attempts to perform sorting based on the first attribute of the input relation. These two sort plans are attached as the left and right child plans of the similarity set operation plan node. On the top of the similarity set operation plan node, a sort node and a Unique node are added to produce unique output. The Unique node assumes an input relation that is sorted on all its attributes, therefore, we apply this type of sort (Sort All).

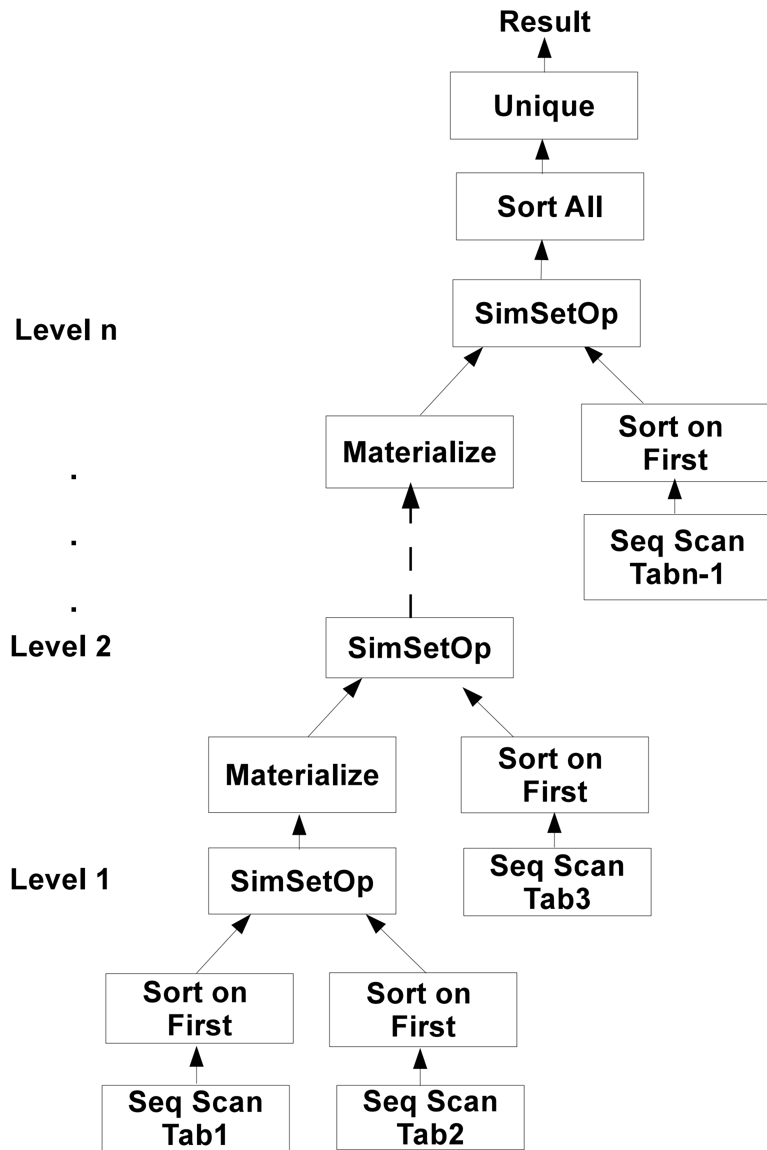


Figure 5: Similarity-aware set operator plan

In the case of a query tree with two set operations, the lower-level set operator node (that is assigned a *SetOpStart* type) has the same sub-plans as the *SetOpStartAndEnd*. However, since the first set operator sends its output to another set operator (in this case, *SetOpEnd* node), it is followed

by a materialize-plan node that stores the output to be read later by the next set operation node. Notice that we use a materialize node, not a sort node, because we need the output order to be preserved to be able to perform the multi-way similarity-aware intersection. For the node (*SetOpEnd*), the planner sorts only its right input (sort node on top of sequential scan node) and its left plan is already planned (the materialize node that is on top of the *SetOpStart* node). Then, the upper node is planned to be a sort node followed by a Unique node as in the *SetOpStartAndEnd* node. When having more than two set operators, the *SetOpIn* (internal) node exists in the tree. In this case, the planner sorts only the right input of this node, the same as the *SetOpEnd*, the output is materialized to be processed by the next set operator and finally the left node is planned in a previous stage to be a materialize node.

