The Similarity-Aware Relational Intersect Database Operator

Wadha J. Al Marri[1], Qutaibah Malluhi[1], Mourad Ouzzani[2], Mingjie Tang[3], and Walid G. Aref[3]

[1] Qatar University, Doha, Qatar
200450064@student.qu.edu.qa, qmalluhi@qu.edu.qa
[2] Qatar Computing Research Institute, Doha, Qatar
mouzzani@qf.org.qa
[3] Purdue University, West Lafayette, IN, USA
tang49@purdue.edu, aref@cs.purdue.edu

Abstract. Identifying similarities in large datasets is an essential operation in many applications such as bioinformatics, pattern recognition, and data integration. To make the underlying database system similarity-aware, the core relational operators have to be extended. Several similarity-aware relational operators have been proposed that introduce similarity processing at the database engine level, e.g., similarity joins and similarity group-by. This paper extends the semantics of the set intersection operator to operate over similar values. The paper describes the semantics of the similarity-based set intersection operator, and develops an efficient query processing algorithm for evaluating it. The proposed operator is implemented inside an open-source database system, namely PostgreSQL. Several queries from the TPC-H benchmark are extended to include similarity-based set intersetion predicates. Performance results 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 [1], data compression [2], data integration [3], and statistical classification [4] mandate that their underlying database systems provide similarity-aware capabilities as a means for identifying similar objects. Several similarity-aware relational operators have been proposed that introduce similarity processing at the database engine level, e.g., similarity joins and similarity group-by's [5], [6], [7]. In this paper, we introduce similarity-aware set intersection as an extended relational database operator.

In standard SQL, relational set operations are based on exact matching. However, assume that we want to find common readings that are produced by two sensors. Assume further that the sensor readings are stored in two separate tables. The standard SQL set intersect operator is not suitable for intersecting these two tables to get the common sensor readings; sensor readings may be similar but not necessarily identical. Thus, it is desirable to perform similarity

A.J. Machado Traina et al. (Eds.): SISAP 2014, LNCS 8821, pp. 164–175, 2014.
DOI: 10.1007/978-3-319-11988-5_15 © Springer International Publishing Switzerland 2014

set intersection to find similar readings in the two tables. While the focus of this paper is on the similarity set intersection operator, we study the other similarity set operators, namely similarity-based set union and similarity-based set difference in [8]. We omit their description for space limitation.

Several relational database operators have introduced similarity into SQL. The similarity group-by operator assigns every object to a group based on a similarity condition, e.g., as in [9,10,3]. Similarity join retrieves pairs of objects that overlap based on a join attribute using a predefined threshold. Several types of similarity join have been proposed, e.g., [11,12,5,13,14]. While the similariy join reports the joining objects, similarity intersection requires union-compatible input relations and returns all similar objects from both relations. An extension to SQL to support nearest-neighbor queries has been studied extensively, e.g., see [15]. k-Nearest-neighbor can be viewed as one form of similarity as each point or tuple is connected with its k-closest (or most similar) values. SIREN [16,17] allows expressing similarity queries in SQL and executing them via a similarity retrieval engine. SIREN is a middle-tier implemented between an RDBMS and application programs that processes and answers similarity-based SQL queries issued by the application. In [18], extensions to SQL make similarity operators first-class database operators by implementing the operators inside the database engine. None of the previous work addresses similarity-based set interesection, which is the focus of our paper.

The contributions of this paper are as follows. (1) We introduce the Similarity-aware Set Intersection Operator that extends the standard SQL set intersection to produce results based on similarity rather than on equality (Section 2). (2) We develop an efficient algorithm for the proposed operator (Section 3) and implement it inside PostgreSQL, an open-source relational database management system [19]. (3) We evaluate the performance of the proposed algorithm and its scalability properties using the TPC-H benchmark [20]. We extend several queries from the TPC-H benchmark to include similarity-based set intersetion predicates. Performance results demonstrate up to three orders of magnitude enhancement in performance over equivalent queries that only employ regular relational operators (Section 4).

2 Semantics of Similarity-Based Relational Intersect

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 compatible. 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 $dis_t(A_i.a_t, B_j.b_t)$ be the distance corresponding to Attribute a_t between the tuple pair (A_i, B_j) using the distance function dis_t.

