Abstract—Similarity joins have been studied as key operations in multiple application domains, e.g., record linkage, data cleaning, multimedia and video applications, and phenomena detection on sensor networks. Multiple similarity join algorithms and implementation techniques have been proposed. They range from out-of-database approaches for only in-memory and external memory data to techniques that make use of standard database operators to answer similarity joins. Unfortunately, there has not been much study on the role and implementation of similarity joins as database physical operators. In this paper, we focus on the study of similarity joins as first-class database operators. We present the definition of several similarity join operators and study the way they interact among themselves, with other standard database operators, and with other previously proposed similarity-aware operators. In particular, we present multiple transformation rules that enable similarity query optimization through the generation of equivalent similarity query execution plans. We then describe an efficient implementation of two similarity join operators, \( \varepsilon \)-Join and Join-Around, as core DBMS operators. The performance evaluation of the implemented operators in PostgreSQL shows that they have good execution time and scalability properties. The execution time of Join-Around is less than 5% of the one of the equivalent query that uses only regular operators while \( \varepsilon \)-Join’s execution time is 20 to 90% of the one of its equivalent regular operators based query for the useful case of small \( \varepsilon \) (0.01% to 10% of the domain range). We also show experimentally that the proposed transformation rules can generate plans with execution times that are only 10% to 70% of the ones of the initial query plans.

I. INTRODUCTION

The shift from systems that focus on exact semantics of data and queries to systems that focus on approximate and imprecise semantics is recognized as one of the main current paradigm transitions in data management systems. Different areas have made important contributions to this paradigm shift, among them: similarity-aware query processing in database systems, integration of information retrieval and database operations, and uncertain or probabilistic databases. The study of the similarity-aware counterparts of common database operations, i.e., selection, join, and grouping is a central goal of the work on similarity query processing. Similarity joins (SJ) are operations that combine two sets of data using similarity join predicates that match tuples with similar or approximate values. Similarity joins have been studied as key components to solve multiple problems, e.g., record linkage, data cleaning, phenomena detection on sensor networks, marketing analysis, multimedia and video applications, etc.

Multiple SJ algorithms and implementation techniques have been proposed. They range from out-of-database approaches for only in-memory or external memory data, to techniques that use standard database operators to answer SJs. However, there has not been much study on the role and implementation of similarity joins as database operators. Fig. 1 compares several approaches to implement Similarity Joins. The implementation of SJ as integrated database operators has the following key advantages: (i) SJ database operators can be interleaved with other regular and similarity-aware operators and their results pipelined for further processing; (ii) important optimization techniques, e.g., pushing certain filtering operators to lower levels of the execution plan, pre-aggregation, and the use of materialized views can be extended to the new operators; and (iii) the implementation of these operators can reuse and extend other operators and structures to handle large datasets, and use the cost-based query optimizer machinery to enhance query execution time.

This paper focuses on the study of similarity joins as first-class database operators. Its main contributions are:

- We study the similarity join as a first-class database operator, its interaction with other non-similarity and similarity-based operators, and its implementation as integrated component of the DBMS query processing and optimization engine.
- We present the different types of similarity joins, introduce a new useful similarity join type, the Join-Around, and propose SQL syntax to express similarity join predicates.
- We analyze multiple transformation rules for the SJ operators. These rules enable query optimization through the generation of equivalent query execution plans. We

---

* This work was partially supported by NSF Grant IIS-0811954.

---

Fig. 1 Comparison of similarity join implementation approaches

<table>
<thead>
<tr>
<th>Supported Join types</th>
<th>Integrated in DB Engine</th>
<th>Using Basic SQL Operators</th>
<th>Outside of DB</th>
<th>As Stored Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>All</td>
<td>Certain types may be unfeasible or require very complex queries</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>Implementation complexity</td>
<td>Can reuse and extend DB operators and structures</td>
<td>Queries use a complex mix of joins and aggregations</td>
<td>Requires specialized structures, mechanisms to deal with large data sets, etc.</td>
<td>Requires specialized structures, spilling mechanisms, etc.</td>
</tr>
<tr>
<td>Composable with other DB operators</td>
<td>Yes (full pipelining of results)</td>
<td>Yes (resulting queries can be highly complex)</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Take advantage of DB optimizer</td>
<td>Yes (trans. rules, pre-aggregation, MVs, etc.)</td>
<td>No directly</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
study: (i) multiple core equivalence rules for SJ operators; (ii) the main theorem of Eager and Lazy aggregation for queries with similarity join and similarity group-by; (iii) the scenarios in which similarity predicates can be pushed from similarity join to similarity group-by; and (iv) equivalence rules between different SJ operators and between SJ and the similarity group-by operator.

- We describe an efficient implementation of two SJ operators, the Epsilon-Join and Join-Around, as core DBMS operators. We consider the case of multiple SJ predicates and one-dimensional (1D) attributes.

- We evaluate the performance and scalability properties of our implementation of the Epsilon-Join and Join-Around operators in PostgreSQL. The execution time of Join-Around is less than 5% of the one of the equivalent query that uses only regular operators while E-Join’s execution time is 20 to 90% of the one of its equivalent regular operators based query for the useful case of small $\varepsilon$ (0.01% to 10% of the domain range).

- We also evaluate experimentally the effectiveness of the proposed transformation rules and show they can generate plans with execution times that are only 10% to 70% of the ones of the initial query plans.

The rest of this paper is organized as follows. Section II discusses the related work. Section III presents the different types of SJ and the proposed syntax to specify their similarity predicates. Section IV studies the equivalence rules among SJ and other regular and similarity-aware operators. Section V presents implementation guidelines based on a prototype realization of two SJ operators within PostgreSQL. Section VI reports the performance evaluation of the implemented operators and Section VII presents the conclusions and directions for future research.

