2012

Similarity-Aware Query Processing and Optimization

Yasin N. Silva
Arizona State University, ysilva@asu.edu

Walid G. Aref
Purdue University, aref@cs.purdue.edu

Per-Ake Larson
Microsoft Research, palarson@microsoft.com

Mohamed H. Ali
Microsoft Corporation, mali@microsoft.com

Report Number:
12-006

TECHNICAL REPORT

SIMILARITY-AWARE QUERY PROCESSING AND OPTIMIZATION

YASIN N. SILVA, Arizona State University, ysilva@asu.edu

WALID G. AREF, Purdue University, aref@cs.purdue.edu

PER-AKE LARSON, Microsoft Research, palarson@microsoft.com

MOHAMED H. ALI, Microsoft Corporation, mali@microsoft.com

Department of Computer Science

Purdue University

May 16, 2012

ABSTRACT

Many application scenarios, e.g., marketing analysis, sensor networks, and medical and biological applications, require or can significantly benefit from the identification and processing of similarities in the data. Even though some work has been done to extend the semantics of some operators, e.g., join and selection, to be aware of data similarities; there has not been much study on the role, interaction, and implementation of similarity-aware operations as first-class database operators. The focus of this thesis work is the proposal and study of several similarity-aware database operators and a systematic analysis of their role as query operators, interactions, optimizations, and implementation techniques. This work presents a detailed study of two core similarity-aware operators: Similarity Group-by and Similarity Join. We describe multiple optimization techniques for the introduced operators. Specifically, we present: (1) multiple non-trivial equivalence rules that enable similarity query transformations, (2) Eager and Lazy aggregation transformations for Similarity Group-by and Similarity Join to allow pre-aggregation before potentially expensive joins, and (3) techniques to use materialized views to answer similarity-based queries. We also present the main guidelines to implement the presented operators as integral components of a database system query engine and several key performance evaluation results of this implementation in an open source database system. We introduce a comprehensive conceptual evaluation model for similarity queries with multiple similarity-aware predicates, i.e., Similarity Selection, Similarity Join, Similarity Group-by. This model clearly defines the expected correct result of a query with multiple similarity-aware predicates. Furthermore, we present multiple transformation rules to transform the initial evaluation plan into more efficient equivalent plans.

CHAPTER 1 INTRODUCTION AND RELATED WORK

1.1. Introduction

It is widely recognized that the move from exact semantics of data and Boolean semantics of queries to imprecise and approximate semantics of data and queries is one of the key paradigm shifts in data management. This shift is fueled in part by the recognition that many application scenarios, e.g., marketing analysis, sensor networks, data warehousing, data cleaning, etc., require or can significantly benefit from the identification of similarities in the data. Several techniques have been proposed to extend some data operations, e.g., join and selection, to take advantage of data similarities. Unfortunately, there has not been much study on the role, interactions, and implementation of similarity-aware operations as first-class database operators. In this context, the research questions that drive our work are:

1. How can database systems take advantage of similarities in the data to answer complex similarity-based queries required in multiple application scenarios?

2. How can conventional database operators be extended to use similarities on the data?

3. How do these similarity-aware database operators interact among themselves and with the regular operators?

4. Which optimization and implementation techniques can be used to effectively realize the similarity-aware operators?

We argue that similarity-aware operators should be implemented as first-class database operators because, as shown in Figure 1, this approach has the following key advantages: (1) the similarity-aware operators can be interleaved with other regular or similarity-aware operators and its results pipelined for further processing; (2) 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 (3) 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. Therefore, the focus of our work is the proposal and study of several similarity-aware database operators and a systematic analysis of their role, interactions, optimizations, and implementation techniques.

	Similarity Operator Implementation Approach			
	Integrated in DB Engine	Using Basic SQL Operators	Outside of DB	As Stored Procedures
Supported Operator Instances	All	Certain types may be unfeasible or require very complex queries	All	All
Implementation complexity	Can reuse and extend DB operators and structures	Queries use a complex mix of joins and aggregations	Requires specialized structures, mechanisms to deal with large data sets, etc.	Requires specialized structures, spilling mechanisms, etc.
Composable with other DB operators	Yes (full pipelining of results)	Yes (resulting queries can be highly complex)	No	No
Take advantage of DB optimizer	Yes (use of MVs, pre-aggregation, etc.)	No directly	No	No

Figure 1-1 Comparison of Similarity Operator Implementation Approaches

As part of this paper, we present the results of the detailed study of two core similarity-aware database operators, i.e., Similarity Group-by (SGB) and

Similarity Join (SJ). We study optimization and implementation techniques for both SGB and SJ operators and systematically evaluate their performance. We also introduce a generic conceptual evaluation order for similarity queries with multiple similarity-aware operations. We present a rich set of generalized equivalence rules to extend cost-based query optimization to the case of similarity-aware operators.

The contributions of our work are as follows:

1. We introduce the Similarity Group-by (SGB) operator which extends standard Group-by to allow the formation of groups based on similarity rather than equality of the data.

2. We present a generic definition of the SGB operator and three instances to support: (1) the formation of groups based on fundamental group properties, e.g., group compactness and group size, (2) the formation of groups around points of interest, and (3) the formation of groups delimited by a set of limiting points. The proposed instances support similarity grouping of one or more independent one-dimensional attributes.

3. We extend the standard optimization techniques for regular aggregations to the case of SGB. In particular, we introduce the main theorem of Eager and Lazy similarity aggregations, an extension of the corresponding regular aggregation based theorem; and the requirements that a materialized view must satisfy to be used to answer a similarity aggregation query.

4. We implement the proposed SGB operators in PostgreSQL (an open source database system) and study their performance and scalability properties. We use SGB in modified TPC-H queries to answer interesting business questions and show that the execution time of all implemented SGB's instances is at most only 25% larger than that of the regular Group-by.

5. We study the Similarity Join (SJ) as a first-class database operator, its interaction with other non-similarity and similarity-based operators, and its implementation as integrated component of the query processing and optimization engine of Database Management Systems (DBMSs).

6. We present the different types of Similarity Join operators, introduce a new useful Similarity Join type, the Join-Around, and propose SQL syntax to express Similarity Join predicates.

7. We analyze multiple transformation rules for the SJ operators. These rules enable query optimization through the generation of equivalent query execution plans. We study: (1) multiple core equivalence rules for SJ operators; (2) the main theorem of Eager and Lazy aggregation for queries with Similarity Join and Similarity Group-by; (3) the scenarios in which similarity predicates can be pushed from Similarity Join to Similarity Group-by; and (4) equivalence rules between different SJ operators and between SJ and the SGB operator.

8. 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.

9. 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 Ɛ-Join's execution time is 20 to 90% of the one of its equivalent regular operators based query for the useful case of small Ɛ (0.01% to 10% of the domain range).

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

11. We introduce a conceptual evaluation order for similarity queries with multiple similarity-aware operations, i.e., Similarity Group-by, Similarity Join, and Similarity Selection. This evaluation order specifies a clear and consistent way to execute a similarity query. It also specifies unambiguously what the results of a similarity query are, even in the presence of various similarity aware operations.

12. We present many equivalence rules to transform query plans with multiple similarity-aware operations. These rules represent a generalized version of the rules proposed for SGB and SJ. Particularly, these rules can be used to transform the conceptual evaluation plan of a similarity query into equivalent plans with potentially better execution time.

We have previously published parts of the work presented in this technical report [52], [53], [54], [55], [56]. The work on Similarity Group-by is presented in [52]. The study of the Similarity Join operator is presented in [53]. In [54], we study the way SGB operators can be extensively used to implement a Decision Support System. In [55] we present *SimDB*, a Similarity-aware Database system that support multiple SGB and SJ operators. In [56] we present a synopsis of our work on similarity-aware query processing.

The rest of this paper is organized as follows. The remaining part of this chapter presents the related work. Chapter 2 introduces and discusses the Similarity Group-by Operator. Chapter 3 discusses the Similarity Join Operator. Chapter 4 introduces the conceptual evaluation order for similarity queries and presents many generalized transformation rules. Chapter 5 presents the conclusions and directions for future research.

1.2. Related Work

Clustering, one of the oldest similarity-aware operations, has been studied extensively, e.g., in pattern recognition, machine learning, physiology, biology, statistics, and data mining. In some of these application scenarios, finding the groups with certain similarity properties is the goal of data analysis while in

others finding the groups is just the first step for other operations, e.g., for data compression or discovery of hidden patterns or relationships among the data items. Jain et al. present an overview of clustering from a statistical perspective [1]. Berkhin surveys clustering techniques used in data mining [2]. These techniques consider the special data mining computational requirements due to very large datasets and many attributes of different types. Given that the result of the clustering process depends on the specific clustering algorithm and its parameter settings, it is important to assess the quality of the results. This evaluation process is termed *cluster validity* [3], [4]. Of special interest is the work on clustering of very large datasets. Single scan versions of the well-known clustering algorithms K-means and Cobweb for large datasets is proposed in [5] and [6]. CURE [7] and BIRCH [8] are two alternative clustering algorithms based on sampling and summaries, respectively. They use only one pass over the data and hence reduce notably the execution time of clustering. However, their execution times are still significantly slower than the one of the standard Group-by. The main differences between these operations and the Similarity Group-by operators we propose are: (1) the execution times of the SGB operators are very close to that of the regular Group-by; (2) SGB are fully integrated with the query engine allowing the direct use of their results in complex query pipelines for further analysis; and (3) the computation of aggregation functions in SGB is integrated in the grouping process and considers all the tuples in each group, not a summary or a subset based on sampling. The last feature allows for fast generation of cluster representatives with the exact values of the aggregation functions that can be used immediately by other operators in the query pipeline. Algorithms similar to CURE or BIRCH would require extra steps to evaluate aggregation functions or to make available their results to SQL queries. Several clustering algorithms have been implemented in data mining systems. In general, the use of clustering is via a complex data mining model and the implementation is not integrated with the standard query processing engine. The work in [9] proposes some SQL constructs to make clustering facilities available from SQL

in the context of spatial data. Basically, these constructs act as wrappers of conventional clustering algorithms but no further integration with database systems is studied. Li et al. extend the Group-by operator to approximately cluster all the tuples in a pre-defined number of clusters [10]. Their framework makes use of conventional clustering algorithms, e.g., K-means; and employs summaries and bitmap indexes to integrate clustering and ranking into database systems. Our study differs from [10] in that (1) we focus on similarity grouping operators independent of the support and tight coupling to ranking; (2) we introduce a framework that does not depend on possibly costly conventional clustering algorithms, but rather allows the specification of the desired grouping using descriptive properties such as group size and compactness; and (3) we consider optimization techniques of the proposed Similarity Group-by operators.

In the context of data reconciliation, Schallehn et al. propose SQL extensions to allow the use of user-defined similarity functions for grouping purposes [11] and similarity grouping predicates [12], [13]. They focus on string similarity and similarity predicates to reconcile records. Although they can be used for this purpose, the proposed SGB operators are more general and are designed to be part of a DBMS's query engine.

Significant work has also been carried out on the extension of certain common operations, i.e., Join and Selection, to make use of similarities in the data. This work introduced the semantics of the extended operations and proposed techniques to implement them primarily as standalone operations outside of a DBMS engine rather than as integrated database operators.

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) [14], [15], [16], [17], [18], [19], [20], [21] k-Distance join (retrieves the k most-similar pairs) [22], and kNN-join (retrieves, for each tuple in one table, the k nearest-neighbors in the other table) [23], [24], [25]. The range distance join, also known as the ε-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 [17] and D-index [18]. 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 Ɛ-Join. EGO [19], GESS [20], and QuickJoin [21] are three of the most relevant non-index-based algorithms. The Epsilon Grid Order (EGO) algorithm [19] 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 [20] creates hypersquares centered on each data point with epsilon length sides, and joins these hypersquares using a spatial join on rectangles. The Quickjoin algorithm [21] 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 [21]. Some Similarity Join techniques have been employed as building blocks to implement common clustering algorithms [26]**.** Kriegel et al. extend the work on Similarity Join to uncertain data [27].

Also, of importance is the work on Similarity Join techniques that make use of relational database technology [28], [29], [30]. 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 SQL statements. Indices on the pre-processed data are used to improve performance. A key difference between this work and our contributions 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 chapter are also applicable to the operators proposed in [28] and [29]. Moreover, the implementation component of our work focuses on SJ on numerical data rather than string data.

A related type of join is the band join introduced in [31]. The join predicate of this join type has the form $S.s-\mathcal{E}1 \leq R.r \leq S.s+\mathcal{E}2$. A key difference between our work and 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 $\mathcal{E}1=\mathcal{E}2$ is a special case of Ɛ-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 [31] 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 [32], i.e., reporting groups of nearby points instead of every join link; efficient algorithms for in-memory Similarity Join with edit distance constraints [33]; algorithms for near duplicate detection that exploit the ordering of tokens in a record to reduce the number of required distance computations [34]; and Similarity Join algorithms that exploit sorting and searching capabilities of GPUs [35].

The special cases of Similarity Joins with one-tuple inner relations correspond to several types of Similarity Selection. Among key recent contributions on Similarity Selection we have: the study of fast indices and algorithms for set-based Similarity Selection using semantic properties that allow pruning large percentages of the search space [36], a quantitative cost-based approach to

build high-quality grams to support selection queries on strings [37], 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 [38], and flexible dimensionality reduction techniques to support similarity search using the Earth Mover's Distance [39].

The optimization techniques we present for SGB and SJ operators build on previous work on optimization of regular aggregation queries. Larson et al. study pull-up and push-down techniques that enable the query optimizer to move aggregation operators up and down the query plan [40], [41]. These techniques allow complete [40] or partial [41] pre-aggregation that can reduce the input size of a join and consequently decrease significantly the execution time of an aggregation query. Galindo-Legaria proposes a general framework for optimization of queries with subqueries and aggregations [42]. Another technique that can provide substantial improvements in query processing is the use of materialized views to answer aggregation queries. This technique is presented in [43] for the case of *sum* and *count* aggregation functions, and is extended in [44] and [45] to arbitrary aggregation functions.

The work in [46] 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 [47] 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, [47] does not consider Similarity Joins or queries that combine non-similarity and similarity predicates. [48] proposes an extended SQL syntax to express queries that use both non-similarity and similarity predicates. The work in [49] 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 [48] and [49] only consider range distance and knn-joins. A framework for similarity query optimization is presented in [50]. This work makes use of simple

equivalence rules to generate multiple alternative query plans. The main difference between [46], [47], [48] and our work is that we focus on analyzing in detail the properties among different types of similarity-aware operators, among different instances of the same similarity operator, and among regular and similarity-aware operators. Furthermore, we study the extension of query optimization techniques, e.g., lazy and eager aggregation transformations, and the use of materialized views to answer queries, to the case of similarity-based queries.

CHAPTER 2 THE SIMILARITY GROUP-BY DATABASE OPERATOR

Group-by is a core database operation that is used extensively in OLTP, OLAP, and decision support systems. In many application scenarios, it is required to group similar but not necessarily equal values. In this chapter we propose a new SQL construct that supports similarity-based Group-by (SGB). SGB is not a new clustering algorithm, but rather is a practical and fast similarity grouping query operator that is compatible with other SQL operators and can be combined with them to answer similarity-based queries efficiently. In contrast to expensive clustering algorithms, the proposed Similarity Group-by operator maintains low execution times while still generating meaningful groupings that address many application needs. The chapter presents a general definition of the Similarity Group-by operator and gives three instances of this definition. The chapter also discusses how optimization techniques for the regular Group-by can be extended to the case of SGB. The proposed operators are implemented inside PostgreSQL. The performance study shows that the proposed similarity-based Group-by operators have good scalability properties with at most only 25% increase in execution time over the regular Group-by.

2.1. Similarity Group-By: Definition

This section presents the general definition of the Similarity Group-by operator along with three instances that enable: (1) grouping tuples based on desired group properties, e.g., group size and group compactness, (2) grouping tuples around points of interest, and (3) segmenting the tuples based on given limiting values.

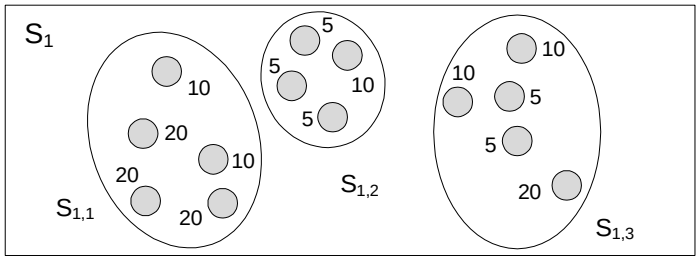


Figure 2-1 Example of the Use of the Generic SGB Definition

2.1.1. Generic Definition

We define the Similarity Group-by operator as follows:

$$_{(G_1,S_1),\ldots,(G_n,S_n)}\gamma_{F_1(A_1),\ldots,F_m(A_m)}(R)$$

where *R* is a relation name, G_i is an attribute of *R* that is used to generate the groups, i.e., a similarity grouping attribute, S_i is a segmentation of the domain of G_i in non-overlapping segments, F_i is an aggregation function, and A_i is an attribute of *R*. The formation of groups has two steps:

1. For each tuple *t*, each value v_i of $t.G_i$ is replaced by the identifier of the segment (member of S_i) that contains v_i. If no segment contains v_i, *t* is dismissed.

2. The resulting tuples are merged to form the similarity groups. Two tuples are in the same group if their new $G_1,\ldots,G_n$ values are the same.

The aggregation functions F_i are applied over each group similar to a standard aggregation operation. Figure 2-1 illustrates an example segmentation S_1 that groups a two-dimensional data set into three segments $S_{1,1}$, $S_{1,2}$, and $S_{1,3}$ based on some notion of similarity. Let the dots in the figure represent the tuples of a relation $R(G_1, A_1)$, where the value of G_1 is the position of the dot and the value of A_1 is the value next to the dot. The result of:

$$_{(G_1,S_1)}\gamma_{Sum(A_1)}(R)$$

is: {($S_{1,1}$, *80*), ($S_{1,2}$, *25*), ($S_{1,3}$, *50*)}.

2.1.2. Instantiating the General Definition

The general definition of Similarity Group-by (SGB) allows the use of any kind of segmentation on the grouping attributes. The segmentation could be the result of any clustering algorithm. For example, the previously proposed clustering approaches for large datasets [5], [6], [7], [8] can be modeled as instances of this generic definition. The generic definition is useful for reasoning with the new SGB operation and for deriving equivalences that allow the optimization of queries (as in Section 2.2). Naturally, this generic form of SGB is not to be implemented directly. Below, we present three implementable instances of the generic SGB. The main factors considered in the selection of the proposed instances are: (1) the ability to generate meaningful and useful groups, e.g., around a set of points of interest or groups that satisfy key properties such as group size and group compactness; (2) the viability of a fast implementation, e.g., using a single-pass plane-sweep approach; and (3) the usefulness of the instances in practical scenarios; the specific scenarios considered in this chapter are: business decision support systems (Section 2.4.2.3) and sensor networks (Section 2.1.2). The proposed instances represent middle ground between the regular Group-by and standard clustering algorithms. The proposed Similarity Group-by instances are intended to be much faster than regular clustering algorithms and generate groupings that capture similarities on the data not captured by regular Group-by. On the other hand, the quality of the generated groupings is not expected to be always as high as the ones generated by more complex and costly clustering algorithms. The presentation in this section focuses on the case of one or multiple independent grouping attributes (multiple independent dimensions).

2.1.2.1. Unsupervised Similarity Group-by (SGB-U)

This operator groups a set of tuples in an unsupervised fashion, i.e., with no extra data provided to guide the process. The SGB-U operator uses the following two clauses to control the group size and the group compactness:

1. MAXIMUM_ELEMENT_SEPARATION *s*: If the distance between two neighbor elements (consecutive elements, for the one-dimensional case) is greater than *s*, then these elements belong to different groups.

2. MAXIMUM_GROUP_DIAMETER *d*: For each formed group, the distance between the extreme elements of a group should be less than or equal to *d*.

The SQL syntax of the SGB-U operator is:

SELECT *select_expr*, ...
FROM *table_references* WHERE *where_condition*
GROUP BY *col_name*
[MAXIMUM_ELEMENT_SEPARATION *s*]
[MAXIMUM_GROUP_DIAMETER *d*], ...

In the case of one-dimensional attributes, the Similarity Group-by operator forms the groups in the following way:

1. If neither of the clauses MAXIMUM_ELEMENT_ SEPARATION, or MAXIMUM_GROUP_DIAMETER is specified, we assume *d*=0 and *s*=0. This case is equivalent to the standard Group-by.

2. If only one clause is specified, we assume that the value of the other is ∞.

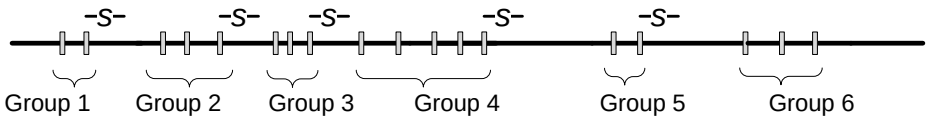
3. If MAXIMUM_ELEMENT_SEPARATION is specified, the elements are grouped first using this criterion. If only MAXIMUM_GROUP_DIAMETER is specified, all the elements form the unique resulting group of this step.

4. If MAXIMUM_GROUP_DIAMETER is specified, the groups formed in the previous step are further divided until the group diameter. The criterion to divide a group can be: (1) split a group “breaking” the longest link in the group, or (2) process the elements in ascending order and end current group as soon as the distance from the start of the group to the current element *E* is greater than *d*. We use this approach in our examples.

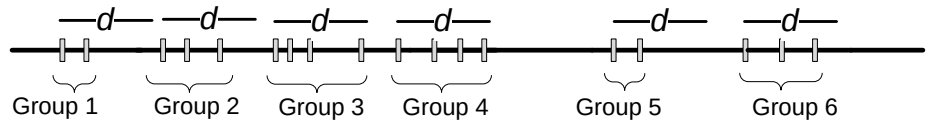
One way to extend the semantics of group diameter and element separation to higher dimensions is as follows. Assume that we build the minimum spanning tree that connects all the elements. Group diameter is the distance between the two most separated elements of a group. Element separation is defined for each pair of elements connected by a link of the tree, and its value is equal to the length of this link. Initially, all the elements connected by the tree form a group. If MAXIMUM_ELEMENT_SEPARATION is specified, all the links whose length is greater than *s* are "broken". If MAXIMUM_GROUP_DIAMETER is specified, we further divide the resulting connected groups until the group diameter of each group is less than or equal to *d*. To split a group, we *break* the longest link of its spanning tree. The following example groups a set of sensor readings such that in each formed group, the distance between two consecutive values is at most 2 degrees. Similar to the regular Group-by, each tuple that belongs to the result of the query represents one group.

```
SELECT Min(Temperature), Max(Temperature),
        Count(Temperature), Avg(Temperature)
FROM SensorsReadings WHERE Temperature > 0
GROUP BY Temperature MAXIMUM_ELEMENT_SEPARATION 2
```

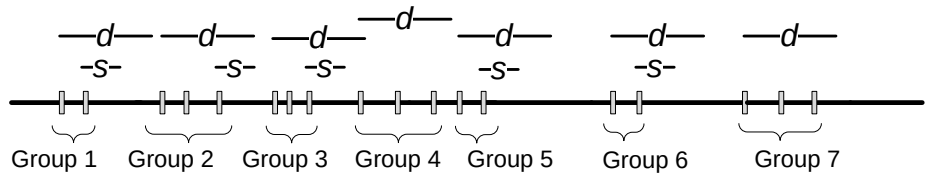
Figure 2-2.a gives one possible output of the previous example. The different temperature readings are represented as marks on a line. Figures 2-2.b and 2-2.c give the output when using the other two possible combinations of the clauses of this operator. In practice, different combinations can be more suitable for different grouping purposes. As evident from Figure 2-2, the use of group size and element separation to guide the process of similarity grouping captures important aspects of the natural formation of groups. These key properties are actually the building elements of more sophisticated clustering algorithms (e.g., as in [1]).



a) GROUP BY Temperature MAXIMUM_ELEMENT_SEPARATION 2



b) GROUP BY Temperature MAXIMUM_GROUP_DIAMETER 6



c) GROUP BY Temperature MAXIMUM_ELEMENT_SEPARATION 2
MAXIMUM_GROUP_DIAMETER 6

Figure 2-2 Examples of Unsupervised Similarity Grouping Limiting the Groups Based on Group Size and Compactness

2.1.2.2. Supervised Similarity Group Around (SGB-A)

The SGB-A similarity grouping operator groups tuples based on a set of guiding points, named central points, such that groups are formed around the central points and each tuple is assigned to the group of its closest central point. Additionally, the SQL syntax of SGB-A provides two clauses that are similar to the ones for the SGB-U operator (Section 2.1.2.1) to restrict the size and compactness of a group. The SQL syntax of the operator is:

SELECT *select_expr*, ...

FROM *table_references* WHERE *where_condition*

GROUP BY *col_name* AROUND *central-points*

[MAXIMUM_GROUP_DIAMETER *2r*]

[MAXIMUM_ELEMENT_SEPARATION *s*], ...

The central points can be specified directly using a list of points or, more generally, by another select statement. The latter option is very useful when the location of the central points depends on dynamic data. In the case of one-dimensional attributes, SGB-A forms the groups as follows:

1. Each tuple is assigned the group with closest central point.

2. If neither clause (MAXIMUM_ELEMENT_ SEPARATION, MAXIMUM_GROUP_DIAMETER) is specified, the groups formed in the previous step are the output of this operator.

3. If only one clause is specified, we assume that the value of the other is ∞.

4. If MAXIMUM_ELEMENT_SEPARATION is specified, the extent of each group is restricted such that each pair of consecutive elements of a group is separated at most by *s*. For this step we can consider the central point of each group to be one additional data point. The elements that are not connected to the central point under this compactness restriction are discarded.

5. If MAXIMUM_GROUP_DIAMETER is specified, the groups formed in the previous steps are further narrowed by removing all the elements whose distance from their central point is greater than *r*.

For multidimensional attributes, the semantics of group diameter and element separation can be extended as follows:

1. If MAXIMUM_GROUP_DIAMETER is specified, the groups are formed around the central points such that the distance from each point of a group to its central point is less than *r*.

2. If MAXIMUM_ELEMENT_SEPARATION is specified, the groups are further reduced such that it is possible to build a path from each element to its central point in which the length of every link is at most *s*.

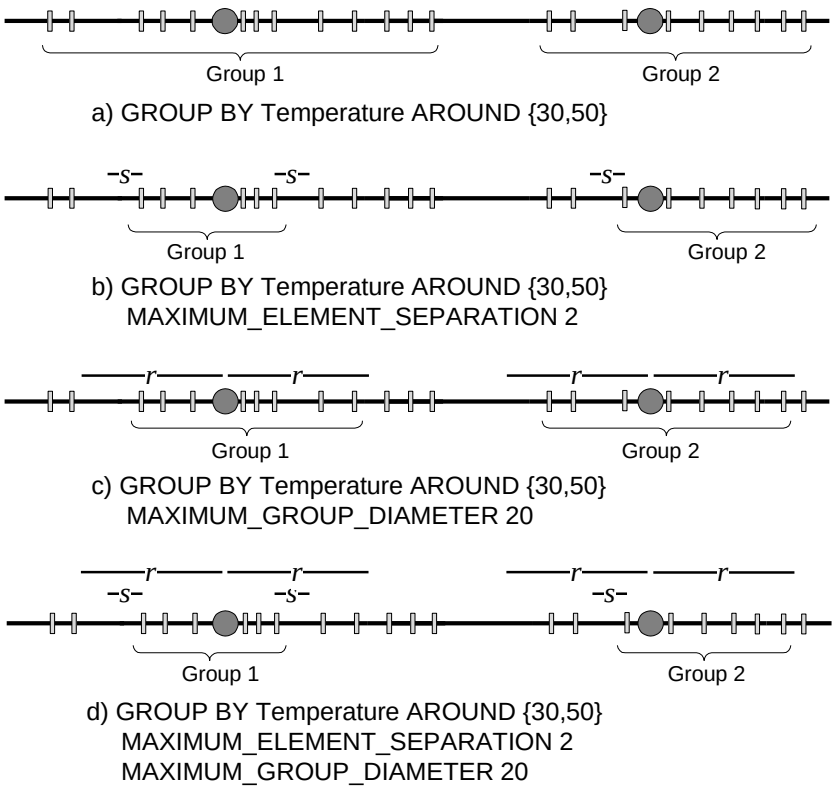


Figure 2-3 Examples of Supervised Similarity Grouping around Two Points under Various Conditions on the Group Size and Compactness

Unlike operator SGB-U of Section 2.1.2.1, operator SGB-A generates at most as many groups as central points are provided and all the elements that do not belong to any group are not considered in the output. Alternatively, all the discarded tuples could form a special group, i.e., group of outliers. Continuing with the scenario of applying similarity grouping to data retrieved from sensors, the following example groups the temperature readings around two temperature values of interest (30 and 50 degrees). Furthermore, the groups are restricted to include only readings whose distance from their central point is at most 10.

```
SELECT Min(Temperature),  Avg(Temperature)
FROM SensorsReadings WHERE Temperature > 0
GROUP BY Temperature AROUND {30,50}
MAXIMUM-GROUP-DIAMETER 20
```

Figure 2-3.c gives one possible output of the previous example. The given central points are represented as small circles. Figures 2-3.a, 2-3.b, and 2-3.d give the output when using the other three possible combinations of the clauses of SGB-A. From these figures, we observe that SGB-A can identify the naturally formed groups around certain points of interest.

In the operators defined so far, clauses to describe desired properties of the groups are combined implicitly using the AND operator. Although not shown in this chapter, we can combine the conditions using other logic operators.

2.1.2.3. Supervised SGB using Delimiters (SGB-D)

The SGB-D similarity grouping operator forms groups based on a set of delimiting points that can be provided directly or specified using a select statement.

In the case of one-dimensional attributes, this operator is especially useful when the partition of the line representing all the possible values of an attribute cannot be obtained using a set of central points. Figure 2-4.a gives an example of this scenario. SGB-D should be used when the natural way to form the required groups is to partition the range of all possible values in predefined or dynamic segments. SGB-D's syntax is:

SELECT *select_expr*, …

FROM *table_references* WHERE *where_condition*

GROUP BY *col_name* DELIMITED BY *limit-points*

The following example groups the temperate readings in groups delimited by the result of a select statement on Table Thresholds.

SELECT Count(Temperature), Avg(Temperature)

FROM SensorsReadings WHERE Temperature > 0

GROUP BY Temperature

DELIMITED BY (SELECT Value FROM Thresholds)

Figure 2-4.b gives the output of the previous example. The result of the internal select is represented by vertical dotted line segments.

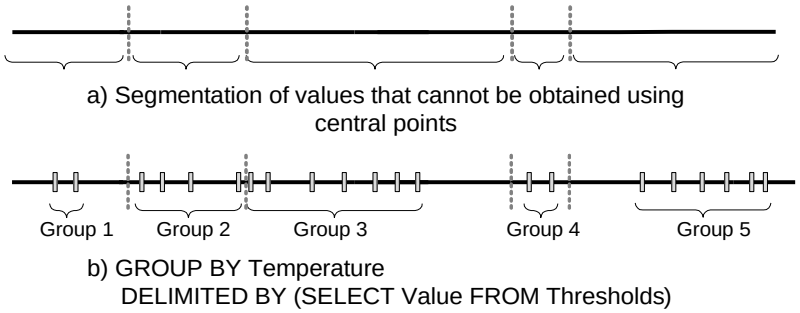


Figure 2-4 Example of Supervised Similarity Grouping Based on a Dynamic Set of Delimiting Points

Extending the semantics of SGB-D to multidimensional attributes can be achieved replacing *limit-points* by a set of geometrical objects, e.g., lines or planes, that partition the multidimensional space containing the elements to be grouped.

An important property of all the presented operators is that multiple executions of the operators on the same data set and same reference points, i.e., central and delimiting points, will generate the same results.

The generic definition of SGB specifies how similarity groups should be formed when several similarity grouping attributes (SGAs) are used. In general, we assume that the segmentation of each SGA is generated using a different similarity grouping instance. The main definition assumes that the SGAs are independent, i.e., the segmentation associated with each SGA *A* depends only on the values of *A* in the data tuples, and the reference points and conditions used with this SGA. According to this generic definition, the result of SGB when multiple SGAs are used is obtained intersecting the segmentations of all the (independent) SGAs. Therefore, the order in which the grouping attributes are specified in a similarity grouping query does not affect its final result. Clustering and segmentation based on correlated attributes is beyond the scope of this

chapter. From an implementation point of view, all the similarity grouping strategies associated with the different operators presented so far can be integrated into one single Similarity Group-by operator. This integration facilitates the use of several similarity grouping strategies in the same SQL statement. The following example applies Similarity Group Around (SGB-A) on attribute Pressure and Similarity Group-by with Delimiters (SGB-D) on attribute Temperature. The sets of elements delimited by dashed lines in Figure 2-5 represent the output of this query.

```
SELECT Avg(Temperature), Avg(Pressure)
FROM SensorsReadings GROUP BY
Pressure AROUND {30,50} MAXIMUM_ELEMENT_SEPARATION 3,
Temperature DELIMITED BY (SELECT Value FROM Thresholds)
```

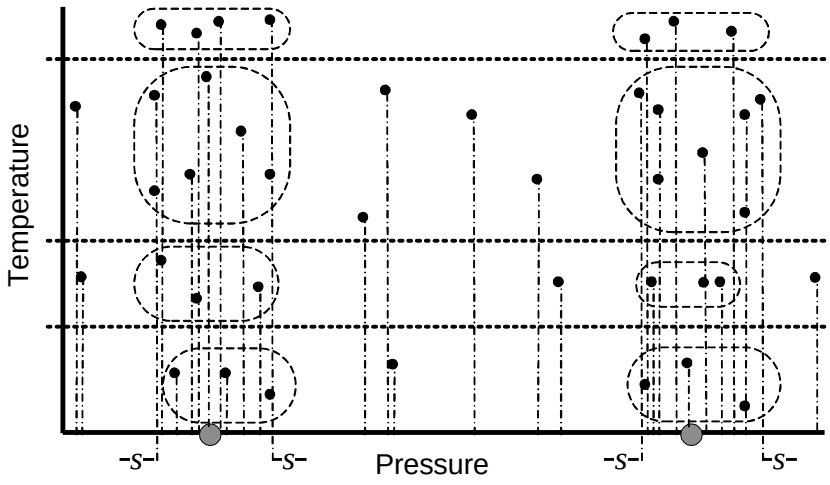


Figure 2-5 Similarity Grouping with Two Grouping Attributes

2.2. Optimizing Similarity Group-by

Several approaches have been proposed to improve the performance of regular aggregation queries. This section presents a study of how these approaches can be extended to the case of similarity grouping. An important approach to optimize

queries with regular aggregations is the use of pull-up and push-down techniques to move the Group-by operator up and down the query tree. The main Eager and Lazy aggregations theorem presented in [40] is a fundamental theorem that enables several pull-up and push-down techniques. Its application allows the pre-aggregation of data, i.e., aggregation before join, and thus potentially reduces the number of tuples to be processed by the join operator. Eager and Lazy similarity aggregations are query transformation classes that extend their regular aggregation counterparts. Figure 2-6 illustrates the transformations of the main theorem for Eager and Lazy similarity aggregation. The single similarity-based 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 that intermediate information to calculate the final results after the join. Similar to the case of non-similarity-based aggregations, it is important to consider both the Eager and Lazy versions of a similarity aggregation query because neither approach is the best in all scenarios. Joins with high selectivity tend to benefit the Lazy approach while aggregations that reduce significantly the number of flowing tuples in the pipeline tend to benefit the Eager approach. Section 2.4.2.3 presents real world scenarios in which each of the approaches performs better.

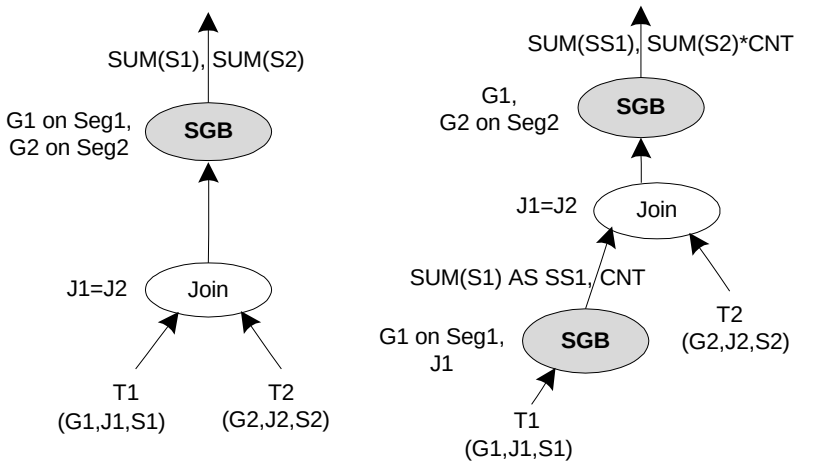


Figure 2-6 Eager and Lazy Aggregation Transformations - The Main Theorem

The algebraic notation used in this section is similar to that in [40]. *g[GA; Seg]R* represents similarity grouping of relation *R* on grouping attributes *GA* using segmentations *Seg*. The domain of the n^{th} element of *GA* is partitioned by the n^{th} element of *Seg*. This operation can be represented by a query that replaces in *R* each value of a grouping attribute by the representative value of the segment that contains it, and sorts the result by *GA*. Each segmentation is assumed to cover the whole domain of its associated attribute. The extension of the main theorem to the case in which this is not true is straightforward. *F[AA]R* represents the aggregation operation of a previously grouped table *R*. *F* and *AA* are sets of aggregation functions and columns, respectively. ×, σ, π_D, π_A, and U_A represent Cartesian product, selection, projection with and without duplicate elimination, and set union without duplicate elimination operations, respectively.

The presentation of the main theorem uses the following notation. R_d is a table that always contains aggregation attributes. R_u is a table that may or may not contain such attributes. Let GA_d and GA_u be the grouping columns of R_d and R_u, respectively, *AA* be all the aggregation columns, AA_d and AA_u be the subsets of *AA* that belong to R_d and R_u, respectively, C_d and C_u be the conjunctive predicates on columns of R_d and R_u, respectively, C_0 be the conjunctive

predicates involving columns in both R_u and R_d, $\alpha(C_0)$ be the columns involved in C_0, $GA_d^+ = GA_d \cup \alpha(C_0) - R_d$ be the columns that participate in the join and grouping, F be the set of all aggregation functions, F_d and F_u be the members of F applied on AA_d and AA_u, respectively, FAA be the resulting columns of the application of F on AA in the first grouping operation of the eager strategy, Seg be the set of segmentation of the attributes in GA, Seg_d and Seg_u be the subsets of Seg for the attributes in GA_d and GA_u, respectively, NGA_d be a set of columns in R_d, CNT be the column with the result of $Count(*)$ in the first aggregation operation of the eager approach, FAA_d be the set of columns, other than CNT, produced in the first aggregation operation of the eager approach, and F_{ua} be the duplicated aggregation function of F_u, e.g., if $F_u=(SUM,MAX)$, then $F_{ua}=(SUM, MAX, count) = (SUM*count, MAX)$. Let $A \sim B$ denote that A and B belong to the same similarity group, and $A \not\sim B$ denote the opposite.

Theorem 2-1 Eager/Lazy Similarity Aggregation Main Theorem. The following two expressions:

$$E_1: F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]$$
$$\gamma [GA_d, GA_u; Seg]\sigma[C_d \wedge C_0 \wedge C_u] (R_d \times R_u)$$

$$E_2: \pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u,CNT], F_{d2}[FAA_d])$$
$$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]$$
$$\gamma [GA_d, GA_u; Seg_u]\sigma[C_0 \wedge C_u]$$
$$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]$$
$$\gamma [NGA_d; Seg_d]\sigma[C_d]R_d) \times R_u)$$

are equivalent if (1) F_d can be decomposed into F_{d1} and F_{d2}, (2) F_u contains only class C or D aggregation functions [40], (3) $NGA_d \rightarrow GA_d^+$ holds in $\sigma[C_d]R_d$, and (4) $\alpha(C_0) \cap GA_d = \emptyset$.

Expression E_1 represents the Eager approach while expression E_2 represents the Lazy approach.

Proof sketch of Theorem 2-1

Consider a group G_d generated by $g\ [NGA_d,\ Seg_d]\sigma[C_d]r_d$ for some instance r_d of R_d. Due to conditions (3) and (4), all the rows of G_d have the same values of GA_d and the joining attributes. Every tuple of G_d joins with the same set of tuples $SA_u(G_d)$. Let $S_u(G_d)$ be the subset of $SA_u(G_d)$ that has a unique value of GA_u. Consider two groups of $g\ [NGA_d,\ Seg_d]\sigma[C_d]r_d$: R_{d1} and R_{d2}. There are two cases to be considered.