We adopt the following similarity predicate: Given r thresholds $\epsilon_1, \epsilon_2, ..., \epsilon_r$ that are assigned to each of the attributes $a_1, a_2, \ldots, a_r$, respectively, where $r \leq$

k, we say two tuples A_i and B_j match iff: $pred(A_i, B_j) = dis_1(A_i.a_1, B_j.b_1) \leq \epsilon_1$ AND $dis_2(A_i.a_2, B_j.b_2) \leq \epsilon_2 \ldots$ AND $dis_r(A_i.a_r, B_j.b_r) \leq \epsilon_r$. 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.

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 identical (or compatible) schemas, and a smilarity predicate $pred(A, B)$, the similarity-aware set intersection operation is defined as follows.

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

Example: Consider the following two tables Q and P; each having a single compatible attribute, where attribute values x and $\widetilde{x}$ are assumed to be similar. $Q = \{a, b, c, d, e, f, g, z\}$ 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, $P \,\widetilde{\cap}\, Q = \{a, b, c, \widetilde{a}, \widetilde{b}, \widetilde{c}, z\}$.

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.

We extend SQL with the similarity-aware set intersect operator in the following way.

(**SELECT** $a_1, a_2, ...$ **FROM** $table_1$
INTERSECT
SELECT $a_1, a_2, ...$ **FROM** $table_2$
INTERSECT
...
SELECT $a_1, a_2, ...$ **FROM** $table_n$
) **WITHIN VALUES** (ϵ_1,ϵ_2,...)

where the phrase **WITHIN VALUES** provides the similarity thresholds for each of the attributes participating in the similarity intersection operation. Notice that the similarity intersect operator can be expressed using standard relational operators as the query evaluation tree in Fig. 1 demonstrates.

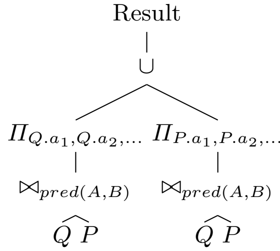


Fig. 1. Expressing Similarity Set Intersection Using Relational Operators

3 Processing the Similarity-Aware Intersect Operator

In this section, we present how the proposed similarity-aware set intersect operator is evaluated. The query processing algorithm for similarity intersect is an extension of the sort-merge join algorithm. So, the first step of the algorithm sorts both input tables unless they are already sorted. In high-level terms, similarity intersect 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 tuples between two relations (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.

The semantics of the similarity intersect operator is implementation independent. Therefore, the order of processing these relations will not impact the result. However, the order can impact the performance and therefore it should be part of query optimization. The current implementation simply uses left associativity to processes 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 sent to the next stage. In the first stage (first level), the intermediate result is constructed in such a way 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 (i.e., the first tuple is from the first relation, the second tuple is from the second relation, and so on).

Algorithm 1 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 repositioning the inner cursor to the marked tuple later in the process.

Algorithm 1 . 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: }
```

This procedure is demonstrated in Figure 2 for the similarity intersection of 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 requirements. For this example, the threshold is selected to be 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 the first attribute. 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 compared to the previous inner, and if both match on the first attribute, the inner cursor is restored to the marked position (Lines 25-28). In the 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 $dist(2.5, 2.3) \leq 0.5$. Then, the process repeats the search for other matching tuples.

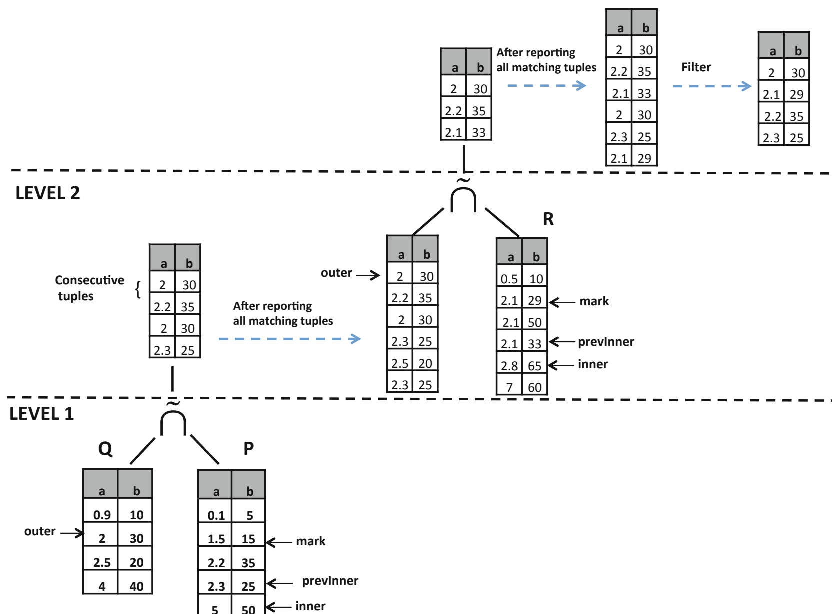


Fig. 2. Sample execution: Sim-Intersect. Threshold list={0.5,5}.

ADVANCEOUTER, COMPARE, and REPORTMATCHINGTUPLES (Algorithms 2, 3, and 4, resp.), work based on the level of the similarity intersection operator. In $Level1$, the outer is advanced once to perform any process, while in $Level2$, the outer is advanced twice, and so on. The reason is that similar tuples of the outer are consecutive to each other in the pipeline. 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 $Level2$. REPORTMATCHINGTUPLES produces the output by first reporting the two similar consecutive outer tuples (2,30) and (2.2,35), since they are in $Level2$, then it reports the current matching inner tuple, i.e., (2.1,33). Then, these three similar tuples are pipelined into $Level3$ for further processing, if any.

Algorithm 2 Advance Outer