II. RELATED WORK

Several types of similarity join, and corresponding implementation strategies, have been proposed in the literature, e.g., range distance join (retrieves all pairs whose distances are smaller than a pre-defined threshold) [1], [2], [3], [8], [9], [10], k-Distance join (retrieves the k most-similar pairs) [4], and kNNS-join (retrieves, for each tuple in one table, the k nearest-neighbors in the other table) [5], [6], [7]. The range distance join, also known as the E-Join, has been the most studied type of similarity join. Among its most relevant implementation techniques, we find approaches that rely on the use of pre-built indices, e.g., eD-index [8] and D-index [9]. These techniques strive to partition the data while clustering together similar objects. However, this approach may require rebuilding the index to support queries with different similarity parameter values, i.e., epsilon. Furthermore, eD-index and D-index are directly applicable only to the case of self-joins. Several non-index-based techniques have also been proposed to implement the E-Join. EGO [10], GESS [11], and QuickJoin [12] are three of the most relevant non-index-based algorithms. The Epsilon Grid Order (EGO) algorithm [10] imposes an epsilon-sized grid over the space and uses an efficient schedule of reads of blocks to minimize I/O. The Generic External Space Sweep (GESS) algorithm [11] creates hypersquares centered on each data point with epsilon length sides, and joins these hypersquares using a spatial join on rectangles. The Quickjoin algorithm [12] recursively partitions the data until the subsets are small enough to be efficiently processed using a nested loop join. The algorithm makes recursive calls to process each partition and a separate recursive call to process the “windows” around the partition boundary. Quickjoin has been shown to perform better than EGO and GESS [12].

Also, of importance is the work on similarity join techniques that make use of relational database technology [17], [18], [19]. These techniques are applicable only to string or set-based data. The general approach pre-processes the data and query, e.g., decomposes data and query strings into sets of q-grams, and stores the results of this stage on separate relational tables. Then, the result of the similarity join can be obtained using standard aggregate/group-by/join SQL statements. Indices on the pre-processed data are used to improve performance. A key difference of this work with our contributions in this paper is that we focus on studying the properties, optimization techniques, e.g., pre-aggregation and query transformation rules, and implementation techniques of several types of similarity joins as database operators themselves rather than studying the way a SJ can be answered using standard operators. In fact, several of the discussed properties for epsilon-join in this paper are also applicable to the operators proposed in [17] and [18]. Moreover, the implementation section of our work focuses on SJ on numerical data rather than string data.

A related type of join is the band join introduced in [32]. The join predicate of this join type has the form $S.s_1 \leq E1 \leq R.r \leq S.s_2 + E2$. A key difference of our work with the work on band joins is that band joins represent only a special case of one of the four types of joins considered in our study. Specifically, a band join where $E1 = E2$ is a special case of E-Join for the case of 1D data. We propose transformation rules and properties for similarity joins that apply in general to multi-dimensional data. Moreover, a key goal of our implementation is to take advantage of the mechanisms and data structures already available in most DBMS’ engines to facilitate the integration of similarity joins into real world DBMSs. The implementation of band joins in [32] makes use of specialized sampling, partitioning, and page replacement mechanisms.

Some recent work in the area of similarity joins has focused on: proposing a compact way to represent the output of an epsilon join [11], i.e., reporting groups of nearby points instead of every join link; efficient algorithms for in-memory similarity join with edit distance constraints [14]; algorithms for near duplicate detection that exploit the ordering of tokens in a record to reduce the number of required distance computations [15]; and similarity join algorithms that exploit sorting and searching capabilities of GPUs [16].

The extension of other standard operations to their similarity-based counterparts, e.g., similarity selection [20], [21], [22], [23], and similarity grouping [24], has been studied previously. Among the important recent contributions in this
The study of fast indices and algorithms for set similarity selection using semantic properties that allow pruning large percentages of the search space [20], a quantitative cost-based approach to build high-quality grams to support selection queries on strings [21], a method that finds all data objects that match with a given query object in a low-dimensional subspace instead of the original full space [22], and flexible dimensionality reduction techniques to support similarity search using the Earth Mover’s Distance [23]. Of special interest is the work on Similarity Group-by (SGB) presented in [24]. SGB is an extension of the group-by database operator that supports the formation of groups of similar objects. Three SGB instances are introduced, i.e., group-around, unsupervised group-by, and group-by with delimiters; and are shown to have good execution time and scalability properties with at most only 25% increase in execution time over the regular group-by [24]. We study the interaction and equivalences between SJ and SGB. Furthermore, we discuss scenarios in which the similarity predicate of SJ can be pushed partially or totally to SGB.

The work in [25] proposes an algebra for similarity-based queries. This work presents the extension of simple algebra rules, e.g., pushing selection into join, to the case of similarity operators. The work in [26] proposes an extension to the relational algebra to support similarity queries with several similarity predicates combined using the Boolean operators and, or, and not. However, [26] does not consider similarity joins or queries that combine non-similarity and similarity predicates. [27] proposes an extended SQL syntax to express queries that use both non-similarity and similarity predicates. The work in [28] presents a cost model to estimate the number of I/O accesses and distance calculations to answer similarity queries over data indexed using metric access methods. Both [27] and [28] only consider range distance and kNN-joins. A framework for similarity query optimization is presented in [29]. This work makes use of simple equivalence rules to generate multiple alternative query plans. The main difference between [25], [26], [27] and our work is that we focus on analyzing in detail the properties and equivalence rules that involve the different kinds of similarity join. Our study considers four types of SJ, the equivalences among them and with the similarity group-by operator. Furthermore, we study extensions of the important Lazy and Eager aggregation transformations to the case of similarity join queries.

Some of the optimization techniques of SJ presented in this paper build on previous work on optimization of regular non-similarity queries. Larson et al. study pull-up and push-down techniques that allow the query optimizer to move aggregation operators up and down the query plan [30], [31]. These techniques enable complete [30] or partial [31] pre-aggregation that can reduce significantly the input size of a join and decrease the execution time of an aggregation query.

III. SIMILARITY JOIN OPERATORS

The generic definition of the Similarity Join (SJ) operator is as follows:

\[ A \bowtie_{\theta_s} B = \{(a, b) \mid \theta_s(a, b), a \in A, b \in B\} \]

where \( \theta_s \) represents the similarity join predicate. This predicate specifies the similarity-based conditions that the pairs \( <a, b> \) need to satisfy to be in the similarity join output. The similarity join predicates for the similarity join operators considered in our study are as follows.