Case 1: $G_{d1}[GA_d] \sim G_{d2}[GA_d]$ and $S_u(G_{d1})[GA_u] \sim S_u(G_{d2})[GA_u]$. In E_2, the results of the join operations represented by the following two expressions are merged into the same similarity group by the second Similarity Group-by.

i. $((F_{d1}[AA_d],\ COUNT)\pi[NGA_d,\ GA_d^+,\ AA_d]G_{d1}) \times S_u(G_{d1})$

ii. $((F_{d1}[AA_d],\ COUNT)\pi[NGA_d,\ GA_d^+,\ AA_d]G_{d2}) \times S_u(G_{d2})$

In E_1, each row of G_{d1} and G_{d2} joins with $S_u(G_{d1})$ and $S_u(G_{d2})$ respectively and all the resulting rows are also merged by the second Similarity Group-by. Due to (1), the aggregation values in the resulting row of the following expressions in E_1 and E_2 respectively are the same.

iii. $F_d[AA_d]\pi_A[GA_d,GA_u,AA_d]\ ((G_{d1} \times S_u(G_{d1}))\ U_A\ (G_{d2} \times S_u(G_{d2})))$

iv. $F_{d2}[FAA_d]\pi_A[GA_d,GA_u,FAA_d]$
$(((F_{d1}[AA_d]\pi_A[NGA_d,\ GA_d^+,\ AA_d]G_{d1}) \times S_u(G_{d1}))$
$U_A\ ((F_{d1}[AA_d]\pi_A[NGA_d,\ GA_d^+,\ AA_d]G_{d2}) \times S_u(G_{d2}))$

Due to (2), the aggregation values in the resulting row of the following expressions in E_1 and E_2, respectively, are the same.

v. $F_u[AA_u]\pi_A[GA_d,GA_u,AA_u]\ ((G_{d1} \times S_u(G_{d1}))\ U_A\ (G_{d2} \times S_u(G_{d2})))$

vi. $F_{ua}[AA_u,CNT]\pi_A[GA_d,GA_u,\ AA_u,\ CNT]$
$(((COUNT\ \pi_A[NGA_d,\ GA_d^+]G_{d1}) \times S_u(G_{d1}))$

U_A $((COUNT\ \pi_A[NGA_d, GA_d^+]G_{d2}) \times S_u(G_{d2}))$

Case 2: $G_{d1}[GA_d]\ !\sim\ G_{d2}[GA_d]$ or $S_u(G_{d1})[GA_u]\ !\sim\ S_u(G_{d2})[GA_u]$. In E_2, the results of the join operations represented by (i) and (ii) are not merged into the same similarity group by the second Similarity Group-by. In E_1, each row of G_{d1} and G_{d2} joins with $S_u(G_{d1})$ and $S_u(G_{d2})$, respectively, but the resulting rows are not merged by the second SGB. Due to (1), the aggregation values in the resulting row of the following expressions in E_1 and E_2, respectively, are the same.

vii. $F_d[AA_d]\pi_A[GA_d,GA_u,AA_d](G_{d1} \times S_u(G_{d1}))$

viii. $F_{d2}[FAA_d]\pi_A[GA_d,GA_u,FAA_d]$

$((F_{d1}[AA_d]\pi_A[NGA_d, GA_d^+, AA_d]G_{d1}) \times S_u(G_{d1}))$

Due to (2), the aggregation values in the resulting row of the following expressions in E_1 and E_2, respectively, are the same.

ix. $F_u[AA_u]\pi_A[GA_d,GA_u,AA_u]\ ((G_{d1} \times S_u(G_{d1}))$

x. $F_{ua}[AA_u,CNT]\pi_A[GA_d,GA_u, AA_u, CNT]$

$((COUNT\ \pi_A[NGA_d, GA_d^+]G_{d1}) \times S_u(G_{d1}))$ †

Similar to the case of regular Group-by, several other query transformation techniques can be derived from the main theorem. The way the main theorem is extended in the case of similarity grouping follows closely the way the equivalent theorem is extended in the case of Group-by [40], [41], [42].

The use of materialized views to answer aggregation queries [43], [44], [45] is another important optimization technique that can yield considerable query processing time improvements and can be extended to the case of similarity grouping. Goldstein et al. propose a view matching algorithm [43] that determines if a query can be answered from existing materialized views with aggregation functions *sum* and *count*. Similarity aggregation queries and views should be treated as a SPJ query followed by a similarity aggregation operation. The

requirements that a view must satisfy to be used to answer a SPJG query with similarity-based aggregations are a slight variation of the requirements for queries with regular aggregation. These requirements are:

1. The SPJ component of the view contains all rows needed by the SPJ component of the query with the same duplication factor.

2. All columns required by compensating predicates are part of the view output.

3. The view does not contain aggregations or is less aggregated than the query, i.e., the query output can be computed by further aggregating the view output.

4. In case further aggregation is required, all the columns needed are available in the view output.

5. All the columns required to compute the query aggregation expressions are part of the view output.

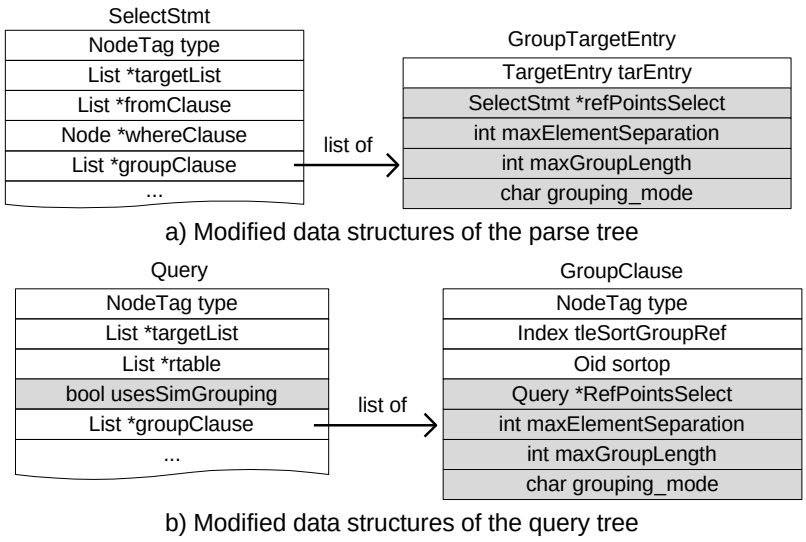
Steps 1, 2, 4, and 5 can be enforced similar to the case of regular aggregation queries. To satisfy Step 3, the algorithm has to consider that a query with regular Group-by on attributes *GA*, can be computed from a view with regular Group-by on a superset of *GA*; a query with Similarity Group-by on attributes *GA*, can be computed from a view with regular Group-by on a superset of *GA*; and a query with Similarity Group-by on attributes *GA*, can be computed from a view with Similarity Group-by on a superset of *GA*. For instance, a view grouped on attributes *A* on *Seg1*, *B* on *Seg2*, *C, D* can be used to compute the results of queries grouped on (1) *A* on *Seg1*; (2) *A* on *Seg1, C*; (3) *C, D*; or (4) *C* on *Seg3*.

2.3. Implementing Similarity Group-by

This section presents the guidelines to implement the similarity grouping operators introduced in Section 2.1 inside the query engine of standard

Relational Database Management Systems (RDBMSs). Although the presentation is intended to be applicable to any RDBMS, some specific details refer to our implementation in PostgreSQL. The SGB operators can be implemented as different database operators or they can be combined with the regular Group-by operator given that there are no conflicts in their syntax. We use the latter approach as it reduces the required changes in the query engine and facilitates the integration of SGB with other query processing mechanisms, e.g., generation of query trees, optimization tasks, etc.

To add support for similarity grouping in the parser, the raw-parsing grammar rules, e.g., the *yacc* rules in the case of PostgreSQL, are extended to recognize the syntax of the different new grouping approaches. This stage also identifies the grouping strategy, i.e., *regular*, *similarityAround*, *similarityDelimitedBy*, or *similarityUnsupervized*, being used with each grouping attribute. The parse-tree and query-tree data structures are extended to include the information related to similarity grouping as shown in Figure 2-7. The routines in charge of transforming the parse tree into the query tree are updated to process the new fields of the parse tree. The transformation of the parse tree section that represents the query of the reference points can be easily performed calling recursively the same function that is used to parse regular select statements, e.g., *do_parse_analyze* in PostgreSQL.



a) Modified data structures of the parse tree

b) Modified data structures of the query tree

Figure 2-7 Modifications in the Main Query Processing Data Structures (PostgreSQL)

2.3.1. The Optimizer

Traditionally, the aggregation nodes of execution plans have only one input plan tree, i.e., a data input plan tree, which represents the query that generates the data to be grouped. To support supervised similarity grouping, the aggregation nodes make use of a second input plan tree to receive the reference points data. Given that in many query engine implementations all the plan tree nodes inherit from a generic plan node that supports two input plan trees; aggregation nodes can make use of a second input plan tree without major changes to the plan tree's data structures. Figure 2-8.a presents the structure of the plan trees when one SGA is used. A sort node that orders by the grouping attribute is added on top of the data input plan tree, and in the case of supervised grouping, another sort node is added on top of the reference-points input plan tree. This order is assumed by the routines that form the similarity groups. When multiple SGAs are used, they are processed one at the time. Figure 2-8.b gives the structure of the plan trees generated when two SGAs *a1* and *a2* are used. The bottom aggregation node applies similarity grouping on *a1* and regular aggregation on *a2*. The result of this node is further aggregated by the top aggregation node that

applies similarity grouping on *a2* and regular aggregation on *a1*. This approach can be extended directly to support any number of attributes.

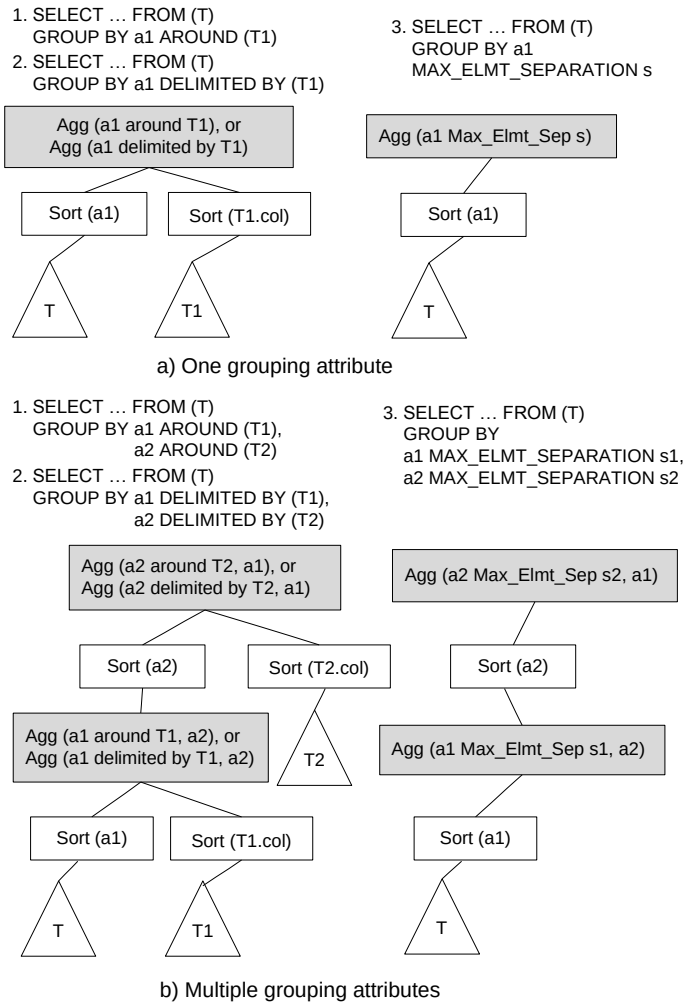


Figure 2-8 Path/Plan Trees for Similarity Grouping

A similarity-based group can combine tuples that have different values of the grouping attribute. Thus, the value of a grouping attribute *A* in an output tuple *T* is a representative of the values of this attribute in the tuples that form *T*. In our implementation, the central point of a group is selected as the representative value when SGB-A is used, the smaller delimiting point when SGB-D is used, and the average of the minimum and maximum values of *A* in the tuples that form *T* when SGB-U is used. Each aggregation node is able to process one SGA

and any number of regular grouping attributes. The group formation routines are presented in Section 2.3.2. Some additional modifications have to be implemented to ensure the correct calculation of the aggregation functions when the aggregation operation is divided into several aggregation nodes. For aggregation functions *F* for which *F(SetA U SetB)* cannot be computed from *F(SetA)* and *F(SetB)*, e.g., *Avg*, the bottom aggregation nodes calculate intermediate information, e.g., *Sum* and *Count*, instead of directly computing the values of the aggregation function *F*. The top aggregation node processes the intermediate information and computes the correct final results. For the aggregation function *Count* for which *Count*(*SetA U SetB*) is not equal to *Count*(*Count*(*SetA*),*Count*(*SetB*)) but equivalent to *Sum*(*Count*(*SetA*), *Count*(*SetB*)), the bottom aggregation node uses the function *Count* while the upper nodes aggregate the intermediate result using *Sum*. Another important change in the optimizer is in the way the number of groups generated by a similarity aggregation operation is estimated. This key estimation is used to compare different query execution paths and is commonly based on the number of groups each grouping attribute would generate if used alone (NA). In regular grouping, NA is the number of different values of a grouping attribute and appropriate statistics are maintained to estimate it. In the case of supervised similarity grouping, NA should be estimated as the number of tuples of the reference points query. In the case of unsupervised similarity grouping, NA can be estimated as the number of different values of the grouping attribute divided by a constant. The estimated number of groups (ENG) can be used to reduce the cost of queries with several similarity aggregation attributes. Given that the order of processing these attributes does not change the final result, they can be arranged to reduce the number of tuples that flow to upper nodes.

2.3.2. The Executor

When several SGAs are used, the constructed query plan uses several aggregation nodes where the result of each aggregation node is pipelined to the next one. The hash-based executor routines that form the groups in each

aggregation node are expected to be able to handle one SGA and zero or more regular grouping attributes. The tuples received from the input plans of the data and reference points have been previously sorted by sort nodes added in the plan construction stage as explained in Section 2.3.1. The executor routines process the input tuples sequentially and form the similarity groups following a plane sweep approach. A vertical line is swept across the sorted data tuples from left to right. At any time, a set of current groups is maintained and each time the line reaches a tuple the system evaluates whether this tuple belongs to the current groups, does not belong to any group, or starts a new set of groups. The main execution routine is modified to call appropriate subroutines that handle the different grouping strategies. In the regular implementation of PostgreSQL, this routine calls the subroutines *agg_fill_hash_table* and *agg_retrieve_hash_table*. The first routine forms the groups using a hash table, and the second retrieves the resulting tuples, one tuple at the time. In the case of similarity grouping, the main routine calls extensions of these two routines that form and retrieve the similarity groups. The rest of this section describes the extensions of these subroutines for the case of SGB-A.

To simplify the presentation we do not distinguish between a tuple and its value, this should be clear from the context. If the value is being used, it corresponds to the value of the SGA of this node, or the attribute representing the central points. In *agg_fill_hash_table_around*, both, the tuples to be grouped and the central points are processed sequentially. At any point, the routine maintains the current and next central points and it processes the data tuples to form the group(s) around the current central point. The sequence of values of the grouping attribute that satisfies the conditions MAXIMUM_GROUP_ DIAMETER and MAXIMUM_ELEMENT_SEPARATION is called a chain. When the distance of at least one of the values of the chain to the central point is smaller than MAXIMUM_ELEMENT_SEPARATION we say that the chain is *connected*. Tuples that belong to a chain are considered candidates to form similarity groups. The hash table entries corresponding to these potential groups are

marked *active*. If the routine finds that the current chain is connected then it changes the status of the entries to *final*. If there is no element that connects the chain to the central element, the entries are marked *inactive*. Tuples that do not belong to any group under the current SGA are also assigned to hash table entries. These entries are marked as *outlier*. Outlier entries are maintained to allow the correct group formation in subsequent similarity grouping nodes when several SGAs are used. This ensures that the final result of a Similarity Group-by query is not affected by the order in which its SGAs are processed. Outlier entries are not considered to calculate the results of aggregation functions since the final groups are composed only by tuples that belong to some group under each SGA. Additionally, the tuple structure is extended with a status field that is used to determine if a tuple is an outlier or not. For each data tuple *T*, the routine performs a test to check if the distance from *T* to the current central point *C* is smaller than the value of the parameter MAXIMUM_GROUP_DIAMETER/2 (i.e., the radius) and that *T* is closer to the current central point than to the next one. If the test fails and *T* is located to the left of *C*, *T* is an outlier. Consequently, the value of the SGA of this tuple is replaced by a constant and this modified tuple is inserted in the hash table marking the associated entry as *outlier*. If the test fails and *T* is located to the right of *C*, the routine finishes processing the current groups, starts the formation of the groups around the next central point, and processes *T* with the new central point. If the test succeeds and *T* has not been marked *outlier* previously, T is processed with the current central point. All the possible arrangements of the previous and current data tuples and current and next central points are considered and appropriate actions taken in each case. For instance, if (1) the distance between the previous and current tuples is greater than MAXIMUM_ELEMENT_ SEPARATION, (2) the current tuple is connected to the current central point, and (3) the current chain (without considering the current tuple) is not connected; the current groups are dismissed, i.e., marked *inactive*, a new chain is started having the current tuple *T* as its first element, and if *T* is not an outlier, the aggregation calculations of the associated

group are updated with the values of *T*. The process of advancing a tuple, i.e., updating the aggregation calculations of the associated group with the values of the tuple, uses a similarity version of the tuple replacing the grouping attribute value with the value of the current central point. The *agg_retrieve_hash_table_around* routine is a variation of *agg_retrieve_hash_table*. It returns the entries marked *final* when called from the last SGA of a SGB query. Otherwise, it returns the entries marked *final* or *outlier*.

The changes in the executor required to support the other similarity grouping strategies can be implemented using similar guidelines. The cost of group formation in SGB nodes is very close to the one of the regular Group-by since each tuple is processed once and in almost constant time. The additional cost of the SGB operators is due to the additional comparison operations and hash table status maintenance. Although we focus on the hash-based approach, some of the basic mechanisms employed by this approach to control the extent of the groups can be used by a simpler sort-based approach to answer single-GA similarity aggregation queries.

2.4. Performance Evaluation

We implemented the proposed SGB operators inside the PostgreSQL 8.2.4 query engine. This section presents the results of the performance study of these operators. The main cost considered is the query execution time.

2.4.1. Test Configuration

The dataset used in the performance evaluation is based on the one specified by the TPC-H benchmark [51]. The tables, additional attributes, and queries used in the tests are presented in Figures 2-9 and 2-10. The default dataset scale factor (SF) is 1, i.e., the dataset size is about 1GB. All the experiments are performed on an Intel Dual Core 1.83GHz machine with 2GB RAM running Linux as operating system. We use the default values for all PostgreSQL configuration parameters. The results presented in this section consider the average of the

warm performance numbers having 95% confidence and an error margin less than ±5%.

2.4.2. Performance Evaluation

The focus of the performance evaluation is to study the scalability and overhead of the Similarity Group-by operators and compare them with the ones of the regular Group-by.

TPC-H Tables
Part(P), Supplier(S), PartSupp(PS), Customer(C), Orders(O), LineItem(L), Nation(N)
C.c_acctbal_xb: Similar to C_acctbal but without values in SF*50 segments of length 1.1 around the points of RefPoints_1b
C.c_acctbal_x: Similar to C_acctbal
C.c_segment_x: Integer. Random [0,19]. Represents ways to segment clients
O.o_clerkType: Integer. Random [1,50]. Represents a way to segment clerks
Reference Points Tables
RefPoints_all: All values used by C_acctbal
RefPoints_1b: 50*SF-1 points that partition C_acctbal's domain in 50*SF segments of equal length. For SF=1: {-780,560,...,9780}
RefPoints_x: 50*SF points that correspond to the center of the segments of RefPoints_1b. For SF=1: {-890,-670, ...,9890}
RefRevLevels: 10 order revenue levels. {20000,60000,…,380000}
MktCmpRefDates: Marketing campaign dates. Random in the range of O_orderdate.
RefDiscLevel: 5 discount levels. {0.010, 0.030, ..., 0.090}

Figure 2-9 Performance Evaluation Dataset

Queries used in Section 2.4.2.1	
GB	SELECT c_acctbal count(c_acctbal), min(c_acctbal), max(c_acctbal), sum(c_acctbal), avg(c_acctbal) FROM C GROUP BY c_acctbal
GB(SGB)	<GB> AROUND <RefPoints_all>
SGB-A	<GB> AROUND <RefPoints_1>
SGB(GB)	SELECT count(R2.A), min(R2.A),max(R2.A),sum(R2.A), avg(R2.A) FROM (SELECT c_acctbal as A, min(abs(c_acctbal - refpoint)) as B FROM C, RefPoints_1 GROUP BY C.c_acctbal) as R1, (SELECT c_acctbal as A, refpoint as C, abs(c_acctbal - refpoint) as B FROM C, RefPoints_1) as R2 WHERE R1.A=R2.A and R1.B=R2.B GROUP BY R2.C
SGB-A_MR	SGB-A + 'MAXIMUM_GROUP_DIAMETER *2r*'. *r* =11000/(100*SF)
SGB-A_MS	SGB-A + MAXIMUM_ELEMENT_SEPARATION 1
SGB-D	<GB> DELIMITED BY <RefPoints_1b>
SGB-U_MR	<GB> MAXIMUM_GROUP_DIAMETER *d*. *d* =11000/(50*SF)
SGB-U_MS	SGB-U_MR using 'MAXIMUM_ELEMENT_SEPARATION 1' instead of 'MAXIMUM_GROUP_DIAMETER *d*'
Queries used in Section 2.4.2.2. n=number of similarity grouping attributes (SGAs)	
GB	SELECT sum(c_acctbal_1), …, sum(c_acctbal_n), c_acctbal_1, …, c_acctbal_n FROM C GROUP BY c_acctbal_1,…, c_acctbal_n
SGB	SELECT sum(c_acctbal_1), …, sum(c_acctbal_n), c_acctbal_1, …, c_acctbal_n FROM C GROUP BY c_acctbal_1 AROUND <RefPoints_1> … c_acctbal_n AROUND <RefPoints_n>
SGB_MR	SGB +'MAXIMUM_GROUP_DIAMETER 220' in each SGA
SGB_MS	SGB +'MAXIMUM_ELEMENT_SEPARATION 1' in each SGA
<Query>+5	<Query> + 'c_acctbal_1b, …, c_segment_5' in the GROUP BY clause
Queries used in Section 2.4.2.3	
Business question: Study the discount level (DL) given by each type of clerk	
Lazy1	SELECT L.l_discount as DcntLevel, O.o_clerkType, sum(L.l_discount) FROM L, O WHERE L.l_orderkey=O.o_orderkey GROUP BY O.o_clerkType, L.l_discount AROUND <RefDiscLevel>
Eager1	SELECT R1.l_discount as DcntLevel, O.o_clerkType, sum(R1.CNT) FROM O, (SELECT L.l_discount, L.l_orderkey, count(L.l_discount) as CNT FROM L GROUP BY L.l_orderkey, L.l_discount AROUND <RefDiscLevel>) AS R1 WHERE R1.l_orderkey=O.o_orderkey GROUP BY R1.l_discount, O.o_clerkType
Business question: Study the DL given by each type of clerk in the past six months	
Lazy2 (Eager2)	Lazy1 (Eager1) + 'AND O.o_orderdate between '1994-06-17' and '1995-06-17' ' in the WHERE clause
Business question: Retrieve the unshipped orders with the highest value	
GB1	Same as TPC-H Q3
Business question: Clusters the unshipped orders around revenue levels of interest	
SGB1	SELECT revenue as RevLevel, count(revenue), min(revenue), max(revenue), avg (revenue) FROM (SELECT l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue FROM C, O, L WHERE c_mktsegment = 'BUILDING' and c_custkey = o_custkey and l_orderkey = o_orderkey and o_orderdate < date '1995-03-15' and l_shipdate > date '1995-03-15' GROUP BY l_orderkey) as R1 GROUP BY revenue AROUND <RefRevLevels>
Business question: Report profit on a given line of parts (by supplier nation and year)	
GB2	Same as TPC-H Q9
Business question: Report profit of a line of parts during marketing campaigns	
SGB2	SELECT nation, o_orderdate as MktCmpRefDate, sum(amount) as sum_profit FROM (SELECT n_name as nation, o_orderdate, l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount FROM P, S, L, PS, O, N WHERE s_suppkey = l_suppkey and ps_suppkey = l_suppkey and ps_partkey = l_partkey and p_partkey = l_partkey and o_orderkey = l_orderkey and s_nationkey = n_nationkey and p_name like '%green%') as profit GROUP BY nation, o_orderdate AROUND <MktCmpRefDates> MAXIMUM_GROUP_DIAMETER interval '14 day' ORDER BY nation
Business question: Retrieve large volume customers	
GB3	Same as TPC-H Q18
Business question: Retrieve clusters of customers with similar buying power	
SGB3	SELECT TotalBuy as TotalBuyLevelRef, min(TotalBuy), max(TotalBuy), count(TotalBuy), avg(TotalBuy) FROM (SELECT c_name,c_custkey,sum(l_extendedprice) as TotalBuy FROM C, O, L WHERE c_custkey = o_custkey and o_orderkey = l_orderkey and o_orderkey IN (SELECT l_orderkey FROM L GROUP BY l_orderkey HAVING sum(l_quantity) > 300) GROUP BY c_name,c_custkey) GROUP BY TotalBuy MAXIMUM_GROUP_DIAMETER 200000 MAXIMUM_ELEMENT_SEPARATION 20000

Figure 2-10 Performance Evaluation Queries

2.4.2.1. Increasing Dataset Size

Figure 2-11 gives the execution time of several aggregation queries for different dataset sizes. The number of tuples in table Customer is 15,000*SF while the number of tuples in the reference points tables is 50*SF. The key result of this experiment is that the execution times of all the queries that use Similarity Group-by, i.e., *SGB-X*, are very close to the execution time of the regular aggregation query *GB* for all the dataset sizes. Even in the worst case scenario represented by *GB(SGB)_X*, i.e., *SGB* query produces the same result as *GB*, the execution time of *GB(SGB)* is at most only 25% bigger than the one of *GB*. The optimizer selected the sort-based approach to execute *GB*. *GB(SGB)_H* and *GB(SGB)_S* use the hash-based and sort-based similarity grouping approaches respectively. The SGB parameters and the data used in this test have been selected such that all the SGB queries generate approximately the same result. *SGB-A_H* and *SGB-A_S* are queries that use Group-by-around without additional clauses. They are executed using the hash-based and sort-based approaches respectively. The execution time of *SGB-A_H* is about 12% bigger than that of *GB* while the execution time of *SGB-A_S* is about 2% bigger than that of *GB*. The execution time of *SGB-A_S* is about 9% smaller than the one of *SGB-A_H* because the hash-based approach makes use of an additional sort node. Given that the hash-based approach supports queries with multiple similarity grouping attributes (SGAs), the execution time of the other SGB queries consider this approach. The execution time of *SGB-A_MD* and *SGB-A_MS*, variants of *SGB-A* that use parameters MAXIMUM_GROUP_DIAMETER and MAXIMUM_ELEMENT_ SEPARATION respectively, are around 2% and 6% bigger than the one of the simple *SGB-A* query. This is due to the extra calculations that need to be performed to ensure that the produced groups comply with the specified parameters, and the overhead of keeping track of the status of hash table entries. As expected, the Group-by-delimited-by query *SGB-D* performs almost exactly as *SGB-A*, and the queries with unsupervised similarity grouping, i.e., *SGB-U_MD* and *SGB-U_MS*, perform similarly to *SGB-A_MD* and *SGB-A_MS*

respectively. In all the cases the difference is less than 2%. In the following experiments we use Group-by-around as a representative of the Similarity-Group-by queries.

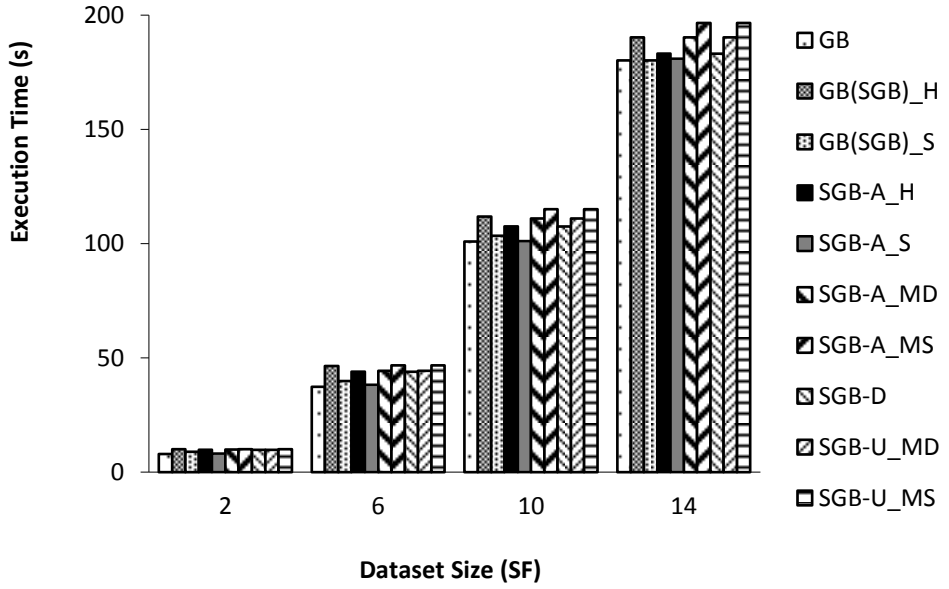


Figure 2-11 Performance while Increasing Dataset Size

Although in general it is not possible to produce the output of SGB queries using only regular SQL operations, this is feasible in the following special cases: (1) SGB-A without conditions (assuming there are no points whose distance to the closest two central points are the same) can be obtained using a complex mix of aggregations and joins as presented in query *SGB(GB)* of Figure 2-10; SGB-A with MAXIMUM_GROUP_ DIAMETER can be implemented using further selection predicates; and (2) SGB-D can be obtained using a complex query similar to *SGB(GB)*. Figure 2-12 compares the execution time of *SGB(GB)* with that of *SGB-A*. The presented results show that the execution time and scalability properties of the query that uses Similarity Group-by is much better than those of the query that uses only regular SQL operations. The execution time of *SGB(GB)*

grows from being 500% bigger than that of *SGB-A* for SF=1 to being 1300% bigger for SF=14.

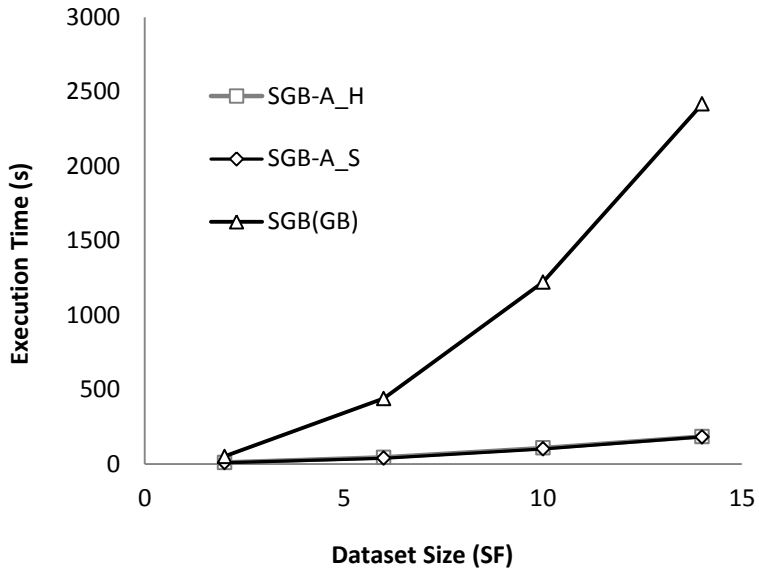


Figure 2-12 Performance of Generating Similarity Groups with Group-by Vs. Similarity Group-by

2.4.2.2. Increasing the Number of SGAs

Figure 2-13 gives the execution time of SGB queries when the number of SGAs increases. As in the previous test, all the SGB queries generate similar results. The query *GB* is included as a reference. The optimizer selected sort-based grouping to execute this query. Even though the implementation to support multiple SGAs makes use of one aggregation node per similarity grouping attribute, the execution times of all the SGB queries, i.e., *SGB*, *SGB_MD*, and *SGB_MS*, scale well when the number of SGAs increases. Furthermore, the way they scale is similar to the one the regular aggregation query *GB* scales. Each query *QRY*+5 represents the query *QRY* with five additional regular grouping attributes. In all the cases, these extra attributes have a very small effect (1% to 5% of additional cost) on the execution time of similarity aggregation queries

because they are handled using the same hash tables used in the similarity-based aggregation nodes.

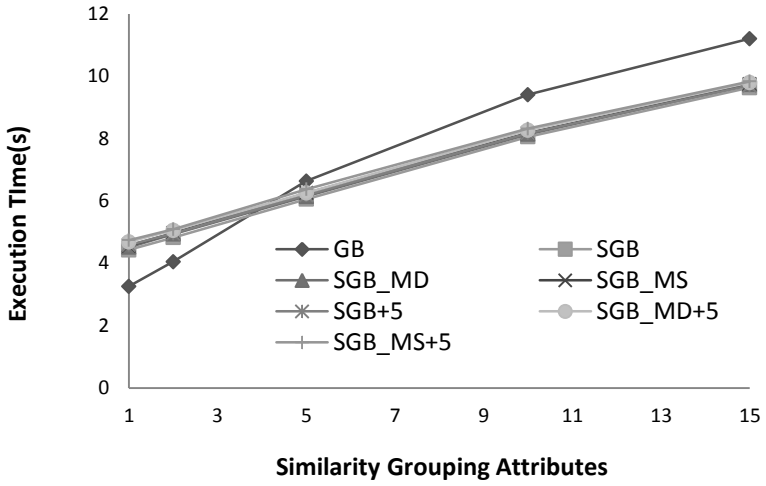


Figure 2-13 Performance while Increasing Number of SGAs

2.4.2.3. Complex Queries

Figure 2-14 gives the execution time of several real world similarity aggregation queries and presents scenarios in which the Eager and Lazy query transformation techniques presented in Section 2.2 are used. Figure 2-10 gives the details of the queries used in this section and the business question they help to answer. The similarity-based queries used in this experiment are a small representative set of the queries that can be built using the introduced similarity operators to answer real world business questions. *Lazy1* and *Eager1* are equivalent queries that obtain information about discount levels given by the different clerk types. The discount values are grouped around a set of discount levels of interest. *Lazy1* performs first the join and after that the similarity grouping while *Eager1* preaggregates all the discount values in table *Lineitem* that correspond to the same order, joins the result with table *Orders*, and finally aggregates all the orders that belong to the same clerk type. The execution time of *Eager1* is 13% smaller than that of *Lazy1*. The reason is that the similarity-

based preaggregation step reduces significantly the number of tuples to be processed by the join operator. *Lazy2* and *Eager2* are also equivalent queries, and are similar to *Lazy1* and *Eager1*, respectively, but only consider the orders made in the past six months. In this case, the execution time of *Lazy2* is 40% smaller than that of *Eager2*. In this case the join is significantly more selective and reduces in *Lazy2* the number of tuples to be processed by the similarity aggregation operator. *SGB1*, *SGB2*, and *SGB3* are three variants of the TPC-H queries *Q3* (*GB1*), *Q9* (*GB2*), and *Q18* (*GB3*) respectively. They all provide richer information and are potentially more useful for the decision maker than their regular aggregation counterparts. For instance, *GB2* reports the profits on a given line of parts while *SGB2* reports how those profits change during marketing campaigns; *GB3* retrieves large volume customers while *SGB3* clusters those costumers in groups of similar buying power. In all cases, the similarity aggregation queries have a comparable execution time to the ones of their regular aggregation counterparts.

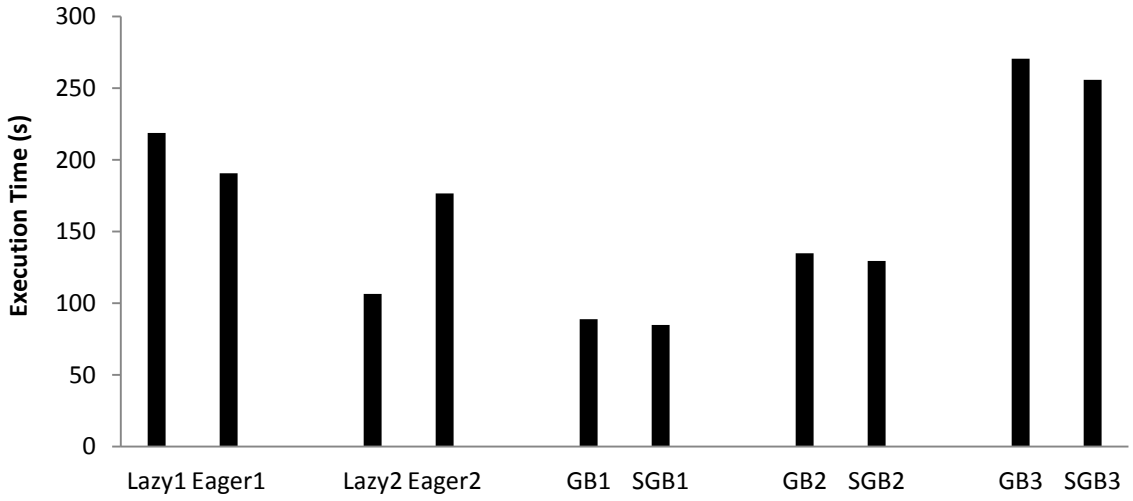


Figure 2-14 Performance of Complex Queries

CHAPTER 3 THE SIMILARITY JOIN DATABASE OPERATOR

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 chapter, 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, Ɛ-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 Ɛ-Join's execution time is 20 to 90% of the one of its equivalent regular operators based query for the useful case of small Ɛ (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.

3.1. Similarity Join Operators

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

$$A \bowtie_{\theta_S} B = \{\langle a, b\rangle \mid \theta_S(a, b), a \in A, b \in B\}$$

where θ_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.

1. Range Distance Join (Ɛ-Join):

   $\theta_\varepsilon \equiv dist(a, b) \leq \varepsilon$

2. kNN-Join:

   $\theta_{kNN} \equiv b\ is\ a\ k - closest\ neighbor\ of\ a$

3. k-Distance-Join (kD-Join):

   $\theta_{kD} \equiv \langle a, b\rangle\ is\ one\ of\ the\ overall\ k - closest\ pairs$

4. Join-Around (A-Join):

   $\theta_{A,MD=2r} \equiv b\ is\ the\ closest\ neighbor\ of\ a\ and\ dist(a, b) \leq r$

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 3.2, the Join-Around operator with *MD*=∞ 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.

Figure 3-1 shows an extension of SQL syntax to express the different types of Similarity Join predicates. Figure 3-2 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. Figure 3-3 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 conventional TPC-H queries [51]. The similarity queries in Figure 3-3 show that the use of Similarity Joins allows answering more complex and interesting business questions.