5.1.3. The Executor

In contrast to the standard set operator execution node that accepts only one tuple at a time, we extend the execution node structure *SetOpState* to hold references to the outer and inner tuples returned by the left and right nodes. Other extensions are performed to hold the pointers to previous inner, marked inner, marked outer, and other parameters. To implement the similarity-aware set operator in an efficient way, we build a finite state machine that expresses the states of a similarity set node and their transitions as given in Figure 6 for similarity set intersection.

5.2. Experiments

In this section, we evaluate the performance of the proposed similarity set operators and discuss the experimental results. We run the experiments on an Ubuntu Linux machine with a 2.4GHz Intel Core i5 CPU and 4GB memory. Experiments are performed on real data sets (Intel Berkeley Research lab, 2015), synthetic data, as well as using the TPC-H benchmark data (TPC, 2015). We first show the effect of varying the number of attributes using a real dataset. Then, we compare the proposed operator against (i) the standard set operators to show that the overhead introduced by the operator is acceptable, and (ii) the equivalent queries that use regular SQL operations to produce the same results as the corresponding similarity-aware query to show that our proposed algorithm yields much better performance. The equivalent queries are presented in Table 1

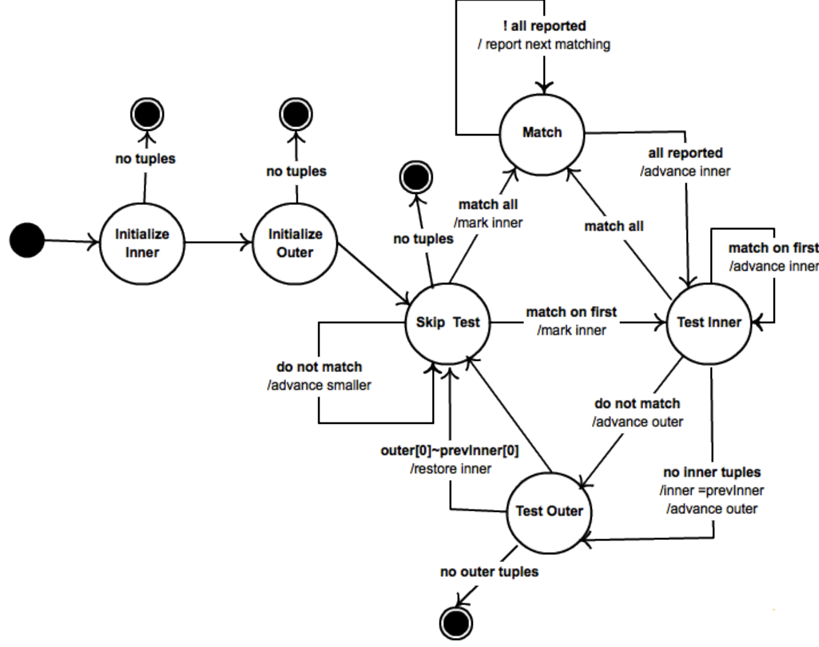


Figure 6: Finite state machine for similarity-aware set intersection.

Similarity-aware Set Operator	Equivalent Query using Regular Operations
(SELECT $a_1, \ldots, a_n$ FROM $tab1$ INTERSECT SELECT $a_1, \ldots, a_n$ FROM $tab2$) WITHIN VALUES $(\epsilon_1, \ldots, \epsilon_n)$;	SELECT $tab1.a_1, \ldots, tab1.a_n$ FROM $tab1, tab2$ WHERE abs($tab1.a_1 - tab2.a_2$)$\leq\epsilon_1$ ... and abs($tab1.a_n - tab2.a_n$)$\leq\epsilon_n$ UNION SELECT $tab2.a_1, \ldots, tab2.a_n$ FROM $tab1, tab2$ WHERE abs($tab1.a_1 - tab2.a_2$)$\leq\epsilon_1$ ... and abs($tab1.a_n - tab2.a_n$)$\leq\epsilon_n$
(SELECT $a_1, \ldots, a_n$ FROM $tab1$ EXCEPT SELECT $a_1, a_2, \ldots, a_n$ FROM $tab2$) WITHIN VALUES $(\epsilon_1, \ldots, \epsilon_n)$;	SELECT $ta_1, \ldots, tab1.a_n$ FROM $tab1$ EXCEPT SELECT $tab1.a_1, \ldots, tab1.a_n$ FROM $tab1, tab2$ WHERE abs($tab1.a_1 - tab2.a_2$)$\leq\epsilon_1$ ... and abs($tab1.a_n - tab2.a_n$)$\leq\epsilon_n$