```
1: function advanceOuter(outer,level)
2:     while level ≠ 0 do
3:         advance outer
4:         level ← level − 1
5:     end while
6: end function
```

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
```

3.1 Analysis

As mentioned in the previous section, the proposed algorithm assumes sorted inputs, and is 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 intersect operator: Assume that the n outer tuples each iterates on average over c tuples of the inner relation, then 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 may take place when having a large similarity threshold, e.g., a big fraction of the domain range. In our algorithm, the threshold assigned to the first attribute is the one influencing the performance the most.
– Filtering the output: Filtering is usually performed by sorting the input, then grouping the duplicates. Assume that there are k output tuples, then the complexity is $O(klogk + k)$.

Thus, the average case complexity is $O(nlogn)$ while the worst case complexity is $O(n^2)$, which is similar to sort-merge join. Typically, a threshold value is expected to be small compared to the domain size. Therefore, the complexity of the similarity intersect algorithm is closer to the average case. Thus, the performance is comparable to that of the standard set intersect, as demonstrated in the experimental section.

4 Experimental Results

We have modified PostgresSQL to support similarity intersect as an operator. We extended the Parser, Optimizer, and Executor modules of PostgreSQL for this purpose. We skip the details of how each of the PostgreSQL components is extended to support similarity intersect. The reader is referred to [8] for more details. Below, we present a summary of the performance results under various real and synthetic data sets as well as using some extensions to the TPC-H benchmark to support similarity queries.

Table 1. Equivalent regular operations

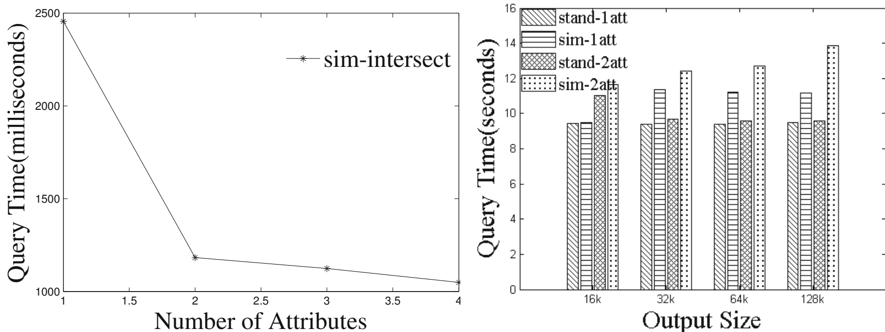
Similarity-aware Set Op.	Equivalent Query using Regular Ops.
(SELECT $a_1, a_2, \ldots, a_n$ FROM $tab1$ INTERSECT SELECT $a_1, a_2, \ldots, a_n$ FROM $tab2$) WITHIN VALUES $(\epsilon_1, \epsilon_2, \ldots, \epsilon_n)$;	SELECT $tab1.a_1, tab1.a_2, \ldots, tab1.a_n$ FROM $tab1, tab2$ WHERE abs($tab1.a_1$ − $tab2.a_2$)$\leq\epsilon_1$ and abs($tab1.a_2$ − $tab2.a_2$)$\leq\epsilon_2$ ... and abs($tab1.a_n$ − $tab2.a_n$)$\leq\epsilon_n$ UNION SELECT $tab2.a_1, tab2.a_2, \ldots, tab2.a_n$ FROM $tab1, tab2$ WHERE abs($tab1.a_1$ − $tab2.a_2$)$\leq\epsilon_1$ and abs($tab1.a_2$ − $tab2.a_2$)$\leq\epsilon_2$... and abs($tab1.a_n - tab2.a_n$)$\leq\epsilon_n$