ε-Join: SELECT … FROM A, B WHERE A.a **WITHIN *ε* OF** B.b
Around-Join: SELECT … FROM A, B WHERE A.a **AROUND** B.b **[MAX_DIAMETER *2r*]**
kNN-Join: SELECT ... FROM A, B WHERE B.b ***k* NEAREST_NEIGHBOR_OF** A.a
kD-Join: SELECT ... FROM A, B WHERE A.a ***k* TOP_CLOSEST_PAIRS** B.b

Figure 3-1 Extended SQL Syntax for Similarity Join Predicates

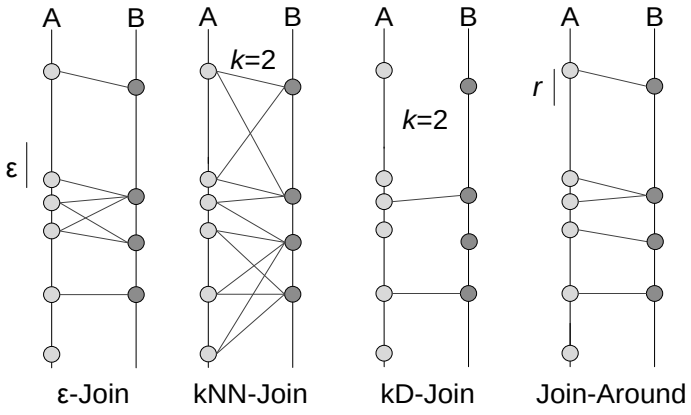


Figure 3-2 Types of Similarity Join

Similarity Query Example 1
Original TPC-H Query
Q4 – Business Question: Study how well the order priority system is working in a given quarter
Similarity-aware Query
Business Question: Study how well the order priority system works around dates of interest (holydays, marketing campaigns, etc.)
Select **d_refdate**, o_orderpriority, count(*) as order_count from orders, **DatesOfInterest** Where **o_orderdate AROUND d_refdate** and exists (Select * from lineitem Where l_orderkey = o_orderkey and l_commitdate < l_receiptdate) group by o_orderpriority, **d_refdate** order by o_orderpriority, **d_refdate**
Similarity Query Example 2
Original TPC-H Query
Q5 – Business Question: Study the revenue volume done between suppliers and customers of the same country
Similarity-aware Query
Business Question: Study the revenue volume done between local (nearby) suppliers and customers (Revenue of “short distance”orders)
Select n_name, sum(l_extendedprice * (1 - l_discount)) as revenue From customer, orders, lineitem, supplier, **nationSupp NS**, **nationCust NC**, region Where c_custkey = o_custkey and l_orderkey = o_orderkey and l_suppkey = s_suppkey **and c_location WITHIN Ɛ TO s_location** and c_nationkey = **NC.n_nationkey** and s_nationkey = **NS.n_nationkey** and NC.n_regionkey = NS.n_regionkey and NC.n_regionkey = r_regionkey and r_name = '[REGION]' and o_orderdate >= date '[DATE]' and o_orderdate<date '[DATE]'+interval '1' year group by n_name order by revenue desc
Similarity Query Example 3
Original TPC-H Query
Q6 – Business Question: Forecast revenue change that would have resulted from eliminating certain discounts in a given year
Similarity-aware Query
Business Question: Forecast revenue change that would have resulted from eliminating certain discounts on certain date ranges of interest (holydays, marketing campaigns, etc.)
Select **d_refdate**, sum(l_extendedprice*l_discount) as revenue From lineitem, **DatesOfInterest** Where **l_shipdate AROUND d_refdate MAX_SIZE 'D' day** and l_discount between [DISCOUNT] - 0.01 and [DISCOUNT] + 0.01 and l_quantity < [QUANTITY] Group by **d_refdate**;
Similarity Query Example 4
Original TPC-H Query
Q18 – Business Question: Find large volume(quantity) customers. Large volume orders are the ones with a total quantity greater than a given level.
Similarity-aware Query
Business Question: Classify customers based on their buying power
Select **c_name, c_custkey, r_refRevlevel** From (Select c_name, c_custkey, sum(l_extendedprice) as TotalBuy From customer, orders, lineitem Where o_orderkey in (Select l_orderkey From lineitem Group by l_orderkey Having sum(l_quantity) > [QUANTITY]) and c_custkey = o_custkey and o_orderkey = l_orderkey Group by c_name, c_custkey), **RevenueLevelsOfInterest** Where **TotalBuy AROUND r_refRevlevel** Order by r_refRevlevel

Figure 3-3 Examples of the Use of Similarity Join

3.2. 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: (1) core equivalence rules that exploit specific properties of SJs, (2) equivalence rules between multiple SJ operators and between SJ and Similarity Group-by (SGB) operators, and (3) 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.

3.2.1. 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 3.2.2 use the notation presented in Figure 3-4. The examples assume the following relations' content: E_1=E_2=E_3={1,2,...,100}, and E_4={21,22,...,25}.

E_i	a relation	
e_i	an attribute of E_i	
σ and ⋈	the selection and join operators respectively	
θ	a non similarity predicate	
θ_{ε}, θ_{kNN}, θ_{kD}, θ_{A}	the different similarity join predicates as defined in section III	
${}_{GA}\gamma_{F(AA)}(R)$	the aggregation operator	
	R	is the relation being aggregated
	AA	the aggregation attributes
	F	the aggregation functions
	GA	the grouping attributes. It can be a simple attribute in the case of regular grouping, or an expression like $E_1.e_1$ *around* $E_2.e_2$ in the case of Similarity Group Around (*SGB-A*), a type of similarity grouping that groups the tuples of E_1 around a set of central points (tuples of E_2) assigning every tuple of E_1 to the group of the central point with the minimum *dist* ($E_1.e_1$, $E_2.e_2$) [24]

Figure 3-4 Notation for Equivalence Rules

3.2.1.1. 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 θ involve only the attributes of one of the expressions being joined (E_1):

E1. $\sigma_\theta(E_1 \bowtie_{\theta_\varepsilon} E_2) \equiv (\sigma_\theta(E_1)) \bowtie_{\theta_\varepsilon} E_2$

E2. $\sigma_\theta(E_1 \bowtie_{\theta_{kNN}} E_2) \equiv (\sigma_\theta(E_1)) \bowtie_{\theta_{kNN}} E_2$

E3. $\sigma_\theta(E_1 \bowtie_{\theta_A} E_2) \equiv (\sigma_\theta(E_1)) \bowtie_{\theta_A} E_2$

When the selection predicates $\theta 1$ and $\theta 2$ involve only the attributes of E_1, and E_2, respectively:

E4. $\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_{\theta_\varepsilon} E_2) \equiv (\sigma_{\theta 1}(E_1)) \bowtie_{\theta_\varepsilon} (\sigma_{\theta 2}(E_2))$

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. Figure 3-5.a presents an example of Rule E1. 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.

3.2.1.2. Pushing Selection Predicate under Originally Unrelated Join Operand

In the equivalence rules presented in Section 3.2.1.1, each selection predicate θ is pushed only under the join operand that contains all the attributes referenced in θ. In the case of Ɛ-Join, the filtering benefits of pushing θ can be further improved by pushing it under both operands of the join as shown in Rule E5.

E5. $\sigma_\theta(E_1 \bowtie_{\theta_\varepsilon} E_2) \equiv (\sigma_\theta(E_1)) \bowtie_{\theta_\varepsilon} (\sigma_{\theta \pm \varepsilon}(E_2))$

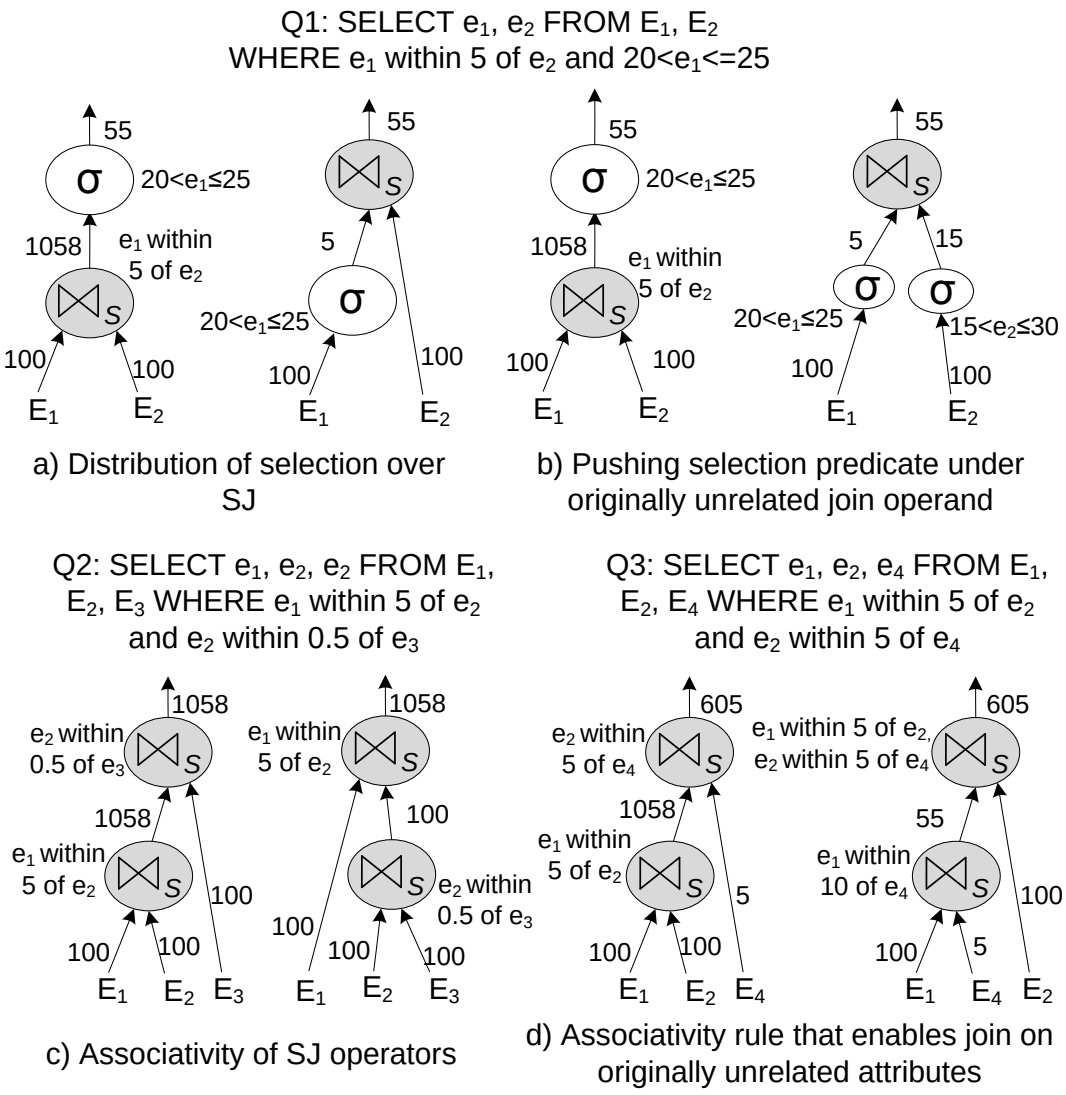


Figure 3-5 Extended SQL Syntax for Similarity Join Predicates

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

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 generated applying Rule E1. Figure 3-5.b presents an example of Rule E5

where 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 20 tuples.

3.2.1.3. Basic Associativity of SJ Operators

Similarity Join operators are associative using the following rules.

Rules with the same type of Similarity Join:

E6. $\left(E_1 \bowtie_{\theta_{\varepsilon 1}} E_2\right) \bowtie_{\theta_{\varepsilon 2} \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta_{\varepsilon 1} \wedge \theta} \left(E_2 \bowtie_{\theta_{\varepsilon 2}} E_3\right)$

E7. $\left(E_1 \bowtie_{\theta_{A1}} E_2\right) \bowtie_{\theta_{A2} \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta_{A1} \wedge \theta} \left(E_2 \bowtie_{\theta_{A2}} E_3\right)$

E8. $\left(E_1 \bowtie_{\theta_{kNN1}} E_2\right) \bowtie_{\theta_{kNN2} \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta_{kNN1} \wedge \theta} \left(E_2 \bowtie_{\theta_{kNN2}} E_3\right)$

Rules that combine different types of similarity and regular join:

E9. $\left(E_1 \bowtie_{\theta_{A1}} E_2\right) \bowtie_{\theta_{kNN2} \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta_{A1} \wedge \theta} \left(E_2 \bowtie_{\theta_{kNN2}} E_3\right)$

E10. $\left(E_1 \bowtie_{\theta_{kNN1}} E_2\right) \bowtie_{\theta_{A2} \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta_{kNN1} \wedge \theta} \left(E_2 \bowtie_{\theta_{A2}} E_3\right)$

E11. $\left(E_1 \bowtie_{\theta_{\varepsilon 1}} E_2\right) \bowtie_{\theta 2 \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta_{\varepsilon 1} \wedge \theta} \left(E_2 \bowtie_{\theta 2} E_3\right)$

E12. $\left(E_1 \bowtie_{\theta 1} E_2\right) \bowtie_{\theta_{A2} \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta 1 \wedge \theta} \left(E_2 \bowtie_{\theta_{A2}} E_3\right)$

E13. $\left(E_1 \bowtie_{\theta 1} E_2\right) \bowtie_{\theta_{kNN2} \wedge \theta} E_3 \equiv E_1 \bowtie_{\theta 1 \wedge \theta} \left(E_2 \bowtie_{\theta_{kNN2}} E_3\right)$

where $\theta 1$, $\theta_{\varepsilon 1}$, θ_{A1}, and θ_{kNN1} involve attributes from only E_1 and E_2; $\theta 2$, $\theta_{\varepsilon 2}$, θ_{A2}, and θ_{kNN2} involve attributes from only E_2 and E_3.

Usage: Given an expression with several SJ operations, 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. Figure 3-5.c presents an example of Rule E6. The LHS expression computes first the less 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.

3.2.1.4. Associativity Rule that Enables Join on Originally Unrelated Attributes

In the equivalence rules presented in Section 3.2.1.3, each join predicate involves the same attributes in both sides of the rule. In the case of Ɛ-Join, when the attributes e_1 of E_1 and e_2 of E_2 are joined using $\mathcal{E}1$ and the result joined with attribute e_3 of E_3 using $\mathcal{E}2$, there is an implicit relationship between e_1 and e_3 that is exploited by the following equivalence rule.

E14. $\left(E_1 \bowtie_{e_1\ \theta_{\varepsilon 1}\ e_2} E_2\right) \bowtie_{e_2\ \theta_{\varepsilon 2}\ e_3} E_3 \equiv \left(E_1 \bowtie_{e_1\ \theta_{\varepsilon 1+\varepsilon 2}\ e_3} E_3\right) \bowtie_{(e_1\ \theta_{\varepsilon 1}\ e_2)\wedge(e_2\ \theta_{\varepsilon 2}\ e_3)} E_2$

Notice that this rule is expressed using an extended notation that specifies explicitly the attributes being joined.

Usage: The RHS expression of this rule produces a bottom join that joins attributes that are not joined in the LHS expression. The transformation from the LHS expression to the RHS one has the potential to generate a lower cost plan when the RHS' bottom join outputs a low number of tuples. Figure 3-5.d presents an example of Rule E14. The LHS expression processes a total of 200 tuples in the first SJ and 1063 tuples in the second one. The LHS expression processes 105 tuples in the first SJ and 155 tuples in the second one. Notice that the top RHS' SJ has a slightly more complex SJ predicate.

3.2.1.5. Commutativity of SJ Operators

Some similarity Join operations are commutative:

E15. $E_1 \bowtie_{\theta_\varepsilon} E_2 \equiv E_2 \bowtie_{\theta_\varepsilon} E_1$

E16. $E_1 \bowtie_{\theta_{kD}} E_2 \equiv E_2 \bowtie_{\theta_{kD}} E_1$

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.

3.2.2. Equivalence among Similarity Operators

The Join-Around and the Similarity Group Around (SGB-A) operators are equivalent in the following way:

E17. $_{e_1\ around\ E_2.e_2}\gamma_{F(AA)}(E_1) \equiv {}_{e_2}\gamma_{F(AA)}(E_1 \bowtie_{e_1\ \theta_A\ e_2} E_2)$

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.

The following rules describe the special cases in which different Similarity Join operators are equivalent.

E18. $E_1 \bowtie_{\theta_{A,MD=\infty}} E_2 \equiv E_1 \bowtie_{\theta_{kNN(k=1)}} E_2$

E19. $E_1 \bowtie_{\theta_{A,MD=2\varepsilon}} E_2 \equiv E_1 \bowtie_{\theta_\varepsilon} E_2,$

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

E20. $E_1 \bowtie_{\theta_{kD}} E_2 \equiv E_1 \bowtie_{\theta_\varepsilon} E_2,$

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

3.2.3. 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 [40] 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 was extended in section 2.2 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 3-7, 3-8, 3-9, and 3-10 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 Figure 3-6. This notation is used because: (1) it allows a direct comparison with analogous theorems for regular operators [40] that use a similar notation, and (2) 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 (1) regular join and Group-by [40], and (2)

regular join and Similarity Group-by are presented next. These theorems are referenced in the new extensions of the theorem studied later in this section.

$g[GA]R$	regular grouping of relation R on grouping attributes GA
$g[GA; Seg]R$	similarity grouping of relation R on grouping attributes GA using segmentations Seg. The domain of the n^{th} element of GA is partitioned by the n^{th} element of Seg
$F[AA]R$	aggregation operation of a previously grouped table R
F and AA	sets of aggregation functions and columns, respectively
σ, π_D, π_A, U_A $\bowtie$ and $\widetilde{\bowtie}$	selection, projection with and without duplicate elimination, set union without duplicate elimination, theta-join, and similarity join respectively
R_d	a table that always contains aggregation attributes
R_u	a table that may or may not contain aggregation attributes
GA_d and GA_u	the grouping columns of R_d and R_u, respectively
AA	all the aggregation columns
AA_d and AA_u	the subsets of AA that belong to R_d and R_u, respectively
C_d and C_u	the conjunctive predicates on columns of R_d and R_u, respectively
C_0	the conjunctive predicates involving columns in both R_u and R_d
$\alpha(C_0)$	the columns involved in C_0
GA_d^+	$= GA_d \cup \alpha(C_0) - R_d$, columns that participate in join and grouping
F	the set of all aggregation functions
F_d and F_u	the members of F applied on AA_d and AA_u, respectively
FAA	the resulting columns of the application of F on AA in the first grouping operation of the eager strategy
Seg	the set of segmentation of the attributes in GA
Seg_d and Seg_u	the subsets of Seg for the attributes in GA_d and GA_u, respectively
NGA_d	a set of columns in R_d
CNT	the column with the result of Count(*) in the first aggregation operation of the eager approach
FAA_d	the set of columns, other than CNT, produced in the first aggregation operation of the eager approach
F_{ua}	the duplicated aggregation function of Fu, e.g., if Fu=(SUM, MAX), then Fua=(SUM, MAX, count) = (SUM*count, MAX)

Figure 3-6 Algebraic Notation for Eager and Lazy Transformation Theorems

Theorem 3-1 Eager/Lazy Aggregation Main Theorem for Group-by and Join. The following two expressions:

$$E_1: F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]$$

$$g[GA_d, GA_u]\sigma[C_d \wedge C_u](R_d \bowtie_{C0} R_u)$$

E_2: $\pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u,CNT], F_{d2}[FAA_d])$

$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]$

$g\ [GA_d, GA_u]\sigma[C_u]$

$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]$

$g\ [NGA_d]\sigma[C_d]R_d) \bowtie_{C0} R_u)$

are equivalent if (1) F_d can be decomposed into F_{d1} and F_{d2}, (2) F_u contains only class *C* or *D* aggregation functions [40], (3) $NGA_d \rightarrow GA_d^+$ holds in $\sigma[C_d]R_d$, and (4) $\alpha(C_0) \cap GA_d = \emptyset$.

Expression E_1 represents the Lazy approach while expression E_2 represents the Eager approach.

Theorem 3-2 Eager/Lazy Aggregation Main Theorem for Similarity Group-by and Join. The following expressions:

E_1: $F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]$

$g\ [GA_d, GA_u; Seg]\sigma[C_d \wedge C_u]\ (R_d \bowtie_{C0} R_u)$

E_2: $\pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u,CNT], F_{d2}[FAA_d])$

$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]$

$g\ [GA_d, GA_u; Seg_u]\sigma[C_u]$

$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]$

$g\ [NGA_d; Seg_d]\sigma[C_d]R_d) \bowtie_{C0} R_u)$

are equivalent under the same conditions as the ones of Theorem 3-1.

3.2.3.1. 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 3-3.

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

E_1: $F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]$

$g\,[GA_d, GA_u]\sigma[C_d \wedge C_u]\,(R_d \tilde{\bowtie}_{C0} R_u)$

E_2: $\pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u, CNT], F_{d2}[FAA_d])$

$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]$

$g\,[GA_d, GA_u]\sigma[C_u]$

$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]$

$g\,[NGA_d]\sigma[C_d]R_d)\ \tilde{\bowtie}_{C0} R_u)$

where $\tilde{\bowtie}_{C0}$ is kNN-Join, Ɛ-Join, or A-Join; are equivalent under the same conditions as the ones of Theorem 3-1.

Usage: Figure 3-7 illustrates an example of the application of this theorem. The SJ of the Lazy aggregation expression processes a total of 7 tuples while the grouping node processes 5 tuples. In the Eager aggregation expression all the tuples of *T1* get combined into one tuple in the bottom grouping node and the SJ and top grouping operators only need to process 3 and 1 tuples respectively. In scenarios where *T1* has a significant number of tuples with the same value of (*G1*, *J1*) the optimizer will probably favor the Eager approach; otherwise the Lazy approach will probably be selected.

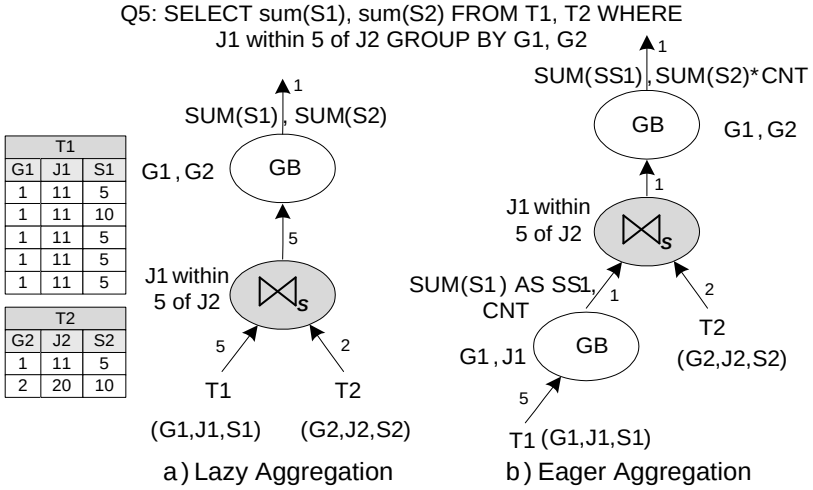


Figure 3-7 Eager/Lazy Transformation with GB and SJ

Proof sketch of Theorem 3-3

The validity of this theorem relies on the following properties.

P1. Given R_d' and R_u' instances of R_d and R_u respectively, the result of $(R_d' \tilde{\bowtie}_{C0} R_u')$ is equivalent to the result of $(R_d' \bowtie_\theta R_u')$ where θ = disjunction of $(R_d.C0_d=x \wedge R_u.C0_u=y)$ for every different link (x,y) of the result of $(R_d' \tilde{\bowtie}_{C0} R_u')$.

P2. θ, as defined in P1, remains unchanged and valid when R_d' is augmented with tuples that have already present values of $R_d'.C0_d$, i.e., duplicates, or when such tuples are removed from R_d'.

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

For every $R_{d'}$ and $R_{u'}$ instances of R_d and R_u, respectively,

1. E_1: $F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]$
   $g\,[GA_d, GA_u]\sigma[C_d \wedge C_u]\,(R_{d'} \tilde{\bowtie}_{C0} R_{u'})$

is equivalent to

$E_{1'}$: $F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]$
$g\,[GA_d, GA_u]\sigma[C_d \wedge C_u]\,(R_{d'} \bowtie_\theta R_{u'})$,

where θ is defined as in P1.

2. $E_{1'}$: $F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]\; g\,[GA_d, GA_u]\sigma[C_d \wedge C_u]\,(R_{d'} \bowtie_\theta R_{u'})$

is equivalent to

$E_{2'}$: $\pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u,CNT], F_{d2}[FAA_d])$
$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]$
$g\,[GA_d, GA_u]\sigma[C_u]$
$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]\; g\,[NGA_d]\sigma[C_d]R_{d'}) \bowtie_\theta R_{u'})$

because of Theorem 3-1.

3. $E_{2'}$: $\pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u,CNT], F_{d2}[FAA_d])$

$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]$

$g\,[GA_d, GA_u]\sigma[C_u]$

$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]$

$g\,[NGA_d]\sigma[C_d]R_{d'}) \bowtie_\theta R_{u'})$

is equivalent to

E_2: $\pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u,CNT], F_{d2}[FAA_d])$

$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]$

$g\,[GA_d, GA_u]\sigma[C_u]$

$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]$

$g\,[NGA_d]\sigma[C_d]R_{d'}) \widetilde{\bowtie}_{C0} R_{u'})$

since the grouping operation before the join merges only tuples that share the same value of $R_{d'}.C0_d$, and P2.

3.2.3.2. 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 3-4.

Theorem 3-4 Eager/Lazy Aggregation Main Theorem for Similarity Group-by and Similarity Join. The following two expressions:

E_1: $F[AA_d, AA_u]\pi_A[GA_d, GA_u, AA_d, AA_u]$

$g\,[GA_d, GA_u; Seg]\sigma[C_d \wedge C_0 \wedge C_u]\,(R_d \widetilde{\bowtie}_{C0} R_u)$

E_2: $\pi_D[GA_d, GA_u, FAA](F_{ua}[AA_u,CNT], F_{d2}[FAA_d])$

$\pi_A[GA_d, GA_u, AA_u, FAA_d, CNT]\; g\,[GA_d, GA_u; Seg_u]\sigma[C_0 \wedge C_u]$

$(((F_{d1}[AA_d], COUNT)\pi_A[NGA_d, GA_d^+, AA_d]$

$g\,[NGA_d; Seg_d]\sigma[C_d]R_d) \widetilde{\bowtie}_{C0} R_u)$

where $\tilde{\bowtie}_{C0}$ is kNN-Join, Ɛ-Join, or A-Join; are equivalent under the same conditions as the ones of Theorem 3-1.

Usage: An example of the use of this theorem is presented in Figure 3-8. 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: (1) the same value of J1 and (2) values of G2 that belong to the same similarity group. In the example all the tuples of T1 are merged even though they have different values of G1.

Proof sketch of Theorem 3-4

The validity of this theorem relies on the validity of Theorem 3-2 and Theorem 3-3.

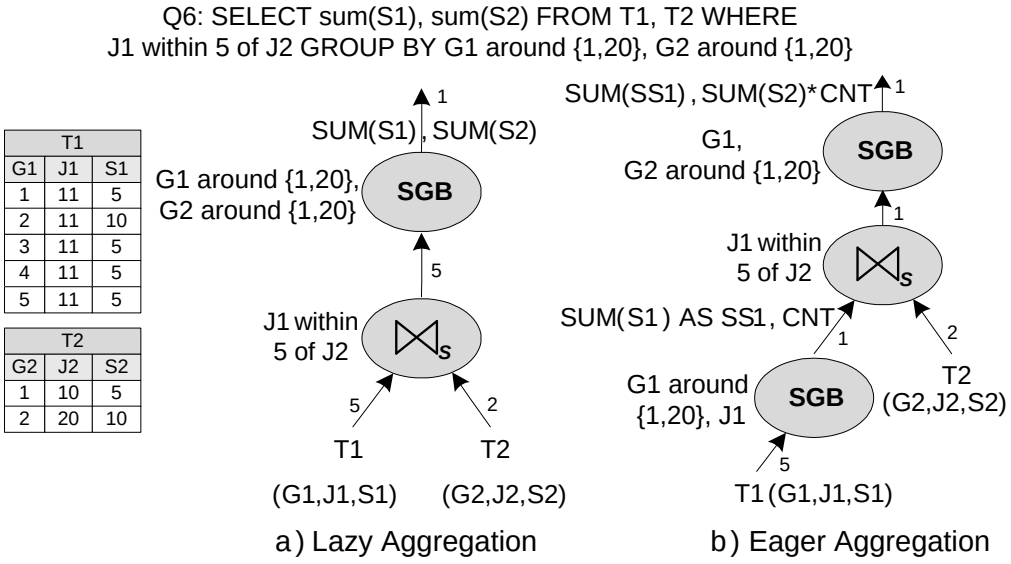


Figure 3-8 Eager/Lazy transformation with SGB and SJ

3.2.3.3. Pushing Similarity Predicate from Ɛ-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 3.2.3.1.

The similarity predicate of the Ɛ-Join can be (partially) pushed down to a grouping operator as shown in Figure 3-9. The bottom aggregation of the Eager approach performs regular aggregation on *G1* and similarity aggregation SGB-A' on *J1* around *J2* with MAX_GROUP_DIAMETER = *2Ɛ*. SGB-A' is a variation of similarity group around (SGB-A) that only merges tuples that are linked to only one central point (*J2*) by the Ɛ-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 Ɛ-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 Figure 3-9, 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 3.2.3.1 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: (1) if two tuples $t1_a$ and $t1_b$ are grouped by the bottom aggregation of the Eager approach around a center point tuple, say *t2*, then $t1_a$ and $t1_b$ will always be matched with *t2* by the Ɛ-Join of the Lazy approach; and (2) 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.

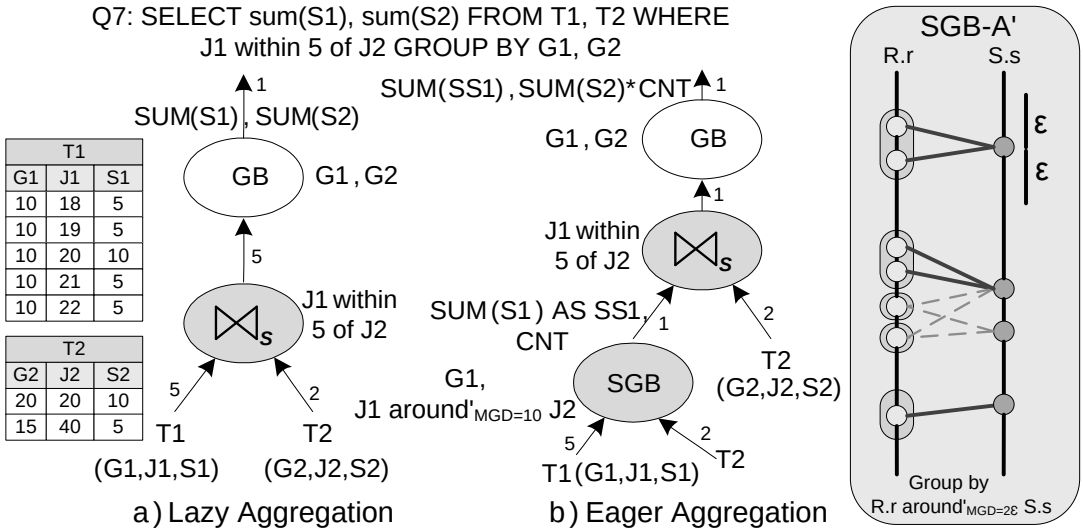


Figure 3-9 Pushing Similarity Predicate from Ɛ-Join to GB

3.2.3.4. 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 Figure 3-10. The bottom aggregation of the Eager approach performs regular aggregation on *G1* and similarity aggregation SGB-A on *J1* around *J2* with MAX_GROUP_DIAMETER = *2Ɛ*. 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 Figure 3-10, 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 Figure 3-10, 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 3.2.3.1 would produce five tuples instead.

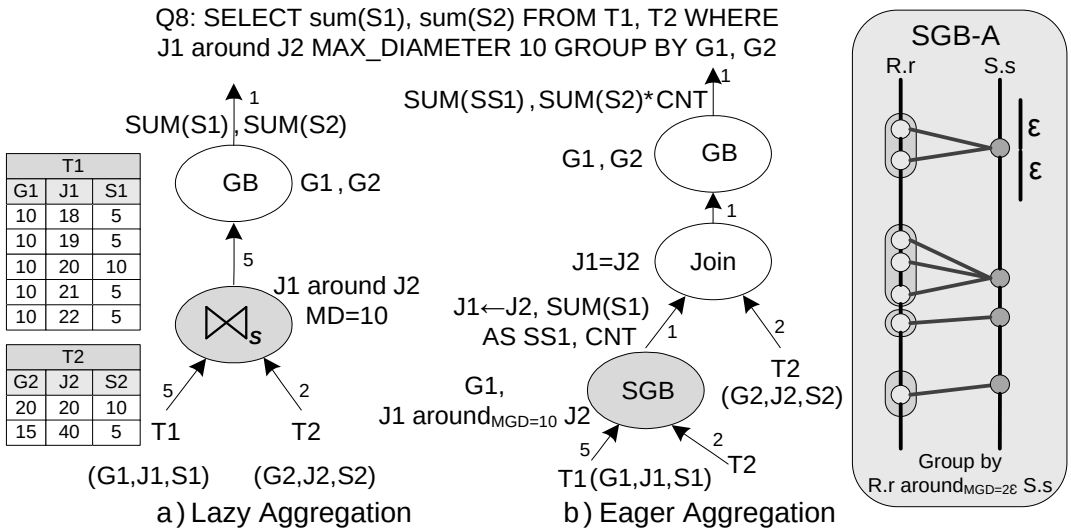


Figure 3-10 Pushing Similarity Predicate from Join-Around to GB

The validity of this equivalence relies on the following properties: (1) if two tuples $t1_a$ and $t1_b$ are grouped by the bottom aggregation of the Eager approach around a center point tuple *t2*, $t1_a$ and $t1_b$ are always matched with *t2* by the Join-Around of the Lazy approach; and (2) if two tuples $t1_a$ and $t1_b$ share the same value of *G1* and are linked to tuple *t2* in the Lazy approach, then $t1_a$ and $t1_b$ will always be grouped by the bottom aggregation of the Eager approach.

3.3. Implementing Similarity Join

This section presents the guidelines to implement two Similarity Join operators, Ɛ-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 Ɛ-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 3.1. The parse-tree and query-tree data structures are extended to include the type and parameters, e.g., $\mathcal{E}$, *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.

3.3.1. The Optimizer

Figure 3-11.a 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. Figure 3-11.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 routines that find the similarity matches are presented in Section 3.3.2. 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 Ɛ-Join, more complex techniques, e.g., employing histograms of the density of elements in metric space [49], can be employed. The number of output tuples of the kNN-Join can be estimated as *(# of tuples of outer input)*min(k, # of tuples of inner input)* while the one of the kD-Join can be estimated as *min(# of tuples of outer input * # 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.

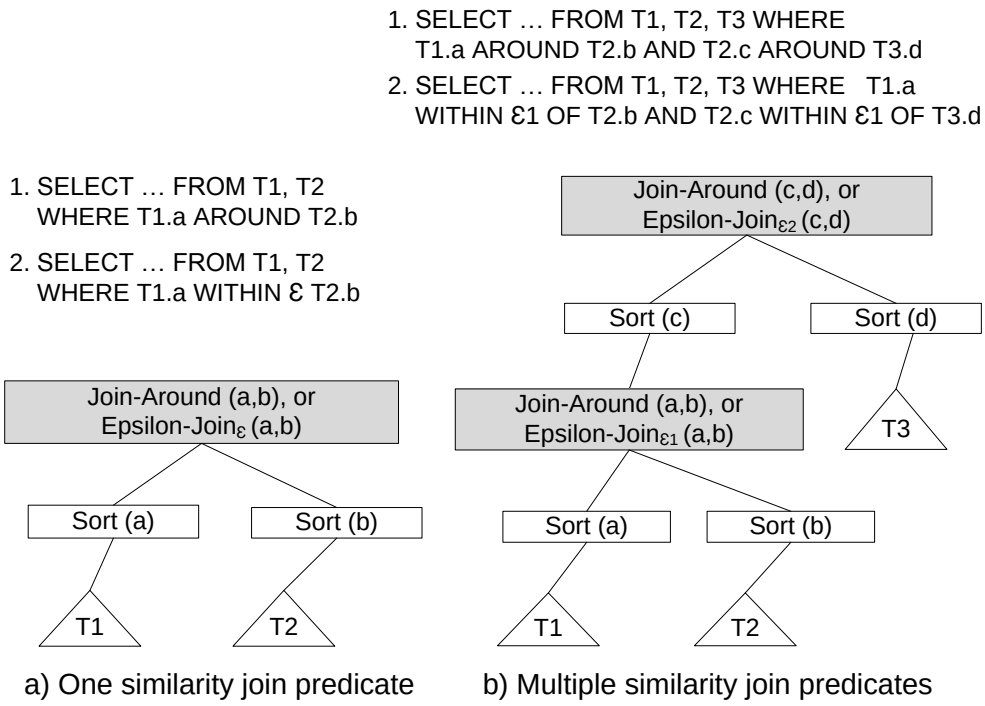


Figure 3-11 Path/Plan Trees for Join-Around and Ɛ-Join

3.3.2. 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 3.3.1. 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 3.3.1. The executor routines process the input tuples synchronously following a plane sweep approach.

Figure 3-12 presents the algorithms of the main operation of the regular Sort Merge Join (3-12.a), Join-Around (3-12.b), and Ɛ-Join (3-12.c). The sections that were modified to support the SJ operators are shown in bold. It is clear from Figure 3-12 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 SMJ algorithm in Figure 3-12.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 Ɛ-Join are presented in Figure 3-12.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 $distance(outer,inner) \leq \varepsilon$. 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 3-12.c and 3-13. 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 Figure 3-12.c, and Lines 2 and 6 in Figure 3-13 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 Figure 3-12.c and Line 4 in Figure 3-13) is also extended. In this case, the similarity predicate *outer~inner* is evaluated as *distance(outer, inner)* < *distance (outer,nextInner)*. The function that evaluates if an inner tuple matches another inner tuple (used in lines 4 and 18 in Figure 3-12.c and in lines 1 and 3 in Figure 3-13) evaluates the regular equality operator on the join attribute values. The expression *outer>inner* in line 1 of Figure 3-13 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 Figure 3-13. 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 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. Figure 3-12.d shows the states associated to the different tasks. The implementation of Ɛ-Join and Join-Around use the same set of states employed by SMJ.

a. Sorted Merge Join	b. Epsilon-Join	c. Join-Around	d. State

SMJoin {
1 get initial outer tuple
2 get initial Inner tuple
3 do forever {
4 while (outer != inner) {
5 if (outer < inner)
6 advance outer
7 else
8 advance inner
9 }
10 mark inner position
11
12 do forever {
13 do{
14 join outer and inner
15
16 advance inner position
17 }
18 while (outer == inner)
19 advance outer position
20 if (outer == mark)
21 restore inner to mark
22
23 else
24 break
25 }
26 }
27 }
	EpsilonJoin {
	1 get initial outer tuple
	2 get initial inner tuple
	3 do forever {
	4 while (outer **!~** inner) {
	5 if (outer < inner)
	6 advance outer
	7 else
	8 advance inner
	9 }
	10 mark inner position
	11
	12 do forever {
	13 do{
	14 join outer and inner
	15 **prevInner ← inner**
	16 advance inner position
	17 }
	18 while (outer **~** inner)
	19 advance outer position
	20 if (outer **~ prevInner)**
	21 restore inner to mark
	22 **break**
	23 else
	24 break
	25 }
	26 }
	27 }
		JoinAround {
		1 get initial outer tuple
		2 get initial inner and **nextInner**
		3 do forever {
		4 while ((**inner != nextInner)&&**
		5 (outer **!~** inner)) {
		6
		7
		8 advance inner and **nextInner**
		9 }
		10 mark inner position
		11 ***if (!check_match()) continue***
		12 do forever {
		13 do{
		14 join outer and inner
		15 **prevInner ← inner**
		16 advance inner **and nextInner**
		17 }
		18 while (**prevInner ==** inner)
		19 advance outer position
		20 if (outer **~ prevInner)**
		21 restore inner to mark
		22 **nextInner ← getNext(inner)**
		23 else
		24 break
		25 }
		26 }
		27 }
			INITIALIZE
			INITIALIZE
			
			SKIP_TEST
			
			SKIPOUTER_ADVANCE
			
			SKIPINNER_ADVANCE
			
			SKIP_TEST
			SKIP_TEST
			
			JOINTUPLES
			NEXTINNER
			NEXTINNER
			
			NEXTOUTER
			TESTOUTER
			TESTOUTER
			TESTOUTER

Figure 3-12 Main Operation of Epsilon-Join and Join-Around Compared to the one of Sorted Merge Join

```
    check_match() {
 1  if ((inner == nextInner) && (outer>inner)){
 2      do {advance inner and nextInner}
 3      while(inner == nextInner)
 4      if (outer ~ inner)
 5         restore inner to mark
 6         nextInner ← getNext(inner)
 7         return True //similarity match
 8      else return False
 9  }
10  return True //no need to advance to check match
11  }
```

Figure 3-13 Routine check_match

The cost of the proposed SJ operators is close to the one of SMJ for reasonably small Ɛ (for Ɛ-Join) and inner datasets without many duplicates (for Join-Around) because: (1) every outer tuple is read once in sequential order; (2) 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; (3) in Ɛ-Join, if the inner cursor is restored, the length of the jump, i.e., distance from previous inner to marked tuple, is at most *2Ɛ*; and (4) 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.

3.4. Performance Evaluation

We implemented the Ɛ-Join and Join-Around, as described in Section 3.3 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.