Table 1: Equivalent regular operations.

5.2.1. Impact of the Number of Attributes

We use a public dataset (Intel Berkeley Research lab, 2015) that contains around 2.3 million readings gathered from 54 sensors deployed in the Intel Berkeley Research lab. The purpose of this experiment is to study the performance of the similarity set intersection and the similarity set difference as the

number of attributes increases. We conduct this experiment by processing the following query:

```
(SELECT epoch, temp, humidity, voltage FROM sensors WHERE moteid=1
INTERSECT/EXCEPT
SELECT epoch, temp, humidity, voltage FROM sensors WHERE moteid=2)
WITHIN VALUES (10,0.1,0.1,0.1);
```

This query returns the similar readings from mote1 and mote2 in case of similarity-aware intersection; and the readings from mote1 that do not have similar readings gathered by mote2 in the case of similarity-aware difference. We start by querying based on one attribute, namely epoch. Then, we repeat the experiment by adding one attribute at a time. From Figure 7, we observe that the execution time is the highest when intersecting two datasets consisting of multiple attributes on their first attribute only and the execution time decreases as we increase the input attributes of these datasets. The reasons for this behavior are as follows. Referring to the algorithm for the similarity-aware set intersection, the number of internal comparison loops is the same for one or more attributes because the algorithm is based on the first attribute value. What differs here is the number of the returned matching tuples. When intersecting on one attribute, it is more likely to have more matching output tuples than when intersecting on two or more attributes. As the number of the output matching tuples increases, the time spent by the sort and the duplicate elimination processes increases. The similarity-aware set difference performs better for one attribute due to the following reason. If the current outer has a similar value as the current inner (or previous inner), this outer is skipped. However, if the outer matches only the first attribute, the algorithm restores the inner position to ensure not having a previous matching tuple. Though, this procedure is not followed if there is only one attribute, which represents the entire tuple, because this tuple either matches or not, so no restoring is performed in this case. Whereas this procedure is increasingly followed when the tuple consists of a larger number of attributes.

5.2.2. Similarity with Standard Operators

In this experiment, we study the performance of the proposed similarity operators against equivalent queries that perform the same functionality and that produce the same output but that use only standard SQL operators. We vary the data size and the similarity threshold value and use the TPC-H data set (TPC, 2015). We run the queries presented in Tables 2 and 3. Through