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 [21], synthetic data, as well as using the TPC-H benchmark data [20]. We use the edit distance in our computations. We first study the effect of varying the number of attributes involved in the similarity intersect operator. Then, we compare the performance of the similarity intersect operator against (i) the standard relational intersect to demonstrate that the overhead of similarity intersect is acceptable, and (ii) the equivalent queries that use regular SQL operations to produce the same results as the corresponding similarity-aware query to demonstrate that similarity intersect yields better performance. The equivalent queries are presented in Table 1.

Impact of the Number of Attributes. We use a public dataset [21] 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 similarity intersect as the number of involved attributes is increased. We conduct this experiment by processing the following query:

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

This query returns similar readings from mote1 and mote2. We start by querying based on one attribute, namely epoch. Then, we repeat the experiment by adding each time one more attribute. Figure 3(a) illustrates 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 increase.



(a) Performance while increasing the number of attributes.

(b) Similarity-aware set intersection vs. standard set intersection.

Fig. 3. Effect of the number of attributes and the output size

Similarity Intersect Using Standard Relational Operators. We study the performance of the proposed similarity intersect operator against an equivalent query that performs the same functionality and that produces the same output but using only standard SQL operators. We vary the data size and the similarity threshold value while using the TPC-H data set [20]. We run the queries presented in Table 2. Through these queries, we can identify similar customer

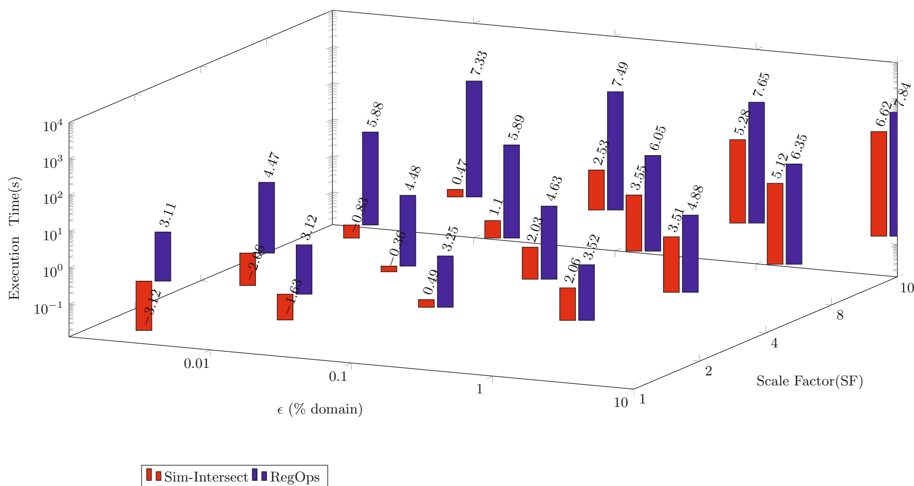


Fig. 4. Similarity-aware set intersection vs. regular operations

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 large 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 performance of similarity intersect when varying the similarity threshold value from 0.01% to 10% of the attribute 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 range [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 Figure 4 that demonstrate a substantial query processing speedup of the similarity set intersection query over the equivalent query that only employs regular operators. The speedup ranges between 1000 and 4 times for similarity threshold values ranging between 0.01% and 10% of the attribute domain range, respectively.