- **Range Distance Join (ε-Join):**
  \[ \theta_{\text{ε}} \equiv \text{dist}(a, b) \leq \varepsilon \]

- **kNN-Join:**
  \[ \theta_{\text{kNN}} \equiv b \text{ is a k - closest neighbor of } a \]

- **k-Distance-Join (kD-Join):**
  \[ \theta_{\text{kD}} \equiv (a, b) \text{ is one of the overall } k \text{ - closest pairs} \]

- **Join-Around (A-Join):**
  \[ \theta_{\text{A,MD=2r}} \equiv b \text{ is the closest neighbor of } a \text{ and } \text{dist}(a, b) \leq r \]

The range distance, kNN, and k-Distance join operators are common and extensively used types of similarity join. The Join-Around is a new useful type of similarity join that combines some properties of both the range distance and kNN joins. Every value of the first joined set is assigned to its closest value in the second set. Additionally, only the pairs separated by a distance of at most \( r \) are part of the join output. MD stands for Maximum Diameter and \( r=MD/2 \) represents the Maximum Radius. As presented in Section IV, the Join-Around operator with \( MD=\infty \) is equivalent to the kNN-Join for \( k=1 \). Some queries that show the usefulness of this new type of similarity join are presented later in this section.

Fig. 2 shows an extension of SQL syntax to express the different types of similarity join predicates. Fig. 3 shows examples of the four types of similarity join operators when they are applied to two numerical datasets.

Similarity joins are core operations in multiple application domains, e.g., data cleaning, pattern recognition, bioinformatics, multimedia, phenomena detection on sensor networks, marketing analysis, etc. Many of these scenarios, e.g., pattern recognition and bioinformatics, inherently need the support of similarity joins on multidimensional data. However, there are also many application scenarios, e.g., marketing analysis and phenomena detection on sensor
networks, that can greatly benefit from the use of similarity joins on one dimensional data. Fig. 4 gives four similarity queries that use similarity joins to answer business-oriented questions in a decision support system. The presented similarity queries are extensions of several non-similarity-based TPC-H queries [33]. The similarity queries in Fig. 4 illustrate that the use of similarity joins allows answering more complex and interesting business questions.

IV. Optimizing Similarity Joins

This section presents the study of similarity join properties and techniques that enable the optimization of similarity join queries through the generation of alternative execution plans. This section introduces: (i) core equivalence rules that exploit specific properties of SJs, (ii) equivalence rules between multiple SJ operators and between SJ and similarity group-by (SGB) operators, and (iii) the study of Eager and Lazy transformation techniques that exploit pre-aggregation using group-by and similarity group-by to significantly reduce the amount of data to be processed by SJs.

A. Core Equivalence Rules

This section presents multiple equivalence rules that involve the different SJ operators. This section not only considers the extension of common equivalence rules to the case of similarity joins, but particularly also studies scenarios that exploit certain specific properties of SJs to enable more effective query transformations. The rules in this section and in section IV.B use the notation presented in Fig. 5. The examples assume the following relations’ content: E1 = (1, 2, ..., 100), and E2 = (21, 22, ..., 25).

1) Basic Distribution of Selection over SJ. The regular selection operation distributes over the similarity join operations according to the following rules.

When all the attributes of the selection predicate \( \theta \) involve only the attributes of one of the expressions being joined (\( E_1 \)): a.\( \sigma_{\theta_1}(E_1) \bowtie_{\theta_2}(E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_2}(E_2) \) \( \bowtie \) b.\( \sigma_{\theta_1}(E_1) \bowtie_{\theta_{kn}}(E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_{kn}}(E_2) \) \( \bowtie \) c.\( \sigma_{\theta_1}(E_1) \bowtie_{\theta_A}(E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_A}(E_2) \)

When the selection predicates \( \theta_1 \) and \( \theta_2 \) involve only the attributes of \( E_1 \) and \( E_2 \), respectively:

d.\( \sigma_{\theta_1 \theta_2}(E_1) \bowtie_{\theta_A}(E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_A}(\sigma_{\theta_2}(E_2)) \)

Usage: In the RHS of these rules, the selection operator is pushed under the SJ operators to reduce the number of tuples to be processed by the join. The transformation from the LHS expression to the RHS one can generate low cost plans because in general SJ operators are expected to be more costly than selection filters. Fig. 6.a presents an example of rule 1.a. The numbers next to the arrows represent the number of flowing tuples in the query pipeline. The SJ operator of the LHS expression processes a total of 200 tuples while the one of the RHS expression only processes a total of 105 tuples.

2) Pushing Selection Predicate under Originally Unrelated Join Operand: In the equivalence rules presented in Section

<table>
<thead>
<tr>
<th>Similarity Query Example 1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Business Question</strong>: Study how well the order priority system works around dates of interest (holidays, marketing campaigns, etc.) <strong>Similarity-aware Query</strong></td>
</tr>
<tr>
<td><strong>Select</strong> ( d \textunderscore \text{refdate} ), ( c \textunderscore \text{orderpriority} ) <strong>as</strong> ( \text{order}<em>\text{count} ) <strong>from</strong> ( \text{order} \text{fact} ) <strong>where</strong> ( o \textunderscore \text{orderkey} \bowtie \text{around} \text{d}</em>{\text{refdate}} \text{and exists} \langle \text{SELECT} \ast \text{from} \text{lineitem} \rangle \text{where} \text{l}<em>{\text{orderkey}} \bowtie o</em>{\text{orderkey}} \text{and l}<em>{\text{commitdate}} \bowtie &lt; \text{l}</em>{\text{receiptdate}} \rangle \text{group by} \text{o}<em>{\text{orderpriority}} \text{d}</em>{\text{refdate}} \text{order by} \text{o}<em>{\text{orderpriority}} \text{d}</em>{\text{refdate}} \rangle</td>
</tr>
</tbody>
</table>
pushing $\theta$ under both operands of the join as shown in the following equivalence rule.

\[ \sigma_E(E_1 \bowtie_{\theta} E_2) \equiv (\sigma_E(E_1)) \bowtie_{\theta_E} (\sigma_{\theta \bowtie E_2}(E_2)) \]

where all the attributes of the selection predicate $\theta$ involve only the attributes of $E_1$, and the selection predicate $\theta \bowtie E$ represents a modified version of $\theta$ where each condition is “extended” by $E$ and is applied on the join attribute of $E_2$. For example, if $\theta = 10 \leq e_1 \leq 20$, then $\theta \bowtie E = 10 - \bowtie E e_2 \leq 20 + \bowtie E$.