3.4.1. Test Configuration

The dataset used in the performance evaluation is based on the one specified by the TPC-H benchmark [51]. The Reference points tables and queries used in the tests are presented in Figure 3-14. 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.

Reference Points Table	
AccBalLevels1(R1): 110 account balance values in the range of C_acctbal [0,11000]	
AccBalLevels2(R2): 11000 account balance values in the range of C_acctbal [0,11000]	
Queries	
SJ-JoinAround	SELECT c_custkey, C_acctbal, refpoint FROM CUSTOMER, AccBalLevels1 WHERE C_acctbal **AROUND** refpoint;
RegOps-JoinAround	SELECT T1.c_custkey, T1.C_acctbal, T2.refpoint FROM (SELECT c_custkey, C_acctbal, min(dist) as mindist FROM (SELECT c_custkey, C_acctbal, refpoint, abs(C_acctbal - refpoint) as dist FROM CUSTOMER, AccBalLevels1) AS C1 GROUP BY c_custkey, C_acctbal) AS T1, AccBalLevels1 T2 WHERE R1.mindist = abs(T1.C_acctbal - T2.refpoint);
SJ-EpsJoin	SELECT * FROM CUSTOMER, AccBalLevels1 WHERE C_acctbal **WITHIN Ɛ OF** refpoint;
RegOps-EpsJoin	SELECT * FROM CUSTOMER, AccBalLevels1 WHERE abs(C_acctbal - refpoint) <= Ɛ;
AssocRule	SELECT * FROM CUSTOMER, AccBalLevels1 R1 , AccBalLevels2 R2 WHERE C_acctbal **WITHIN 11 OF** R1.refpoint AND R1.refpoint **WITHIN 11 OF** R2.refpoint;
PushSel	SELECT * FROM CUSTOMER, AccBalLevels2 WHERE C_acctbal **WITHIN 11 OF** refpoint AND 2200<C_acctbal AND C_acctbal<=6600
Lazy-Eager [N]	SELECT refpoint, sum(C_acctbal) FROM CUSTOMER, AccBalLevels[N] WHERE C_acctbal **WITHIN 11 OF** refpoint GROUP BY refpoint

Figure 3-14 Reference Points Table and Queries Used in Performance Evaluation

3.4.2. 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., [17], [18], [19], [20], [21], 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.

3.4.2.1. Join-Around Performance while Increasing Dataset Size

Figure 3-15 gives the execution time of the *SJ-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. 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.

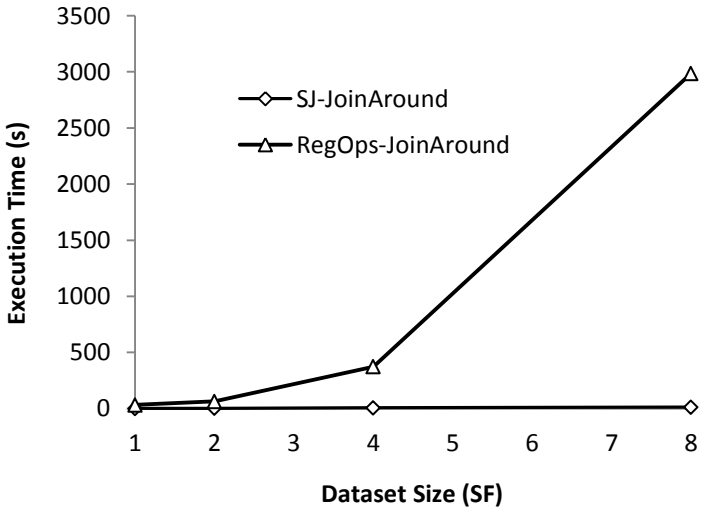


Figure 3-15 Performance of Join-Around

3.4.2.2. Ɛ-Join Performance while Increasing Ɛ

Figure 3-16 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 ε. The value of ε 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 $\mathcal{E}$. For instance, when $\mathcal{E}$=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 Ɛ-Join over the regular query gets reduced as the value of $\mathcal{E}$ increases and is almost negligible when the size of $\mathcal{E}$ is about 20% of the domain range. Having a good performance for small values of $\mathcal{E}$ is of key importance for the Ɛ-Join operator since Similarity Join queries with small $\mathcal{E}$ are among the most common and useful types of similarity-based operations. The performance of *SJ-EpsJoin* is better for small values of $\mathcal{E}$ 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 $\mathcal{E}$. The additional cost for high values of $\mathcal{E}$ is due to the increase in the number of links to be reported.

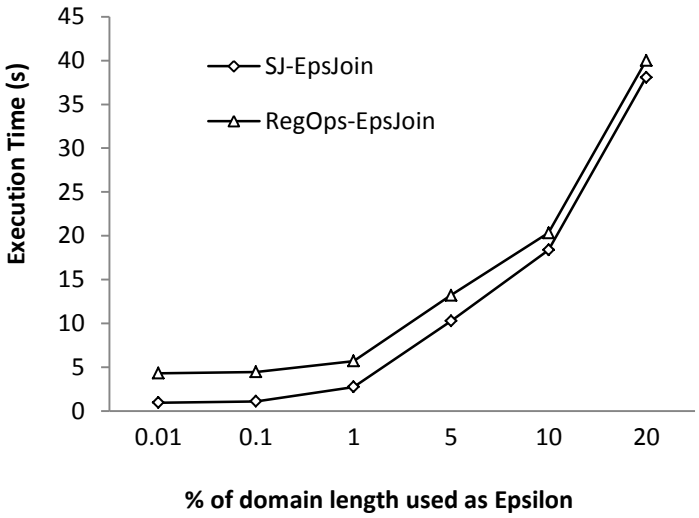


Figure 3-16 Performance of Ɛ-Join

3.4.2.3. Effectiveness of Associativity Transformation

AssocRule_ LHS and *AssocRule_RHS* in Figure 3-17 represent the query *AssocRule* executed using plans that corresponds to the LHS and RHS of the

Rule E6 respectively. The execution time of *AssocRule_RHS* is 9.2% of that of *AssocRule_LHS*. *AssocRule_LHS* joins (Ɛ-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.

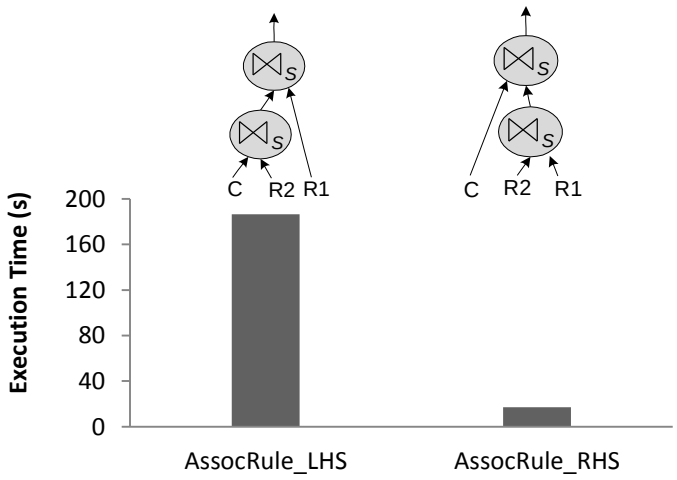


Figure 3-17 Effectiveness of Associativity Transformation

3.4.2.4. Effectiveness of Pushing Selection under SJ

PushSel_LHS, *PushSel_RHS1*, and *PushSel_RHS2* in Figure 3-18 represent the query *PushSel* executed using plans that corresponds to the LHS and RHS of Rule E1, and the RHS of Rule E5 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 furher 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*.

3.4.2.5. Effectiveness of Lazy and Eager Aggregation Transformations

In Figure 3-19, *LazyN* and *EagerN* represent the query *LazyEager* executed using plans that corresponds to the expressions E_1 and E_2 of Theorem 3-3 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. *Eager2* has an execution time that is only 9% of that of *Lazy2*.

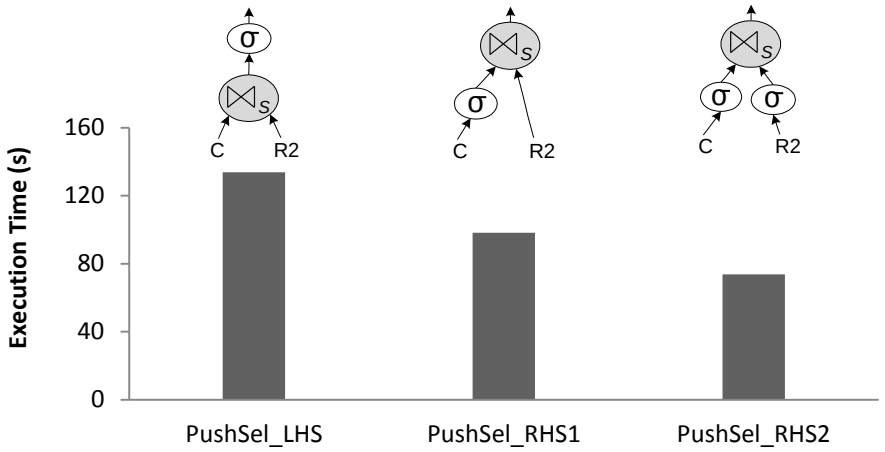


Figure 3-18 Effectiveness of Pushing Selection under SJ

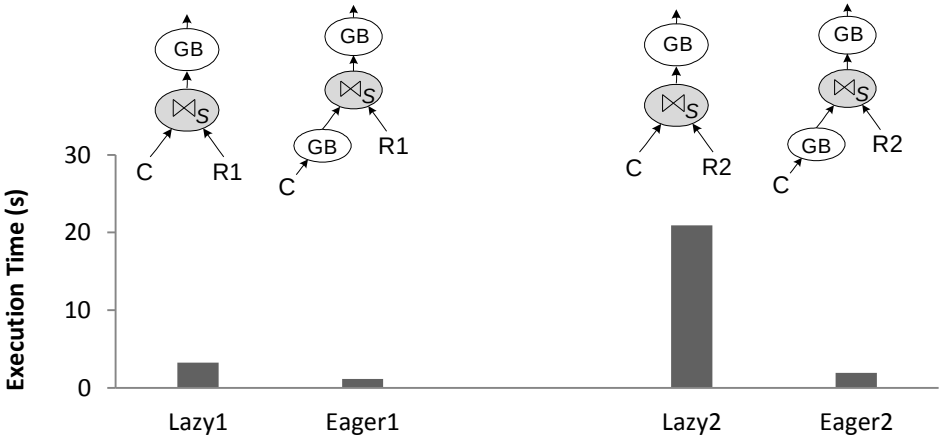


Figure 3-19 Effectiveness of Lazy and Eager Aggregation Transformations

CHAPTER 4 CONCEPTUAL EVALUATION OF SIMILARITY QUERIES AND SIMILARITY QUERY TRANSFORMATIONS

4.1. Supported Similarity-aware Operators

This section specifies the similarity-aware operations considered in this chapter. The supported operations are:

1. Similarity Group-by (SGB)
    - Unsupervised SGB (U-SGB)
    - Similarity Group Around (SGB-A)
    - SGB with Delimiters (SGB-D)
2. Similarity Join (SJ)
    - Range Distance Join (Ɛ-Join, Eps-Join)
    - kNN Join (kNN-Join)
    - kDistance Join (kD-Join)
    - Join Around (Join-Around)
3. Similarity Selection (SS)
    - Range Dist. Selection (Ɛ-Selection, Eps-Selection)
    - kNN Selection (kNN-Selection)

The Similarity Group-by and Similarity Join operators are defined in Chapters 2 and 3, respectively. The Similarity Selection operators can be seen as special

cases of the join operators where one of the input relations of the join consists of a single tuple. The Range Distance selection operator is a special case of the Range Distance join and the kNN selection operator is a special case of the kNN Join.

The generic definition of the Similarity Selection (SJ) operator is as follows.

$$\sigma_{\theta_S}(A) = \{a \mid \theta_S(a), a \in A\}$$

where θ_s represents the Similarity Selection predicate. This predicate specifies the similarity-based conditions that tuple *a* needs to satisfy to be in the Similarity Selection output. The Similarity Selection predicates for the Similarity Selection operators considered in our study are as follows. Let *C* be a constant value.

Range Distance Selection (Eps-Selection): $\theta_{\varepsilon,C} \equiv dist(a, C) \leq \varepsilon$

kNN-Selection: $\theta_{kNN,C} \equiv a\ is\ a\ k - closest\ neighbor\ of\ C$

For simplicity of this presentation, we require that all the relations involved in the k-based operations, i.e., kNN-Join, kDistance-Join and kNN-Selection, have a primary key. This requirement allows the correct computation of the results when the relations have duplicates or have been combined with other relations, and using only the values of the attributes involved in the operations' predicates (and the required keys). Figure 4-1 shows a scenario that highlights the need for primary keys to correctly compute a kNN-Join operation. This figure shows two sets of relations E_1 and E_2. The results of the kNN-Join between E_1 (outer) and E_2 (inner) are represented by the lines between the values of the joined relations. We want to be able to compute the kNN-Join even if we previously combine E_1 and E_2, e.g., using the cross product operation. However, both sets of relations generate the same cross product making it impossible to compute the kNN-Join without additional information. The use of primary keys in E_1 and E_2 solve the problem because these keys uniquely identify the tuples of the original joined tables even after they have been combined.

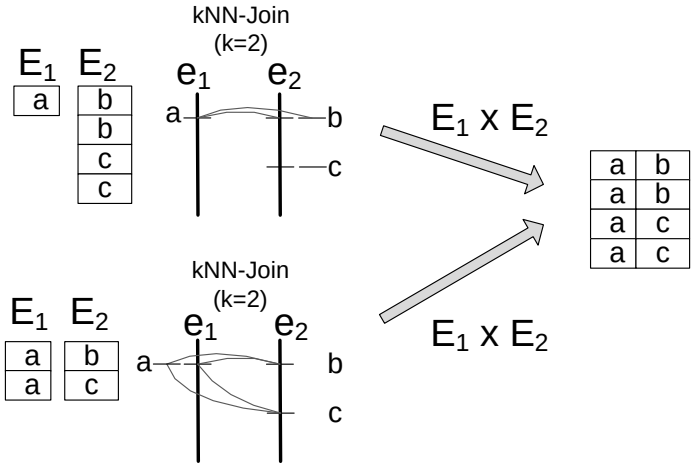


Figure 4-1 The Need of Primary keys for kNN-Join

4.2. Notation Used in Similarity-aware Expressions

Unless otherwise specified, the expressions in this chapter use the following notation conventions:

1. The default letter to represent a relation is *E*. The default attribute name of relation E_i is e_i. When expressions require multiple attributes of a relation E_i, we use a second component in the subscript, e.g., E_{i_1}, E_{i_2}, etc.

2. Similarity Join predicates are specified using the expression θ_{Sa_b}. The subscript *a* refers to the outer relation while *b* refers to the inner relation. The value of *S* determines the type of Similarity Join: ε represents Epsilon-Join, *kNN* represents kNN-Join, *A* represents Join-Around and *kD* represents kDistance-Join. For example, the predicate $\theta_{\varepsilon 1\_2}$ represents an Epsilon-Join operation between relations E_1 (outer) and E_2 (inner). Furthermore, by default, $\theta_{\varepsilon 1\_2}$ represents a join on the attributes e_1 (outer) and e_2 (inner). Regular, i.e., non similarity, join uses a similar notation without the component *S*.

3. Similarity Selection predicates are specified using the expression $\theta_{Sa,C}$. The subscript *a* refers to the input relation while *C* refers to the constant parameter. The value of *S* determines the type of Similarity Selection: ε represents Epsilon-Selection and *kNN* represents kNN-Selection. For example

the predicate $\theta_{\varepsilon 1,C1}$ represents an Epsilon-Selection operation on E_1 around $C1$. Regular selection uses a similar notation without the components S and C.

4. We say that the attributes of an expression have a single direction when the expression is composed by join predicates and their attribute graph is of the form $e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n$. The attribute graph is built as follows. The vertices of the graph are the join attributes and each join is represented as a directed edge from the outer attribute (left attribute of the join predicate) to the inner one (right attribute of the join predicate).

4.3. Conceptual Evaluation of Similarity-aware Queries

The conceptual evaluation order of queries specifies a clear and consistent way to evaluate queries and the expected correct results. In practice, database systems generate an initial plan for a given query and the query optimizer transforms this plan into an equivalent one trying to find a better way to execute the query. Having a conceptual order of evaluation of queries is important because it provides a clear and consistent way to specify a query, which will generate the same results independently of the database system implementation.

We present a conceptual evaluation order for similarity-aware queries with multiple similarity-aware operators. This evaluation order is particularly important because the order in which the similarity operations are evaluated affects the results of a query. For instance, consider the left hand side (LHS) plan of Figure 4-2 which shows a similarity query with two Similarity Selection predicates: an Epsilon-Selection predicate and a kNN-Selection predicate. Figure 4-2 shows two ways in which this query could be evaluated and the different results obtained under each evaluation. The middle plan in the figure corresponds to evaluating first the kNN-Selection predicate and applying the Epsilon-Selection over the output of the first operator. The right hand side (RHS) plan corresponds to evaluating first the Epsilon-Selection predicate and then the kNN-Selection. It

is not clear which way this query should be evaluated and without a clear conceptual evaluation order of similarity queries, multiple users may write the same query expecting different results.

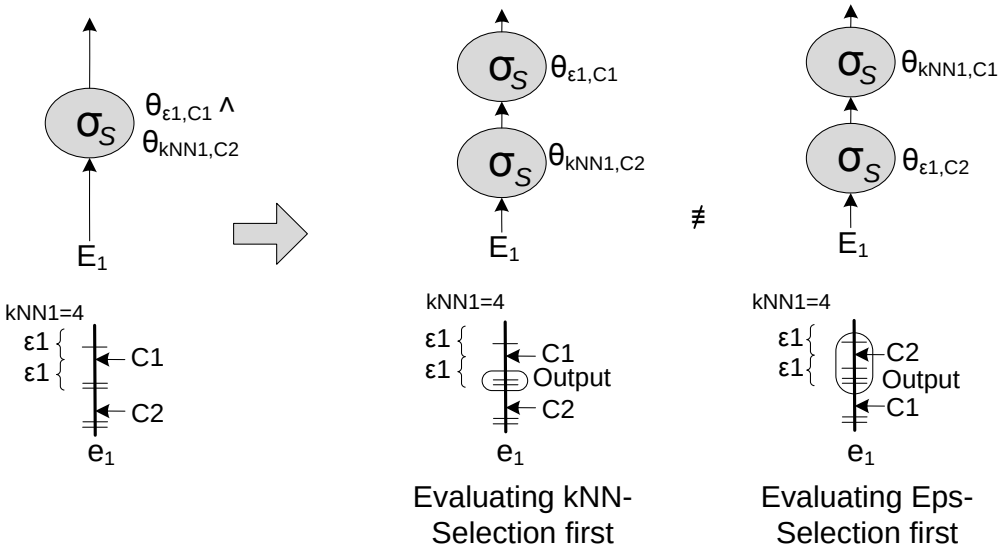


Figure 4-2 Different Ways to Combine Eps-Selection and kNN-Selection

Figure 4-3 presents a conceptual evaluation order for similarity-aware queries. The conceptual query plan makes use of a generic similarity-selection node that combines multiple similarity-selection and similarity-join predicates using the conventional intersection operator as shown in Figure 4.4. Based on the conceptual evaluation order presented in Figure 4.3, a generic similarity-aware query composed by multiple SGB, SJ and SS operators is evaluated as follows. At the bottom of the plan, all the relations involved in the query get combined using cross product. A generic Similarity Selection is evaluated after the cross product operation. This step is equivalent to intersecting the results of evaluating independently each SS and SJ predicate. The regular and similarity grouping operations are evaluated over the results of the selection node. Finally, an optional TOP operator selects the top K tuples using the order established by *SortExpr*.

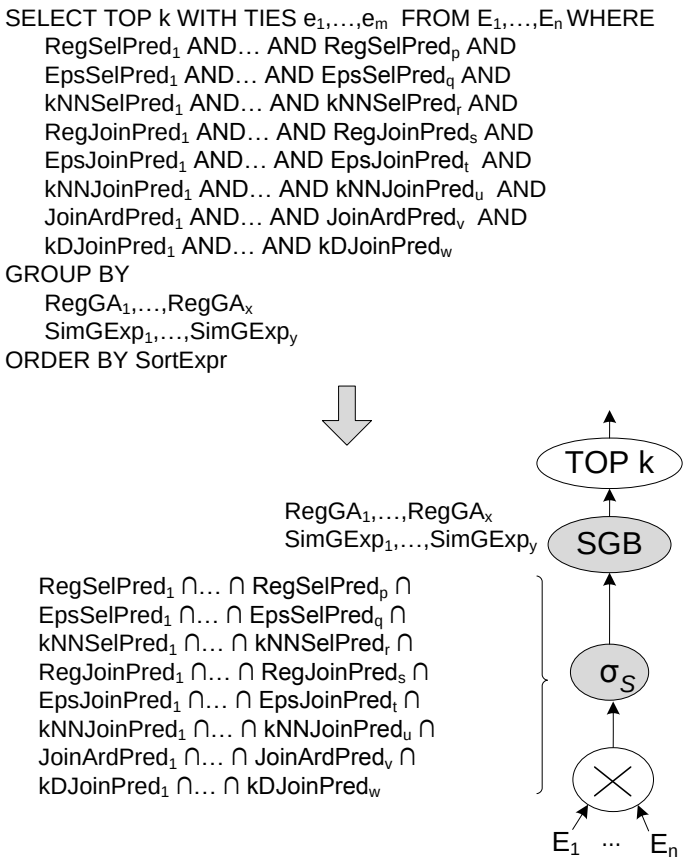


Figure 4-3 Conceptual Evaluation Order of Similarity Queries

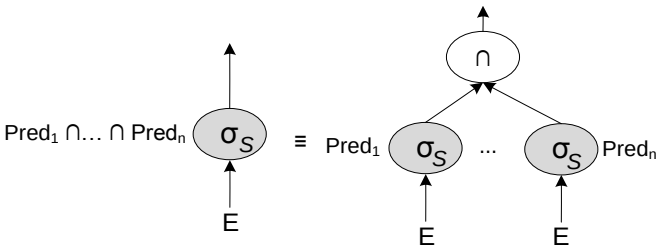


Figure 4-4 Combining Multiple Similarity-aware Predicates

For a given similarity query, the presented conceptual evaluation order makes it clear what the query's expected results are. For example, Figure 4-5 shows how the query presented in Figure 4-2 is evaluated using the conceptual evaluation order. This figure also shows that the conceptual evaluation plan of this query is equivalent to evaluating first the kNN-Selection operator and applying the Epsilon-Selection on the results of the first operator. Figure 4-6 shows another

example of the use of the conceptual evaluation order. This figure shows the SQL version of a similarity query with multiple similarity-aware predicates and the corresponding conceptual evaluation plan.

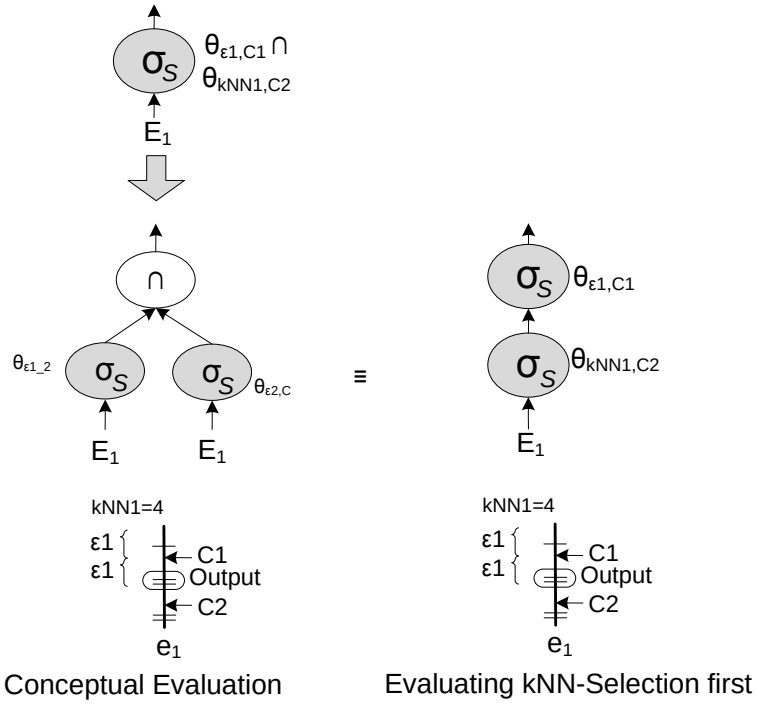


Figure 4-5 Using the Conceptual Evaluation Order

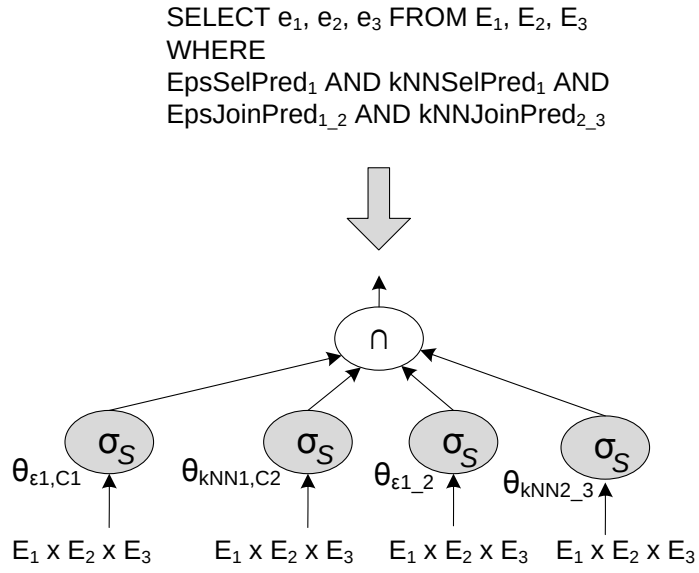


Figure 4-6 Conceptual Evaluation of a Query with Multiple Similarity Predicates

4.4. Similarity Query Transformations

Section 4.3 introduced a conceptual evaluation order for similarity-aware queries. Similar to conventional query processing, the conceptual evaluation of a similarity query is not, in many cases, an efficient way to evaluate the query. Conventional database systems often make use of equivalence rules to transform a query plan into equivalent plans that generate the same results. Cost-based query optimizers compute the cost of each equivalent plan and return the plan with the smallest cost for execution. This section presents multiple equivalence rules that allow the transformation of a similarity query from its conceptual evaluation plan into multiple plans that generate the same results. These equivalence rules allow the extension of cost-based optimization techniques to the case of similarity-aware queries. This section presents proof sketches of several equivalence rules and counterexamples to show the correctness of several non-equivalence rules. Proof sketches for other rules can be constructed in a similar way.

4.4.1. Rules to Combine/Separate Similarity-aware Predicates

The rules presented in this section can be used to *serialize* the operations involved in a query. For instance, given a similarity query composed of two Epsilon-selection predicates applied over the same attribute, the conceptual evaluation will evaluate each selection predicate separately. This evaluation will require reading and processing the input relation twice and then applying an intersection operation over the intermediate results. Using the equivalences presented in this section we are able to obtain an equivalent plan that serializes both selection operations. This new plan only reads from the original relation once to process the first selection. The second selection is applied over the output of the first operation. In all the rules that allow the separation of multiple similarity-aware predicates we assume that the input relation is composed by the cross product of all the relations involved in the similarity-aware predicates. Note that this is always the case in the plans obtained using the conceptual evaluation of similarity queries.

4.4.1.1. Combining Similarity Selection with Cross Product

Similarity Selection operators can be combined with cross product using the following rules.

R1. $\sigma_{\theta_{\varepsilon 1\_2}}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2$

R2. $\sigma_{\theta_{kNN1\_2}}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta_{kNN1\_2}} E_2$

R3. $\sigma_{\theta_{kD1\_2}}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta_{kD1\_2}} E_2$

R4. $\sigma_{\theta_{A1\_2}}(E_1 \times E_2) \equiv E_1 \bowtie_{\theta_{A1\_2}} E_2$

Note that the selection predicates correspond to Similarity Join operations. Figure 4-7 shows a graphic representation of these rules.

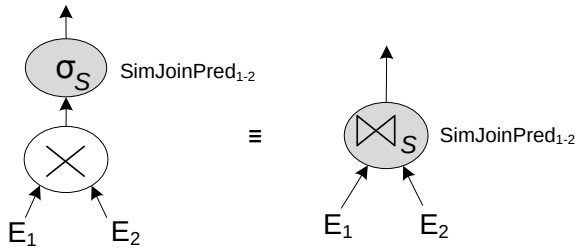


Figure 4-7 Combining Similarity Selection with Cross Product

Proof sketch of Rule R1

Consider a generic tuple t_{E1} of E_1. We will show that for any possible pair (t_{E1},t_{E2}), where t_{E2} is a tuple of E_2, the results generated by the plans of both sides of the rule are the same. Figure 4-8 shows a graphical representation of Rule R1. This figure also shows the domain of the join attributes e_1 and e_2, the location of $t_{E1}.e_1$ (the value of attribute e_1 in tuple t_{E1}), and the different possible regions for the value of $t_{E2}.e_2$.

1. When the value of $t_{E2}.e_2$ belongs to A. In the LHS plan, the cross product will generate the tuple (t_{E1},t_{E2}). However, the tuple will not be selected by the

Similarity Selection operator since *dist($t_{E1}.e_1$,$t_{E2}.e_2$)>ε1_2*. In the RHS plan, due to the definition of Eps-Join, the tuple (t_{E1},t_{E2}) is not part of the output.

2. When the value of $t_{E2}.e_2$ belongs to B. In the LHS plan, the cross product will generate the tuple (t_{E1},t_{E2}). In this case, this tuple is selected by the Similarity Selection operator since *dist($t_{E1}.e_1$,$t_{E2}.e_2$)≤ε1_2*. In the RHS plan, due to the definition of Eps-Join, the tuple (t_{E1},t_{E2}) is part of the output.

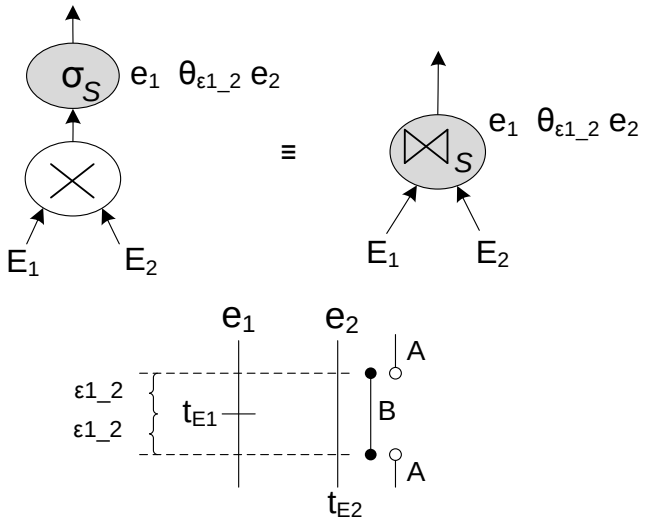


Figure 4-8 Combining Similarity Selection with Cross Product – Proof Sketch

4.4.1.2. Combining/Separating Similarity Selection Predicates

Multiple Similarity Selection predicates can be combined or separated using the following rules.

R5. $\sigma_{\theta_{\varepsilon 1\_1}}\left(\sigma_{\theta_{\varepsilon 1\_2}}(E_1)\right) \equiv \sigma_{\theta_{\varepsilon 1\_1} \cap \theta_{\varepsilon 1\_2}}(E_1)$

R6. $\sigma_{\theta_{kNN1\_1}}\left(\sigma_{\theta_{kNN1\_2}}(E_1)\right) \not\equiv \sigma_{\theta_{kNN1\_1} \cap \theta_{kNN1\_2}}(E_1)$

R7. $\sigma_{\theta_{\varepsilon 1}}\left(\sigma_{\theta_{kNN1}}(E_1)\right) \equiv \sigma_{\theta_{\varepsilon 1} \cap \theta_{kNN1}}(E_1)$

R8. $\sigma_{\theta_{kNN1}}\left(\sigma_{\theta_{\varepsilon 1}}(E_1)\right) \not\equiv \sigma_{\theta_{kNN1} \cap \theta_{\varepsilon 1}}(E_1)$

Rule R5, presented in Figure 4-9, states that multiple Eps-Selection predicates can be combined or separated.

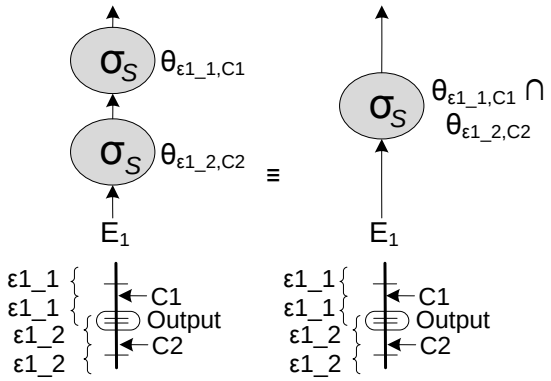


Figure 4-9 Combining/Separating Eps-Selection and Eps-Selection

Proof sketch of Rule R5

Consider a generic tuple t_{E1} of E_1. We will show that for any possible value of t_{E1}, the results generated by the plans of both sides of the rule are the same. The top part of Figure 4-10 shows a graphical representation of Rule R5. Using the conceptual evaluation order of similarity queries, we can transform the left part of the rule to an equivalent expression that uses the intersection operation as represented in the middle part of Figure 4-10. We will use this second version of the rule in the remaining part of the proof. The bottom part of Figure 4-10 shows the different possible regions for the value of $t_{E1}.e_1$.

1. When the value of $t_{E1}.e_1$ belongs to A. In the LHS plan, t_{E1} is not selected in any of the Eps-Selection operators since it does not satisfy any of the selection predicates. Thus, no output is generated by this plan. In the RHS plan, t_{E1} is filtered out by the bottom selection. No tuple flows to the top selection. Thus, no output is generated by this plan either.

2. When the value of $t_{E1}.e_1$ belongs to B. In the LHS plan, t_{E1} is selected in the left Eps-Selection but not in the right one. The intersection operator does not produce any output and consequently no output is generated by this plan. In the

RHS plan, t_{E1} is filtered out by the bottom selection. No tuple flows to the top selection. Thus, no output is generated by this plan either.

3. When the value of $t_{E1}.e_1$ belongs to C. In the LHS plan, t_{E1} is selected by both Eps-Selection operators. Consequently, t_{E1} belongs to the output of the intersection operator. t_{E1} belongs to the output of the LHS plan. In the RHS plan, t_{E1} is selected by the bottom Eps-Selection. t_{E1} is also selected by the top Eps-Selection. Thus, t_{E1} belongs also to the output of the RHS plan.

4. When the value of $t_{E1}.e_1$ belongs to D. In the LHS plan, t_{E1} is selected in the right Eps-Selection but not in the left one. The intersection operator does not produce any output and consequently no output is generated by this plan. In the RHS plan, t_{E1} is selected by the bottom Eps-Selection but filtered out by the top one. Thus, no output is generated by this plan either.

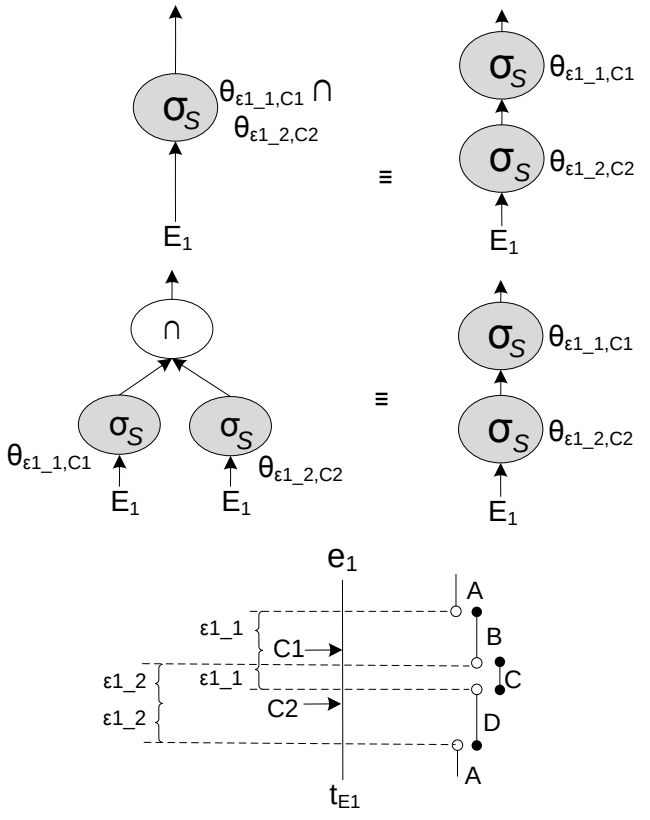


Figure 4-10 Combining Eps-Selection and Eps-Selection – Proof Sketch

Rule R6 states that kNN-Selection predicates cannot be combined or separated. Figure 4-11 shows that plans with separated and combined kNN-Selection predicates generate different results.

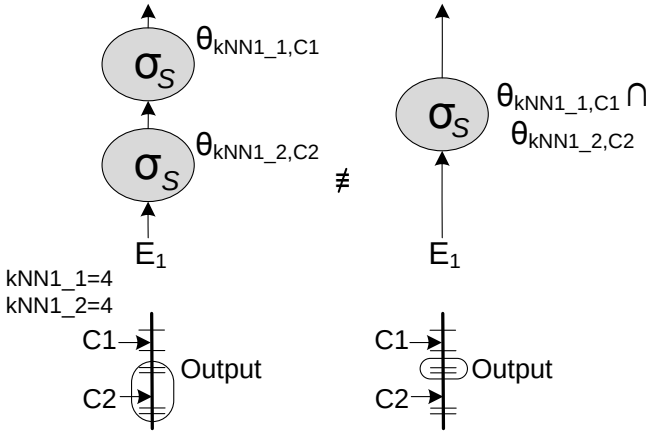


Figure 4-11 Combining/Separating kNN-Selection and kNN-Selection

Rules R7 and R8 specify the way in which Eps-Selection and kNN-Selection predicates can be combined. According to Rule R7, the plan that combines these two types of Similarity Selection is equivalent to executing first the kNN-Selection operation and then the Eps-Selection operation as shown in Figure 4-12. Rule R8 states that we cannot separate these selection predicates executing first the Eps-Selection and then the kNN-Selection. Figure 4-13 shows an example of Rule R8 where the plans have different output.

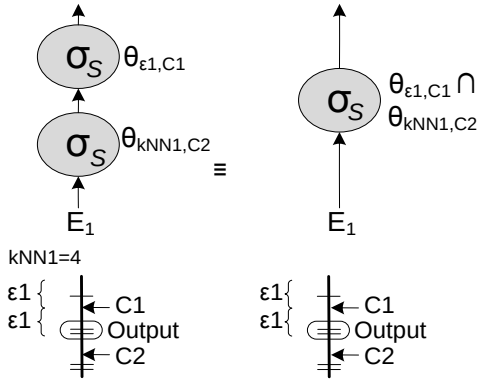


Figure 4-12 Combining/Separating Eps-Selection and kNN-Selection

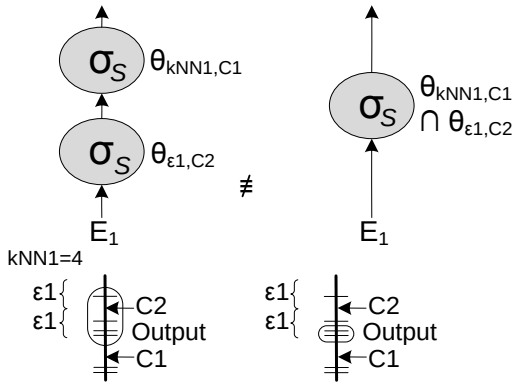


Figure 4-13 Combining/Separating kNN-Selection and Eps-Selection

Proof sketch of Rule R7

Consider a generic tuple t_{E1} of E_1. We will show that for any possible value of t_{E1}, the results generated by the plans of both sides of the rule are the same. The top part of Figure 4-14 shows a graphical representation of Rule R7. Using the conceptual evaluation order of similarity queries, we can transform the left part of the rule to an equivalent expression that uses the intersection operation as represented in the middle part of Figure 4-14. We will use this second version of the rule in the remaining part of the proof. The bottom part of Figure 4-14 shows the different possible regions for the value of $t_{E1}.e_1$. Note that the region marked as kNN1 (which comprises regions C and D) represents the region that contains the kNN1 closest neighbors of C2.