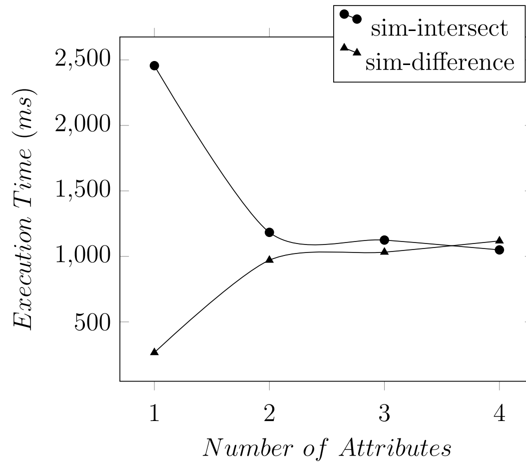
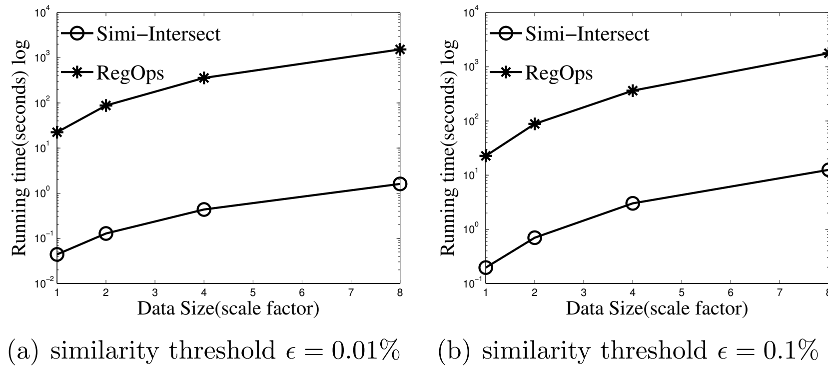
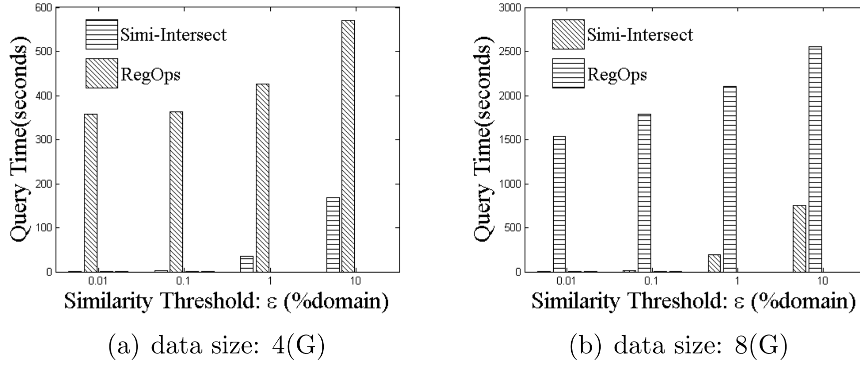


Figure 7: Performance of similarity-aware set operators while increasing the number of attributes.



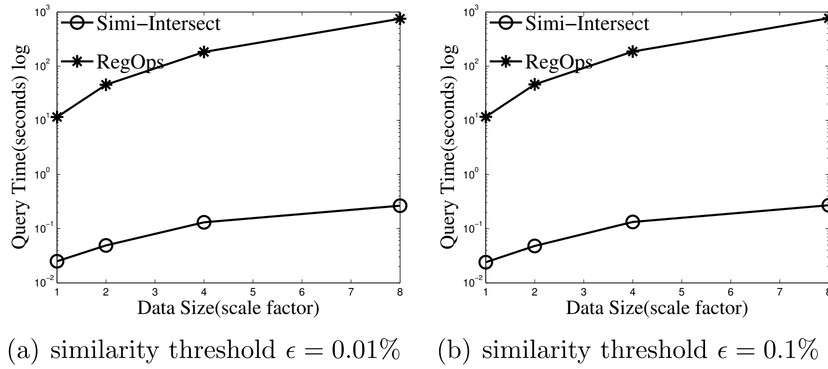
(a) similarity threshold $\epsilon = 0.01\%$ (b) similarity threshold $\epsilon = 0.1\%$

Figure 8: Sim-Intersect: Effect of Data Size.



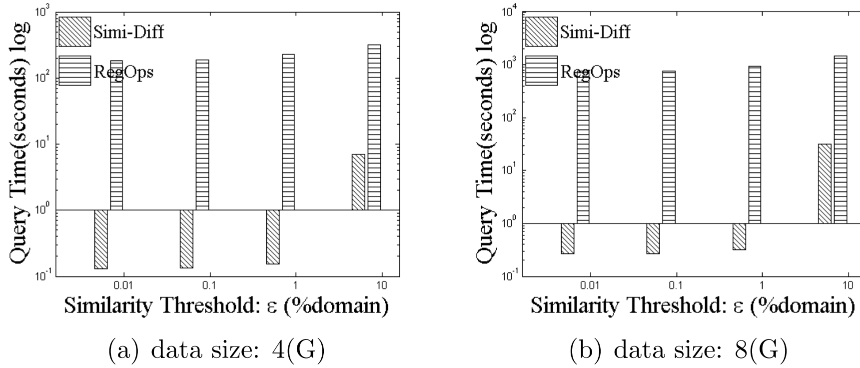
(a) data size: 4(G) (b) data size: 8(G)

Figure 9: Sim-Intersect: Effect of Similarity Threshold: ϵ.