**Usage:** The single selection operator of the LHS expression is used to filter both inputs of the join in the RHS expression. The transformation from the LHS expression to the RHS one can generate a plan with even lower cost than the one of the RHS expression only processes a total of 200 tuples while the one of the RHS expression only processes a total of 20 tuples.

3) **Basic Associativity of SJ Operators:** Similarity Join operators are associative using the following rules.

**Rules with the same type of similarity join:**

a. $(E_1 \bowtie_{\theta_1} E_2) \bowtie_{\theta_{2,\bowtie}} E_3 \equiv E_1 \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} (E_2 \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} E_3)$

b. $(E_1 \bowtie_{\theta_{1,\bowtie}} E_2) \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} E_3 \equiv E_1 \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} (E_2 \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} E_3)$

c. $(E_1 \bowtie_{\theta_{1,\bowtie}} E_2) \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} E_3 \equiv E_1 \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} (E_2 \bowtie_{\theta_{2,\bowtie} \bowtie_{\theta_1}} E_3)$

**Rules that combine different types of similarity and regular join:**

d. $(E_1 \bowtie_{\theta_{1,\bowtie}} E_2) \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}} E_3 \equiv E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}} (E_2 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}} E_3)$

e. $(E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}} E_2) \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3 \equiv E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} (E_2 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3)$

f. $(E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}} E_2) \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3 \equiv E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} (E_2 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3)$

g. $(E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}} E_2) \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3 \equiv E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} (E_2 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3)$

h. $(E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}} E_2) \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3 \equiv E_1 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} (E_2 \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie} \bowtie_{\theta_{1,\bowtie}}}} E_3)$

where $\theta_1$, $\theta_2$, and $\theta_3$ involve attributes from only $E_1$ and $E_2$; $\theta_2$, $\theta_3$, and $\theta_4$ involve attributes from only $E_2$ and $E_3$.

**Usage:** Given an expression with several SJ operators, the plan cost depends on how many tuples need to be processed by each SJ operator and the processing cost of each specific type of SJ. Thus, the cost depends on which SJ operation is computed first. This will determine the number of flowing tuples to be processed by the remaining SJ operators. Fig. 6.c presents an example of rule 3.a. The LHS expression computes first the last selective SJ and processes a total of 1158 tuples in the second one. The RHS expression computes first the most selective SJ and processes only 200 tuples in the second one. The optimizer will probably select the RHS plan.

4) **Associativity Rule that Enables Join on Originally Unrelated Attributes:** In the equivalence rules presented in Section IV.A.3, each join predicate involves the same attributes in both sides of the rule. In the case of E-Join, when the attributes $e_1$ of $E_1$ and $e_2$ of $E_2$ are joined using $E_1$ and the result joined with attribute $e_3$ of $E_3$ using $E_2$, there is an implicit relationship between $e_1$ and $e_3$ that is exploited by the following equivalence rule.

5) **Commutativity of SJ Operators:** Some similarity Join operations are commutative:

a. $E_1 \bowtie_{\theta_E} E_2 \equiv E_2 \bowtie_{\theta_E} E_1$

b. $E_1 \bowtie_{\theta_{kN}} E_2 \equiv E_2 \bowtie_{\theta_{kN}} E_1$

kNN-Join and Join-Around operators are not commutative.

**Usage:** Similarly to the case of regular join, the cost of a given implementation of a SJ operator can be different when considering the larger relation to be joined as the inner or outer input of the operator. This rule is used to consider both cases during cost-based optimization.

Additionally, other rules like the distribution of projection over SJ and the combination of selection predicates with SJ predicates apply to the case of SJs in a similar way they do to the case of non-similarity joins.

**B. Equivalence Among Similarity Operators**

The Join-Around and the Similarity Group Around (SGB-A) operators are equivalent in the following way:
a. $e_1$ around $E_2$ are $F(A)(E_1)\equiv e_2F(A)(E_1 \setminus e_1 \theta_A e_2)$

i.e., a SGB-A operation can be transformed into a regular Group-by applied to the result of a Join-Around operation.

Usage: This rule can be used to support a similarity grouping operation using the implementation of the Join-Around operation.

The following rules describe the special cases in which different similarity join operators are equivalent.

b. $E_1 \bowtie_{\theta,A,MD=\infty} E_2 \equiv E_1 \bowtie_{\theta,KN (k=1)} E_2$

c. $E_1 \bowtie_{\theta,A,MD=2} E_2 \equiv E_1 \bowtie_{\theta} E_2,$

if the joins operate on one-dimensional data and $2E < \min$ distance of consecutive points in $E_2$, i.e., there is no overlap in the $MD$ ranges.

d. $E_1 \bowtie_{\theta/k_0} E_2 \equiv E_1 \bowtie_{\theta} E_2,$

if $E = \text{distance of the } k\text{-th (longest) link in LHS}.$

C. Eager and Lazy Transformations with SJ and SGB

An important query optimization approach is the use of pull-up and push-down techniques to move the grouping operator up and down the query tree. The main Eager and Lazy aggregations theorem introduced in [30] enables several pull-up and push-down techniques for the regular, i.e., non-similarity, join and group-by operators. This theorem allows the pre-aggregation of data before the join operator to reduce its input size. The main theorem is extended in [24] to the case of regular join and similarity group-by (SGB). This subsection presents the extension of the main theorem to the case of similarity join and (regular or similarity) group-by. Furthermore, we study scenarios in which the similarity predicate of SJ operators can be pushed totally or partially to the grouping operator.