1. When the value of $t_{E1}.e_1$ belongs to A. In the LHS plan, t_{E1} is not selected in any of the selection operators since it does not satisfy any of the Similarity Selection predicates. Thus, no output is generated by this plan. In the RHS plan, t_{E1} is filtered out by the kNN-Selection. No tuple flows to the Eps-Selection. Thus, no output is generated by this plan either.

2. When the value of $t_{E1}.e_1$ belongs to B. In the LHS plan, t_{E1} is selected in the Eps-Selection but not in the kNN-Selection. The intersection operator does not produce any output and consequently no output is generated by this plan. In

the RHS plan, t_{E1} is filtered out by the kNN-Selection. No tuple flows to the Eps-Selection. Thus, no output is generated by this plan either.

3. When the value of $t_{E1}.e_1$ belongs to C. In the LHS plan, t_{E1} is selected by both Similarity Selection operators. Consequently, t_{E1} belongs to the output of the intersection operator. t_{E1} belongs to the output of the LHS plan. In the RHS plan, t_{E1} is selected by the kNN-Selection. t_{E1} is also selected by the Eps-Selection. Thus, t_{E1} belongs also to the output of the RHS plan.

4. When the value of $t_{E1}.e_1$ belongs to D. In the LHS plan, t_{E1} is selected in the kNN-Selection but not in the Eps-Selection. The intersection operator does not produce any output and consequently no output is generated by this plan. In the RHS plan, t_{E1} is selected by the kNN-Selection but filtered out by the Eps-Selection. Thus, no output is generated by this plan either.

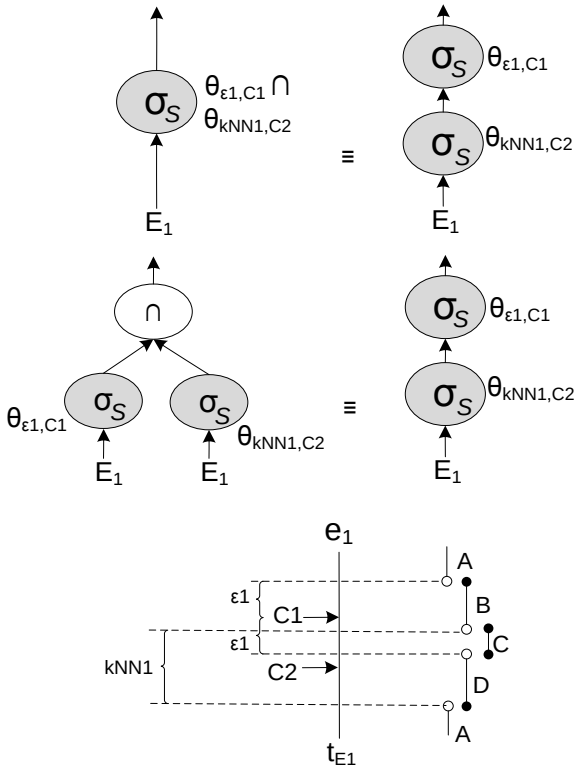


Figure 4-14 Combining Eps-Selection and kNN-Selection – Proof Sketch

4.4.1.3. Combining/Separating Similarity Join and Similarity Selection

Similarity Selection and Similarity Join predicates can be combined or separated using the following rules.

1. Eps-Join and Eps-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R9. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{\varepsilon 2,C}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{\varepsilon 2,C}}(E)) \equiv \sigma_{\theta_{\varepsilon 2,C}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R10. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{\varepsilon 1,C}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{\varepsilon 1,C}}(E)) \equiv \sigma_{\theta_{\varepsilon 1,C}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

2. Eps-Join and kNN-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R11. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kNN2,C}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{kNN2,C}}(E))$

R12. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kNN2,C}}(E) \not\equiv \sigma_{\theta_{kNN2,C}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R13. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kNN1,C}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{kNN1,C}}(E))$

R14. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kNN1,C}}(E) \not\equiv \sigma_{\theta_{kNN1,C}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

3. kNN-Join and Eps-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R15. $\sigma_{\theta_{kNN1\_2} \cap \theta_{\varepsilon 2,C}}(E) \not\equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{\varepsilon 2,C}}(E))$

R16. $\sigma_{\theta_{kNN1\_2} \cap \theta_{\varepsilon 2,C}}(E) \equiv \sigma_{\theta_{\varepsilon 2,C}}(\sigma_{\theta_{kNN1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R17. $\sigma_{\theta_{kNN1\_2} \cap \theta_{\varepsilon 1,C}}(E) \equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{\varepsilon 1,C}}(E)) \equiv \sigma_{\theta_{\varepsilon 1,C}}(\sigma_{\theta_{kNN1\_2}}(E))$

4. kNN-Join and kNN-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R18. $\sigma_{\theta_{kNN1\_2} \cap \theta_{kNN2,C}}(E) \not\equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{kNN2,C}}(E))$

R19. $\sigma_{\theta_{kNN1\_2} \cap \theta_{kNN2,C}}(E) \not\equiv \sigma_{\theta_{kNN2,C}}(\sigma_{\theta_{kNN1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R20. $\sigma_{\theta_{kNN1\_2} \cap \theta_{kNN1,C}}(E) \equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{kNN1,C}}(E)) \equiv \sigma_{\theta_{kNN1,C}}(\sigma_{\theta_{kNN1\_2}}(E))$

5. kD-Join and Eps-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R21. $\sigma_{\theta_{kD1\_2} \cap \theta_{\varepsilon 2,C}}(E) \not\equiv \sigma_{\theta_{kD1\_2}}(\sigma_{\theta_{\varepsilon 2,C}}(E))$

R22. $\sigma_{\theta_{kD1\_2} \cap \theta_{\varepsilon 2,C}}(E) \equiv \sigma_{\theta_{\varepsilon 2,C}}(\sigma_{\theta_{kD1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R23. $\sigma_{\theta_{kD1\_2} \cap \theta_{\varepsilon 1,C}}(E) \not\equiv \sigma_{\theta_{kD1\_2}}(\sigma_{\theta_{\varepsilon 1,C}}(E))$

R24. $\sigma_{\theta_{kD1\_2} \cap \theta_{\varepsilon 1,C}}(E) \equiv \sigma_{\theta_{\varepsilon 1,C}}(\sigma_{\theta_{kD1\_2}}(E))$

6. kD-Join and kNN-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R25. $\sigma_{\theta_{kD1\_2} \cap \theta_{kNN2,C}}(E) \not\equiv \sigma_{\theta_{kD1\_2}}(\sigma_{\theta_{kNN2,C}}(E))$

R26. $\sigma_{\theta_{kD1\_2} \cap \theta_{kNN2,C}}(E) \not\equiv \sigma_{\theta_{kNN2,C}}(\sigma_{\theta_{kD1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R27. $\sigma_{\theta_{kD1\_2} \cap \theta_{kNN1,C}}(E) \not\equiv \sigma_{\theta_{kD1\_2}}(\sigma_{\theta_{kNN1,C}}(E))$

R28. $\sigma_{\theta_{kD1\_2} \cap \theta_{kNN1,C}}(E) \not\equiv \sigma_{\theta_{kNN1,C}}(\sigma_{\theta_{kD1\_2}}(E))$

7. Join-Around and Eps-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R29. $\sigma_{\theta_{A1\_2} \cap \theta_{\varepsilon 2,C}}(E) \not\equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{\varepsilon 2,C}}(E))$

R30. $\sigma_{\theta_{A1\_2} \cap \theta_{\varepsilon 2,C}}(E) \equiv \sigma_{\theta_{\varepsilon 2,C}}(\sigma_{\theta_{A1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R31. $\sigma_{\theta_{A1\_2} \cap \theta_{\varepsilon 1,C}}(E) \equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{\varepsilon 1,C}}(E)) \equiv \sigma_{\theta_{\varepsilon 1,C}}(\sigma_{\theta_{A1\_2}}(E))$

8. Join-Around and kNN-Selection

When the selection predicate attribute is the inner attribute in the join predicate

R32. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN2,C}}(E) \not\equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{kNN2,C}}(E))$

R33. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN2,C}}(E) \not\equiv \sigma_{\theta_{kNN2,C}}(\sigma_{\theta_{A1\_2}}(E))$

When the selection predicate attribute is the outer attribute in the join predicate

R34. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN1,C}}(E) \equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{kNN1,C}}(E))$

R35. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN1,C}}(E) \not\equiv \sigma_{\theta_{kNN1,C}}(\sigma_{\theta_{A1\_2}}(E))$

In Rules R9 to R35, we consider two generic cases: when the selection predicate attribute is the outer or inner attribute in the join predicate. An intuitive but important generic observation is that this classification is relevant, i.e., generate different equivalence rules in both cases, when the Similarity Join operation is not commutative (kNN-Join and Join-Around). In general, if the join operation is

commutative (Epsilon-Join and kDistance-Join), the rules for both cases are the same. We will discuss commutative join operations in Section 4.4.2.

Rules R9 and R10 state that Eps-Join and Eps-Selection operations can be combined or separated. Figure 4-15 shows an example of Rule R9.

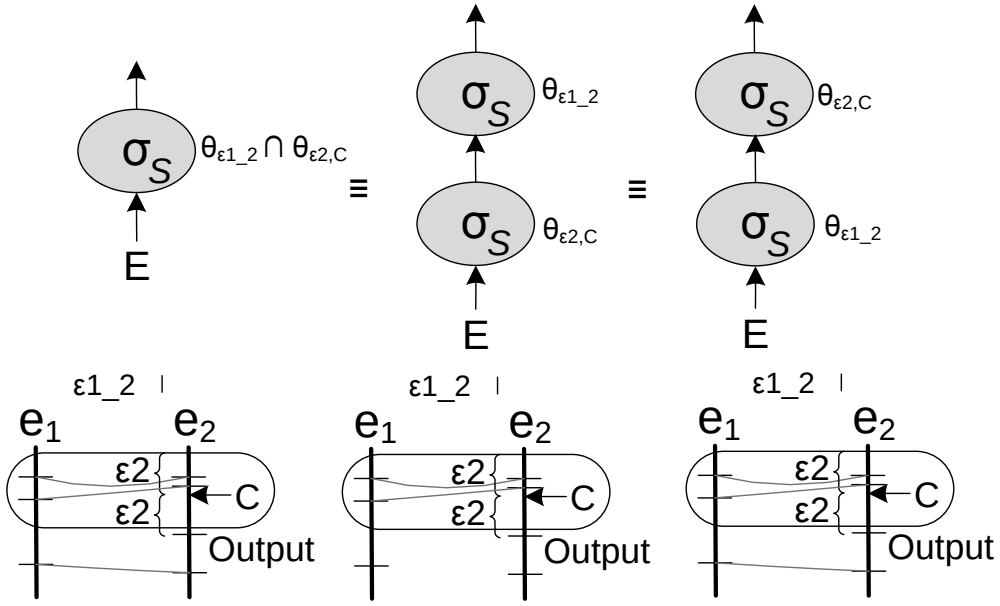


Figure 4-15 Combining/Separating Eps-Join and Eps-Selection

Proof sketch of Rule R9

Eps-Join is defined over two relations. Assume that $\theta_{\varepsilon 1\_2}$ is defined over relations E_1 and E_2, and that the input relation E is the cross product of all the relations involved in the similarity-aware predicates, i.e., $E = E_1 \times E_2$. Furthermore, we assume that the join attributes are $E_1.e_1$ and $E_2.e_2$. Consider a generic tuple t_{E1} of E_1. We will show that for any possible pair (t_{E1},t_{E2}), where t_{E2} is a tuple of E_2, the results generated by the plans of both sides of the rule are the same. The top part of Figure 4-16 shows a graphical representation of Rule R9. Using the conceptual evaluation order of similarity queries, we can transform the left part of the rule to an equivalent expression that uses the intersection operation as

represented in the middle part of Figure 4-16. We will use this second version of the rule in the remaining part of the proof. The bottom part of Figure 4-16 shows the different possible regions for the value of $t_{E2}.e_2$.

1. When the value of $t_{E2}.e_2$ belongs to A. In the LHS plan, the pair (t_{E1},t_{E2}) is not selected in any similarity-aware operator since it does not satisfy any of their predicates. Thus, no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2}) is filtered out by the bottom selection since $dist(t_{E2}.e_2, C1) > \varepsilon 2$. No tuple flows to the top operator. Thus, no output is generated by this plan either.

2. When the value of $t_{E2}.e_2$ belongs to B. In the LHS plan, the pair (t_{E1},t_{E2}) is selected in the left Similarity Selection but not in the right one. The intersection operator does not produce any output and consequently no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2}) is filtered out by the bottom selection since $dist(t_{E2}.e_2, C1) > \varepsilon 2$. No tuple flows to the top operator. Thus, no output is generated by this plan either.

3. When the value of $t_{E2}.e_2$ belongs to C. In the LHS plan, the pair (t_{E1},t_{E2}) is selected in both similarity-aware operators. Consequently, (t_{E1},t_{E2}) belongs to the output of the intersection operator. (t_{E1},t_{E2}) belongs to the output of the LHS plan. In the RHS plan, (t_{E1},t_{E2}) is selected by the bottom selection since $dist(t_{E2}.e_2, C1) \leq \varepsilon 2$. ($t_{E1}$,$t_{E2}$) is also selected by the top selection since $dist(t_{E1}.e_1, t_{E2}.e_2) \leq \varepsilon 1\_2$. Thus, the pair ($t_{E1}$,$t_{E2}$) belongs also to the output of the RHS plan.

4. When the value of $t_{E2}.e_2$ belongs to D. In the LHS plan, the pair (t_{E1},t_{E2}) is selected in the right Similarity Selection but not in the left one. The intersection operator does not produce any output and consequently no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2}) is selected in the bottom selection since $dist(t_{E2}.e_2, C1) \leq \varepsilon 2$ but it is filtered out by the top selection. Thus, no output is generated by this plan either.

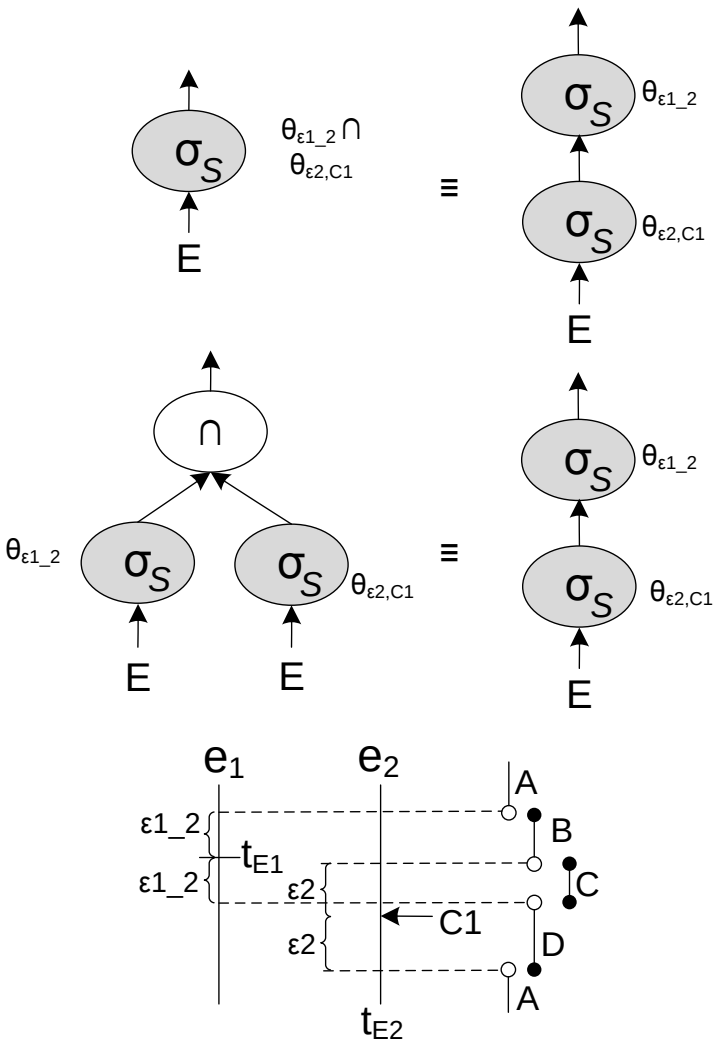


Figure 4-16 Combining Eps-Join and Eps-Selection – Proof Sketch

Rules R11, R12, R13 and R14 state that Eps-Join and kNN-Selection predicates can be separated as long as the kNN-Selection operation is executed first. Figure 4-17 shows examples of Rules R11 and R12.

Rules R15, R16 and R17 state the way kNN-Join and Eps-Selection predicates can be combined or separated. In this case, the equivalence rules depend on whether the selection attribute is the inner or outer attribute of the join predicate. According to Rules R15 and R16, when the selection attribute is the inner attribute of the join predicate, the similarity operations can be separated executing first the kNN-Join and then the Eps-Selection as shown in Figure 4-18.

According to Rule R17, when the selection attribute is the outer attribute of the join predicate, the similarity operations can be separated in any order as shown in Figure 4-19.

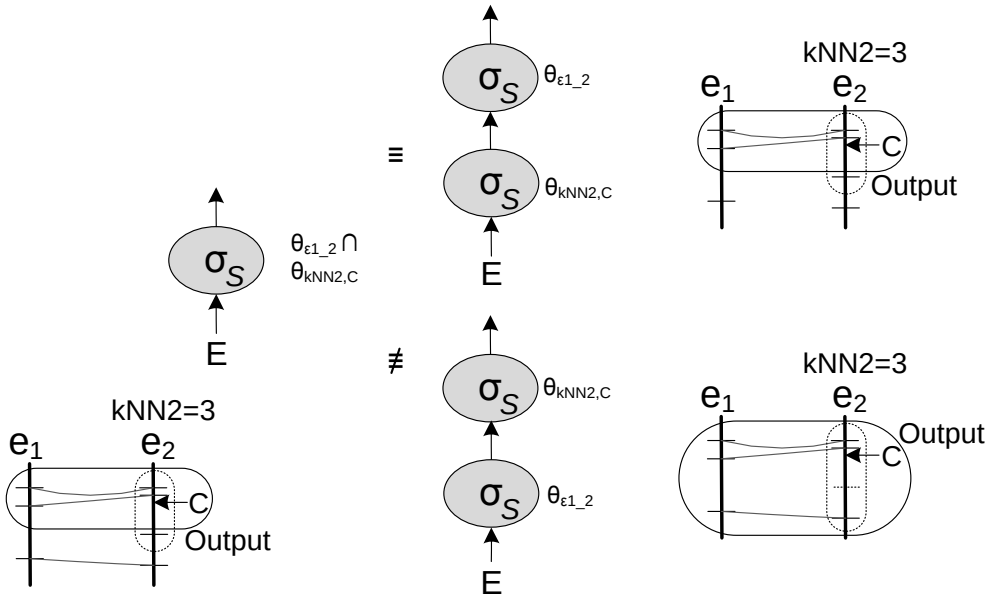


Figure 4-17 Combining/Separating Eps-Join and kNN-Selection

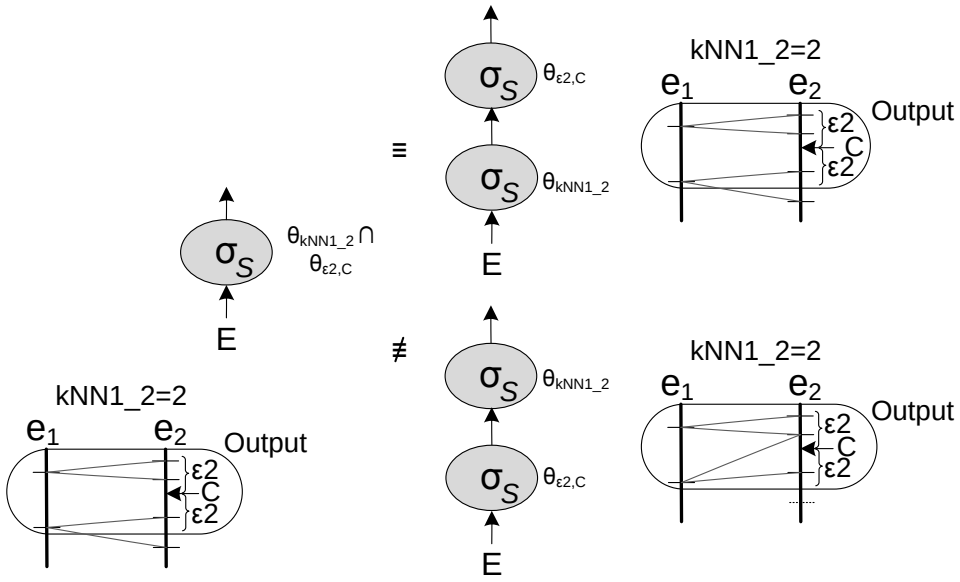


Figure 4-18 Combining/Separating kNN-Join and Eps-Selection - When the Selection Predicate Attribute is the Inner Attribute in the Join Predicate

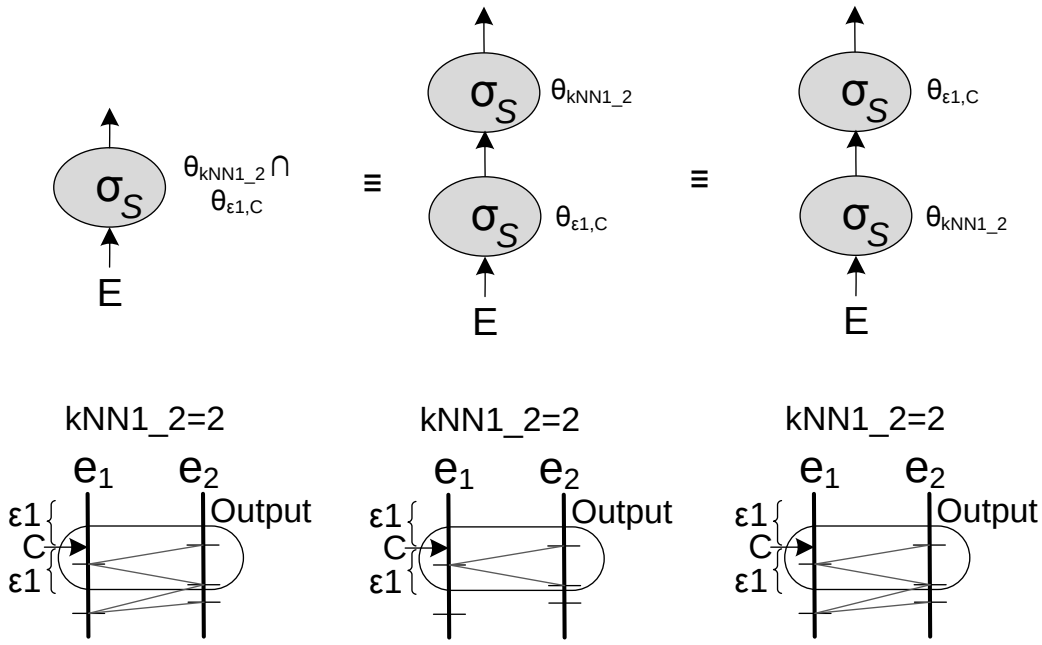


Figure 4-19 Combining/Separating kNN-Join and Eps-Selection - When the Selection Predicate Attribute is the Outer Attribute in the Join Predicate

Proof sketch of Rule R16

kNN-Join is defined over two relations. Assume that θ_{kNN1_2} is defined over relations E_1 and E_2, and that the input relation E is the cross product of all the relations involved in the similarity-aware predicates, i.e., $E = E_1 \times E_2$. Furthermore, we assume that the join attributes are $E_1.e_1$ and $E_2.e_2$. Consider a generic tuple t_{E1} of E_1. We will show that for any possible pair (t_{E1},t_{E2}), where t_{E2} is a tuple of E_2, the results generated by the plans of both sides of the rule are the same. The top part of Figure 4-20 shows a graphical representation of Rule R16. Using the conceptual evaluation order of similarity queries, we can transform the left part of the rule to an equivalent expression that uses the intersection operation as represented in the middle part of Figure 4-20. We will use this second version of the rule in the remaining part of the proof. The bottom part of Figure 4-20 shows the different possible regions for the value of $t_{E2}.e_2$. Note that the region marked as *kNN1_2* (which comprises regions B and C) represents the region that contains the *kNN1_2* closest neighbors of t_{E1} in E_2.

1. When the value of $t_{E2}.e_2$ belongs to A. In the LHS plan, the pair (t_{E1},t_{E2}) is not selected in any similarity-aware operator since it does not satisfy any of their predicates. Thus, no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2}) is filtered out by the bottom selection since t_{E2} is not one of the *kNN1_2* closest neighbors of t_{E1} in E_2. No tuple flows to the top operator. Thus, no output is generated by this plan either.

2. When the value of $t_{E2}.e_2$ belongs to B. In the LHS plan, the pair (t_{E1},t_{E2}) is selected in the left Similarity Selection but not in the right one. The intersection operator does not produce any output and consequently no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2}) is selected in the bottom selection since t_{E2} is one of the *kNN1_2* closest neighbors of t_{E1} in E_2. However, (t_{E1},t_{E2}) is filtered out by the top selection because $dist(t_{E2}.e_2, C1) > \varepsilon 2$. Thus, no output is generated by this plan either.

3. When the value of $t_{E2}.e_2$ belongs to C. In the LHS plan, the pair (t_{E1},t_{E2}) is selected in both similarity-aware operators. Consequently, (t_{E1},t_{E2}) belongs to the output of the intersection operator. (t_{E1},t_{E2}) belongs to the output of the LHS plan. In the RHS plan, (t_{E1},t_{E2}) is selected by the bottom selection since t_{E2} is one of the *kNN1_2* closest neighbors of t_{E1} in E_2. (t_{E1},t_{E2}) is also selected by the top selection since $dist(t_{E2}.e_2, C1) \leq \varepsilon 2$. Thus, ($t_{E1}$,$t_{E2}$) belongs also to the output of the RHS plan.

4. When the value of $t_{E2}.e_2$ belongs to D. In the LHS plan, the pair (t_{E1},t_{E2}) is selected in the right Similarity Selection but not in the left one. The intersection operator does not produce any output and consequently no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2}) is filtered out by the bottom selection. No tuple flows to the top operator. Thus, no output is generated by this plan either.

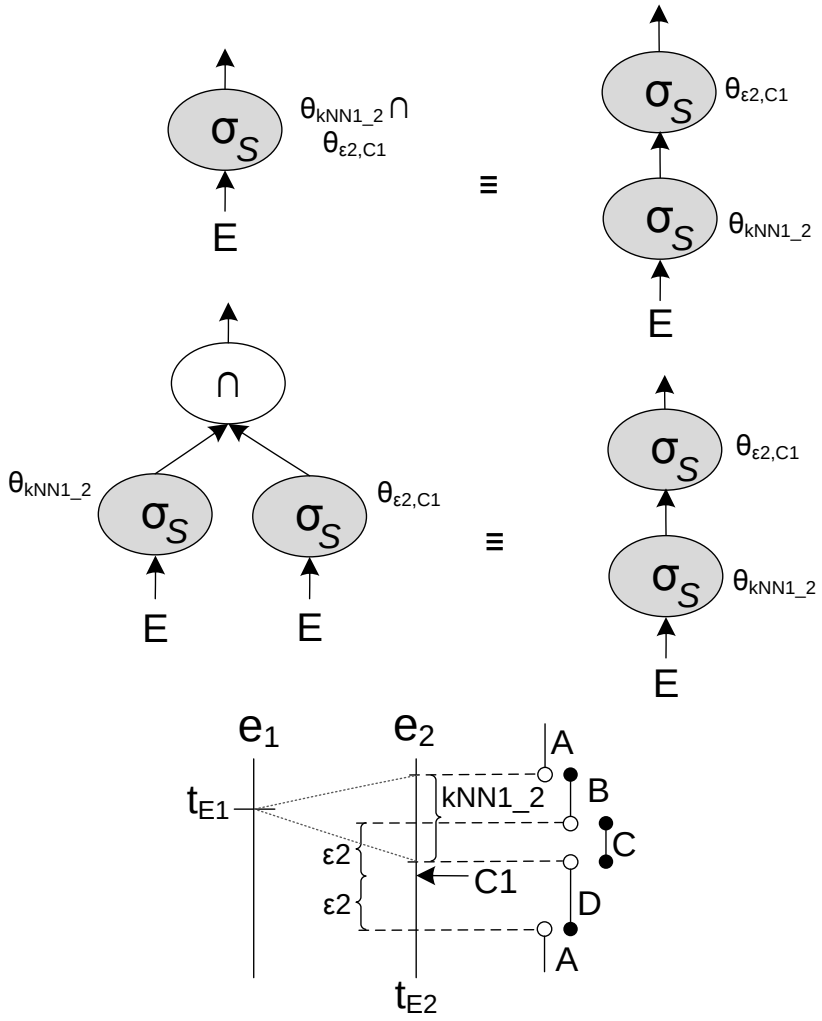


Figure 4-20 Combining kNN-Join and Eps-Selection - When the Selection Predicate Attribute is the Inner Attribute in the Join Predicate – Proof Sketch

Rules R18, R19 and R20 specify the way kNN-Join and kNN-Selection predicates can be combined or separated. In this case, the similarity operations can be combined or separated only if the selection attribute is the outer attribute of the join predicate. Figure 4-21 shows that when the selection attribute is the inner attribute of the join predicate, the plans with combined and separated similarity predicates produce different results (R18 and R19). Figure 4-22 shows an example of separating kNN-Join and kNN-Selection when the selection attribute is the outer attribute of the join predicate (R20).

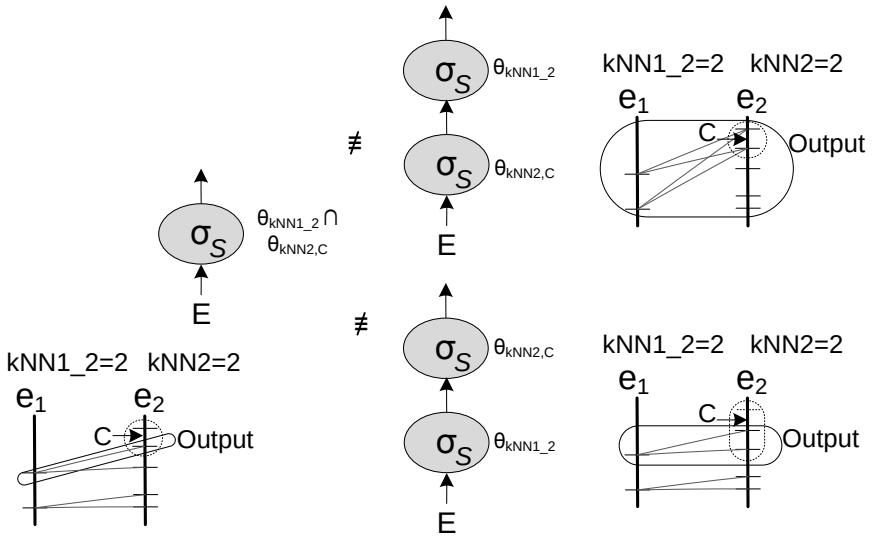


Figure 4-21 Combining/Separating kNN-Join and kNN-Selection - When the Selection Predicate Attribute is the Inner Attribute in the Join Predicate

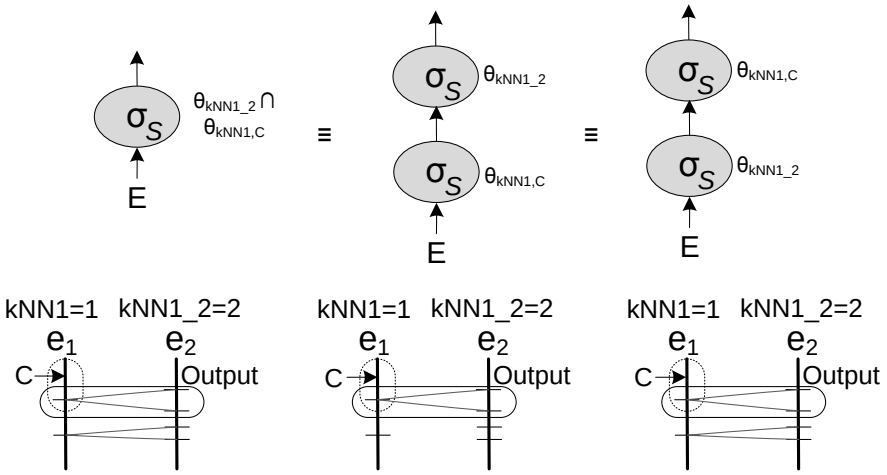


Figure 4-22 Combining/Separating kNN-Join and kNN-Selection - When the Selection Predicate Attribute is the Outer Attribute in the Join Predicate

Rules R21, R22, R23 and R24 specify the way kD-Join and Eps-Selection predicates can be combined or separated. According to these rules, the similarity operations can be separated executing first the kD-Join and then the Eps-Selection as shown in Figure 4-23. Rules R25, R26, R27 and R28 state that plans with combined or separated kD-Join and kNN-Selection predicates produce different results. Figure 4-24 shows examples of Rules R25 and R26.

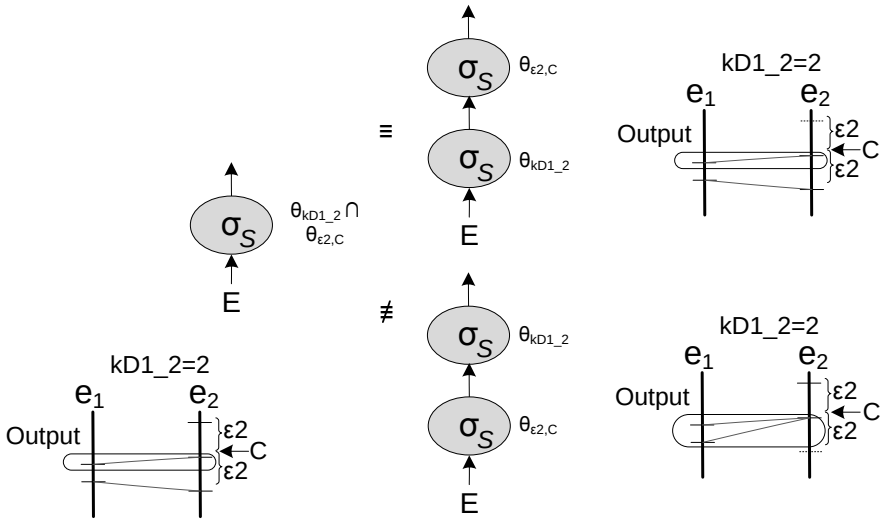


Figure 4-23 Combining/Separating kD-Join and Eps-Selection

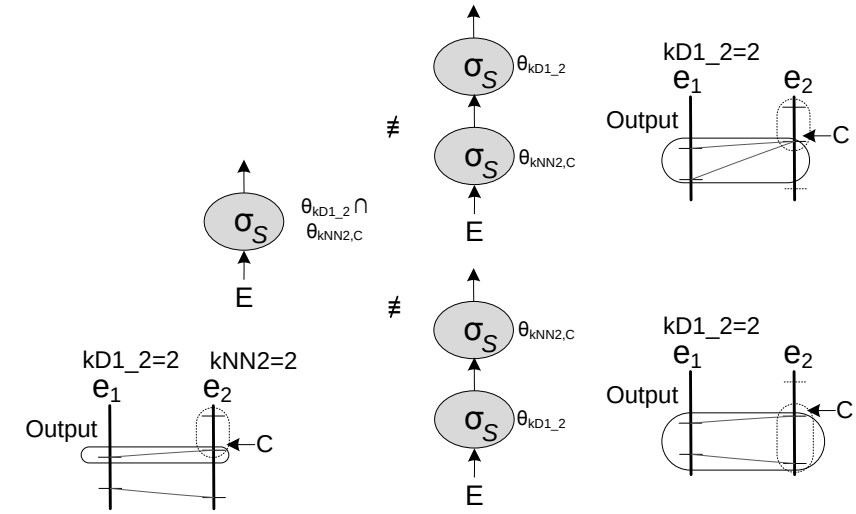


Figure 4-24 Combining/Separating kD-Join and kNN-Selection

Rules R29, R30 and R31 state the way Join-Around and Eps-Selection predicates can be combined or separated. Rules R32, R33, R34 and R35 state the way Join-Around and kNN-Selection predicates can be combined or separated. Given that Join-Around is a hybrid between the kNN-Join with k=1 and the Eps-Join, the way this operation can be combined with Similarity

Selection corresponds to the most restricted way in which kNN-Join or the Eps-Join can be combined with Similarity Selection.

4.4.1.4. Combining/Separating Similarity Join Predicates

Multiple Similarity Join predicates can be combined or separated using the following rules.

1. Eps-Join and Eps-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R36. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{\varepsilon 2\_3}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{\varepsilon 2\_3}}(E)) \equiv \sigma_{\theta_{\varepsilon 2\_3}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2 ←e3)

R37. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{\varepsilon 3\_2}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{\varepsilon 3\_2}}(E)) \equiv \sigma_{\theta_{\varepsilon 3\_2}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

2. kNN-Join and kNN-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R38. $\sigma_{\theta_{kNN1\_2} \cap \theta_{kNN2\_3}}(E) \equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{kNN2\_3}}(E)) \equiv \sigma_{\theta_{kNN2\_3}}(\sigma_{\theta_{kNN1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2 ←e3)

R39. $\sigma_{\theta_{kNN1\_2} \cap \theta_{kNN3\_2}}(E) \not\equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{kNN3\_2}}(E))$

R40. $\sigma_{\theta_{kNN1\_2} \cap \theta_{kNN3\_2}}(E) \not\equiv \sigma_{\theta_{kNN3\_2}}(\sigma_{\theta_{kNN1\_2}}(E))$

3. kD-Join and kD-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R41. $\sigma_{\theta_{kD1\_2} \cap \theta_{kD2\_3}}(E) \not\equiv \sigma_{\theta_{kD1\_2}}(\sigma_{\theta_{kD2\_3}}(E))$

R42. $\sigma_{\theta_{kD1\_2} \cap \theta_{kD2\_3}}(E) \not\equiv \sigma_{\theta_{kD2\_3}}(\sigma_{\theta_{kD1\_2}}(E))$

When predicates' attributes do not have a single direction (e1→e2, e2 ←e3)

R43. $\sigma_{\theta_{kD1\_2} \cap \theta_{kD3\_2}}(E) \not\equiv \sigma_{\theta_{kD1\_2}}(\sigma_{\theta_{kD3\_2}}(E))$

R44. $\sigma_{\theta_{kD1\_2} \cap \theta_{kD3\_2}}(E) \not\equiv \sigma_{\theta_{kD3\_2}}(\sigma_{\theta_{kD1\_2}}(E))$

4. Eps-Join and kNN-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R45. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kNN2\_3}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{kNN2\_3}}(E)) \equiv \sigma_{\theta_{kNN2\_3}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R46. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kNN3\_2}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{kNN3\_2}}(E))$

R47. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kNN3\_2}}(E) \not\equiv \sigma_{\theta_{kNN3\_2}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

5. Eps-Join and kD-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R48. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kD2\_3}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{kD2\_3}}(E))$

R49. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kD2\_3}}(E) \not\equiv \sigma_{\theta_{kD2\_3}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2 ←e3)

R50. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kD3\_2}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{kD3\_2}}(E))$

R51. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{kD3\_2}}(E) \not\equiv \sigma_{\theta_{kD3\_2}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

6. kNN-Join and kD-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R52. $\sigma_{\theta_{kNN1\_2} \wedge \theta_{kD2\_3}}(E) \not\equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{kD2\_3}}(E))$

R53. $\sigma_{\theta_{kNN1\_2} \wedge \theta_{kD2\_3}}(E) \not\equiv \sigma_{\theta_{kD2\_3}}(\sigma_{\theta_{kNN1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2 ←e3)

R54. $\sigma_{\theta_{kNN1\_2} \wedge \theta_{kD3\_2}}(E) \not\equiv \sigma_{\theta_{kNN1\_2}}(\sigma_{\theta_{kD3\_2}}(E))$

R55. $\sigma_{\theta_{kNN1\_2} \wedge \theta_{kD3\_2}}(E) \not\equiv \sigma_{\theta_{kD3\_2}}(\sigma_{\theta_{kD1\_2}}(E))$

7. Join-Around and Join-Around

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R56. $\sigma_{\theta_{A1\_2} \cap \theta_{A2\_3}}(E) \equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{A2\_3}}(E)) \equiv \sigma_{\theta_{A2\_3}}(\sigma_{\theta_{A1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2 ←e3)

R57. $\sigma_{\theta_{A1\_2} \cap \theta_{A3\_2}}(E) \not\equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{A3\_2}}(E))$

R58. $\sigma_{\theta_{A1\_2} \cap \theta_{A3\_2}}(E) \not\equiv \sigma_{\theta_{A3\_2}}(\sigma_{\theta_{A1\_2}}(E))$

8. Eps-Join and Join-Around

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R59. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{A2\_3}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{A2\_3}}(E)) \equiv \sigma_{\theta_{A2\_3}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R60. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{A3\_2}}(E) \equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{A3\_2}}(E))$

R61. $\sigma_{\theta_{\varepsilon 1\_2} \cap \theta_{A3\_2}}(E) \not\equiv \sigma_{\theta_{A3\_2}}(\sigma_{\theta_{\varepsilon 1\_2}}(E))$

9. Join-Around and kNN-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R62. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN2\_3}}(E) \equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{kNN2\_3}}(E))$

R63. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN2\_3}}(E) \equiv \sigma_{\theta_{kNN2\_3}}(\sigma_{\theta_{A1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R64. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN3\_2}}(E) \not\equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{kNN3\_2}}(E))$

R65. $\sigma_{\theta_{A1\_2} \cap \theta_{kNN3\_2}}(E) \not\equiv \sigma_{\theta_{kNN3\_2}}(\sigma_{\theta_{A1\_2}}(E))$

10. Join-Around and kD-Join

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R66. $\sigma_{\theta_{A1\_2} \cap \theta_{kD2\_3}}(E) \equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{kD2\_3}}(E))$

R67. $\sigma_{\theta_{A1\_2} \cap \theta_{kD2\_3}}(E) \not\equiv \sigma_{\theta_{kD2\_3}}(\sigma_{\theta_{A1\_2}}(E))$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R68. $\sigma_{\theta_{A1\_2} \cap \theta_{kD3\_2}}(E) \not\equiv \sigma_{\theta_{A1\_2}}(\sigma_{\theta_{kD3\_2}}(E))$

R69. $\sigma_{\theta_{A1\_2} \cap \theta_{kD3\_2}}(E) \not\equiv \sigma_{\theta_{kD3\_2}}(\sigma_{\theta_{A1\_2}}(E))$

In Rules R36 to R69, we consider two generic cases: when the attributes in the predicates have a single direction, e.g., e1→e2, e2→e3; and when they do not, e.g., e1→e2, e2 ←e3. In general, this classification is relevant, i.e., generate different equivalence rules in both cases, when at least one of the Similarity Join operations is not commutative (kNN-Join and Join-Around). Commutative join operations are discussed in Section 4.4.2.