(a) similarity threshold $\epsilon = 0.01\%$ (b) similarity threshold $\epsilon = 0.1\%$

Figure 10: Sim-Difference: Effect of Data Size.



(a) data size: 4(G) (b) data size: 8(G)

Figure 11: Sim-Difference: Effect of Similarity Threshold: ϵ.

these queries, we can identify similar/different customer profiles from two countries. We may consider customer profiles to be defined by the amount of money spent. For this case, we can run queries that use one attribute (total price). However, some customers may spend a huge amount of money on a small quantity of items or may spend a small amount of money on a large quantity of items. Therefore, we run a more precise query that uses two attributes (total price and total quantity) to represent the customer profile. Notice that the assigned threshold to custkey attribute is -1. This value is used to express the infinity value because we want to count the customers with similar profiles regardless of whether their customer keys match or not.

We study the similarity set intersection and similarity set difference performance by varying the threshold value from 0.01% of the attribute domain range to a reasonable value, which is 10% of the domain range. We vary the threshold of the first attribute only because the algorithm is influenced highly by its value. The threshold assigned to the second attribute is fixed to be 0.1% of the attribute domain range. Specifically, the customer total price domain and total quantity domain use values in the ranges [11020, 6289000] and [10, 4000], respectively. We vary the input size by repeating the experiment using different TPCH scale factors (from SF=1 to SF=8).

The results are given in Figures 8, 9, 10 and 11. They demonstrate a substantial query processing speedup of the similarity set intersection and similarity set difference queries over the equivalent queries that only employ regular operators. The speedup ranges between 1000 and 4 times for the similarity set intersection and ranges between 3000 to 47 for the similarity set difference for similarity threshold values ranging between 0.01% and 10% of the attribute domain range.

5.2.3. Comparison with Standard Queries

This section evaluates the performance of the similarity set intersection and similarity set difference operators as compared to the standard SQL set operators. We compare queries that have similar selectivities (i.e., queries that produce similar output size for a given input size). We control the output size by careful generation of synthetic input data. For each operator, at a specified output size, we generate two tables of 1,000,000 tuples with uniformly distributed values. We initially fix the output size to 16k and repeat the experiment by doubling the output size until we reach around 10% of the table size. We start with tables of one attribute and repeat the experiments with two attributes. The generated tuples in the two tables are

Operator Type	Syntax
Similarity-aware SetOp, one attribute	(SELECT c_acctbal FROM customer WHERE c_nation=1 INTERSECT/EXCEPT SELECT c_acctbal FROM customer WHERE c_nation=2) WITHIN VALUES (ϵ)
Regular Operations, one attribute for sim-intersect	SELECT c1.c_acctbal FROM (SELECT c_acctbal FROM customer WHERE c_nation=1) c1, (SELECT c_acctbal FROM customer WHERE c_nation=2) c2 WHERE abs(c1.c_acctbal-c2.c_acctbal)$\leq \epsilon$ UNION SELECT c2.c_acctbal FROM (SELECT c_acctbal FROM customer WHERE c_nation=1) c1, (SELECT c_acctbal FROM customer WHERE c_nation=2) c2 WHERE abs(c1.c_acctbal-c2.c_acctbal)$\leq \epsilon$
Regular Operations one attribute for sim-diff	SELECT c_acctbal FROM customer WHERE c_nation=1 EXCEPT SELECT c1.c_acctbal FROM (SELECT c_acctbal FROM customer WHERE c_nation=1) c1, (SELECT c_acctbal FROM customer WHERE c_nation=2) c2 WHERE abs(c1.c_acctbal-c2.c_acctbal)$\leq \epsilon$
Similarity-aware SetOp, two attributes	SELECT count(*) FROM ((SELECT p1.priceSum, p1.qtySum, p1.custkey FROM (SELECT sum(o.o_totalprice) as priceSum, sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c, (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=1 GROUP BY o.o_custkey) p1 INTERSECT/EXCEPT SELECT p2.priceSum,p2.qtySum,p2.custkey FROM (SELECT sum(o.o_totalprice) as priceSum, sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c, (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=2 GROUP BY o.o_custkey) p2) WITHIN VALUES (ϵ_1,ϵ_2,-1)) as result;