General usage: Figures 8, 9, 10, and 11 illustrate several cases of the eager and lazy transformations that will be studied in detail later in this section. In general, the single aggregation operator of the Lazy approach is split into two parts in the Eager approach. The first part pre-evaluates some aggregation functions and calculates the count before the join. The second part uses the intermediate information to calculate the final results after the join. Both the eager and lazy versions of a query should be considered during query optimization since neither of them is the best approach in all scenarios. Joins with high selectivity tend to benefit the Lazy approach while aggregations that reduce considerably the number of tuples that flow in the pipeline tend to benefit the Eager approach.

The presentation of the theorems and proofs in this section use the notation presented in Fig. 7. This notation is used because: (i) it allows a direct comparison with analogous theorems for regular operators [30] and for similarity grouping [24] that use a similar notation, and (ii) it uses a convenient representation of operators’ arguments that facilitates the presentation of the theorems and proofs. The Eager and Lazy aggregation theorems for the case of (i) regular join and group-by [30], and (ii) regular join and similarity group-by [24] are presented next. These theorems are referenced in the new extensions of the theorem studied later in this section.

Theorem 1 Eager/Lazy Aggregation Main Theorem for Group-by and Join: The following two expressions

are equivalent under the same conditions as Theorem 1. 1) Eager and Lazy Transformations with GB and SJ: The Eager and Lazy aggregation transformations can be extended to the case of similarity joins as shown in Theorem 3.

Theorem 3 Eager/Lazy Aggregation Main Theorem for Group-by and Similarity Join: The following expressions

are equivalent under the same conditions as Theorem 1. 1) Eager and Lazy Transformations with GB and SJ: The Eager and Lazy aggregation transformations can be extended to the case of similarity joins as shown in Theorem 3.

Theorem 2 Eager/Lazy Aggregation Main Theorem for Similarity Group-by and Join: The following expressions

are equivalent under the same conditions as Theorem 1. 1) Eager and Lazy Transformations with GB and SJ: The Eager and Lazy aggregation transformations can be extended to the case of similarity joins as shown in Theorem 3.
Proof sketch: The validity of this theorem relies on the following properties. Given $R_j'$ and $R_j''$ instances of $R_j$ and $R_j''$ respectively, the result of $(R_j' \bowtie \tilde{\sigma}_{C_0} R_j'')$ is equivalent to the result of $(R_j'' \bowtie \theta R_j')$ where $\theta = \text{disjunction of } (R_j C_0=x \land R_j C_0=y)$ for every different link $(x,y)$ of the result of $(R_j' \bowtie \tilde{\sigma}_{C_0} R_j'')$. (1) $\theta$, as defined in (1), remains unchanged and valid when $R_j''$ is augmented with tuples that have already present values of $R_j' C_0 i.e.,$ duplicates, or when such tuples are removed from $R_j''$. (2)

The validity of Theorem 3 can be shown by following these steps:

For every $R_d$ and $R_d''$ instances of $R_d$ and $R_d''$, respectively,

1. $E_1: F[AA_d AA_d] \pi_d[GA_d GA_w AA_d AA_d]$
   $g[GA_d GA_w] \sigma[C_d \land C_d] (R_d \bowtie \tilde{\sigma}_{C_0} R_d'')$
   is equivalent to $E_1': F[AA_d AA_d] \pi_d[GA_d GA_w AA_d AA_d]$
   $g[GA_d GA_w] \sigma[C_d \land C_d] (R_d'' \bowtie \theta R_d)$
   where $\theta$ is defined as in (1).

2. $E_2: F[AA_d AA_d] \pi_d[GA_d GA_w AA_d AA_d]$
   $g[GA_d GA_w] \sigma[C_d \land C_d] (R_d'' \bowtie \theta R_d)$
   is equivalent to $E_2': \pi_d[G_d GA_w FA] (F\mu[AA_w CNT], F\mu[FAA_d])$
   $\pi_d[GA_d GA_w AA_d FA_d CNT]$
   $g[GA_d GA_w] \sigma[C_d]$
   $(((F\mu[AA_d], COUNT)\pi_d[NGA_d GA_d], AA_d)]$
   $g[NGA_d] \sigma[C_d] (R_d'' \bowtie \theta R_d)$
   because of Theorem 1.

3. $E_3: \pi_d[G_d GA_w FA] (F\mu[AA_w CNT], F\mu[FAA_d])$
   $\pi_d[GA_d GA_w AA_d FA_d CNT]$
   $g[GA_d GA_w] \sigma[C_d]$
   $(((F\mu[AA_d], COUNT)\pi_d[NGA_d GA_d], AA_d)]$
   $g[NGA_d] \sigma[C_d] (R_d'' \bowtie \theta R_d)$
   is equivalent to $E_3': \pi_d[G_d GA_w FA] (F\mu[AA_w CNT], F\mu[FAA_d])$
   $\pi_d[GA_d GA_w AA_d FA_d CNT]$
   $g[GA_d GA_w] \sigma[C_d]$
   $(((F\mu[AA_d], COUNT)\pi_d[NGA_d GA_d], AA_d)]$
   $g[NGA_d] \sigma[C_d] (R_d'' \bowtie \theta R_d)$
   since the grouping operation before the join merges only tuples that share the same value of $R_d C_0$, and (2).

2) Eager and Lazy Transformations with SGB and SJ: The Eager and Lazy Aggregation transformations can be extended to the case of similarity join and similarity group-by as shown in Theorem 4.

**Theorem 4 Eager/Lazy Aggregation Main Theorem for Similarity Group-by and Similarity Join**: The following two expressions

$E_1: F[AA_d AA_d] \pi_d[GA_d GA_w AA_d AA_d]$
$g[GA_d GA_w] \sigma[C_d \land C_d] (R_d \bowtie \tilde{\sigma}_{C_0} R_d)$

$E_2: \pi_d[G_d GA_w FA] (F\mu[AA_w CNT], F\mu[FAA_d])$
$\pi_d[GA_d GA_w AA_d FA_d CNT]$
$g[GA_d GA_w] \sigma[C_d]$
$(((F\mu[AA_d], COUNT)\pi_d[NGA_d GA_d], AA_d)]$
$g[NGA_d] \sigma[C_d] (R_d'' \bowtie \theta R_d)$

where $\bowtie \tilde{\sigma}_{C_0}$ is kNN-Join, $\bowtie$-Join, or $\bowtie$-Join; are equivalent under the same conditions as Theorem 1.

