Similarity Group-by*

Yasin N. Silva[1], Walid G. Aref[1], Mohamed H. Ali[2]

[1]*Department of Computer Science, Purdue University, Indiana, USA*
{ysilva,aref}@cs.purdue.edu

[2]*Microsoft Corporation, Washington, USA*
mali@microsoft.com

Abstract— **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 paper 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 paper presents a general definition of the similarity group-by operation and gives three instances of this definition. The paper 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.**

I. Introduction

One of the most important paradigm shifts in data management is the move from systems that focus on exact semantics of data and Boolean semantics of queries to systems that focus on imprecise and approximate semantics of data and queries. Among the areas driving this paradigm shift are probabilistic databases, the integration of information retrieval and database systems, and similarity query processing. Previous work on similarity query processing has focused on solving the problem of joining two sets of data using similarity join predicates that match tuples with similar or approximate values. The goal of this paper is to extend similarity-based processing to another core database operation, the *group-by*, to group objects with similar or approximate values. Grouping capabilities have been extensively studied and implemented in data management systems. In DBMSs, the most common support for grouping is implemented through the standard group-by operator. This operator has relatively good execution time and scalability properties. However, while the semantics of group-by is simple, it is also limited because it is based only on equality, i.e., all the tuples in a group have exactly the same values of the grouping attributes. Grouping has also been studied in data warehouses, where several facilities have been implemented to enrich the output of the regular group-by operator with multiple levels of subtotals. Additionally, analysis techniques, e.g., OLAP and data mining, provide advanced features for grouping. OLAP provides several tools to summarize data organized under the dimensional model but the formation of groups is still based on the equality semantics. Data mining clustering tools employ complex and sophisticated algorithms to discover groups naturally formed in the data. Unfortunately, the use of these analysis techniques requires the definition of elaborate data mining models; is not integrated into the regular query processing engine, and is not standard among the different commercial database systems.

	Similarity Grouping Implementation Approach			
	Integrated in DB Engine	Using Basic SQL Operators	Using User-Defined Aggr.	As Stored Procedures
Supported grouping strategies	All (Supervised & Unsupervised)	Only Supervised (partially)	Only Supervised (partially)	All (Sup. & Unsup.)
Implementation complexity	Low (reuses and extends DB operators and structures)	Queries use a complex mix of joins and aggregations	UDAs require the support of vectors for reference points, spilling mechanisms, etc.	SPs require the support of large hash tables, spilling mechanisms, etc.
Execution time	Very Low	High (see Fig. 13)	Medium	Low
Composable with other operators (pipelining)	Yes	Yes	Yes	No
Take advantage of query optimizer	Yes (pre-aggr. and use of MVs)	No directly	No	No

Fig. 1 Comparison of similarity group-by implementation approaches

Emerging applications, e.g., biological databases and data streaming, require the identification of sets of approximate values. Moreover, many business application scenarios, especially those handling large datasets, can benefit tremendously from SQL constructs that identify groups of similar values to process them further. Current database systems do not provide fast mechanisms that are integrated into the query engine to generate and process groups of similar objects using complex TPC-H-like queries [29]. Figure 1 presents a comparison of several approaches to support similarity-based grouping. The implementation of similarity-based grouping at the DB engine level has the following key advantages: (1) the execution time of the Similarity Group-by operator (SGB) is comparable to that of the regular group-by and is superior to the performance of the other user-level definitions; (2) SGB can be interleaved with other regular operators and its results pipelined for further processing; and (3) important optimization techniques, e.g., pre-aggregation and the use of materialized views can be extended to the new operator. The contributions of this paper are as follows:

- We introduce the similarity group-by (SGB) operator which extends standard group-by to allow the formation

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

of groups based on similarity rather than equality of the data.

- 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.
- 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.
- We implement the proposed SGB operators in PostgreSQL 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.

The rest of this paper proceeds as follows. Section II discusses the related work. Section III presents the general definition of SGB and three instances of this definition. Section IV studies optimization techniques applicable to the new operators. Section V presents implementation guidelines based on a prototype realization of the operators within PostgreSQL. Section VI reports on the performance evaluation of the similarity group-by operators and Section VII presents the conclusions and directions for future research.