Table 2: Queries to evaluate on TPC-H data.

either:

- Matching: To be reported in case of similarity intersection or excluded in case of similarity difference.
- Non-Matching: To be reported in case of similarity difference and excluded in case of similarity intersection.
- Overlapping: For these tuples, there exist tuples in the other relation such that they either match on first attribute or second attribute. These

Operator Type	**Syntax**
Equivalent Regular Operations to sim-intersect	SELECT count(*) FROM (SELECT p1.priceSum, p1.qtySum, p1.custkey FROM (SELECT sum(o.o_totalprice) as priceSum, sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c, (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=1 GROUP BY o.o_custkey) p1, (SELECT sum(o.o_totalprice) as priceSum, sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c, (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=2 GROUP BY o.o_custkey) p2 WHERE abs(p1.priceSum-p2.priceSum)$\leq \epsilon_1$ AND abs(p1.qtySum-p2.qtySum)$\leq \epsilon_2$ UNION SELECT p2.priceSum, p2.qtySum, p2.custkey FROM (SELECT sum(o.o_totalprice) as priceSum, sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c, (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=1 GROUP BY o.o_custkey) p1, (SELECT sum(o.o_totalprice) as priceSum, sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c, (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=2 GROUP BY o.o_custkey) p2 WHERE abs(p1.priceSum-p2.priceSum)$\leq \epsilon_1$ AND abs(p1.qtySum-p2.qtySum)$\leq \epsilon_2$) as result;
Equivalent Regular Operations to sim-diff	SELECT count(*) FROM (SELECT p1.priceSum, p1.qtySum ,p1.custkey FROM (SELECT sum(o.o_totalprice) as priceSum ,sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c , (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=1 GROUP BY o.o_custkey) p1 EXCEPT SELECT p1.priceSum, p1.qtySum ,p1.custkey FROM (SELECT sum(o.o_totalprice) as priceSum ,sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c , (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=1 GROUP BY o.o_custkey) p1, (SELECT sum(o.o_totalprice) as priceSum ,sum(q.qty) as qtySum, o.o_custkey as custkey FROM orders o, customer c , (SELECT l_orderkey as o_key, sum(l_quantity) as qty FROM lineitem GROUP BY l_orderkey) q where o.o_orderkey=q.o_key and c.c_custkey=o.o_custkey and c.c_nationkey=2 GROUP BY o.o_custkey) p2 WHERE abs(p1.priceSum-p2.priceSum)$\leq \epsilon_1$ AND abs(p1.qtySum-p2.qtySum)$\leq \epsilon_2$) as result;

Table 3: Queries to evaluate on TPC-H data (cont'd).

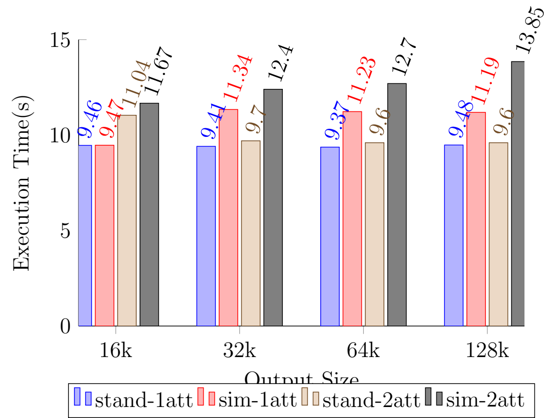
tuples are excluded in case of similarity intersection and reported in case of similarity difference. The purpose from generating these tuples is to include the case of having nested loops.