**Usage**: An example of the use of this theorem is presented in Fig. 9. The number of tuples flowing in the pipelines is similar to the one of the previous example. The bottom grouping node of the Eager approach merges tuples that have: (i) the same value of $J_1$ and (ii) values of $G_2$ that belong to the same similarity group. In the example all the tuples of $T_1$ are merged even though they have different values of $G_1$.

**Proof sketch**: The validity of this theorem relies on the validity of theorems 2 and 3.

3) Pushing Similarity Predicate from $\bowtie$-Join to GB: This subsection and the following one explore ways to further enhance the filtering power of the pre-aggregation step of the Eager approach pushing down the similarity predicates from the SJ operator to the grouping one. The equivalences
described in these subsections are enhancements over the one presented in Section IV.C.1.

The similarity predicate of the E-Join can be (partially) pushed down to a grouping operator as shown in Fig. 10. The bottom aggregation of the Eager approach performs regular aggregation on G1 and similarity aggregation SGB-A' on J1 around J2 with \( \text{MAX\_GROUP\_DIAMETER} = 2\varepsilon \). SGB-A' is a variation of similarity group around (SGB-A) [24] that only merges tuples that are linked to only one central point (J2) by the E-Join. The value of J1 in a resulting tuple of SGB-A' can be the value of the central point, i.e., J2, or any of the values of J1 of the grouped tuples. In both cases, the E-Join of the Eager approach will generate the correct join links. SGB-A' generates at most one group per different value of J2, i.e., tuples with the same value of J2 in T2 are treated as a single central point. The goal of pushing the similarity predicate from SJ to the aggregation operator is to increase the number of pre-aggregated tuples while maintaining a grouping operator that can be executed quickly. SGB-A has been shown to have an execution time not higher than 25% of that of the regular group-by for one dimensional data. SGB-A' is expected to perform similarly.

**Usage:** In the example presented in Fig. 10, the bottom grouping node of the Eager approach merges all the tuples of T1 even though they have different J1 values. Notice that applying the transformation of Section IV.C.1 to this case would generate five tuples rather than one as the result of the bottom grouping node of the Eager approach.

The validity of this equivalence relies on the following properties: (i) if two tuples \( t_a \) and \( t_b \) are grouped by the bottom aggregation of the Eager approach around a center point tuple, say \( t_r \), then \( t_a \) and \( t_b \) will always be matched with \( t_r \) by the E-Join of the Lazy approach; and (ii) tuples that are not merged with others at the bottom aggregation of the Eager approach, are always processed in the same way in both approaches.

4) **Pushing Similarity Predicate from Join-Around to GB:** The similarity predicate of the Join-Around can be (completely) pushed down to a grouping operator as shown in Fig. 11. The bottom aggregation of the Eager approach performs regular aggregation on G1 and similarity aggregation SGB-A [24] on J1 around J2 with \( \text{MAX\_GROUP\_DIAMETER} = 2\varepsilon \). The value of J1 in a resulting tuple of SGB-A is the value of the central point, i.e., J2. This will enable generating the correct links using only a regular join in the Eager approach. This regular join is still required to obtain the values of G2 and S2. SGB-A generates at most one group per different value of J2, i.e., tuples with the same value of J2 in T2 are treated as a single central point.

**Usage:** As illustrated in Fig. 11, the Eager approach avoids completely the use of the SJ operator, using instead a fast similarity group-by operator and a regular join. In the example shown in Fig. 11, the bottom grouping node of the Eager approach merges all the tuples of T1 even though they have different values of J1; applying the transformation of Section IV.C.1 would produce five tuples instead.

**V. IMPLEMENTING SIMILARITY JOIN**

This section presents the guidelines to implement two similarity join operators, E-Join and Join-Around, inside the query engine of standard RDBMSs. Although the presentation is intended to be applicable to any RDBMS, some specific details refer to our implementation in PostgreSQL. One of the goals of the implementation is to reuse and extend already available routines and structures to minimize the effort needed to realize these operators. The E-Join and Join-Around operators are implemented as extensions of the Sort Merge Join (SMJ) operator and consider the case of one dimensional numeric data and multiple similarity join predicates.

To add support for SJs in the parser, the raw-parsing grammar rules, e.g., yacc rules in the case of PostgreSQL, are extended to recognize the syntax of the various new similarity join predicates presented in Section III. The parse-tree and query-tree data structures are extended to include the type and parameters, e.g., \( \varepsilon, \text{MD} \), of SJ predicates. The routines in charge of transforming the parse tree into the query tree are updated accordingly to process the new fields in the parse tree.

**A. The Optimizer**

Fig. 12a presents the structure of the plan tree when one similarity join predicate is used. Given that the
implementation is based on Sorted Merge Join, sort nodes that order by the similarity join attributes are added on top of the input plan trees. This order is assumed by the routines that find the similarity matches, i.e., links. When multiple similarity join predicates are used, they are processed one at a time. Fig. 12.b gives the structure of the plan tree generated when two similarity join predicates, \( a-b \) and \( c-d \), are used. The bottom similarity join makes use of \( a-b \) while the top one uses \( c-d \). The routes that find the similarity matches are presented in Section V.B. Another important change in the optimizer is in the way the number of tuples generated by a similarity aggregation node is estimated. This important estimation is used to compare the cost of different query execution plans. In the case of Join-Around, the number of resulting tuples can be estimated as the number of tuples in the inner input dataset. In the case of E-Join, more complex techniques, e.g., employing histograms of the density of elements in metric space [28], can be employed. The number of output tuples of the kNN-Join can be estimated as \((\# \text{ of tuples of outer input}) \times \min(k, \# \text{ of tuples of inner input})\) while the one of the KD-Join can be estimated as \(\min(\# \text{ of tuples of outer input } \times \# \text{ of tuples of inner input }, k)\). The estimated number of output tuples can be used to reduce the cost of queries with several similarity join predicates. Since the order of processing these predicates does not change the final result, they can be arranged to minimize the overall cost of the query.