II. Related Work

The work on similarity-based query processing has focused on similarity joins. Similarity joins use special types of join predicates to match tuples that have approximate values. Different types of similarity join have been proposed, e.g., range distance join (retrieves all pairs whose distances are smaller than a pre-defined threshold) [1], [7], [8], k-distance join (retrieves the k most-similar pairs) [2], and knn-join (retrieves, for each tuple in one table, the k nearest-neighbours in the other table) [3], [5], [6]. Some similarity join techniques have been employed as building blocks to implement common clustering algorithms [4]**.** Kriegel et al. extend the work on similarity join to uncertain data [9].

The clustering problem 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 [10]. Berkhin surveys clustering techniques used in data mining [11]. 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* [12], [13]. 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 [14] and [15]. CURE [16] and BIRCH [17] 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 of the proposed similarity group-by from these algorithms are: (1) the execution times of the proposed similarity grouping operators are very close to that of the regular group-by; (2) similarity group-by operators 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 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 [18] 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 [28]. 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 [28] 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 [25] and similarity grouping predicates [26], [27]. They focus on string similarity and similarity predicates to reconcile records. Although they can be used for this purpose, the proposed similarity group-by operators in this paper are more general and are designed to be part of a DBMS's query engine.

The optimization techniques of similarity grouping presented in this paper builds 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 tree [19], [20]. These techniques allow complete [19] or partial [20] 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 [21]. 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 [22] for the case of *sum* and *count* aggregation functions, and is extended in [23] and [24] to arbitrary aggregation functions.

III. 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., size and compactness, (2) grouping around points of interest, and (3) segmenting the tuples based on given limiting values.

A. 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 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*)}.

B. 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 [14], [15], [16], [17] can be

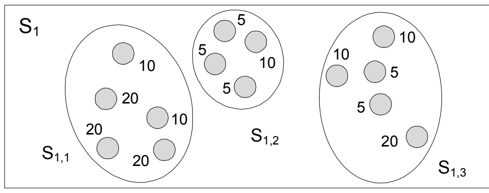


Fig. 2 Example usage of the generic SGB

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 IV). 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 paper are: business decision support systems (Section VI-B.3) and sensor networks (Section III-B). 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).

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:

- 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.
- 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.