Results are illustrated in Figure 12. The similarity set intersection adds a 20% overhead in the one-attribute tables experiment. The overhead varies from 20% to 44% when increasing the output size from 16k to 128k in the two-attribute experiment. However, the similarity set difference may execute faster than the standard set difference, and this behavior can be explained as follows:

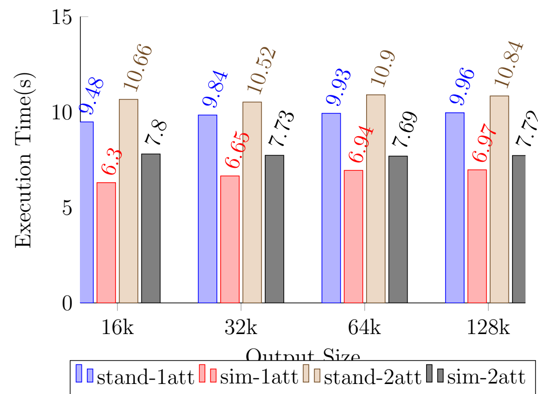
- Sorting two lists each of size N/2 is faster than sorting a list of size N. Here, the standard set difference performs the sorting after combining the two tables while the similarity set difference performs the sort on each table separately.
- The similarity difference has a lower probability of performing multiple inner loops compared to the similarity set intersection. The similarity set intersection restores the pointer as first attributes match regardless of whether the tuples fully match or not. Whereas the similarity set difference compares the current tuple to the current inner or the previous inner and if one matches it skips the current outer tuple.

6. Conclusion

In this paper, we introduced the semantics and extended SQL syntax of the similarity-based set operators. We developed algorithms that are based on the Mark/Restore mechanism. We implemented these algorithms inside PostgreSQL and evaluated their performance. Our implementation outperforms the queries that produce the same result using only regular operations. The speedup ranges between 1000 and 4 times for similarity threshold values ranging between 0.01% and 10% of the attribute domain range. Experimental results have also demonstrated that the added functionality introduced by the proposed similarity operators is achieved without a big overhead when compared to standard operators.



(a) standarad intersect vs similarity intersect



(b) standard set difference vs. similarity difference

Figure 12: Similarity-aware set operators vs. standard set operators.

Acknowledgements

This publication was made possible by the support of an NPRP grant from the QNRF and the National Science Foundation under Grants III-1117766 and III-0964639. The statements made herein are solely the responsibility of the authors.

References

Arasu, A., Ganti, V., and Kaushik, R. (2006). Efficient exact set-similarity joins. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 918–929. VLDB Endowment.

Barioni, M. C. N., Razente, H., Traina, A., and Traina Jr, C. (2006). Siren: A similarity retrieval engine for complex data. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1155–1158. VLDB Endowment.

Barioni, M. C. N., Razente, H. L., Traina Jr, C., and Traina, A. J. (2005). Querying complex objects by similarity in sql. In *SBBD*, volume 5, pages 130–144. Citeseer.

Böhm, C., Braunmüller, B., Krebs, F., and Kriegel, H.-P. (2001). 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, pages 379–388. ACM.

Böhm, C. and Krebs, F. (2004). The k-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6(6):728–749.

Chaudhuri, S., Ganti, V., and Kaushik, R. (2006). A primitive operator for similarity joins in data cleaning. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 5–5. IEEE.

Ciaccia, P., Patella, M., and Zezula, P. (1997). M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Clarkson, K. L. (1997). Nearest neighbor queries in metric spaces. pages 609–617.

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

Gao, L., Wang, M., Wang, X. S., and Padmanabhan, S. (2004). Expressing and optimizing similarity-based queries in sql. In *Conceptual Modeling–ER 2004*, pages 464–478. Springer.

Hjaltason, G. R. and Samet, H. (1998). Incremental distance join algorithms for spatial databases. volume 27, pages 237–248, New York, NY, USA. ACM.

Intel Berkeley Research lab (2015). Intel Lab Data. http://db.csail.mit.edu/labdata/labdata.html.

Jacox, E. H. and Samet, H. (2008). Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2):7:1–7:38.

Lian, X. and Chen, L. (2008). Similarity search in arbitrary subspaces under lp-norm. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 317–326.

Lu, W., Shen, Y., Chen, S., and Ooi, B. C. (2012). Efficient processing of k nearest neighbor joins using mapreduce. *Proceedings of the VLDB Endowment*, 5(10):1016–1027.