### B. The Executor

When several similarity join predicates are used, the constructed query plan uses several similarity join nodes where the result of each node is pipelined to the next one as illustrated in Section V.A. The executor routines that produce the similarity links in a SJ node are expected to handle one similarity join predicate. Additionally, they could be extended to handle any number of regular join predicates. The tuples received from the input plans have been previously sorted as explained in Section V.A. The executor routines process the input tuples synchronously following a plane sweep approach.

Fig. 13 presents the algorithms of the main operation of the regular Sort Merge Join (13.a), Join-Around (13.b), and E-Join (13.c). The sections that were modified to support the SJ operators are shown in bold. It is clear from Fig. 13 that the use of the already implemented machinery that supports Sorted Merge Join as the basis to support similarity joins, allows a fast and efficient implementation of both SJ operators. The Sorted Merge Join algorithm in Fig. 13.a operates as follows. Lines 1 and 2 initialize the outer and inner tuples. Lines 4-9 advance the current inner and outer tuples until a match is found. When a match is found, Line 10 marks the inner tuple. Marking a tuple allows repositioning the inner cursor to the marked tuple later in the process. This key feature is already supported by the access method interface of PostgreSQL. Lines 13-18 join the current outer tuple with the current and following inner tuples as long as there is a match between outer and inner. Once an inner tuple that fails the match is found, the outer tuple is advanced (Line 19). Lines 20 to 24 test if the new outer tuple matches the marked tuple. If this is the case the inner cursor is restored to the marked tuple and the new match is processed, otherwise the process continues looking for a new match.

In the presentation of the algorithms, we assume that there is only one join predicate, i.e., the similarity predicate. The algorithms can be easily extended to handle the case of additional regular join predicates. The required changes to support E-Join are presented in Fig. 13.b. As expected, the function that evaluates if there is match between an outer and an inner tuples (Lines 4, 18, and 20) needs to be extended. In this case, the similarity predicate outer–inner is evaluated as \(\text{distance}(\text{outer}, \text{inner}) \leq \epsilon\). The block that produces the join links, in Lines 13-18, keeps track of the previous processed input tuple, i.e., prevInner. This tuple is used in Line 20 to test if there is a match between outer and prevInner. A positive result of this test means that there is at least one tuple in the range [mark, prevInner] that matches with the current outer. If this is the case, we restore the inner cursor to mark. The break command in Line 22 ensures that the process jumps to Line 4 to look for a match. This is required since outer may not match all the tuples in the range [mark, prevInner].

The required changes to support Join-Around are shown in Figures 13.c and 14. At any point, the algorithm keeps track of the current outer and inner and the next inner tuple, i.e., nextInner. Lines 2, 8, 16, and 22 in Fig. 13.c, and Lines 2 and 6 in Fig. 14 maintain the correct nextInner tuple. The function that evaluates if there is match between an outer and an inner tuples (used in Lines 5 and 20 in Fig. 13.c and Line 4 in Fig. 14) is also extended. In this case, the similarity predicate outer–inner is evaluated as \(\text{distance}(\text{outer}, \text{inner}) < \text{distance}(\text{outer}, \text{nextInner})\). The function that evaluates if an inner tuple matches another inner tuple (used in lines 4 and 18 in Fig. 13.c and in lines 1 and 3 in Fig. 14) evaluates the regular equality operator on the join attribute values. The expression \(\text{outer}>\text{inner}\) in line 1 of Fig. 14 ensures that the similarity join attribute of the outer tuple is greater than the one of the inner tuple. In contrast to the previous algorithms, when the process reaches line 10, there is not necessarily a match. This happens when there are consecutive inner tuples with the same join attribute values and the similarity join attribute of outer is greater than the one of inner. In this case, the inner cursor needs to be advanced until it is possible to check if there is a similarity match. This task is performed by check_match() as presented in Fig. 14. If a match is found, then the inner cursor is restored to mark and the process reports the join links. Otherwise, the process starts looking for a match again in
Fig. 13 Main operation of Epsilon-Join and Join-Around compared to the one of Sorted Merge Join

```
check_match() { 
if ((inner == nextInner) && (outer>inner)){
    do {advance inner ...
    else return False
}
return True //no need to advance to check match
}
```

Fig. 14 Routine check_match

line 4. The block that reports the join links is also modified to keep track of the previous inner, i.e., prevInner. This block (lines 13 to 18) outputs join links for the current inner and the consecutive inner tuples that have the same value of the join attribute. prevInner is used in line 18 to test if two consecutive inner tuples have the same join attribute values. prevInner is also used in line 20 to test if the new outer is closer to prevInner than to inner. Notice that if the result of this test is true, the new outer matches all the tuples in the range [mark, prevInner] and the process continues reporting the join links directly (line 13). The presented algorithms are coded in PostgreSQL in the fashion of a state machine. Fig. 13.d shows the states associated to the different tasks. The implementation of $\epsilon$-Join and Join-Around use the same set of states employed by SMJ.

The cost of the proposed SJ operators is close to the one of SMJ for reasonably small $\epsilon$ (for $\epsilon$-Join) and inner datasets without many duplicates (for Join-Around) because: (i) every outer tuple is read once in sequential order; (ii) the inner tuples are read in an almost sequential order, restoring the inner cursor to a previously read inner tuple is employed to generate the correct SJ links; (iii) in $\epsilon$-Join, if the inner cursor is restored, the length of the jump, i.e., distance from previous inner to marked tuple, is at most $2\epsilon$; and (iv) in Join-Around, if the inner cursor is restored, all the tuples in the range [marked tuple, previous inner tuple] share the same value of the similarity join attribute.

VI. PERFORMANCE EVALUATION

We implemented the $\epsilon$-Join and Join-Around, as described in Section V inside the PostgreSQL 8.2.4 query engine. In this section we evaluate the performance of these operators as well as the effectiveness of several transformation rules for SJs.