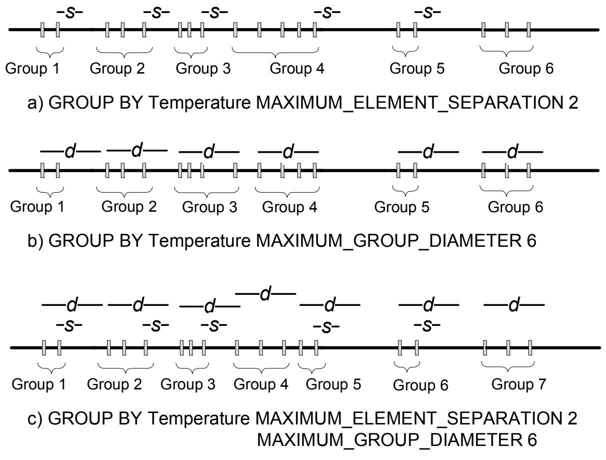
2. If only one clause is specified, we assume that the value of the other is ∞.
3. If MAXIMUM_ELEMENT_SEPARATION is specified, the elements are grouped first using this criterion. If only MAXIMUM_GROUP_DIAMETER is specified, all the elements form the unique resulting group of this step.
4. If MAXIMUM_GROUP_DIAMETER is specified, the groups formed in the previous step are further divided until the group diameter. The criterion to divide a group can be: (i) split a group "breaking" the longest link in the group, or (ii) 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
```

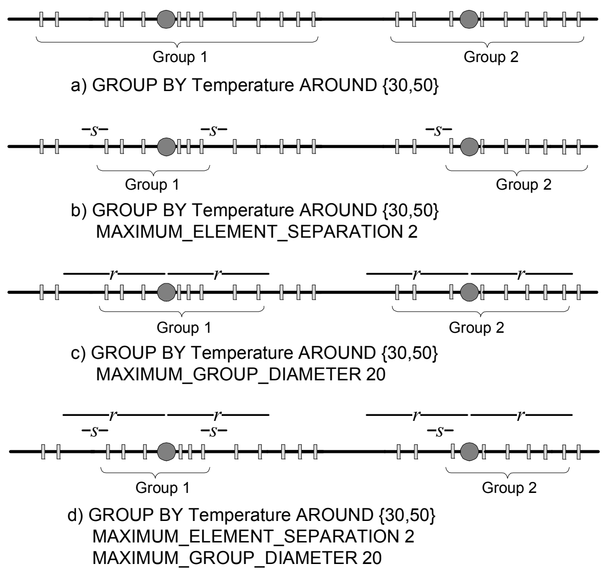
Figure 3.a gives one possible output of the previous example. The different temperature readings are represented as marks on a line. Figures 3.b and 3.c give the output when using the other two possible combinations of the clauses of this operator. In practice, different combinations can be more suitable for different grouping purposes. As evident from Figure 3, 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 [10]).

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,



-s- -s- -s- -s- -s-
Group 1 Group 2 Group 3 Group 4 Group 5 Group 6
a) GROUP BY Temperature MAXIMUM_ELEMENT_SEPARATION 2
—d— —d— —d— —d— —d— —d—
Group 1 Group 2 Group 3 Group 4 Group 5 Group 6
b) GROUP BY Temperature MAXIMUM_GROUP_DIAMETER 6
—d— —d— —d— —d— —d— —d— —d—
-s- -s- -s- -s- -s-
Group 1 Group 2 Group 3 Group 4 Group 5 Group 6 Group 7
c) GROUP BY Temperature MAXIMUM_ELEMENT_SEPARATION 2 MAXIMUM_GROUP_DIAMETER 6

Fig. 3 Examples of unsupervised similarity grouping limiting the groups based on group size and compactness



Group 1 Group 2
a) GROUP BY Temperature AROUND {30,50}
-s- -s- -s-
Group 1 Group 2
b) GROUP BY Temperature AROUND {30,50} MAXIMUM_ELEMENT_SEPARATION 2
r r r r
Group 1 Group 2
c) GROUP BY Temperature AROUND {30,50} MAXIMUM_GROUP_DIAMETER 20
r r r r
-s- -s- -s-
Group 1 Group 2
d) GROUP BY Temperature AROUND {30,50} MAXIMUM_ELEMENT_SEPARATION 2 MAXIMUM_GROUP_DIAMETER 20

Fig. 4 Examples of supervised similarity grouping around two points under various conditions on size and compactness

the SQL syntax of SGB-A provides two clauses that are similar to the ones for the SGB-U operator (Section III-B.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*.

Unlike operator SGB-U of Section III-B.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 4.c gives one possible output of the previous example. The given central points are represented as small circles. Figures 4.a, 4.b, and 4.d give the output when using the other three possible combinations of the clauses of SGB-A. From the 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 the paper, we can combine the conditions using other logic operators.

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.

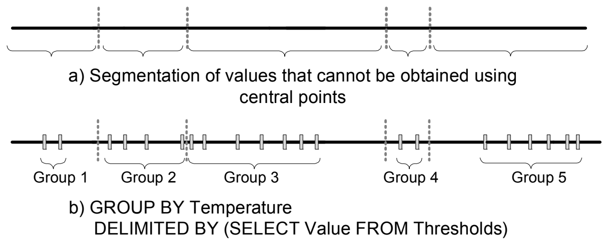

a) Segmentation of values that cannot be obtained using central points
Group 1
Group 2
Group 3
Group 4
Group 5
b) GROUP BY Temperature DELIMITED BY (SELECT Value FROM Thresholds)

Fig. 5 Example of supervised similarity grouping based on a dynamic set of delimiting points

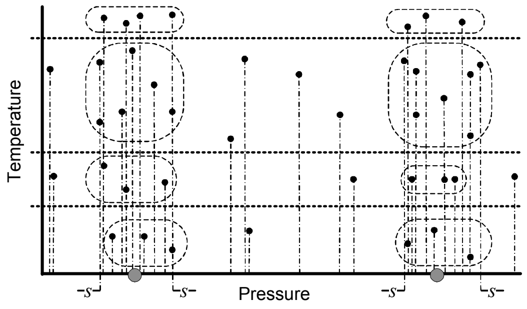


Fig. 6 Similarity grouping with two attributes

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 5.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 5.b gives the output of the previous example. The result of the internal select is represented by vertical dotted line segments.

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 (*SGA*s) 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 *SGA*s 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 *SGA*s are used is obtained intersecting the segmentations of all the (independent) *SGA*s. 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 paper. 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 6 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)
```

IV. 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 [19] 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 7 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 VI-B.3 presents real world scenarios in which each of the approaches performs better.