Rules R36 and R37 specify the way multiple Eps-Join predicates can be combined or separated. According to these rules, Eps-Join predicates can be separated in any order. Figure 4-25 shows an example of Rule R36.

Proof sketch of Rule R36

Every Eps-Join operation is defined over two relations. Assume that $\theta_{\varepsilon 1\_2}$ is defined over relations E_1 and E_2, and $\theta_{\varepsilon 2\_3}$ over relations E_2 and E_3.

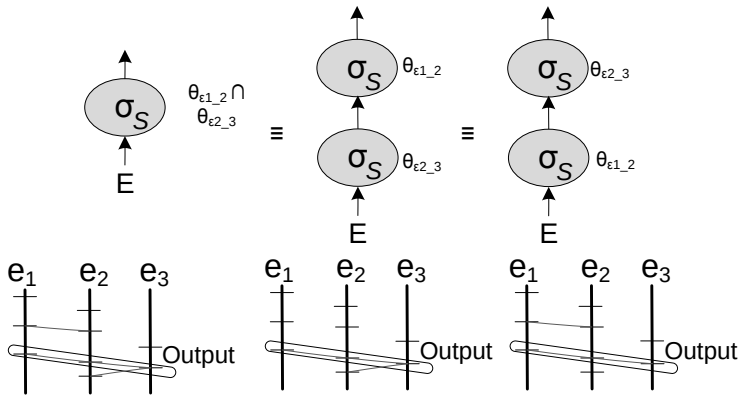


Figure 4-25 Combining/Separating Multiple Eps-Join Predicates

Assume also that the input relation *E* is the cross product of all the relations involved in the similarity-aware predicates, i.e., $E = E_1 \times E_2 \times E_3$. Furthermore, we assume that the join attributes in $\theta_{\varepsilon 1\_2}$ are $E_1.e_1$ and $E_2.e_2$, and in $\theta_{\varepsilon 2\_3}$ are $E_2.e_2$ and $E_3.e_3$. Consider a generic tuple t_{E1} of E_1. We will show that for any possible triplet (t_{E1},t_{E2},t_{E3}), where t_{E2} is a tuple of E_2, and t_{E3} is a tuple of E_3, the results generated by the plans of both sides of the rule are the same. The top part of Figure 4-26 shows a graphical representation of Rule R36. Using the conceptual evaluation order of similarity queries, we can transform the left part of the rule to an equivalent expression that uses the intersection operation as represented in the middle part of Figure 4-26. We will use this second version of the rule in the remaining part of the proof. The bottom part of Figure 4-26 shows the different possible regions for the values of $t_{E2}.e_2$ and $t_{E3}.e_3$. Note that the regions for $t_{E3}.e_3$ have been specified based on a generic tuple t_{E2} with $t_{E2}.e_2$ in region B.

1. When the value of $t_{E2}.e_2$ belongs to A. In the LHS plan, the triplet (t_{E1},t_{E2},t_{E3}) is not selected in any similarity-aware operator since it does not satisfy any of their predicates. Thus, no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2},t_{E3}) is filtered out by the bottom selection since $dist(t_{E1}.e_1,t_{E2}.e_2) > \varepsilon 1\_2$. No tuple flows to the top operator. Thus, no output is generated by this plan either.

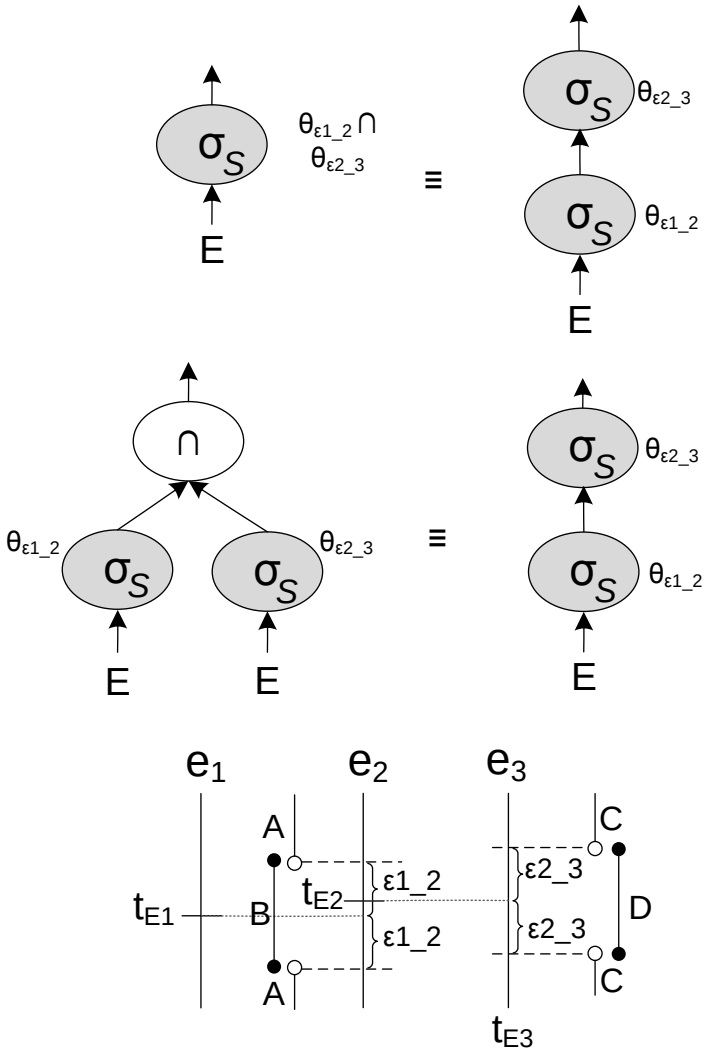


Figure 4-26 Combining/Separating Two Eps-Join Predicates – Proof Sketch

2. When the value of $t_{E2}.e_2$ belongs to B and the value of $t_{E3}.e_3$ belongs to C. In the LHS plan, the triplet (t_{E1},t_{E2},t_{E3}) is selected in the left Similarity Selection since $dist(t_{E1}.e_1,t_{E2}.e_2) \leq \varepsilon 1\_2$ but not in the right one since $dist(t_{E2}.e_2,t_{E3}.e_3) > \varepsilon 2\_3$. The intersection operator does not produce any output and consequently no output is generated by this plan. In the RHS plan, (t_{E1},t_{E2},t_{E3}) is selected in the bottom selection since $dist(t_{E1}.e_1,t_{E2}.e_2) \leq \varepsilon 1\_2$ but it is filtered out by the top selection since $dist(t_{E2}.e_2,t_{E3}.e_3) > \varepsilon 2\_3$. Thus, no output is generated by this plan either.

3. When the value of $t_{E2}.e_2$ belongs to B and the value of $t_{E3}.e_3$ belongs to D. In the LHS plan, the triplet (t_{E1},t_{E2},t_{E3}) is selected in both similarity-aware operators since $dist(t_{E1}.e_1,t_{E2}.e_2) \leq \varepsilon 1\_2$ (left) and $dist(t_{E2}.e_2,t_{E3}.e_3) \leq \varepsilon 2\_3$ (right). Consequently, (t_{E1},t_{E2},t_{E3}) belongs to the output of the intersection operator. (t_{E1},t_{E2},t_{E3}) belongs to the output of the LHS plan. In the RHS plan, (t_{E1},t_{E2},t_{E3}) is selected by the bottom selection since $dist(t_{E1}.e_1,t_{E2}.e_2) \leq \varepsilon 1\_2$. ($t_{E1}$,$t_{E2}$,$t_{E3}$) is also selected by the top selection since $dist(t_{E2}.e_2,t_{E3}.e_3) \leq \varepsilon 2\_3$. Thus, ($t_{E1}$,$t_{E2}$,$t_{E3}$) belongs also to the output of the RHS plan.

Multiple kNN-Join operations can be separated when the attributes of the join predicates have a single direction (R38) but not when this condition is not satisfied (R39 and R40). Figure 4-27 shows an example of Rule R38. This figure also shows that the kNN-Join operations can be separated in any order. Figure 4-28 shows an example of Rules R39 and R40. The figure shows that, when the join attributes do not have a single direction, the plans generated serializing the kNN-Join operations generate different results than the conceptual evaluation plan. Furthermore, the plans corresponding to the two ways to serialize the operations generate different results.

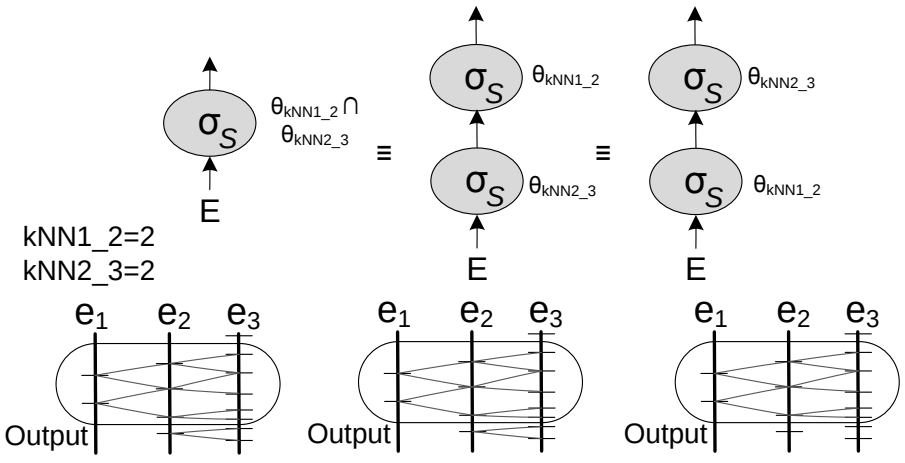


Figure 4-27 Combining/Separating Multiple kNN-Join Predicates - When the Attributes in the Predicates Have a Single Direction (e1→e2, e2→e3)

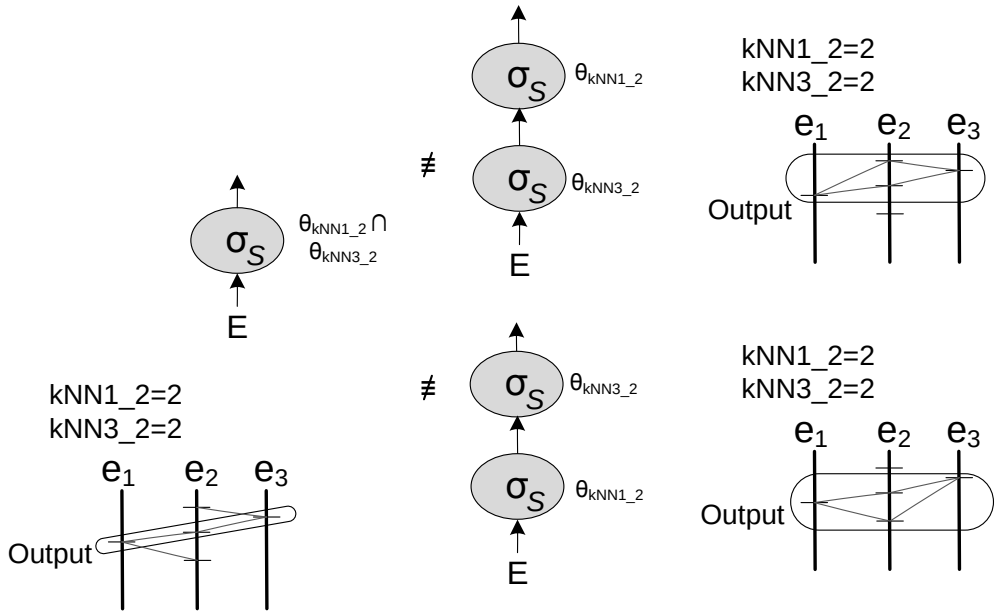


Figure 4-28 Combining/Separating Multiple kNN-Join Predicates - When the Attributes in the Predicates do not Have a Single Direction (e1→e2, e2←e3)

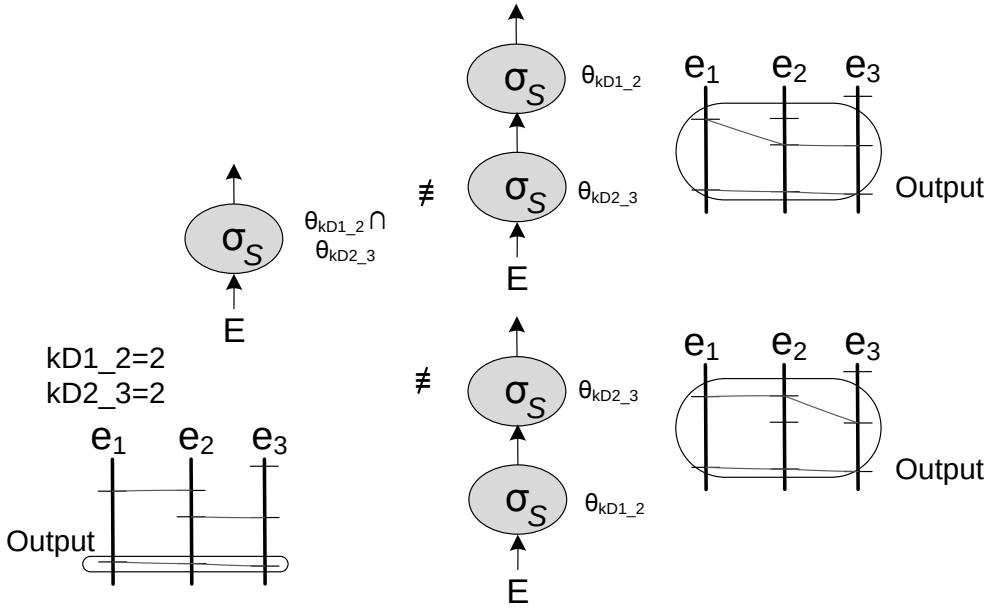


Figure 4-29 Combining/Separating Multiple kD-Join Predicates

Rules R41, R42, R43 and R44 specify that multiple kD-Join operations cannot be separated. Rules R41 and R42 state that the separation cannot be made when

the attributes of the join predicates have a single direction. Rules R43 and R44 state that the separation cannot be made when the single direction requirement is not satisfied. Figure 4-29 shows an example of Rules R41 and R42. This figure shows that the plans that separate the kNN-Join operations generate different results than the plan that combines the join predicates. Furthermore, the plans corresponding to the two ways to serialize the operations generate different results.

Rules R45, R46 and R47 specify the way Eps-Join and kNN-Join predicates can be combined or separated. In this case, the equivalence rules depend on whether the attributes of the join predicates have a single direction or not. According to Rule R45, when the attributes of the join predicates have a single direction, the join operations can be separated in any order as shown in Figure 4-30. According to Rules R46 and R47, when the attributes of the join predicates do not have a single direction, the join operations can be separated executing first the kNN-Join and then the Eps-Join as shown in Figure 4-31.

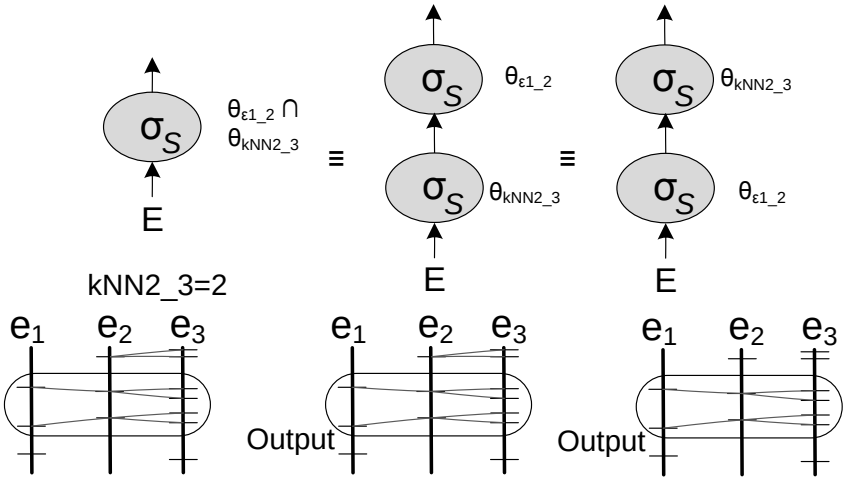


Figure 4-30 Combining/Separating Eps-Join and kNN-Join - When the Attributes in the Predicates Have a Single Direction (e1→e2, e2→e3)

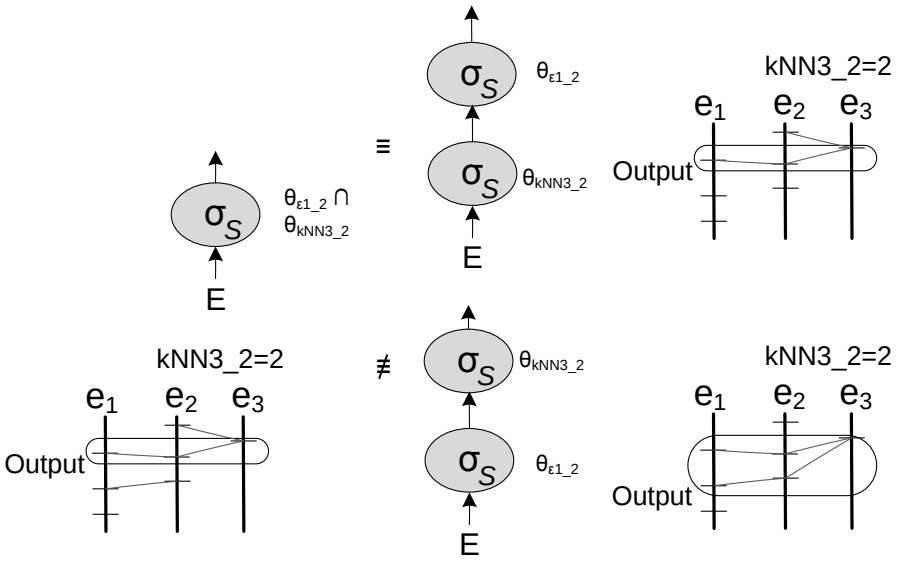


Figure 4-31 Combining/Separating Eps-Join and kNN-Join - When the Attributes in the Predicates do not Have a Single Direction (e1→e2, e2←e3)

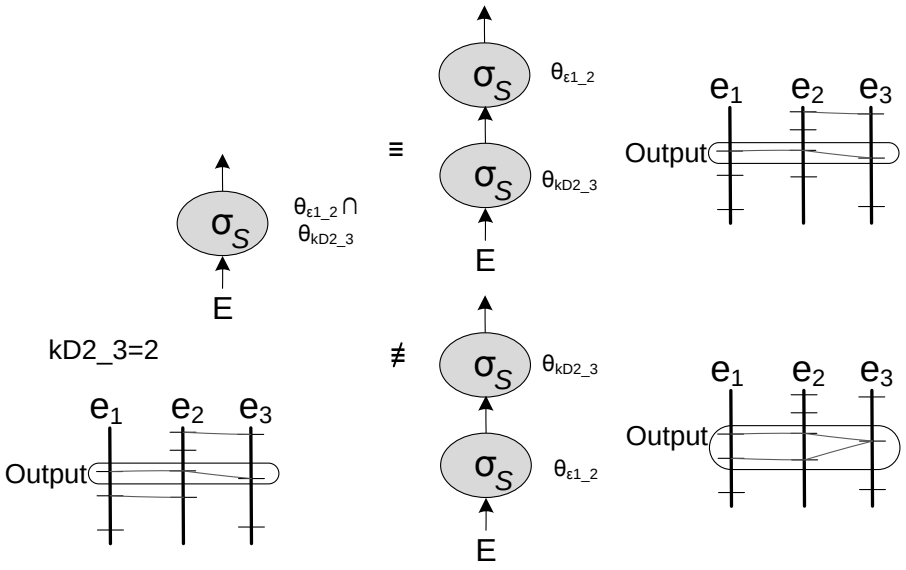


Figure 4-32 Combining/Separating Eps-Join and kD-Join

Rules R48, R49, R50 and R51 specify the way Eps-Join and kD-Join predicates can be combined or separated. The Similarity Join predicates can be separated as long as kD-Join is executed first and Eps-Join is executed on the first join's output. An example of Rules R48 and R49 is presented in Figure 4-32. Given

that both Eps-Join and kD-Join are commutative, the transformation does not depend on whether or not the attributes of the join predicates have a single direction.

According to Rules R52, R53, R54 and R55, kNN-Join and kD-Join predicates cannot be combined or separated in any order. Figure 4-33 shows an example of Rules R52 and R53. This figure shows that the plans that separate the kNN-Join and kD-Join operations generate different results than the plan that combines these predicates. Furthermore, the plans corresponding to the two ways to serialize the operations generate different results.

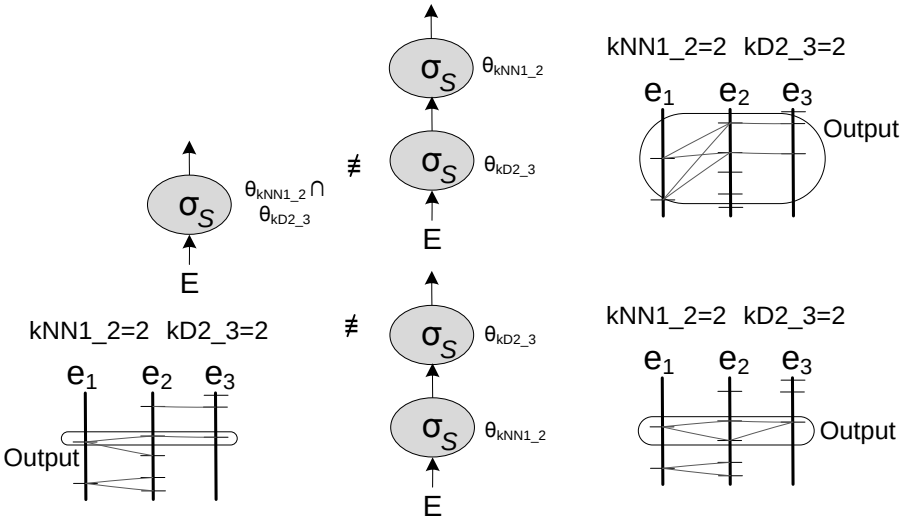


Figure 4-33 Combining/Separating kNN-Join and kD-Join

Rules R56 to R69 specify how Join-Around can be combined with other Similarity Join operations and how join expressions that contain at least one Join-Around predicate can be separated. Join-Around is a hybrid between the kNN-Join with k=1 and the Eps-Join. Therefore, given a specific combination of Join-Around and another type of Similarity Join *Sim-Join*, the equivalence rules for combining or separating Join-Around and *Sim-Join* correspond to the most restrictive rules between combining (1) Eps-Join and *Sim-Join* and (2) kNN-Join and *Sim-Join*.

4.4.2. Other Core Equivalence Rules

4.4.2.1 Commutativity of Similarity Join Operators

Some similarity Join operations are commutative as specified by the following rules. Some additional conditions are given in the description of these rules.

R70. $E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2 \equiv E_2 \bowtie_{\theta_{\varepsilon 2\_1}} E_1;\ where\ \varepsilon 1\_2 = \varepsilon 2\_1$

R71. $E_1 \bowtie_{\theta_{kD1\_2}} E_2 \equiv E_2 \bowtie_{\theta_{kD2\_1}} E_1;\ where\ kD1\_2 = kD2\_1$

R72. $\left(E_1 \bowtie_{\theta_{kNN1\_2}} E_2\right) \not\equiv E_2 \bowtie_{\theta_{kNN2\_1}} E_1;\ where\ kNN1\_2 = kNN2\_1$

R73. $\left(E_1 \bowtie_{\theta_{A1\_2}} E_2\right) \not\equiv E_2 \bowtie_{\theta_{A2\_1}} E_1;\ where\ A1\_2 = A2\_1$

Rules R70 and R71 state that Epsilon-Join and kD-Join are commutative. In addition to the conditions specified in the rules, the distance functions associated to these operations have to be symmetric. Figure 4-34 represent Rules R70 and R71 graphically. Rules R72 and R73 state that kNN-Join and Join-Around are not commutative.

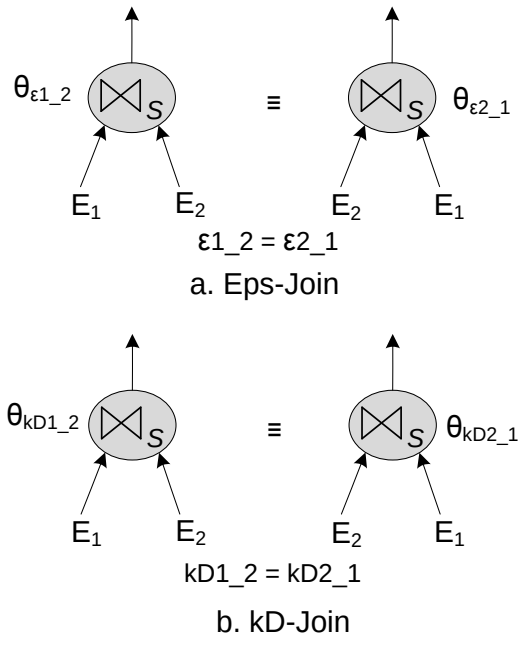


Figure 4-34 Commutativity of Similarity Join Operators

Proof sketch of Rule R70

In the LHS expression of the equivalence, all the join links satisfy dist(e_1,e_2) ≤ ε. Given that the distance function dist is symmetric, dist(e_1,e_2)=dist(e_2,e_1). Consequently, the condition dist(e_2,e_1) ≤ ε in the LHS expression of the equivalence will produce the same set of join links.

4.4.2.2 Distribution of Selection over Similarity Join

The regular selection operation distributes over the Similarity Join operations according to the following rules.

When all the attributes of θ_n involve only the attributes of one of the expressions being joined (E_n):

R74. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_\varepsilon} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_\varepsilon} E_2$

R75. $\sigma_{\theta_2}(E_1 \bowtie_{\theta_\varepsilon} E_2) \equiv E_1 \bowtie_{\theta_\varepsilon} (\sigma_{\theta_2}(E_2))$

R76. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_{kNN}} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_{kNN}} E_2$

R77. $\sigma_{\theta_2}(E_1 \bowtie_{\theta_{kNN}} E_2) \not\equiv E_1 \bowtie_{\theta_{kNN}} (\sigma_{\theta_2}(E_2))$

R78. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_{kD}} E_2) \not\equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_{kD}} E_2$

R79. $\sigma_{\theta_2}(E_1 \bowtie_{\theta_{kD}} E_2) \not\equiv E_1 \bowtie_{\theta_{kD}} (\sigma_{\theta_2}(E_2))$

R80. $\sigma_{\theta_1}(E_1 \bowtie_{\theta_A} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_A} E_2$

R81. $\sigma_{\theta_2}(E_1 \bowtie_{\theta_A} E_2) \not\equiv E_1 \bowtie_{\theta_A} (\sigma_{\theta_2}(E_2))$

When predicates θ_1 and θ_2 involve only the attributes of E_1 and E_2, respectively:

R82. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_\varepsilon} E_2) \equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_\varepsilon} (\sigma_{\theta_2}(E_2))$

R83. $\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta_{kNN}} E_2) \not\equiv (\sigma_{\theta_1}(E_1)) \bowtie_{\theta_{kNN}} (\sigma_{\theta_2}(E_2))$

R84. $\sigma_{\theta_1 \wedge \theta_2}\left(E_1 \bowtie_{\theta_{kD}} E_2\right) \not\equiv \left(\sigma_{\theta_1}(E_1)\right) \bowtie_{\theta_{kD}} \left(\sigma_{\theta_2}(E_2)\right)$

R85. $\sigma_{\theta_1 \wedge \theta_2}\left(E_1 \bowtie_{\theta_A} E_2\right) \not\equiv \left(\sigma_{\theta_1}(E_1)\right) \bowtie_{\theta_A} \left(\sigma_{\theta_2}(E_2)\right)$

According to Rules R74 and R75, the regular selection operation distributes over the Eps-Join operation. Furthermore, the selection operation can be *pushed* under either the outer (R74) or the inner (R75) input of the Eps-Join. Figure 4-35 represents Rule R74 graphically.

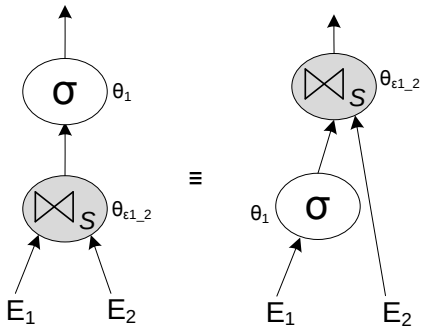


Figure 4-35 Distribution of Selection over Eps-Join

Proof sketch of Rule R74

Assume that the join attributes in $\theta_{\varepsilon 1\_2}$ are $E_1.e_1$ and $E_2.e_2$ and that θ_1 is defined over $E_1.e_1$. Consider a generic tuple t_{E1} of E_1. We will show that for any possible pair (t_{E1},t_{E2}), where t_{E2} is a tuple of E_2, the results generated by the plans of both sides of the rule are the same. The top part of Figure 4-36 shows a graphical representation of Rule R74. The bottom part of Figure 4-36 shows the different possible regions for the values of $t_{E2}.e_2$ and two generic values of $t_{E1}.e_1$. a_2 represents a value that satisfies the predicate θ_1 while a_1 represents a value that does not.

1. When the value of $t_{E1}.e_1$ is a_1. In the LHS plan, the pair (t_{E1},t_{E2}) may or may not belong to the output of the Eps-Join. However, (t_{E1},t_{E2}) will be filtered out by the selection operator since a_1 does not satisfy the predicate θ_1. Thus, no output is generated by this plan. In the RHS plan, t_{E1} is filtered out by the

selection since a_1 does not satisfy θ_1. No tuple flows to the Eps-Join operator form its outer input. Thus, no output is generated by this plan either.

2. When the value of $t_{E1}.e_1$ is a_2 and the value of $t_{E2}.e_2$ belongs to A. In the LHS plan, the pair (t_{E1},t_{E2}) does not belong to the output of the Eps-Join since $dist(t_{E1}.e_1,t_{E2}.e_2)>\varepsilon 1\_2$. No tuple flows to the selection operator. Thus, no output is generated by this plan. In the RHS plan, t_{E1} is selected by the regular selection operator since a_2 satisfies θ_1. However, the pair (t_{E1},t_{E2}) does not belong to the output of the Eps-Join since $dist(t_{E1}.e_1,t_{E2}.e_2)>\varepsilon 1\_2$. Thus, no output is generated by this plan either.

3. When the value of $t_{E1}.e_1$ is a_2 and the value of $t_{E2}.e_2$ belongs to B. In the LHS plan, the pair (t_{E1},t_{E2}) belongs to the output of the Eps-Join since $dist(t_{E1}.e_1,t_{E2}.e_2)\leq\varepsilon 1\_2$. (t_{E1},t_{E2}) is also selected by the regular selection operator since a_2 satisfies θ_1. (t_{E1},t_{E2}) belongs to the output of the LHS plan. In the RHS plan, t_{E1} is selected by the selection operator since a_2 satisfies θ_1. (t_{E1},t_{E2}) belongs to the output of the Eps-Join since $dist(t_{E1}.e_1,t_{E2}.e_2)\leq\varepsilon 1\_2$. Thus, (t_{E1},t_{E2}) belongs also to the output of the RHS plan.

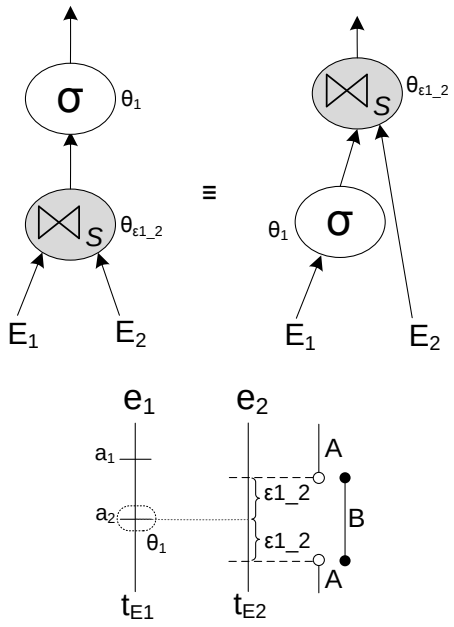


Figure 4-36 Distribution of Selection over Eps-Join – Proof Sketch

Rules R76 and R77 specify the way the regular selection operation distributes over the kNN-Join operation. In this case, the regular selection operation can be pushed only under the outer input of the kNN-Join.

Rules R78 and R79 state that the regular selection operation does not distribute over the kD-Join operation. Figure 4-37 shows an example of Rule R78. This figure shows that the plan that executes the selection after the kD-Join generates an output that is different from that of the plan that pushes selection under the join.

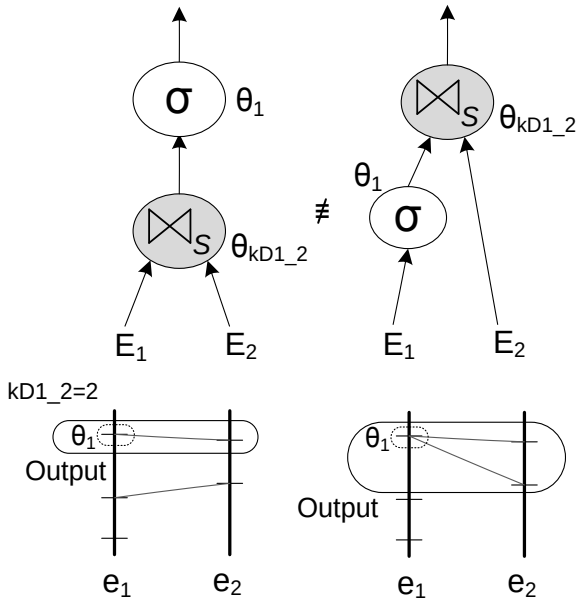


Figure 4-37 Distribution of Selection over kD-Join

Rules R80 and R81 state that the regular selection operation does not distribute over the Join-Around operation. These rules can be explained taking into consideration that Join-Around is a hybrid between the kNN-Join with k=1 and the Eps-Join. The way regular selection distributes over Join-Around corresponds to the most restrictive way in which regular selection distributes over Eps-Join and kNN-Join.

Rules R82, R83, R84 and R85 specify the way the regular selection operation distributes over both inputs of a Similarity Join operation when the selection operations contains two predicates θ_1 and θ_2, and θ_i involves only the attributes of E_i. In this case, regular selection distributes only over Eps-Join. This can be explained considering the rules that specify how selection can be distributed over one input of a Similarity Join (R74 to R81). In these rules, selection can be distributed over either the inner or the outer input of the Similarity Join only in the case of Eps-Join. Figure 4-38 presents Rule 82 graphically.

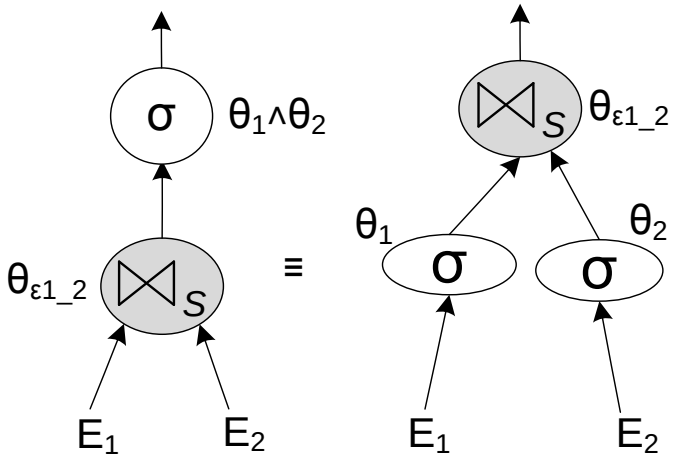


Figure 4-38 Distribution of Selection over Both Inputs of Eps-Join

4.4.2.3 Distribution of Similarity Selection over Join

Similarity Selection operations distribute over the regular join according to the following rules.

R86. $\sigma_{\theta_{\varepsilon 1}}(E_1 \bowtie_\theta E_2) \equiv (\sigma_{\theta_{\varepsilon 1}}(E_1)) \bowtie_\theta E_2$

R87. $\sigma_{\theta_{\varepsilon 2}}(E_1 \bowtie_\theta E_2) \equiv E_1 \bowtie_\theta (\sigma_{\theta_{\varepsilon 2}}(E_2))$

R88. $\sigma_{\theta_{kNN1}}(E_1 \bowtie_\theta E_2) \not\equiv (\sigma_{\theta_{kNN1}}(E_1)) \bowtie_\theta E_2$

R89. $\sigma_{\theta_{kNN2}}(E_1 \bowtie_\theta E_2) \not\equiv E_1 \bowtie_\theta (\sigma_{\theta_{kNN2}}(E_2))$

According to these rules only the Eps-Selection operation distributes over the regular join (R86 and R87). Furthermore, Eps-Selection can be pushed under either the outer (R86) or the inner (R87) input of the join. Figure 4-39 shows Rule R86 graphically. Figure 4-40 presents an example of Rule R88. This figure shows that the plan that executes the kNN-Selection after the join generates an output that is different from that of the plan that pushes the kNN-Selection under the outer input of the join.

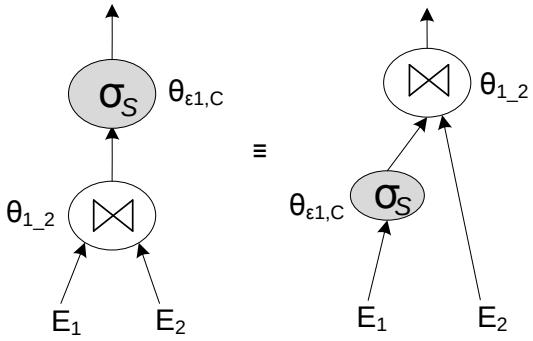


Figure 4-39 Distribution of Eps-Selection over Join

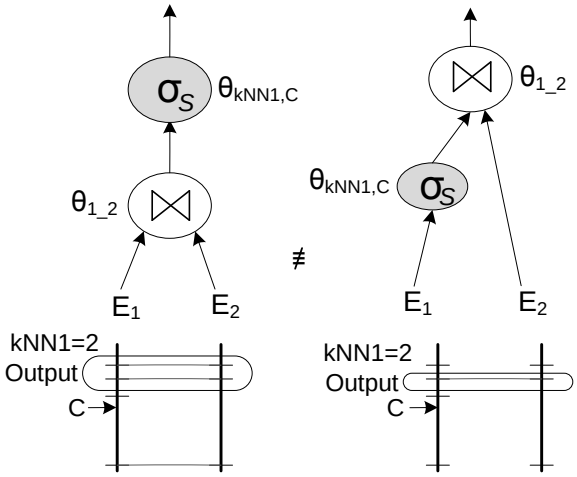


Figure 4-40 Distribution of kNN-Selection over Join

4.4.2.4 Distribution of Similarity Selection over Similarity Join

Similarity Selection operations distribute over Similarity Join operations according to the following rules.