Comparison with Standard Queries. This section evaluates the performance of similarity intersct operator when compared to the standard SQL set intersection operator. We compare queries that have similar selectivities (i.e., queries that produce a similar output size for a given input size). We control the output cardinality by careful generation of synthetic input data. The details of how the data is generated are omitted due to space limitation. The reader is referred

Table 2. Similarity-based intersect queries using TPC-H data

Operator Type	Syntax
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;
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)$\leq$ ϵ_1 AND abs(p1.qtySum-p2.qtySum)$\leq$ ϵ_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)$\leq$ ϵ_1 AND abs(p1.qtySum-p2.qtySum)$\leq$ ϵ_2) as result;

to [8] for further detail. From Figure 3(b), the similarity intersect operator adds a 20% overhead in the case of one-attribute-based similarity while it varies from 20% to 44% when increasing the output size from 16k to 128k in the case of the two-attribute-based similarity.

5 Conclusion

We introduced the semantics and extended SQL syntax of the similarity-based set intersection operator. We developed an algorithm that is based on the Mark/Restore mechanism to avoid the $O(n^2)$ complexity. We implemented this algorithm inside PostgreSQL and evaluated its performance. Our implementation of the proposed operator 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. We also demonstrated that the added functionality is achieved without a big overhead when compared to standard operators.

Acknowledgments. This work was supported by an NPRP grant 4-1534-1-247 from the Qatar National Research Fund and by the National Science Foundation Grants IIS 0916614, IIS 1117766, and IIS 0964639.

References

1. Narayanan, M., Karp, R.M.: Gapped local similarity search with provable guarantees. In: Jonassen, I., Kim, J. (eds.) WABI 2004. LNCS (LNBI), vol. 3240, pp. 74–86. Springer, Heidelberg (2004)
2. Wang, J., Li, G., Feng, J.: Fast-join: An efficient method for fuzzy token matching based string similarity join. In: ICDE (2011)
3. Schallehn, E., Sattler, K.U., Saake, G.: Efficient similarity-based operations for data integration. Data and Knowledge Engineering 48(3) (2004)
4. Mills, P.: Efficient statistical classification of satellite measurements. International Journal of Remote Sensing 32(21) (2011)
5. Silva, Y.N., Aref, W.G., Ali, M.H.: The similarity join database operator. In: ICDE (2010)
6. Silva, Y.N., Aref, W.G., Ali, M.H.: Similarity group-by. In: ICDE (2009)
7. Silva, Y.N., Aref, W.G., Larson, P., Pearson, S., Ali, M.H.: Similarity queries: their conceptual evaluation, transformations, and processing. VLDB J. 22(3) (2013)
8. Marri, W.J.A.: Similarity-aware set operators. Master's thesis, Qatar University (2009)
9. Wang, J., Li, G., Fe, J.: Fast-join: An efficient method for fuzzy token matching based string similarity join. In: ICDE (2011)
10. Schallehn, E., Sattler, K., Saake, G.: Advanced grouping and aggregation for data integration. In: CIKM (2001)
11. Yu, C., Cui, B., Wang, S., Su, J.: Efficient index-based knn join processing for high-dimensional data. Journal of Information and Software Technology 49(4) (2007)
12. Hjaltason, G., Samet, H.: Incremental distance join algorithms for spatial databases. In: SIGMOD (1998)
13. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB (2006)
14. Böhm, C., Krebs, F.: The k-nearest neighbour join: Turbo charging the kdd process. Knowledge and Information Systems 6(6) (2004)
15. Gao, L., Wang, M., Wang, X.S., Padmanabhan, S.: Expressing and optimizing similarity-based queries in sql. In: Atzeni, P., Chu, W., Lu, H., Zhou, S., Ling, T.-W. (eds.) ER 2004. LNCS, vol. 3288, pp. 464–478. Springer, Heidelberg (2004)
16. Barioni, M.C.N., Razente, H.L., Traina Jr., C., Traina, A.J.M.: Querying complex objects by similarity in sql. In: SBBD (2005)
17. Barioni, M.C.N., Razente, H.L., Traina, A.J.M., Traina Jr., C.: Siren: A similarity retrieval engine for complex data. In: VLDB (2006)
18. Silva, Y.N., Aly, A.M., Aref, W.G., Larson, P.Å.: Simdb: a similarity-aware database system. In: SIGMOD (2010)
19. PostgreSQL Global Development Group: Postgresql (2014), http://www.postgresql.org/
20. TPCH: Tpc-h version 2.15.0 (2014), http://www.tpc.org/tpch
21. Intel Berkeley Research lab: Intel lab data (2014), http://db.csail.mit.edu/labdata/labdata.html