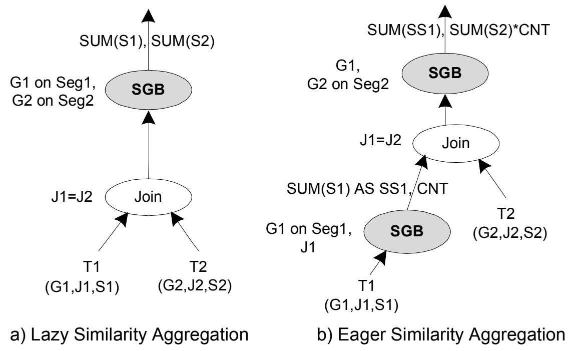


Fig. 7 The Main Theorem

The algebraic notation used in this section is similar to that in [19]. *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. $\times$, 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, $GA_d^+ = GA_d \ U \ \alpha(C_0) - R_d$ 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 ~ B* denote that *A* and *B* belong to the same similarity group, and *A !~ B* denote the opposite.

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

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

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

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 [19], **(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:

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 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.

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 [19], [20], [21].

The use of materialized views to answer aggregation queries [22], [23], [24] 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 [22] 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*.

V. Implementing Similarity Group-By

This section presents the guidelines to implement the similarity grouping operators introduced in Section III 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. 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 8. 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. 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 9.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 *SGA*s are used, they are processed one at the time. Figure 9.b gives the structure of the plan trees generated when two *SGA*s *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. 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 group-by-around is used, the smaller delimiting point when group-by-delimited-by is used, and the average of the minimum and maximum values of *A* in the tuples that form *T* when unsupervised group-by is used. Each aggregation node is able

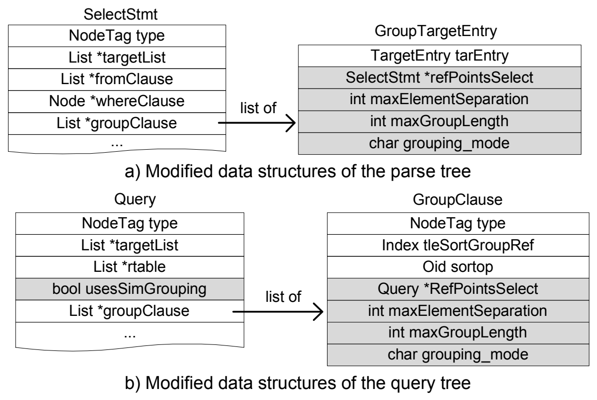


Fig. 8 Modifications in the main query processing data structures (PostgreSQL)

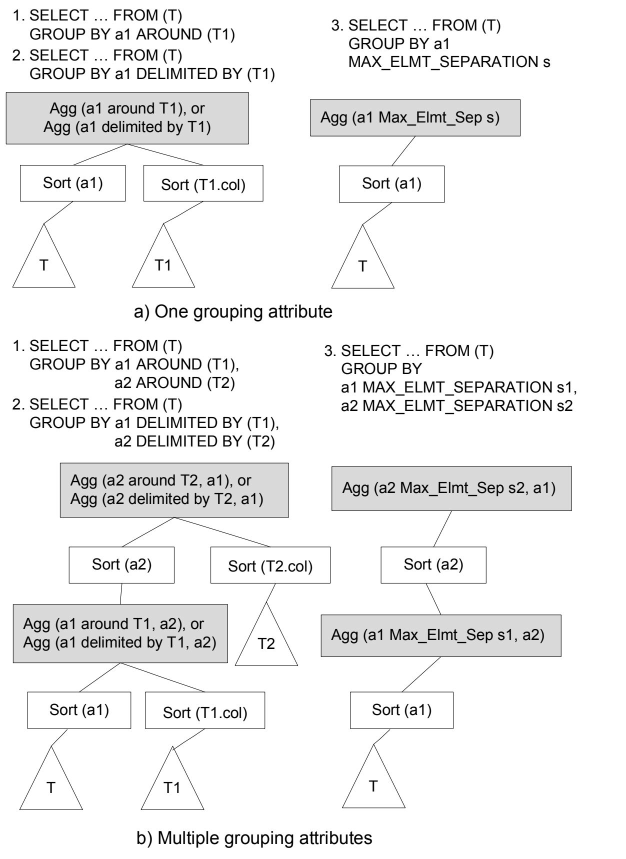


Fig. 9 Path/Plan trees for similarity grouping

to process one *SGA* and any number of regular grouping attributes. The group formation routines are presented in Section V-B. 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.

B. The Executor

When several *SGA*s 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 V-A. 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 group-by-around.