Marri, W. J. A., Malluhi, Q. M., Ouzzani, M., Tang, M., and Aref, W. G. (2014). The similarity-aware relational intersect database operator. In *7th International Conference Similarity Search and Applications, SISAP*, pages 164–175.

Mills, P. (2011). Efficient statistical classification of satellite measurements. *International Journal of Remote Sensing*, 32(21):6109–6132.

Narayanan, M. and Karp, R. (2004). Gapped local similarity search with provable guarantees. In Jonassen, I. and Kim, J., editors, *Algorithms in Bioinformatics*, volume 3240 of *Lecture Notes in Computer Science*, pages 74–86. Springer Berlin Heidelberg.

Pola, I. R., Cordeiro, R. L., Traina Jr, C., and Traina, A. J. (2013). A new concept of sets to handle similarity in databases: The simsets. In *Similarity Search and Applications*, pages 30–42. Springer.

PostgreSQL (2015). PostgreSQL DBMS. http://www.postgresql.org.

Schallehn, E. and Sattler, K.-U. (2003). Using similarity-based operations for resolving data-level conflicts. In *Proceedings of the 20th British National Conference on Databases*, BNCOD'03, pages 172–189. Springer-Verlag, Berlin, Heidelberg.

Schallehn, E., Sattler, K.-U., and Saake, G. (2001). Advanced grouping and aggregation for data integration. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*, CIKM '01, pages 547–549, New York, NY, USA. ACM.

Schallehn, E., Sattler, K.-U., and Saake, G. (2002). Extensible and similarity-based grouping for data integration. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 277–277.

Schallehn, E., Sattler, K.-U., and Saake, G. (2004). Efficient similarity-based operations for data integration. *Data Knowl. Eng.*, 48(3):361–387.

Seidl, T. and Kriegel, H.-P. (1998). Optimal multi-step k-nearest neighbor search. volume 27, pages 154–165, New York, NY, USA. ACM.

Silva, Y., Aref, W., Larson, P., Pearson, S. S., and Ali, M. (2013). Similarity queries: their conceptual evaluation, transformations, and processing. *The VLDB Journal*, 3(22).

Silva, Y. N., Aly, A. M., Aref, W. G., and Larson, P.-A. (2010a). Simdb: a similarity-aware database system. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 1243–1246. ACM.

Silva, Y. N., Aref, W. G., and Ali, M. H. (2009). Similarity Group-By. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE.

Silva, Y. N., Aref, W. G., and Ali, M. H. (2010b). The similarity join database operator. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE.

Silva, Y. N., Aref, W. G., Larson, P.-A., and Ali, M. H. (2012). Similarity-aware query processing and optimization.

Tang, M., Tahboub, R. Y., Aref, W. G., Atallah, M. J., Malluhi, Q. M., Ouzzani, M., and Silva, Y. N. (2014). Similarity group-by operators for multi-dimensional relational data. *CoRR*, abs/1412.4842.

Tang, M., Yu, Y., Aref, W. G., Malluhi, Q. M., and Ouzzani, M. (2015). Efficient processing of hamming-distance-based similarity-search queries over mapreduce. In *Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015.*, pages 361–372.

TPC (2015). TPC-H Version 2.15.0. http://www.tpc.org/tpch.

Traina Jr, C., Traina, A., Faloutsos, C., and Seeger, B. (2002). Fast indexing and visualization of metric data sets using slim-trees. *Knowledge and Data Engineering, IEEE Transactions on*, 14(2):244–260.

Wang, J., Feng, J., and Li, G. (2010). Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *Proceedings of the VLDB Endowment*, 3(1-2):1219–1230.

Wang, J., Li, G., and Fe, J. (2011). Fast-join: An efficient method for fuzzy token matching based string similarity join. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 458–469.

Xiao, C., Wang, W., and Lin, X. (2008). Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *Proceedings of the VLDB Endowment*, 1(1):933–944.

Yu, C., Cui, B., Wang, S., and Su, J. (2007). Efficient index-based KNN join processing for high-dimensional data. *Information & Software Technology*, 49(4):332–344.