A. Test Configuration

The dataset used in the performance evaluation is based on the one specified by the TPC-H benchmark [33]. The Reference points tables and queries used in the tests are presented in Fig. 15. The default dataset scale factor (SF) is 5 (5GB). All the experiments are performed on an Intel Dual Core 1.83GHz machine with 2GB RAM running Linux as OS.
B. Performance Evaluation

We study the performance of the implemented operators comparing their execution time and scalability properties with the ones of queries that get similar results using only regular, i.e., non-similarity-based, operators. Notice that even though many implementation approaches have been proposed for SJs, e.g., [8], [9], [10], [11], [12], most of them have been proposed as standalone implementations not integrated within a DBMS engine and make use of specialized indices, data structures, partitioning, and access methods. The efficient integration of these techniques within a DBMS query engine and evaluation of their performance is a task for future work.

1) Join-Around Performance while Increasing Dataset Size: Fig. 16 gives the execution time of the JoinAround query compared to the one of the RegOps-JoinAround query that produces the same output using only regular operators. This figure compares the performance of both queries for different values of scale factor (SF). The number of customers is 150,000*SF while the number of central points is maintained constant. The execution time of RegOps-JoinAround grows from being about 20 times bigger than that of SJ-JoinAround for SF=1 to being about 200 times bigger for SF=8. The poor performance of RegOps-JoinAround is due to a double nested loop join in its execution plan in addition to the use of an aggregation operation. The Join-Around operator sorts each set once, and processes both sets synchronously.

2) E-Join Performance while Increasing $\varepsilon$: Fig. 17 gives the execution time of the SJ-EpsJoin query compared to the one of the RegOps-EpsJoin query that produces the same output. The results are presented for various values of $\varepsilon$. The value of $\varepsilon$ is a fraction of the domain range. Specifically, the customer account balance domain uses values in the range [0,11000].

This experiment uses SF=1. The key result of this experiment is that the SJ-EpsJoin query performs significantly better than the RegOps-EpsJoin query for small values of $\varepsilon$. For instance, when $\varepsilon=1$, the execution time of RegOps-EpsJoin is 4.32 sec. while the one of SJ-EpsJoin is 0.96 sec., i.e., RegOps-EpsJoin is over 4 times faster. The advantage of the E-Join over the regular query gets reduced as the value of $\varepsilon$ increases and is almost negligible when the size of $\varepsilon$ is about 20% of the domain range. Having a good performance for small values of $\varepsilon$ is of key importance for the E-Join operator since similarity join queries with small $\varepsilon$ are among the most common and useful types of similarity-based operations. The performance of SJ-EpsJoin is better for small values of $\varepsilon$ because it generates shorter restorations of the inner cursor. On the other hand, RegOps-EpsJoin calculates the distance between all the combinations of outer and inner tuples. This requires in general the same amount of I/O independently of the value of $\varepsilon$. The additional cost for high values of $\varepsilon$ is due to the increase in the number of links to be reported.

3) Effectiveness of Associativity transformation: AssocRule_LHS and AssocRule_RHS in Fig. 18 represent the query AssocRule executed using plans that corresponds to the LHS and RHS of the rule IV.A.3.a respectively. The execution time of AssocRule_RHS is 9.2% of that of AssocRule_LHS. AssocRule_LHS joins (E-Join) first Customer (C) and R2 generating 17,241,601 intermediate rows. The execution time of AssocRule_RHS is much smaller because it joins the two smaller tables (R1 and R2) first generating only 2519 intermediate rows.

4) Effectiveness of pushing selection under SJ: PushSel_LHS, PushSel_RHS1, and PushSel_RHS2 in Fig. 19 represent the query PushSel executed using plans that corresponds to the LHS and RHS of rule IV.A.1.a, and the RHS of rule IV.A.2.a
respectively. PushSel LHS performs first the join (7,241,601 intermediate rows) and then the selection. In PushSel RHS1 the selection operation has been pushed to the input corresponding to table Customer (300,872 intermediate rows). The execution time of PushSel RHS1 is 73% of the one of PushSel LHS. In PushSel RHS2 the filtering benefit is further improved by pushing selection operations on both inputs of the join. The execution time of PushSel RHS2 is only 55% of the one of PushSel LHS.

5) Effectiveness of Lazy and Eager aggregation transformations: In Fig. 20, LazyN and EagerN represent the query LazyEager executed using plans that corresponds to the expressions E1 and E2 of Theorem 3 respectively. The execution time of Eager1 is 35% of the one of Lazy1. The advantage of the Eager approach increases when the cardinality of the inner input grows as in Eager2 with an execution time that is only 9% of that of Lazy2.

VII. CONCLUSIONS AND FUTURE WORK

This paper focuses on the study and implementation of the Similarity Join (SJ) as a first-class database operator. Several previously proposed types of similarity join are considered in our study. In addition, a useful extension of the kNN-Join and E-Join, named Join-Around is introduced. The paper studies extensively the query operator properties of the Similarity Join. It presents multiple equivalence rules that not only direct extensions of known relational algebra rules but also exploit specific properties of similarity joins to enable more useful query transformations. The paper also studies the way the Eager and Lazy Aggregation transformation techniques can be applied to queries with JS and addresses the interaction and equivalences of the previously proposed Similarity Group-by (SGB) operators with the studied SJ operators. The paper presents guidelines to implement Join-Around and E-Join as core operators of a DBMS query engine and the performance evaluation of this implementation in PostgreSQL. The performance study shows that the SJ-based queries perform significantly better than queries that get the same result using only regular operators. Furthermore this section shows the effectiveness of several studied transformation rules.

Plans for future work include the study and integration of more complex similarity join strategies as database operators, in particular approaches that support multi-dimensional data; the extension of other operations, e.g., CUBE, ROLLUP, union and selection, as similarity-aware operators and the study of their interaction with SJ and SGB, the application of SJ and SGB operators to the area of privacy preservation and anonymity, and the study of similarity-based joins and aggregations as tools in business decision support systems.

REFERENCES