Distribution of Eps-Selection over Eps-Join

R90. $\sigma_{\theta_{\varepsilon 1}}\left(E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2\right) \equiv \left(\sigma_{\theta_{\varepsilon 1}}(E_1)\right) \bowtie_{\theta_{\varepsilon 1\_2}} E_2$

R91. $\sigma_{\theta_{\varepsilon 2}}\left(E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2\right) \equiv E_1 \bowtie_{\theta_{\varepsilon 1\_2}} \left(\sigma_{\theta_{\varepsilon 2}}(E_2)\right)$

Distribution of Eps-Selection over kNN-Join

R92. $\sigma_{\theta_{\varepsilon 1}}\left(E_1 \bowtie_{\theta_{kNN1\_2}} E_2\right) \equiv \left(\sigma_{\theta_{\varepsilon 1}}(E_1)\right) \bowtie_{\theta_{kNN1\_2}} E_2$

R93. $\sigma_{\theta_{\varepsilon 2}}\left(E_1 \bowtie_{\theta_{kNN}} E_2\right) \not\equiv E_1 \bowtie_{\theta_{kNN1\_2}} \left(\sigma_{\theta_{\varepsilon 2}}(E_2)\right)$

Distribution of Eps-Selection over kD-Join

R94. $\sigma_{\theta_{\varepsilon 1}}\left(E_1 \bowtie_{\theta_{kD1\_2}} E_2\right) \not\equiv \left(\sigma_{\theta_{\varepsilon 1}}(E_1)\right) \bowtie_{\theta_{kD1\_2}} E_2$

R95. $\sigma_{\theta_{\varepsilon 2}}\left(E_1 \bowtie_{\theta_{kD1\_2}} E_2\right) \not\equiv E_1 \bowtie_{\theta_{kD1\_2}} \left(\sigma_{\theta_{\varepsilon 2}}(E_2)\right)$

Distribution of kNN-Selection over Eps-Join

R96. $\sigma_{\theta_{kNN1}}\left(E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2\right) \not\equiv \left(\sigma_{\theta_{kNN1}}(E_1)\right) \bowtie_{\theta_{\varepsilon 1\_2}} E_2$

R97. $\sigma_{\theta_{kNN2}}\left(E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2\right) \not\equiv E_1 \bowtie_{\theta_{\varepsilon 1\_2}} \left(\sigma_{\theta_{kNN2}}(E_2)\right)$

Distribution of kNN-Selection over kNN-Join

R98. $\sigma_{\theta_{kNN1}}\left(E_1 \bowtie_{\theta_{kNN1\_2}} E_2\right) \equiv \left(\sigma_{\theta_{kNN1}}(E_1)\right) \bowtie_{\theta_{kNN1\_2}} E_2$

R99. $\sigma_{\theta_{kNN2}}\left(E_1 \bowtie_{\theta_{kNN1\_2}} E_2\right) \not\equiv E_1 \bowtie_{\theta_{kNN1\_2}} \left(\sigma_{\theta_{kNN2}}(E_2)\right)$

Distribution of kNN-Selection over kD-Join

R100. $\sigma_{\theta_{kNN1}}\left(E_1 \bowtie_{\theta_{kD1\_2}} E_2\right) \not\equiv \left(\sigma_{\theta_{kNN1}}(E_1)\right) \bowtie_{\theta_{kD1\_2}} E_2$

R101. $\sigma_{\theta_{kNN2}}\left(E_1 \bowtie_{\theta_{kD1\_2}} E_2\right) \not\equiv E_1 \bowtie_{\theta_{kD1\_2}} \left(\sigma_{\theta_{kNN2}}(E_2)\right)$

Distribution of Eps-Selection over Join-Around

R102. $\sigma_{\theta_{\varepsilon 1}}\left(E_1 \bowtie_{\theta_{A1\_2}} E_2\right) \equiv \left(\sigma_{\theta_{\varepsilon 1}}(E_1)\right) \bowtie_{\theta_{A1\_2}} E_2$

R103. $\sigma_{\theta_{\varepsilon 2}}\left(E_1 \bowtie_{\theta_{A1\_2}} E_2\right) \not\equiv E_1 \bowtie_{\theta_{A1\_2}} \left(\sigma_{\theta_{\varepsilon 2}}(E_2)\right)$

Distribution of kNN-Selection over Join-Around

R104. $\sigma_{\theta_{kNN1}}\left(E_1 \bowtie_{\theta_{A1\_2}} E_2\right) \not\equiv \left(\sigma_{\theta_{kNN1}}(E_1)\right) \bowtie_{\theta_{A1\_2}} E_2$

R105. $\sigma_{\theta_{kNN2}}\left(E_1 \bowtie_{\theta_{A1\_2}} E_2\right) \not\equiv E_1 \bowtie_{\theta_{A1\_2}} \left(\sigma_{\theta_{kNN2}}(E_2)\right)$

According to Rules R90 and R91, the Eps-Selection operation distributes over the Eps-Join operation. The Eps-Selection operation can be pushed under either the outer (R74) or the inner (R75) input of the Eps-Join. Figure 4-41 represents Rule R74 graphically.

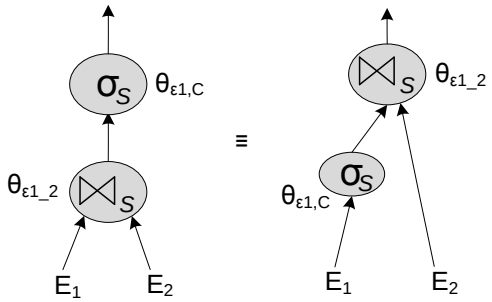


Figure 4-41 Distribution of Eps-Selection over Eps-Join

Rules R92 and R93 specify the way the Eps-Selection operation distributes over the kNN-Join operation. In this case, the Eps-Selection operation can be pushed only under the outer input of the kNN-Join (R92). Figure 4-42 shows an example of the equivalence Rule R92. Figure 4-43 shows an example of Rule R93. This figure shows that the output of the plan that executes the Eps-Selection after the kNN-Join is different from the output of the plan that pushes the Eps-Selection under the inner input of the kNN-Join.

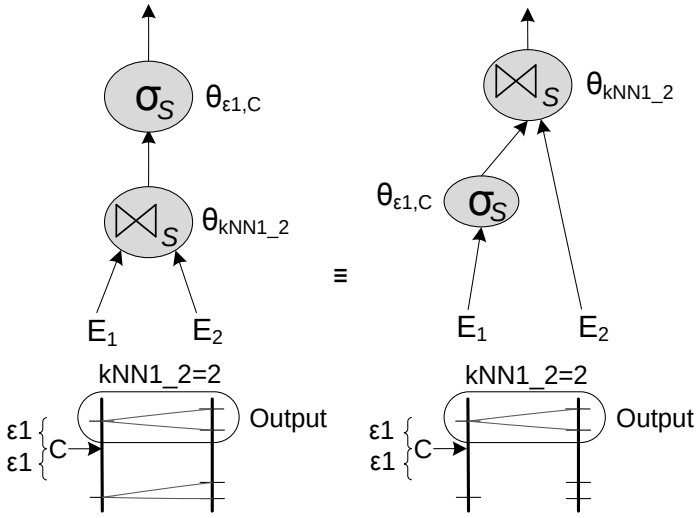


Figure 4-42 Distribution of Eps-Selection over kNN-Join - When Selection is Pushed under the Outer Relation

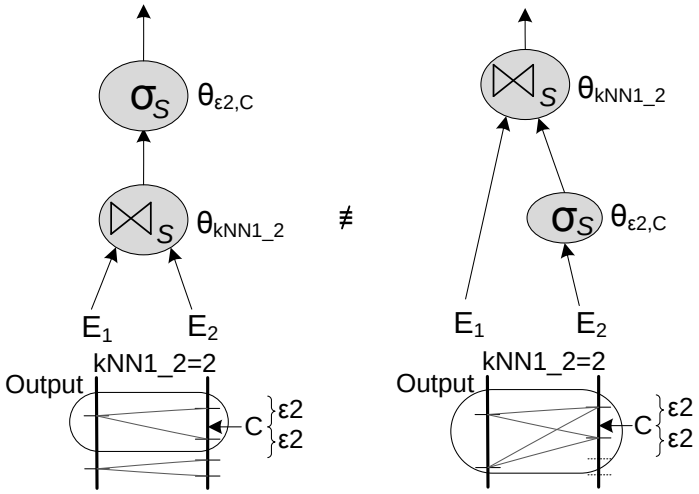


Figure 4-43 Distribution of Eps-Selection over kNN-Join - When Selection is Pushed under the Inner Relation

Rules R94 and R95 specify that the Eps-Selection operation does not distribute over the kD-Join operation. Figure 4-44 shows an example of Rule R94. This figure shows that the plan that executes the Eps-Selection after the kD-Join

generates an output that is different from that of the plan that pushes Eps-Selection under the outer input of the kD-Join.

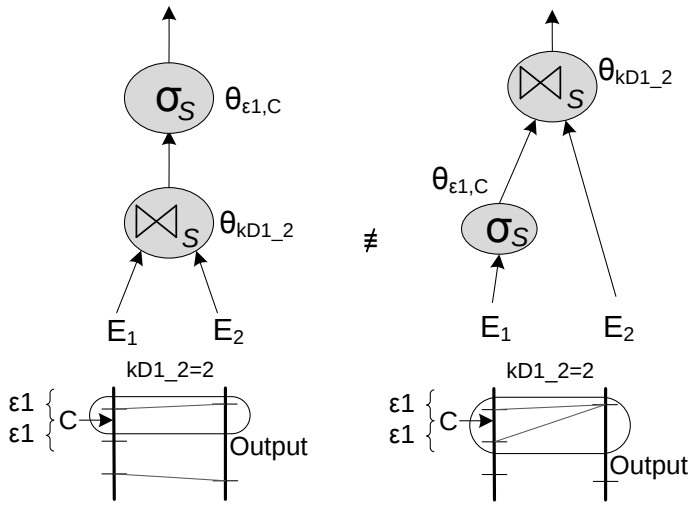


Figure 4-44 Distribution of Eps-Selection over KD-Join

Rules R96 and R97 specify that the kNN-Selection operation does not distribute over the Eps-Join operation. Figure 4-45 shows an example of Rule R96. This figure shows that the output of the plan that executes the kNN-Selection after the Eps-Join is different from the output of the plan that pushes the kNN-Selection under the outer input of the Eps-Join. Intuitively, Rule R97 can be derived from Rule R96 considering that Eps-Join is commutative.

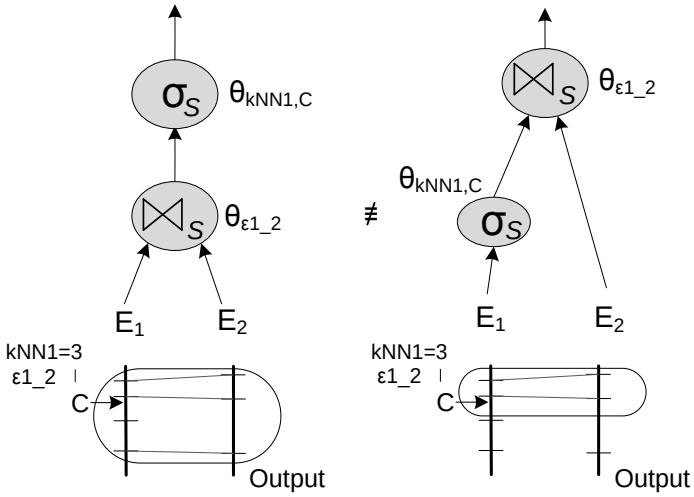


Figure 4-45 Distribution of kNN-Selection over Eps-Join

Rules R98 and R99 specify the way the kNN-Selection operation distributes over the kNN-Join operation. In this case, the kNN-Selection operation can be pushed only under the outer input of the kNN-Join (R98). The reason why the rules depend on whether the selection is pushed under the outer or the inner input is that the kNN-Join operation is not commutative. Figure 4-46 shows an example of Rule R98. Figure 4-47 shows an example of Rule R99. This figure shows that the output of the plan that executes the kNN-Selection after the kNN-Join is different from the output of the plan that pushes the kNN-Selection under the inner input of the kNN-Join.

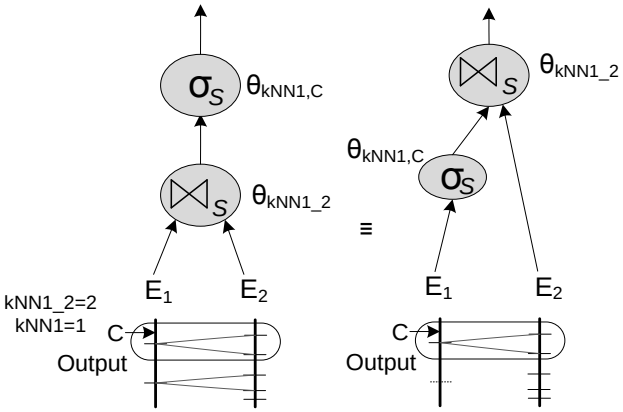


Figure 4-46 Distribution of kNN-Selection over kNN-Join - When selection is Pushed under the Outer Relation

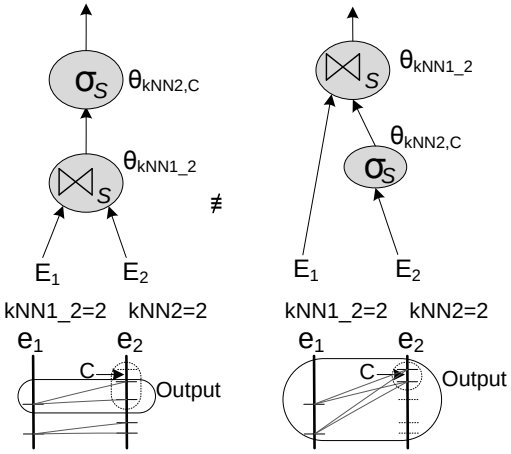


Figure 4-47 Distribution of kNN-Selection over kNN-Join - When selection is Pushed under the Inner Relation

Rules R100 and R101 state that the kNN-Selection operation does not distribute over the kD-Join operation. Figure 4-48 shows an example of Rule R100. In this figure, the output of the plan that executes the kNN-Selection after the kD-Join is different from the output of the plan that pushes the kNN-Selection under the outer input of the kD-Join. Intuitively, Rule R101 can be derived from Rule R100 considering that kD-Join is commutative.

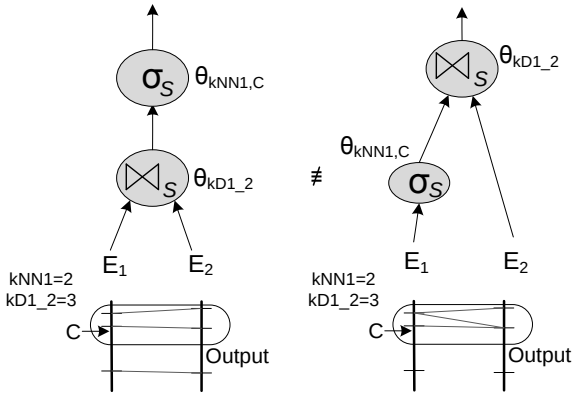


Figure 4-48 Distribution of kNN-Selection over KD-Join

Rules R102 to R105 specify the way Similarity Selection operations distribute over the Join-Around operation. Since Join-Around is a hybrid between kNN-Join and Eps-Join, these rules correspond to the most restricted way in which a given Similarity Selection operation distributes over Eps-Join and kNN-Join.

4.4.2.5. Associativity of Similarity Join Operators

Similarity Join operations are associative according to the following rules.

Associativity of Eps-Join Operators

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R106. $\left(E_1 \ltimes_{\theta_{\varepsilon 1\_2}} E_2\right) \ltimes_{\theta_{\varepsilon 2\_3}} E_3 \equiv E_1 \ltimes_{\theta_{\varepsilon 1\_2}} (E_2 \ltimes_{\theta_{\varepsilon 2\_3}} E_3)$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R107. $E_3 \ltimes_{\theta_{\varepsilon 3\_2}} \left(E_1 \ltimes_{\theta_{\varepsilon 1\_2}} E_2\right) \equiv E_1 \ltimes_{\theta_{\varepsilon 1\_2}} (E_3 \ltimes_{\theta_{\varepsilon 3\_2}} E_2)$

Associativity of kNN-Join Operators

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R108. $\left(E_1 \ltimes_{\theta_{kNN1\_2}} E_2\right) \ltimes_{\theta_{kNN2\_3}} E_3 \equiv E_1 \ltimes_{\theta_{kNN1\_2}} (E_2 \ltimes_{\theta_{kNN2\_3}} E_3)$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R109. $E_3 \bowtie_{\theta_{kNN3\_2}} (E_1 \bowtie_{\theta_{kNN1\_2}} E_2) \not\equiv E_1 \bowtie_{\theta_{kNN1\_2}} (E_3 \bowtie_{\theta_{kNN3\_2}} E_2)$

Associativity of kD-Join Operators

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R110. $(E_1 \bowtie_{\theta_{kD1\_2}} E_2) \bowtie_{\theta_{kD2\_3}} E_3 \not\equiv E_1 \bowtie_{\theta_{kD1\_2}} (E_2 \bowtie_{\theta_{kD2\_3}} E_3)$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R111. $E_3 \bowtie_{\theta_{kD3\_2}} (E_1 \bowtie_{\theta_{kD1\_2}} E_2) \not\equiv E_1 \bowtie_{\theta_{kD1\_2}} (E_3 \bowtie_{\theta_{kD3\_2}} E_2)$

Associativity of Join-Around Operators

When the attributes in the predicates have a single direction (e1→e2, e2→e3)

R112. $(E_1 \bowtie_{\theta_{A1\_2}} E_2) \bowtie_{\theta_{A2\_3}} E_3 \equiv E_1 \bowtie_{\theta_{A1\_2}} (E_2 \bowtie_{\theta_{A2\_3}} E_3)$

When the predicates' attributes do not have a single direction (e1→e2, e2←e3)

R113. $E_3 \bowtie_{\theta_{A3\_2}} (E_1 \bowtie_{\theta_{A1\_2}} E_2) \not\equiv E_1 \bowtie_{\theta_{A1\_2}} (E_3 \bowtie_{\theta_{A3\_2}} E_2)$

Rules R106 and R107 state that Eps-Join operations are associative whether the attributes in the predicates have a single direction or not. Having or not a single direction does not affect the equivalence rules since Eps-Join is commutative. Figure 4-49 shows an example of Rule R106.

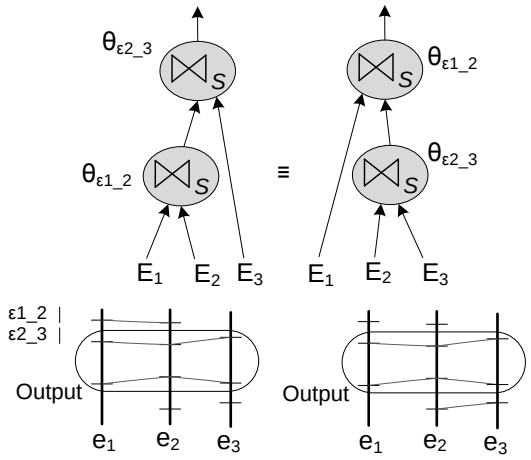


Figure 4-49 Associativity of Eps-Join Operators

Rules R108 and R109 specify when kNN-Join operations are associative. As expected, given that kNN-Join is not commutative, the rules depend on whether or not the attributes in the predicates have a single direction. kNN-Join operations are associative only when the predicate attributes have a single direction (R108). Figure 4-50 shows an example of Rule R108. Figure 4-51 shows an example of Rule R109. This figure shows that the order of evaluation of multiple kNN-Join operations with predicate attributes that do not have a single direction affects the final results.

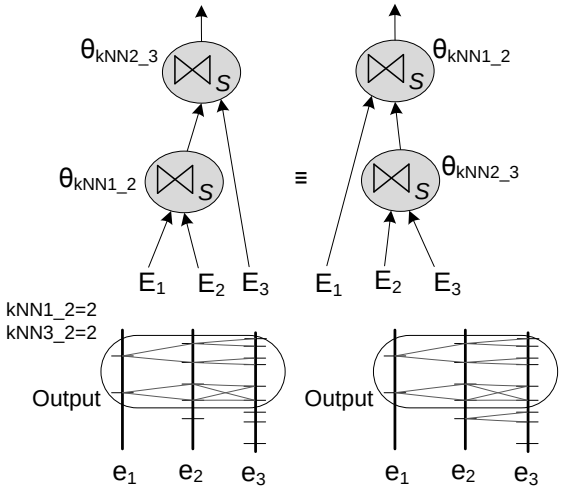


Figure 4-50 Associativity of kNN-Join Operators - When Attributes in Predicates Have a Single Direction: e1→e2, e2→e3

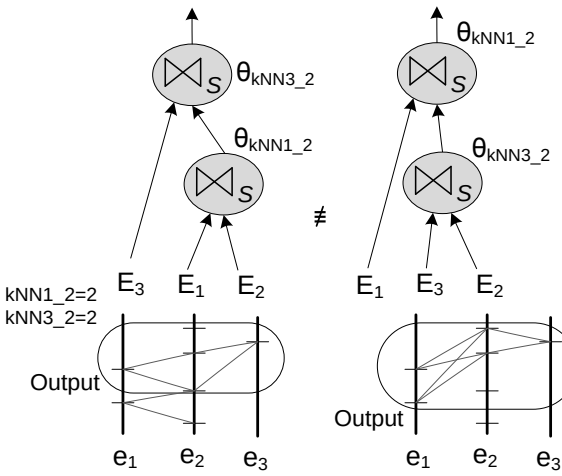


Figure 4-51 Associativity of kNN-Join Operators - When Attributes in Predicates do not have a Single Direction: e1→e2, e2←e3

Proof sketch of Rule R108

Assume that the join attributes in $\theta_{\varepsilon 1\_2}$ are $E_1.e_1$ and $E_2.e_2$ and the join attributes in $\theta_{\varepsilon 2\_3}$ are $E_2.e_2$ and $E_3.e_3$. Consider a generic tuple t_{E1} of E_1. We will show that for any possible triplet (t_{E1},t_{E2},t_{E3}), where t_{E2} is a tuple of E_2 and t_{E3} is a tuple of E_3, the results generated by the plans of both sides of the rule are the same. The top part of Figure 4-52 shows a graphical representation of Rule R108. The bottom part of Figure 4-52 shows the different possible regions for the values of $t_{E2}.e_2$ and $t_{E3}.e_3$. Note that the regions for $t_{E3}.e_3$ have been specified based on a generic tuple t_{E2} with $t_{E2}.e_2$ in region B. The region marked as *kNN1_2* represents the segment that contains the *kNN1_2* closest neighbors of t_{E1} in E_2. The region marked as *kNN2_3* represents the segment that contains the *kNN2_3* closest neighbors of t_{E2} in E_3. Note that for a given kNN-Join (θ_{kNN1_2} or θ_{kNN2_3}) and a given outer tuple *t*, the join identifies the same set of *k* nearest neighbors of *t* in both plans. This is the case since (1) kNN-Join over R_1 and R_2 makes use of primary keys in both input relations ($R_1.pk_1$, $R_2.pk_2$) and ignores tuples in R_2 that have the same primary key, and (2) the set of different values of $R_2.pk_2$ in the inner input of both plans is the same. Furthermore, note that the set of different values of $R_2.pk_2$ in the inner input of both plans corresponds to the set of all different values of $R_2.pk_2$ in the base relation R_2.

1. When the value of $t_{E2}.e_2$ belongs to B and the value of $t_{E3}.e_3$ belongs to D. In the LHS plan, the pair (t_{E1},t_{E2}) belongs to the output of the bottom kNN-Join (θ_{kNN1_2}) since t_{E2} is one of the *kNN1_2* closest neighbors of t_{E1} in E_2. (t_{E1},t_{E2}) flows to the top kNN-Join. The triplet (t_{E1},t_{E2},t_{E3}) belongs also to the output of the top kNN-Join (θ_{kNN2_3}) since t_{E3} is one of the *kNN2_3* closest neighbors of t_{E2} in E_3. Consequently, (t_{E1},t_{E2},t_{E3}) belongs to the output of the LHS plan. In the RHS plan, (t_{E2},t_{E3}) belongs to the output of the bottom kNN-Join (θ_{kNN2_3}) since t_{E3} is one of the *kNN2_3* closest neighbors of t_{E2} in E_3. The triplet (t_{E1},t_{E2},t_{E3}) belongs also to the output of the top kNN-Join (θ_{kNN1_2}) since t_{E2} is one of the *kNN1_2* closest neighbors of t_{E1} in E_2. Thus, (t_{E1},t_{E2},t_{E3}) belongs also to the output of the RHS plan. Note that in the RHS plan, the bottom join (θ_{kNN2_3}) matches each inner tuple of E_2 to its closes *kNN2_3* neighbors in E_3. The output of this join will contain all the values of $E_2.pk_2$ (the primary key of E_2) in the base relation E_2. Consequently, the set of all different values of $E_2.pk_2$ in the inner input of θ_{kNN1_2} is the same in both plans. Therefore, for a given inner tuple *t*, the join θ_{kNN1_2} will find the same set of *kNN1_2* nearest neighbors of *t* in both plans.

2. When the value of $t_{E2}.e_2$ belongs to B and the value of $t_{E3}.e_3$ belongs to C. In the LHS plan, the pair (t_{E1},t_{E2}) belongs to the output of the bottom kNN-Join (θ_{kNN1_2}) since t_{E2} is one of the *kNN1_2* closest neighbors of t_{E1} in E_2. (t_{E1},t_{E2}) flows to the top kNN-Join. However, the triplet (t_{E1},t_{E2},t_{E3}) does not belong to the output of the top kNN-Join (θ_{kNN2_3}) since t_{E3} is not one of the *kNN2_3* closest neighbors of t_{E2} in E_3. Consequently, no output is generated by this plan. In the RHS plan, (t_{E2},t_{E3}) does not belongs to the output of the bottom kNN-Join (θ_{kNN2_3}) since t_{E3} is not one of the *kNN2_3* closest neighbors of t_{E2} in E_3. No tuple flows to the top join. Thus, no output is generated by this plan either.

3. When the value of $t_{E2}.e_2$ belongs to A. In the LHS plan, the pair (t_{E1},t_{E2}) does not belongs to the output of the bottom kNN-Join (θ_{kNN1_2}) since t_{E2} is not one of the *kNN1_2* closest neighbors of t_{E1} in E_2. No tuple flows to the top join. Consequently no output is generated by this plan. In the RHS plan, (t_{E2},t_{E3}) may

or may not belong to the output of the bottom kNN-Join (θ_{kNN2_3}). However, any triplet (t_{E1},t_{E2},t_{E3}) does not belong to the output of the top kNN-Join (θ_{kNN1_2}) since t_{E2} is not one of the *kNN1_2* closest neighbors of t_{E1} in E_2. Thus, no output is generated by this plan either.

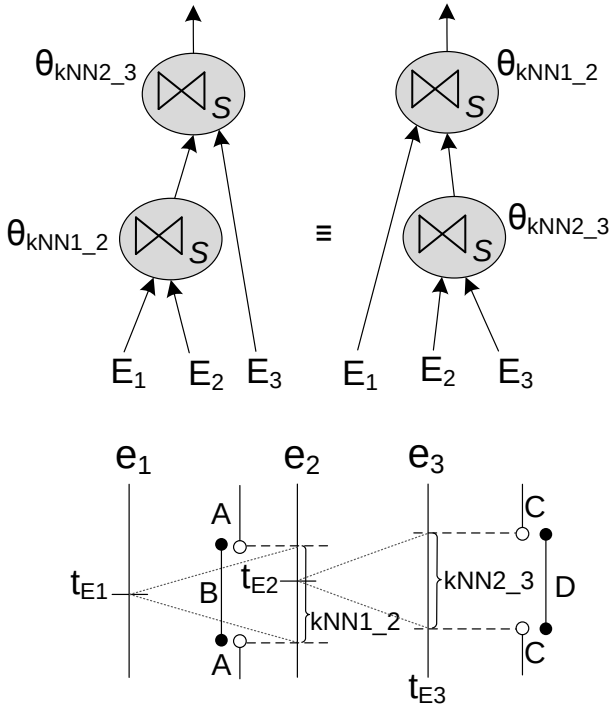


Figure 4-52 Associativity of kNN-Join Operators - When Attributes in Predicates Have a Single Direction: e1→e2, e2→e3 – Proof Sketch

Rules R110 and R111 state that multiple kD-Join operations are not associative. Given that kD-Join is commutative, the transformations do not depend on whether or not the attributes of the join predicates have a single direction. Figure 4-53 shows an example of Rule R110. This figure shows that plans with different evaluation order of the kD-Join operations generate different results.

Rules R112 and R113 specify when multiple Join-Around operations are associative. Since Join Around is a hybrid between Eps-Join and kNN-Join, these rules correspond to the most restrictive rules among the counterpart rules for Eps-Join and kNN-Join.

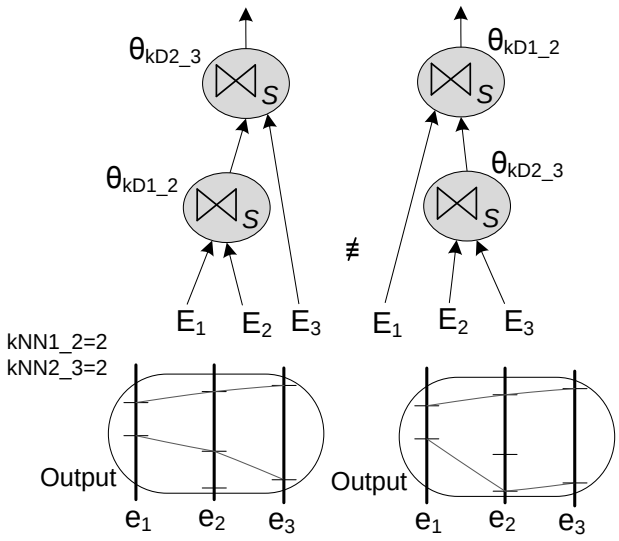


Figure 4-53 Associativity of kD-Join Operators

4.4.3. Rules that use Distance Function properties

This section presents some equivalence rules that take advantage of properties of the distance functions used by the similarity-aware operators. The rules in this section specify explicitly the attributes that are involved in each similarity-aware operation. The selection predicate $\theta_{S,C}(e)$ specifies that the selection condition is applied on the attribute e. The join predicate $e_1\ \theta_S\ e_2$ specifies that e_1 and e_2 are the outer and inner join attributes respectively.

4.4.3.1. Pushing Selection Predicate under Originally Unrelated Eps-Join Operand

In the equivalence rules presented in Section 4.4.2.2, each selection predicate θ is pushed only under the join operand that contains the attribute referenced in θ. In the case of the Ɛ-Join operator, the filtering benefits of pushing selection under the join can be further improved by pushing θ or a variant of it under both operands of the Eps-Join as shown in the following equivalence rule.

R114. $\sigma_{\theta(e_1)}\left(E_1 \bowtie_{e_1\ \theta_\varepsilon\ e_2} E_2\right) \equiv (\sigma_{\theta(e_1)}(E_1)) \bowtie_{e_1\ \theta_\varepsilon\ e_2} (\sigma_{\theta\pm\varepsilon(e_2)}(E_2))$

where (1) the distance function satisfies the properties: Triangular Inequality, Symmetry, and Identity of Indiscernibles; and (2) the selection predicate $\theta\pm\varepsilon$

represents a modified version of θ where each condition is *extended* by $\mathcal{E}$ and is applied on e_2, the join attribute of E_2. $\theta \pm \mathcal{E}$ uses the same distance function used in θ. For example, if $\theta = 10 \leq e_1 \leq 20$, then $\theta \pm \mathcal{E} = 10 - \mathcal{E} \leq e_2 \leq 20 + \mathcal{E}$. This rule is represented graphically in Figure 4-54.

Proof sketch of Rule R114

Notice that pushing the selection operation under the outer input of the Eps-Join has been already studied in Rule R74. We focus here on the validity of pushing the selection operation under the inner input of the Eps-Join. Consider the case of 1D data. Assume that in the LHS part of Rule R114, the selection predicate θ is $e_1 = 10$ and the Eps-Join predicate $e_1 \ \theta_\varepsilon \ e_2$ is $dist(e1,e2) \leq \varepsilon$.

1. Since *dist* satisfies Identity of Indiscernibles, we know that $dist(e_1,10) = 0$.

2. *dist* also satisfies Triangular Inequality, consequently $dist(10,e_2) \leq dist(10,e_1) + dist(e_1,e_2)$.

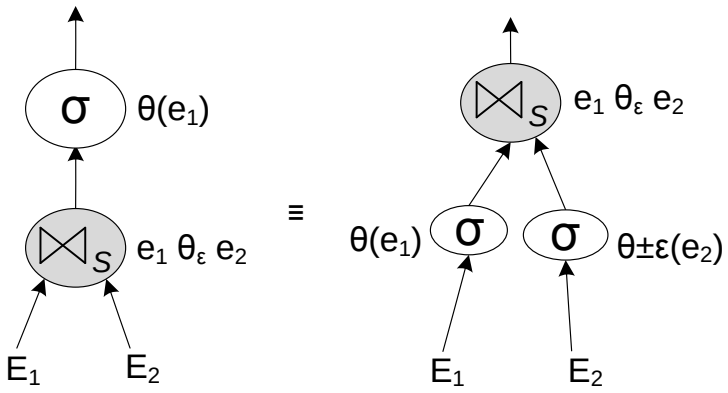


Figure 4-54 Pushing Selection Predicate under Originally Unrelated Eps-Join Operand

3. Due to Commutativity, we have that $dist(10,e_2) \leq dist(e_1,10) + dist(e_1,e_2)$.

4. Replacing (1) in (3), $dist(10,e_2) \leq 0 + dist(e_1,e_2) \leq dist(e_1,e_2)$.

5. Using in (4) the fact that $dist(e_1,e_2) \leq \varepsilon$, $dist(10,e_2) \leq \varepsilon$.

The expression in (5) $dist(10,e_2) \leq \varepsilon$ represents a selection predicate that can be applied on e_2. This predicate is in fact the predicate being applied on e_2 in the inner input of the RHS part of Rule R114. We could extend this analysis to other types of selection conditions.

4.4.3.2. Pushing Eps-Selection Predicate under Originally Unrelated Eps-Join Operand

Section 4.4.2.4 presented multiple rules that enabled pushing Similarity Selection predicates θ_S under the Similarity Join operand that contains the attribute referenced in θ_S. In the case of Eps-Join and Eps-Selection, the filtering benefits of pushing a Similarity Selection predicate θ_S can be further improved by pushing θ_S under one join operand and a variant of θ_S under the other join operand as shown in the following equivalence rule.

R115. $\sigma_{\theta_{\varepsilon 1,C}(e_1)}\left(E_1 \bowtie_{e_1\ \theta_{\varepsilon 2}\ e_2} E_2\right) \equiv \left(\sigma_{\theta_{\varepsilon 1,C}(e_1)}(E_1)\right) \bowtie_{e_1\ \theta_{\varepsilon 2}\ e_2} \left(\sigma_{\theta_{(\varepsilon 1+\varepsilon 2),C}(e_2)}(E_2)\right)$

where (1) all Eps-Selection and Eps-Join operators use the same distance function; (2) the distance function satisfies the Triangular Inequality and Symmetry properties; and (3) the selection predicate $\theta_{(\varepsilon 1+\varepsilon 2),C}$ represents an Eps-Selection predicate with a value of ε equal to $\varepsilon 1 + \varepsilon 2$, *where* $\varepsilon 1$ and $\varepsilon 2$ are the values of epsilon used in the Eps-Selection and Eps-Join operators, respectively. For example, if $\theta_{\varepsilon 1,C}$ is $dist(e_1,C) \leq 10$, *and* $\theta_{\varepsilon 2}$ is $dist(e_1,e_2) \leq 5$, *then* $\theta_{(\varepsilon 1+\varepsilon 2),C}$ is dist (e_2,C) ≤*15*. This rule is represented graphically in Figure 4-55.

Proof sketch of Rule R115

Notice that pushing Eps-Selection under the outer input of the Eps-Join has been already studied in Rule R90. We focus here on the validity of pushing the Eps-Selection operation under the inner input of the Eps-Join. Consider the case of 1D data. Assume that the selection predicate $\theta_{\varepsilon 1,C1}$ is $dist(e_1,C1) \leq \varepsilon 1$ and the join predicate $\theta_{\varepsilon 2}$ is $dist(e_1,e_2) \leq \varepsilon 2$.

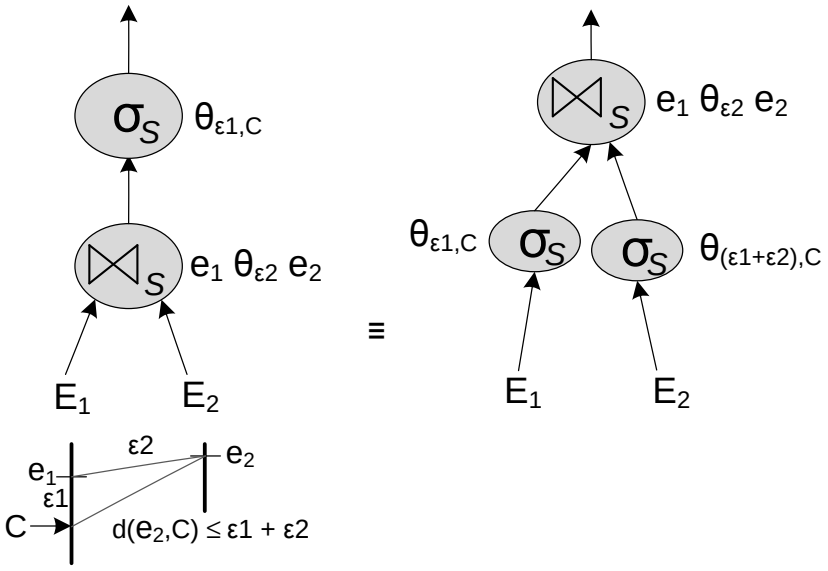


Figure 4-55 Pushing Eps-Selection Predicate under Originally Unrelated Eps-Join Operand

1. Due to Triangular Inequality, $dist(e_2,C1) \leq dist(e_2,e_1) + dist(e_1,C1)$.

2. Due to Commutativity, we have that $dist(e_2,C1) \leq dist(e_1,e_2) + dist(e_1,C1)$.

3. Using in (2) the fact that $dist(e_1,e_2) \leq \varepsilon 2$, $dist(e_2,C1) \leq \varepsilon 2 + dist(e_1,C1)$.

4. Using in (3) the fact that $dist(e_1,C1) \leq \varepsilon 1$, $dist(e_2,C1) \leq \varepsilon 1 + \varepsilon 2$.

The expression in (4) $dist(e_2,C1) \leq \varepsilon 1 + \varepsilon 2$ represents an Eps-Selection predicate that can be applied on e_2. This predicate is in fact the predicate being applied on e_2 in the inner input of the RHS part of Rule R115.

4.4.3.3. Associativity Rule that Enables Join on Originally Unrelated Attributes

In the Associativity rules presented in Section 4.4.2.5, each Similarity Join predicate involves the same attributes in both sides of the rule. In the case of Ɛ-Join, when the attributes e_1 of E_1 and e_2 of E_2 are joined using $\mathcal{E}1$ and the result joined with attribute e_3 of E_3 using $\mathcal{E}2$, there is an implicit relationship between e_1 and e_3 that is exploited by the following equivalence rule.