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 (i) the distance between the previous and current tuples is greater than MAXIMUM_ELEMENT_ SEPARATION, (ii) the current tuple is connected to the current central point, and (iii) 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

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}

Fig. 10 Performance evaluation dataset

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.

VI. Performance Evaluation

We implemented the similarity grouping operators presented in Section III 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.

A. Test Configuration

The dataset used in the performance evaluation is based on the one specified by the TPC-H benchmark [29]. The tables, additional attributes, and queries used in the tests are presented in Figures 10 and 11. 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%.

B. 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.

1) Increasing Dataset Size: Figure 12 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

Queries used in Section 7.2.1	
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 7.2.2. 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 7.2.3	
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

Fig. 11 Performance evaluation queries

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 (*SGA*s), 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 SGB queries.

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: (i) 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 11; SGB-A with MAXIMUM_GROUP_DIAMETER can be implemented using further selection predicates; and (ii) SGB-D can be obtained using a complex query similar to SGB(GB). Figure 13 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 SGB query 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.

2) Increasing the Number of SGAs: Figure 14 gives the execution time of SGB queries when the number of *SGA*s 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 *SGA*s 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 *SGA*s 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

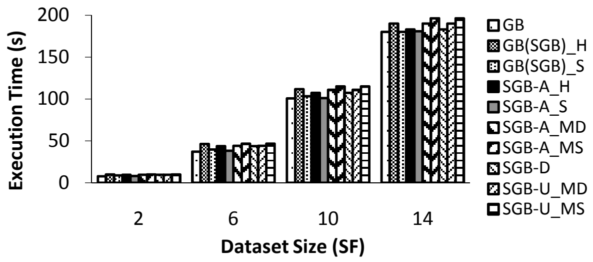


Fig. 12 Performance while increasing dataset size

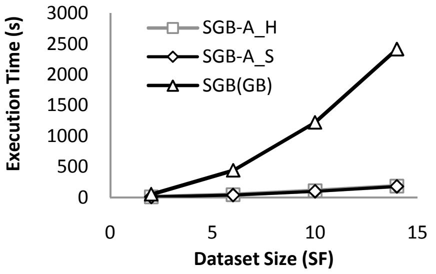


Fig. 13 Performance of generating similarity groups with group-by vs. similarity group-by

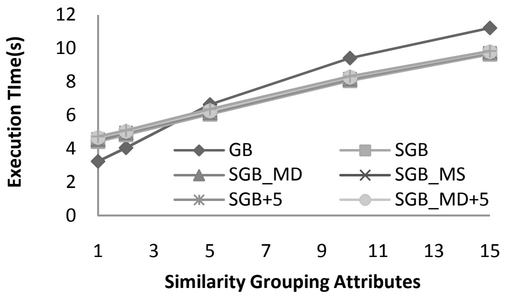


Fig. 14 Performance while increasing number of *SGA*s

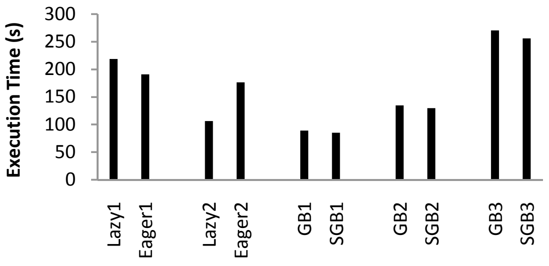


Fig. 15 Performance of complex queries

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.

3) Complex Queries: Figure 15 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 IV are used. Figure 11 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.

VII. CONCLUSIONS AND FUTURE WORK

This paper presents a similarity-based grouping operator, named similarity group-by (SGB), to support the grouping of objects with approximate values. The main goal of SGB is to generate more meaningful and useful groupings than the regular group-by operator while having execution times comparable to those of its non-similarity counterpart. This paper presents a generic definition of SGB and three grouping strategies as instances of this definition. It studies how techniques to optimize standard group-by operations can be extended to the case of similarity group-by and presents the implementation guidelines to implement SGB in the query engine of standard DBMSs. The performance evaluation of the implementation in PostgreSQL shows that the proposed strategies of SGB have a very low cost and scales well when the dataset size or the number of grouping attributes increases. Some paths for future work include: the study of similarity grouping techniques for high-dimensional data, the study of the relationship and integration of similarity grouping techniques with grouping techniques in probabilistic databases, and the study of similarity-based aggregation as a tool for phenomena detection in the context of sensor networks.