R116. $\left(E_1 \bowtie_{e_1\ \theta_{\varepsilon 1}\ e_2} E_2\right) \bowtie_{e_2\ \theta_{\varepsilon 2}\ e_3} E_3 \equiv \left(E_1 \bowtie_{e_1\ \theta_{\varepsilon 1+\varepsilon 2}\ e_3} E_3\right) \bowtie_{(e_1\ \theta_{\varepsilon 1}\ e_2)\wedge(e_2\ \theta_{\varepsilon 2}\ e_3)} E_2$

where (1) all the Eps-Join operators use the same distance function; (2) the distance function satisfies the Triangular Inequality and Symmetry properties; and (3) the predicate $\theta_{\varepsilon 1+\varepsilon 2}$ represents an Eps-Join predicate with a value of ε equal to $\varepsilon 1 + \varepsilon 2$, *where* $\varepsilon 1$ and $\varepsilon 2$ are the values of epsilon used in the Eps-Join operators of the RHS part of the rule. For example, if $\theta_{\varepsilon 1}$ is $dist(e_1,e_2) \leq 10$, *and* $\theta_{\varepsilon 2}$ is $dist(e_2,e_3) \leq 5$, *then* $\theta_{\varepsilon 1+\varepsilon 2}$ is $dist(e_1,e_3) \leq 15$. This rule is represented graphically in Figure 4-56.

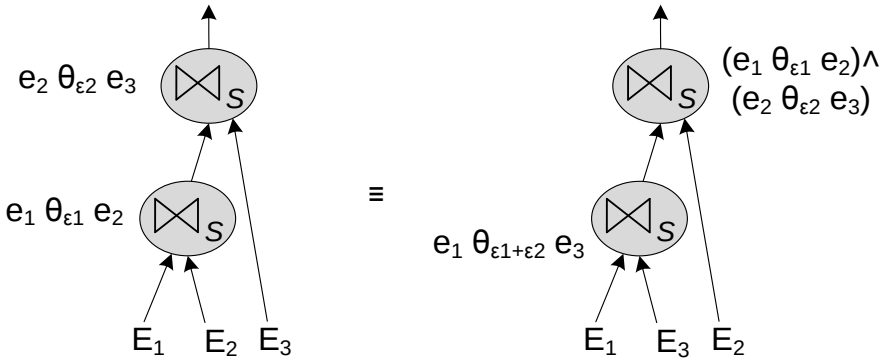


Figure 4-56 Eps-Join Associativity that Enables Join on Originally Unrelated Attributes

Proof sketch of Rule R116

Assume that in the LHS part of Rule R116, the join predicate $\theta_{\varepsilon 1}$ is $dist(e_1,e_2) \leq \varepsilon 1$, and the join predicate $\theta_{\varepsilon 2}$ is $dist(e_2,e_3) \leq \varepsilon 2$. The order of attributes in these expressions is irrelevant because the distance function is Commutative.

1. Due to Triangular Inequality, $dist(e_1,e_3) \leq dist(e_1,e_2)+dist(e_2,e_3)$.

2. Since $dist(e_1,e_2) \leq \varepsilon 1$ and $dist(e_2,e_3) \leq \varepsilon 2$, $dist(e_1,e_3) \leq \varepsilon_1+\varepsilon_2$.

The expression in (2) $dist(e_1,e_3) \leq \varepsilon_1+\varepsilon_2$ represents a join predicate that can be applied on e_2 and e_3.This predicate is in fact the predicate being applied on e_2 and e_3 in the left join of the RHS part of Rule R116. Notice that the RHS part of

the rule requires a second join that applies the two join predicates of the LHS part because some tuples that do not satisfy these predicates can be present in the output of the join between e_1 and e_3.

4.4.3.4. Applicability of Rules for Common Distance Functions

Sections 4.4.3.1, 4.4.3.2, and 4.4.3.3 presented three equivalence rules that take advantage of specific properties of the distance functions used by the similarity-aware operators (Rules R114, R115, and R116). Many distance functions are used in practice, where each distance function can be used with certain types of data, e.g., numeric, text, vector data, etc. Tables 4-1 and 4-2 present several common distance functions and the equivalence rules that can be used with each of them. These tables also present the definition of each distance function and the data types they can be used with.

4.4.4. Examples of the Use of Transformation Rules

The equivalence rules presented in Section 4.4 allow the transformation of similarity query plans into equivalent plans with possibly smaller execution times. Particularly, these rules can be used to transform the conceptual evaluation plan of a similarity query into more efficient equivalent plans. This section presents examples of this type of query transformations.

Figure 4-57 shows the SQL version of a similarity query with Eps-Selection and Eps-Join predicates. The left plan in this figure shows the conceptual evaluation plan of this query. The right plan shows an equivalent plan with potentially better execution time (since each relation is read only once and the Similarity Selection is pushed under the Similarity Join). The following steps show how the query expression of the left plan can be transformed into the one of the right plan.

1. $\sigma_{\theta_{\varepsilon 1\_2}}(E_1 \times E_2) \cap \sigma_{\theta_{\varepsilon 1,\mathrm{C}}}(E_1 \times E_2)$

2. $\equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{\varepsilon 1,\mathrm{C}}}(E_1 \times E_2))$, since Eps-Selection and Eps-Join can be separated or combined (Rule R10)

Table 4-1 Common Distance Functions 1

Distance Function	Definition	Supported Data Types				Applicable Rules		
		Text	Numeric	Vector	Time Series	114	115	116
p-norm distance	p-norm distance of two vectors (x_1, x_2, ...,x_n) and (y_1, y_2, ...,y_n) is defined as: 1-norm distance = $\sum_{i=1}^{n} \lvert x_i - y_i \rvert$ 2-norm distance = $\left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^2\right)^{1/2}$ p-norm distance = $\left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p\right)^{1/p}$ infinity-norm distance = $\lim_{p\to\infty} \left(\sum_{i=1}^{n} \lvert x_i - y_i \rvert^p\right)^{1/p}$		X	X	X	X	X	X
Cosine Distance 1	CD1(A,B) = 1 - CS(A,B), where A and B are vectors and is the CS(A,B) Cosine Similarity. CS(A,B) = (A·B)/(‖A‖‖B‖)			X	X			
Cosine Distance 2	Cosine Distance 2 CD2(A,B)=arccos(CS(A,B))			X	X	X	X	X
Discrete Metric Function	DM(x,y) = 0 if x = y, 1 otherwise, where x and y are numbers.		X			X	X	X
Longest Common Subsequence	LCS(X,Y) = longest subsequence common to strings or time series X and Y.	X			X			

Table 4-2 Common Distance Functions 2

Distance Function	Definition	Supported Data Types				Applicable Rules		
		Text	Numeric	Vector	Time Series	114	115	116
Edit Distance with Equal Weights	ED(X,Y) = minimum number of operations needed to transform string X into string Y. Allowed operations: insertion, deletion, and substitution of a single character.	X				X	X	X
Edit Distance with Different Weights	ED(X,Y) = min(w(E)), where E is a sequence of edit operations that transforms string X into string Y, and w is a weight function that assigns a nonnegative real number w(x, y) to each elementary edit operation.	X						
Hamming Distance	HD(X,Y) = number of positions in which the characters of strings X and Y are different.	X				X	X	X
Jaccard Distance	JD(A,B) = 1- JS(A,B), where JS(A,B) = (|A∩B|/|AUB|). A and B are two generic sets. For string data, JS(A,B) = number of shared tokens/total number of tokens. For vector data, JS(A,B)=number of matching cells/total number of cells.	X		X		X	X	X

3. $\equiv \sigma_{\theta_{\varepsilon 1\_2}}(\sigma_{\theta_{\varepsilon 1,\mathrm{C}}}(E_1) \times E_2)$, since Eps-Selection distributes over Eps-Join (Rule R86)

4. $\equiv \sigma_{\theta_{\varepsilon 1,\mathrm{C}}}(E_1) \bowtie_{\theta_{\varepsilon 1\_2}} E_2$, since Eps-Selection and cross product can be combined (Rule R1)

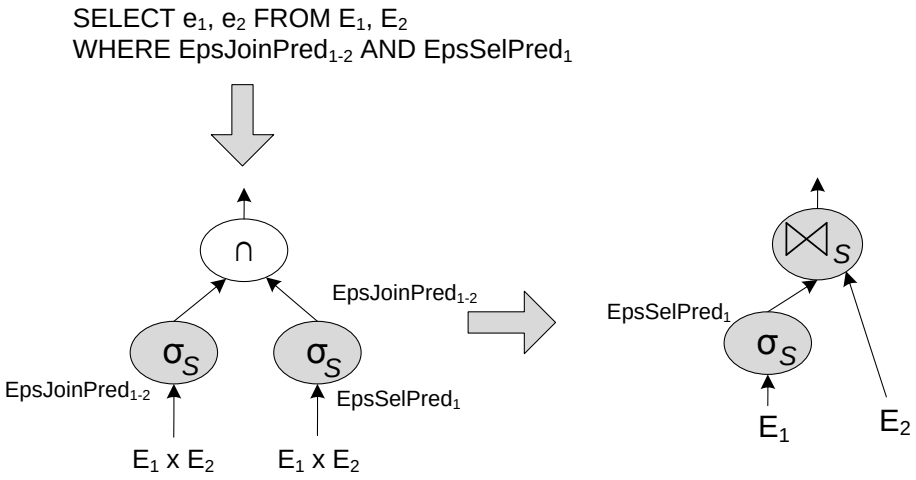


Figure 4-57 Transformation of Query with Eps-Selection and Eps-Join Predicates

Figure 4-58 shows the SQL version of a similarity query two Eps-Join predicates. The left plan in this figure is the conceptual evaluation plan of the query while the right one shows an equivalent plan with potentially better execution time (since each relation is read only once and only the tuples that satisfy the bottom join flow to the outer input of the top join). The following steps show how the query expression of the left plan can be transformed into the one of the right plan.

1. $\sigma_{\theta_{\varepsilon 1\_2}}((E_1 \times E_2) \times E_3) \cap \sigma_{\theta_{\varepsilon 2\_3}}((E_1 \times E_2) \times E_3)$

2. $\sigma_{\theta_{\varepsilon 2\_3}}(\sigma_{\theta_{\varepsilon 1\_2}}((E_1 \times E_2) \times E_3))$, since two Eps-Join predicates can be separated or combined (Rule R36)

3. $\sigma_{\theta_{\varepsilon 2\_3}}(\sigma_{\theta_{\varepsilon 1\_2}}(E_1 \times E_2) \times E_3)$, since Eps-Selection distributes over Eps-Join (Rule R86)

4. $\sigma_{\theta_{\varepsilon 2\_3}}\left((E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2) \times E_3\right)$, since Eps-Selection and cross product can be combined (Rule R1)

5. $(E_1 \bowtie_{\theta_{\varepsilon 1\_2}} E_2) \bowtie_{\theta_{\varepsilon 2\_3}} E_3$, since Eps-Selection and cross product can be combined (Rule R1)

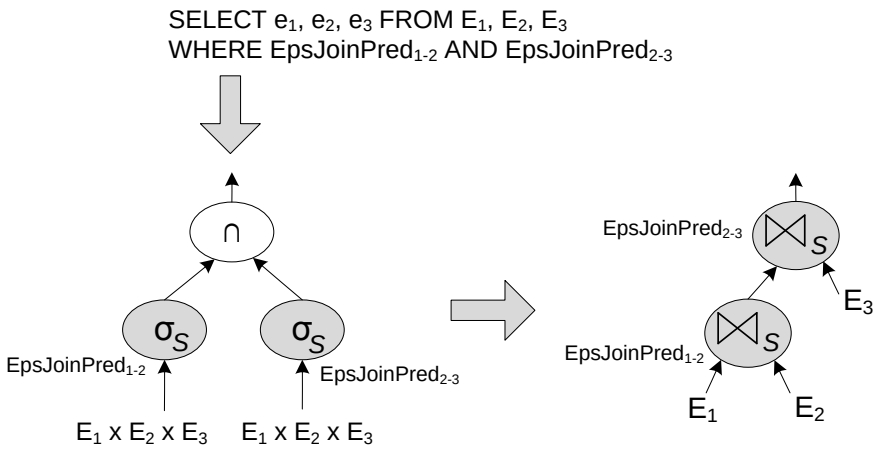


Figure 4-58 Transformation of Query with Multiple Eps-Join Predicates

Figures 4-59, 4-60, 4-61 and 4-62 show examples of more complex similarity query transformations. The final plan presented in these examples can be derived from the corresponding conceptual evaluation plans using the equivalence rules presented in this chapter and rules that generalize them. These queries also show several key general transformation guidelines for similarity query optimization.

Figure 4-59 shows the transformation of a query with multiple Similarity Selection predicates. This figure shows that multiple Eps-Selection operators over the same attribute can be serialized. Multiple kNN-Selection operators cannot be serialized; they need to be executed independently and their results combined using the intersection operator. Eps-Selection and kNN-Selection operations over the same attribute can be serialized executing the kNN-Selection operations first.

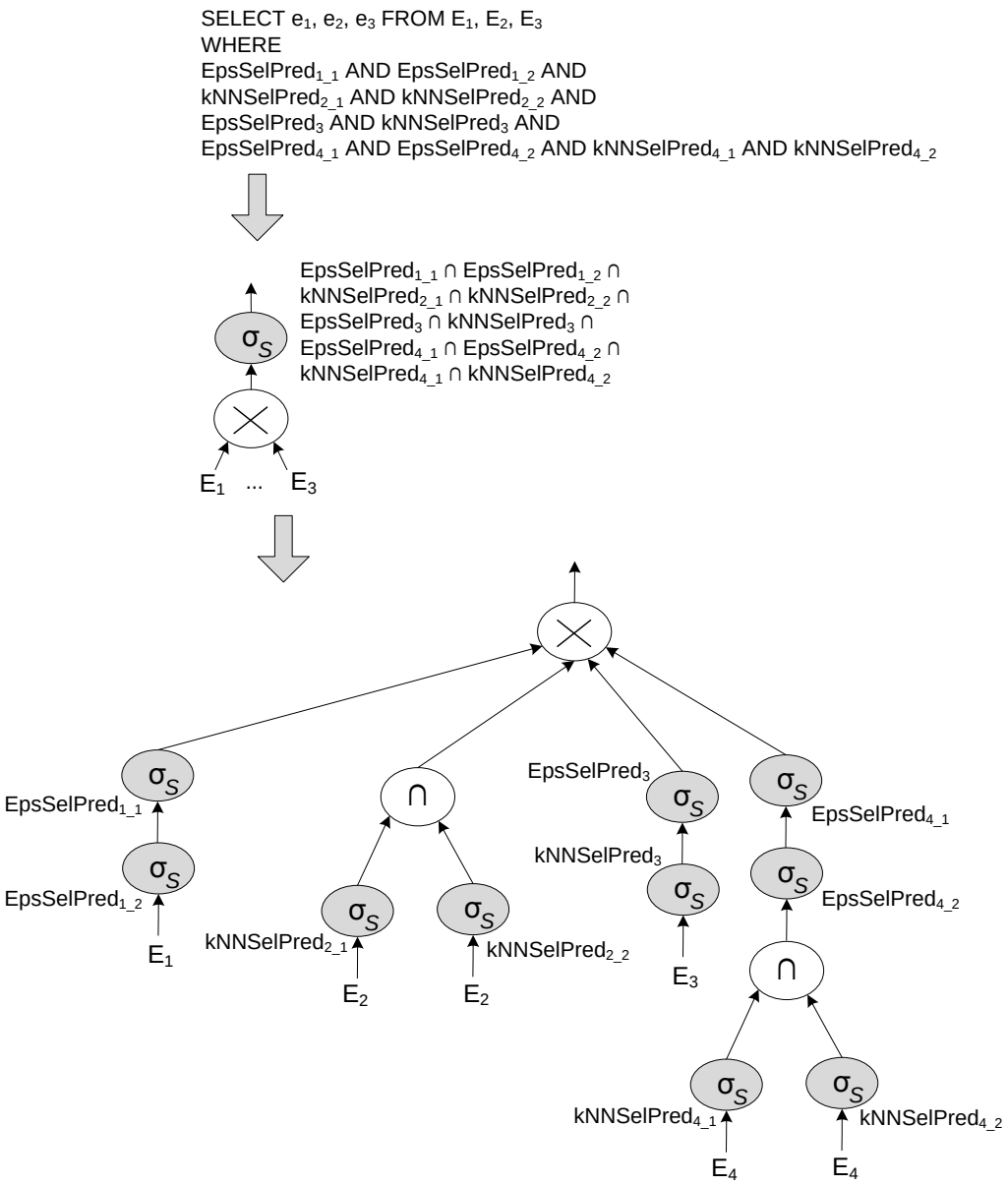


Figure 4-59 Query with Multiple Similarity Selection Predicates

Figure 4-60 shows the transformation of a query with multiple Eps-Join and Similarity Selection predicates. This figure shows that Eps-Selection and kNN-Selection operations can be pushed under any input of an Eps-Join. Multiple Eps-Join operations can be serialized, i.e., the results of a join are sent to the next one.

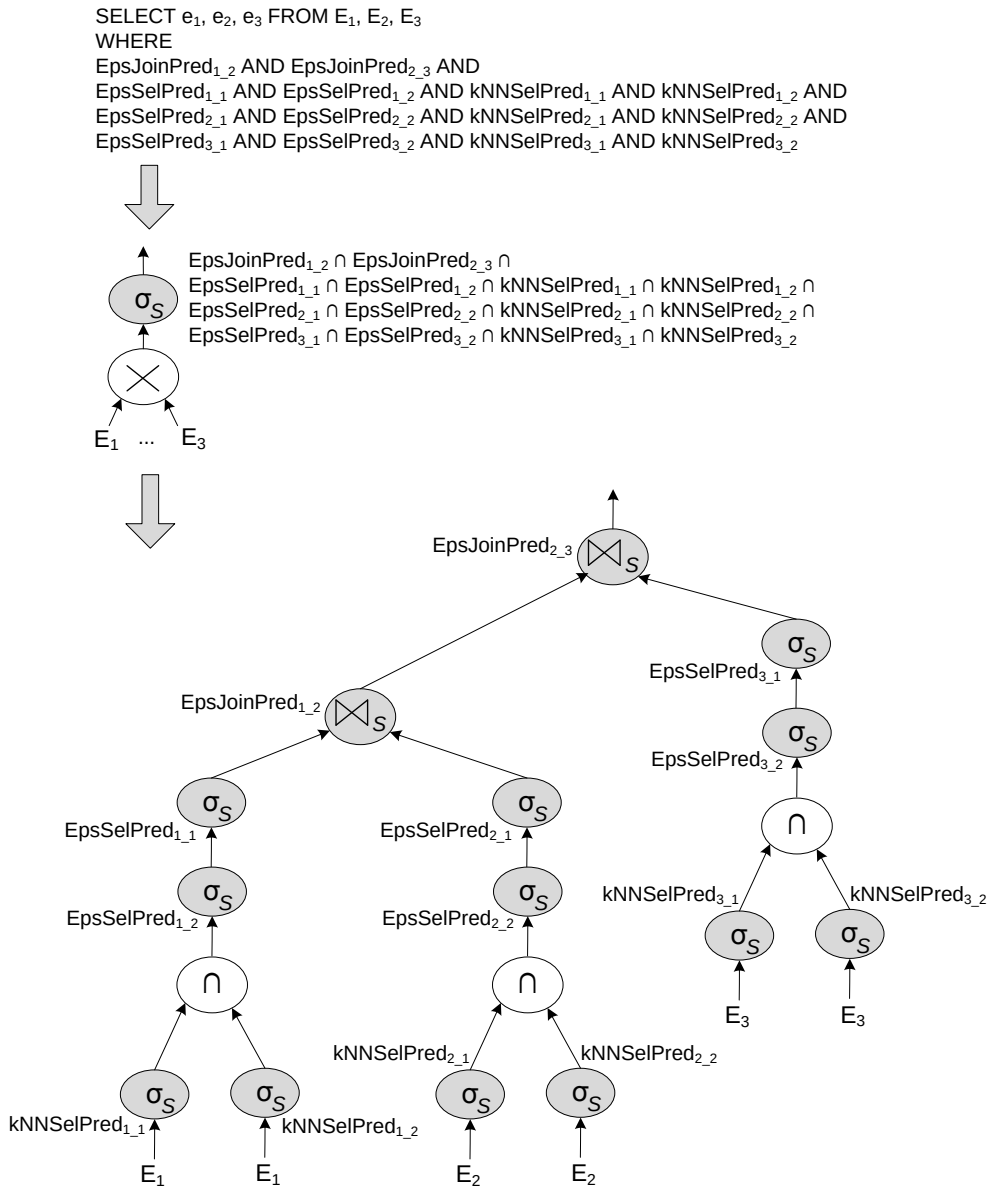


Figure 4-60 Query with Multiple Eps-Join and Similarity Selection Predicates

Figure 4-61 shows the transformation of a query with a kNN-Join and multiple Similarity Selection predicates. This figure shows that Eps-Selection and kNN-Selection can be pushed under the outer input of kNN-Joins. Eps-Selection defined over the inner input attribute of a kNN-Join can be serialized with the join operation executing the kNN-Join first. kNN-Selection defined over the inner input attribute of a kNN-Join cannot be serialized with the join operation. In this

case, the kNN-Join and kNN-Selection operations need to be evaluated independently and the results combined using the intersection operation.

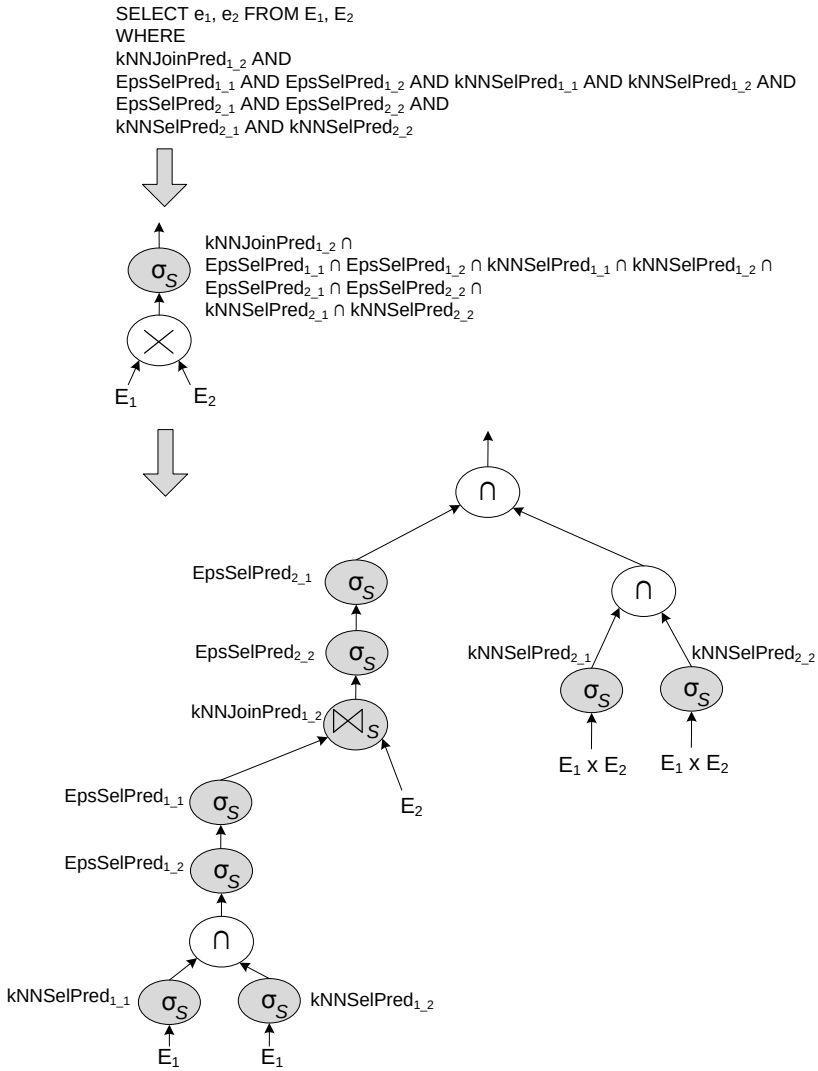


Figure 4-61 Query with kNN-Join and Multiple Similarity Selection Predicates

Figure 4-62 shows the transformation of a generic query with multiple Similarity Join and Similarity Selection predicates. Figure 4-62 shows that multiple kNN-Join operations can be serialized as long as the attributes of the join predicates have a single direction. kNN-Join and Eps-Join can also be serialized executing the kNN-Joins first. Multiple kNN-Join operations whose predicates do not have a

single direction need to be evaluated independently and the results combined using the intersection operation.

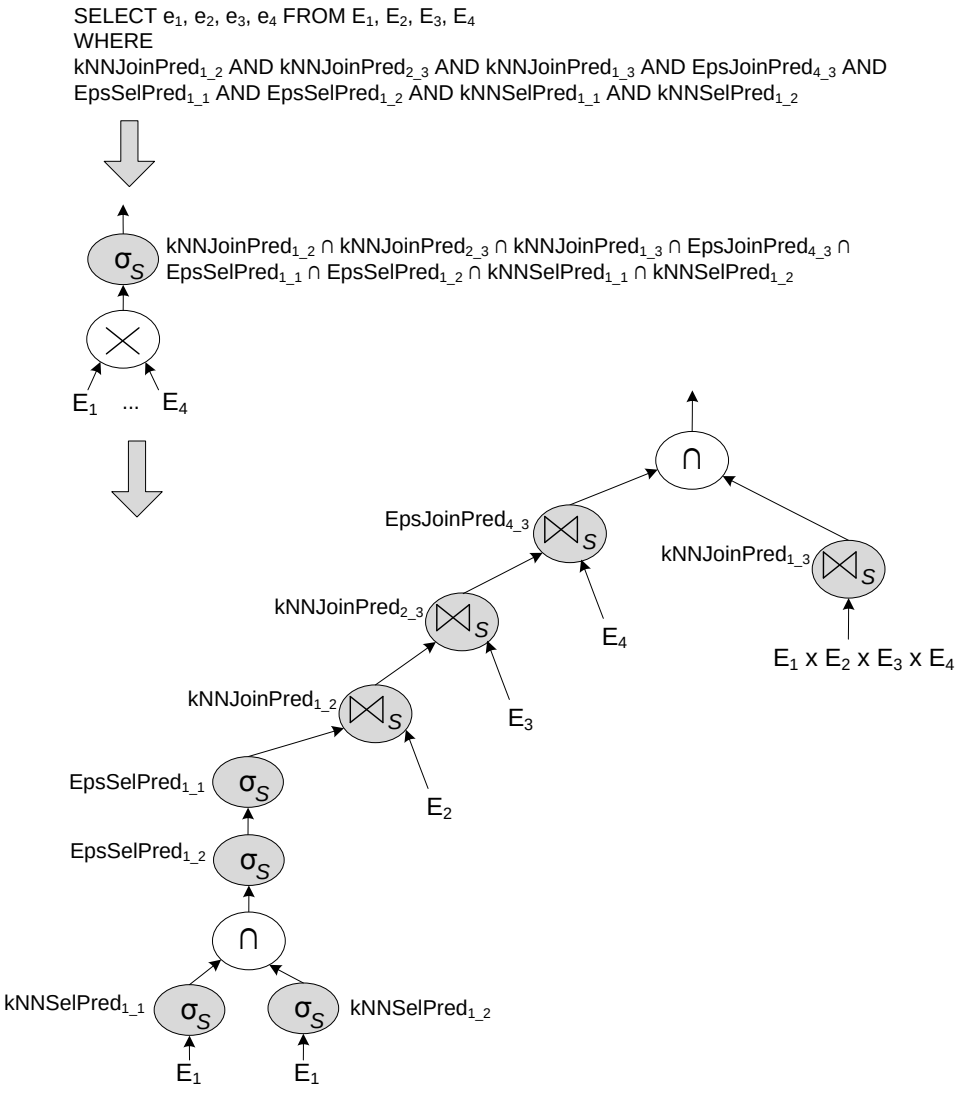


Figure 4-62 Query with Multiple Similarity Join and Similarity Selection Predicates

CHAPTER 5 CONCLUSIONS AND FUTURE WORK

5.1. Conclusions

Many application scenarios can benefit tremendously from database operators that exploit similarities in the data and allow the pipelining of the results for further processing. Related previous work has proposed some similarity-aware operations and standalone implementation techniques that are not fully integrated with the query processing engine of DBMSs.

The focus of this paper is the proposal and study of several similarity-aware database operators and the analysis of their role as physical operators, interactions, optimizations, and implementation techniques.

We demonstrate that Similarity-aware operators can be efficiently implemented taking advantage of structures and mechanisms already available in DBMSs. The performance study shows that similarity queries using the implemented similarity-aware operators perform significantly better than queries that get the same result using only regular operators. Furthermore, some similarity-aware operations cannot be answered using conventional database operators, e.g., Unsupervised Similarity Group-by.

Multiple optimization techniques used in regular operators can be extended to the case of Similarity-aware operators. Particularly, we present (1) multiple transformation rules for SGB and SJ, (2) Eager and Lazy Aggregation transformation techniques for SGB and SJ, and (3) guidelines to answer similarity queries using materialized views.

We demonstrated that it is possible to have a conceptual evaluation model for similarity queries that clearly specifies the way a similarity query should be

evaluated even if the query has multiple similarity-aware operations. The proposed conceptual evaluation model considers Similarity Group-By, Similarity-Join, and Similarity Selection operations.

We presented a rich set of generalized transformation rules for similarity queries with multiple similarity-aware operations. Furthermore, we demonstrated that transformation rules for similarity operators can take advantage of special properties of these operations and the involved distance functions to enable more useful query transformations.

We also demonstrated how the conceptual evaluation plan of a query can be transformed to equivalent plans with potentially better execution times. Furthermore, we identified several core query transformation guidelines for similarity queries, e.g., (1) multiple Eps-Selection or multiple Eps-Join operations can be serialized, (2) multiple kNN-Selection operations need to be executed independently and their results combined using intersection, (3) Eps-Selection and kNN-Selection over the same attribute can be serialized executing the kNN-Selection first, (4) Eps-Selection and kNN-Selection can be pushed under any input of Eps-Joins, (5) kNN-Join and Eps-Selection over the inner input of the join can be serialized executing the kNN-Join first, (6) kNN-Join and kNN-Selection on the inner input need to be executed independently and their results combined using intersection, (7) multiple kNN-Join whose join attributes do not have a single direction also need to be executed independently.

We showed how the SGB and SJ operators can be efficiently used in practice. We used these operators to support the queries of a decision support system.

5.2. Future Work

The paths for future work include:

1. Similarity-aware database for sensor networks. The study and implementation of similarity-aware operators to process sensor data is of

particular interest because of the imprecise nature of the data. In this scenario, operations like SGB and SJ can be extensively used to answer more useful queries.

2. Similarity-aware massively parallel data stream management system. Building this system will involve implementing similarity-aware operations using the Map-Reduce paradigm. These operations will enable the analysis of very large streams of data.

3. Other core similarity-aware database operators. Our previous work focused on the Similarity Group-by, Similarity Join and Similarity Selection operators. Additional operators that can be studied are: duplicate elimination, set intersection, and set difference.

4. Similarity-aware data warehousing operators**.** The CUBE and ROLLUP operators, which are extensively used in data warehousing applications, can be extended to use similarity grouping mechanisms like the ones used in SGB. Different similarity grouping strategies could be used to group the values in different dimensions. These extended CUBE and ROLLUP operators will be able to generate more meaningful and useful summaries of large datasets.

5. Benchmark for Similarity-aware Query Processing**.** This benchmark will evaluate the similarity-aware query processing capabilities of database systems. One of the goals for this benchmark would be the specification of queries that exploit similarities in the data and have broad industry-wide relevance.

REFERENCES

REFERENCES

1. A. K. Jain, M. N. Murty, and P. J. Flynn, “Data clustering: A review,” ACM Computing Surveys, vol. 31, no. 3, pp. 264-323, 1999.

2. P. Berkhin, “Survey of clustering data mining techniques,” Accrue Software, 2002.

3. B. Stein, S. M. Eissen, and F. Wibrock, “On cluster validity and the information need of users,” in 3rd IASTED Int. Conference on Artificial Intelligence and Applications, 2003.

4. M. Halkidi, Y. Batistakis, and M. Vazirgiannis, “Clustering validity checking methods: Part II,” SIGMOD Record, vol. 31, no. 3, pp. 19-27, 2002.

5. M. Li, G. Holmes and B. Pfahringer, “Clustering large datasets using Cobweb and K-Means in tandem,” in 17th Australian Joint Conference on Artificial Intelligence, 2004.

6. F. Farnstrom, J. Lewis, and C. Elkan, “Scalability for clustering algorithms revisited,” SIGKDD Explorations Newsletter, vol. 2, no. 1, pp. 51–57, 2000.

7. S. Guha, R. Rastogi, and K. Shim, “CURE: An efficient clustering algorithm for large databases,” SIGMOD Record, vol. 27, no. 2, pp. 73-84, 1998.

8. T. Zhang, R. Ramakrishnan, and M. Livny, “BIRCH: An efficient data clustering method for very large databases,” SIGMOD Record, vol. 25, no. 2, pp. 103-114, 1996.

9. C. Zhang and Y. Huang, “Cluster By: A new SQL extension for spatial data aggregation,” in 15th Annual ACM international Symposium on Advances in Geographic information Systems, 2007.

10. C. Li, M. Wang, L. Lim, H. Wang, and K. C. Chang, “Supporting ranking and clustering as generalized order-by and group-by,” in ACM SIGMOD International Conference on Management of Data, 2007.

11. E. Schallehn, K. Sattler, and G. Saake, “Extensible grouping and aggregation for data reconciliation,” in 4th Workshop of Engineering Federated Information Systems, 2001.

12. E. Schallehn and K. Sattler, “Using similarity-based operations for resolving data-level conflicts,” in 20th British National Conference on Databases, 2003.

13. E. Schallehn, K. Sattler, and G. Saake, “Efficient similarity-based operations for data integration,” Data & Knowledge Engineering, vol. 48, no. 3, pp. 361-387, 2004.

14. C. Böhm, “The Similarity Join: A powerful database primitive for high performance data mining,” tutorial, in 17th International Conference on Data Engineering, 2001.

15. C. Böhm and H. Kriegel, “A cost model and index architecture for the similarity join,” in 17th International Conference on Data Engineering, 2001.

16. C. Böhm, F. Krebs, and H. Kriegel, “Optimal Dimension Order: A generic technique for the similarity join,” in 4th International Conference on Data Warehousing and Knowledge Discovery, 2002.

17. V. Dohnal, C. Gennaro, and P. Zezula, “Similarity join in metric spaces using eD-Index,” in 14th International Conference on Database and Expert Systems Applications, 2003.

18. V. Dohnal, C. Gennaro, P. Savino, and P. Zezula, “Similarity join in metric spaces,” in 25th European Conference on IR Research, 2003.

19. C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel, “Epsilon Grid Order: An algorithm for the similarity join on massive high-dimensional data,” in ACM SIGMOD International Conference on Management of Data, 2001.

20. J.-P. Dittrich and B. Seeger, “GESS: A scalable similarity join algorithm for mining large data sets in high dimensional spaces,” in 7th ACM SIGKDD international Conference on Knowledge Discovery and Data Mining, 2001.

21. E. H. Jacox and H. Samet, “Metric space similarity joins,” ACM Trans. Database Syst., 33(2):1-38, 2008.

22. G. Hjaltason and H. Samet, “Incremental distance join algorithms for spatial databases,” in ACM SIGMOD International Conference on Management of Data, 1998.

23. C. Böhm and F. Krebs, “The k-Nearest Neighbour Join: Turbo charging the KDD process,” Knowledge and Information Systems, 6(6): 728-749, 2004.

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

25. C. Xia, H. Lu, B. Chin, and O. Hu, “GORDER: An Efficient method for KNN join processing,” in 30th International Conference on Very Large Data Bases, 2004.

26. C. Böhm, B. Braunmüller, M. Breunig, and H. Kriegel, “High performance clustering based on the similarity join,” in 9th International Conference on Information and Knowledge Management, 2000.

27. H. Kriegel, P. Kunath, M. Pfeifle, and M. Renz, “Probabilistic similarity join on uncertain data,” in 11th International Conference on Database Systems for Advanced Applications, 2006.

28. S. Chaudhuri, V. Ganti, and R. Kaushik, “A Primitive operator for similarity joins in data cleaning,” in International Conference on Data Engineering, 2006.

29. S. Chaudhuri, V. Ganti, and R. Kaushik, “Data Debugger: An operator-centric approach for data quality solutions,” Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, 2006.

30. L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, “Approximate string joins in a database (almost) for free,” in International Conference on Very Large Data Bases, 2001.

31. D. J. DeWitt, J. F. Naughton, and D. A. Schneider, “An evaluation of non-equijoin algorithms,” in International Conference on Very Large Data Bases, 1991.

32. B. Bryan, F. Eberhardt, and C. Faloutsos, “Compact similarity joins,” in International Conference on Data Engineering, 2008.

33. C. Xiao, W. Wang, and X. Lin, “EdJoin: An efficient algorithm for similarity joins with edit distance constraints,” in International Conference on Very Large Data Bases, 2008.

34. C. Xiao, W. Wang, X. Lin, and J. X. Yu, “Efficient similarity joins for near duplicate detection,” in 17th international Conference on World Wide Web, 2008.

35. M. D. Lieberman, J. Sankaranarayanan, and H. Samet, “A fast similarity join algorithm using graphics processing units,” in International Conference on Data Engineering, 2008.

36. M. Hadjieleftheriou, A. Chandel, N. Koudas, and D. Srivastava, “Fast indexes and algorithms for set similarity selection queries,” in International Conference on Data Engineering, 2008.

37. X. Yang, B. Wang, and C. Li, “Cost-based variable-length-gram selection for string collections to support approximate queries efficiently,” in ACM SIGMOD International Conference on Management of Data, 2008.

38. X. Lian and L. Chen, “Similarity search in arbitrary subspaces under Lp-norm,” in International Conference on Data Engineering, 2008.

39. M. Wichterich, I. Assent, P. Kranen, and T. Seidl, “Efficient EMD-based similarity search in multimedia databases via flexible dimensionality reduction,” in ACM SIGMOD International Conference on Management of Data, 2008.

40. W. Yan and P. Larson, “Eager aggregation and lazy aggregation,” in 21th International Conference on Very Large Data Bases.

41. P. Larson, “Data reduction by partial preaggregation,” in Proc. 18th International Conference on Data Engineering, 2002.

42. C. Galindo-Legaria and M. Joshi, “Orthogonal optimization of subqueries and aggregation,” SIGMOD Record, vol. 30, no. 2, pp. 571-581, 2001.

43. J. Goldstein and P. Larson, “Optimizing queries using materialized views: A practical, scalable solution,” SIGMOD Record, vol. 30, no. 2, pp. 331-342, 2001.

44. S. Cohen, W. Nutt, and Y. Sagiv, “Rewriting queries with arbitrary aggregation functions using views,” ACM Transactions on Database Systems, vol. 31, no. 2, pp. 672-715, 2006.

45. S. Cohen, “User-defined aggregate functions: Bridging theory and practice,” in ACM SIGMOD International Conference on Management of Data, 2006.

46. S. Adali, P. Bonatti, M. L. Sapino, and V. S. Subrahmanian, “A multi-similarity algebra,” in ACM SIGMOD International Conference on Management of Data, 1998.

47. C. Traina, A. J. M. Traina, M. R. Vieira, A. Arantes, and C. Faloutsos, “Efficient processing of complex similarity queries in RDBMs through query rewriting,” in 15th ACM international Conference on information and Knowledge Management, 2006.

48. M. C. N. Barioni, H. L. Razente, A. J. M. Traina, and C. Traina, “SIREN: A similarity retrieval engine for complex data,” In International Conference on Very Large Data Bases, 2006.

49. G. B. Baioco, A. J. M. Traina, and C. Traina, “Mamcost: Global and local estimates leading to robust cost estimation of similarity queries,” in 19th international Conference on Scientific and Statistical Database Management, 2007.

50. M. R. P. Ferreira, C. Traina, and A. J. M. Traina, “An efficient framework for similarity query optimization,” in 15th Annual ACM international Symposium on Advances in Geographic information Systems, 2007.

51. TPC-H Version 2.6.1. http://www.tpc.org/tpch.

52. Y. N. Silva, W. G. Aref, and M. H. Ali, “Similarity group-by,” in International Conference on Data Engineering, 2009.

53. Y. N. Silva, M. Arshad, and W. G. Aref, “Exploiting similarity-aware grouping in decision support systems,” in 12th International Conference on Extending Database Technology: Advances in Database Technology, 2009.

54. Y. N. Silva, W. G. Aref, and M. Ali, “The similarity join database operator,” in International Conference on Data Engineering, 2010.

55. Y. N. Silva, A. M. Aly, W. G. Aref, and P. Larson, “SimDB: A similarity-aware database system,” in ACM SIGMOD International Conference on Management of Data, 2010.

56. Y. N. Silva and W. G. Aref, “Similarity-aware query processing and optimization,” in International Conference on Very Large Data Bases PhD Workshop, 2009.