REFERENCES

[1] N. Koudas and K. C. Sevcik, "High Dimensional Similarity Joins: Algorithms and Performance Evaluation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 1, pp. 3-18, 2000.

[2] G. Hjaltason and H. Samet, "Incremental distance join algorithms for spatial databases," *SIGMOD Record*, vol. 27, no. 2, pp. 237-248, 1998.

[3] C. Böhm and F. Krebs, "The k-Nearest Neighbour Join: Turbo charging the KDD process," *Knowledge and Information Systems*, vol. 6, no. 6, pp. 728-749, 2004.

[4] C. Böhm, B. Braunmüller, M. Breunig, and H. Kriegel, "High performance clustering based on the similarity join," in *Proc. 9th CIKM*, 2000, pp. 298-305.

[5] C. Yu, B. Cui, S. Wang, and J. Su, "Efficient index-based KNN join processing for high-dimensional data," *Information and Software Technology*, vol. 49, no. 4, pp. 332-344, 2007.

[6] C. Xia, H. Lu, B. Chin, and O. Hu, "GORDER: An Efficient method for KNN join processing," in *Proc. 30th VLDB*, 2004, pp. 756-767.

[7] C. Böhm and H. Kriegel, "A cost model and index architecture for the similarity join," in *Proc. 17th ICDE*, 2001, pp. 411-420.

[8] C. Böhm, F. Krebs, and H. Kriegel, "Optimal Dimension Order: A generic technique for the similarity join," in *Proc. 4th DaWaK*, 2002, pp. 135-149.

[9] H. Kriegel, P. Kunath, M. Pfeifle, and M. Renz, "Probabilistic similarity join on uncertain data," in *Proc. 11th DASFAA*, 2006, pp. 295-309.

[10] A. Jain, M. Murty, and P. Flynn, "Data clustering: a review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264-323, 1999.

[11] P. Berkhin, "Survey of clustering data mining techniques," *Accrue Software*, 2002.

[12] B. Stein, S. zu Eissen, and F. Wibrock, "On cluster validity and the information need of users," in *Proc. 3rd AIA*, 2003, pp. 216-221.

[13] M. Halkidi, Y. Batistakis, and M. Vazirgiannis, "Clustering validity checking methods: part II," *SIGMOD Record*, vol. 31, no. 3, pp. 19-27, 2002.

[14] M. Li, G. Holmes, B. Pfahringer, "Clustering large datasets using Cobweb and K-Means in tandem," in *Proc. 17th Australian Joint Conference on Artificial Intelligence*, 2004, pp. 368-379 .

[15] F. Farnstrom, J. Lewis, and C. Elkan, "Scalability for clustering algorithms revisited," *SIGKDD Explorations Newsletter*, vol. 2, no. 1, pp. 51–57, 2000.

[16] 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.

[17] 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.

[18] C. Zhang and Y. Huang, "Cluster By: A new SQL extension for spatial data aggregation," in *Proc. 15th GIS*, 2007, pp. 1-4.

[19] W. Yan and P. Larson, "Eager Aggregation and Lazy Aggregation," in *Proc. 21th VLDB*, 1995, pp. 345-357.

[20] P. Larson. "Data reduction by partial preaggregation," in *Proc. 18th ICDE*, 2002, pp. 706-715.

[21] C. Galindo-Legaria and M. Joshi, "Orthogonal optimization of subqueries and aggregation," *SIGMOD Record*, vol. 30, no. 2, pp. 571-581, 2001.

[22] J. Goldstein and P. Larson, "Optimizing queries using materialized views: a practical, scalable solution," *SIGMOD Record*, vol. 30, no. 2, pp. 331-342, 2001.

[23] 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.

[24] S. Cohen, "User-defined aggregate functions: bridging theory and practice," in *Proc. SIGMOD*, 2006, pp. 49-60.

[25] E. Schallehn, K. Sattler, and G. Saake, "Extensible Grouping and Aggregation for Data Reconciliation," in *Proc. 4th EFIS*, 2001, pp. 19-32.

[26] E. Schallehn and K. Sattler, "Using similarity-based operations for resolving data-level conflicts," in *Proc. 20th BNCOD*, 2003, pp. 172-189.

[27] 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.

[28] C. Li, M. Wang, L. Lim, H. Wang, and K. C. Chang, "Supporting ranking and clustering as generalized order-by and group-by," in Proc. SIGMOD, 2007, pp. 127-138.

[29] TPC-H Version 2.6.1. [Online]. Available: http://www.tpc.org/tpch