Similarity Group-by Operators for Multi-dimensional Relational Data

Mingjie Tang, Ruby Y. Tahboub, Walid G. Aref, Senior Member, IEEE, Mikhail J. Atallah, Fellow, IEEE and ACM, Qutaibah M. Malluhi, Mourad Ouzzani, Member, IEEE, and Yasin N. Silva

Abstract—The SQL group-by operator plays an important role in summarizing and aggregating large datasets in a data analytics stack. While the standard group-by operator, which is based on equality, is useful in several applications, allowing similarity aware grouping provides a more realistic view on real-world data that could lead to better insights. The Similarity SQL-based Group-By operator (SGB, for short) extends the semantics of the standard SQL Group-by by grouping data with similar but not necessarily equal values. While existing similarity-based grouping operators efficiently materialize this approximate semantics, they primarily focus on one-dimensional attributes and treat multi-dimensional attributes independently. However, correlated attributes, such as in spatial data, are processed independently, and hence, groups in the multi-dimensional space are not detected properly. To address this problem, we introduce two new SGB operators for multi-dimensional data. The first operator is the clique (or distance-to-all) SGB, where all the tuples in a group are within some distance from each other. The second operator is the distance-to-any SGB, where a tuple belongs to a group if the tuple is within some distance from any other tuple in the group. Since a tuple may satisfy the membership criterion of multiple groups, we introduce three different semantics to deal with such a case: (i) eliminate the tuple, (ii) put the tuple in any one group, and (iii) create a new group for this tuple. We implement and test the new SGB operators and their algorithms inside PostgreSQL. The overhead introduced by these operators proves to be minimal and the execution times are comparable to those of the standard Group-by. The experimental study, based on TPC-H and a social check-in data, demonstrates that the proposed algorithms can achieve up to three orders of magnitude enhancement in performance over baseline methods developed to solve the same problem.

Index Terms—similarity query, relational database,

1 INTRODUCTION

The deluge of data accumulated from sensors, social networks, computational sciences, and location-aware services calls for advanced querying and analytics that are often dependent on efficient aggregation and summarization techniques. The SQL group-by operator is one main construct that is used in conjunction with aggregate operations to cluster the data into groups and produce useful summaries. Grouping is usually performed by aggregating into the same groups tuples with equal values on a certain subset of the attributes. However, many applications (i.e., in Section 5) are often interested in grouping based on similar rather than strictly equal values.

Clustering [1] is a well-known technique for grouping similar data items in the multi-dimensional space. In most cases, clustering is performed outside of the database system. Moving the data outside of the database to perform the clustering and then back into the database for further processing results in a costly impedance mismatch. Moreover, based on the needs of the underlying applications, the output clusters may need to be further processed by SQL to filter out some of the clusters and to perform other SQL operations on the remaining clusters. Hence, it would be greatly beneficial to develop practical and fast similarity group-by operators that can be embedded within SQL to avoid such impedance mismatch and to benefit from the processing power of all the other SQL operators.

SQL-based Similarity Group-by (SGB) operators have been proposed in [2] to support several semantics to group similar but not necessarily equal data. Although several applications can benefit from using existing SGB over Group-by, a key shortcoming of these operators is that they focus on one-dimensional data. Consequently, data can only be approximately grouped based on one attribute at a time.

In this paper, we introduce new similarity-based group-by operators that group multi-dimensional data using various metric distance functions. More specifically, we propose two SGB operators, namely SGB-All and SGB-Any, for grouping multi-dimensional data. SGB-All forms groups such that a tuple or a data item, say o, belongs to a group, say g, if o is at a distance within a user-defined threshold from all other data items in g. In other words, each group in SGB-All forms a clique of nearby
data items in the multi-dimensional space. For example, all the two-dimensional points (a-c) in Figure 1 are within distance 3 from each other and hence form a clique. They are all reported as members of one group as they are all part of the output of SGB-All. In contrast, SGB-Any forms groups such that a tuple or a data item, say o, belongs to a group, say g, if o is within a user-defined threshold from at least one other data item in g. For example, all the two dimensional points in Figure 1 form one group. Point a is within Distance 3 from Point c that in turn is within Distance 3 from Points b, d, and f. Furthermore, Point e is within Distance 3 from Point d, and so on. Therefore, Points a-h of Figure 1, are reported as members of one group as part of the output of SGB-Any.

Notice that in the SGB-All operator, a data item may qualify the membership criterion of multiple groups. For example, data item c in Figure 1 forms a clique with two groups. In this case, we propose three semantics, namely, on-overlap join-any, on-overlap eliminate, and on-overlap form-new-group, for handling such a case. We provide efficient algorithms for computing the two proposed SGB operators over correlated multi-dimensional data. The proposed algorithms use a filter-refine paradigm. In the filter step, a fast yet conservative check is performed to identify the data items that are candidates to form groups. Some of the data items resulting from the filter step will end up being false-positives that will be discarded. The refinement step eliminates the false-positives to produce the final output groups. Notice that for the case of SGB-Any, a data item cannot belong to multiple groups. For example, consider a data item, say o, that is a member of two groups, say g1 and g2, i.e., o is within distance $\epsilon$ from at least one other data item in each of g1 and g2. In this case, based on the semantics of SGB-Any, Groups g1 and g2 merge into one encompassing bigger group that contains all members of g1, g2 and common data item o. Specificity, we mainly focus on two and three dimensional data space, leaving higher dimensions for future work.

The contributions of this paper are as follows:

1) We introduce two new operators, namely SGB-All and SGB-Any, for grouping multi-dimensional data from within SQL.

2) We present an extensible algorithmic framework to accommodate the various semantics of SGB-All and SGB-Any along with various options to handle overlapping data among groups. We introduce effective optimizations for both operators.

3) We prototype the two operators inside PostgreSQL and study their performance using the TPC-H benchmark. The experiments demonstrate that the proposed algorithms can achieve up to three orders of magnitude enhancement in performance over the baseline approaches. Moreover, the performance of the proposed SGB operators is comparable to that of relational Group-by, and outperform state-of-the-art clustering algorithm (i.e., $K$-means, DBSCAN and BIRCH) from one to three orders of magnitude.

Fig. 1: The Semantics of Similarity predicates $\epsilon = 3$.

The rest of the paper proceeds as follows. Section 2 discusses the related work. Section 3 provides background material. Section 4 introduces the new SGB operators. Section 5 presents application scenarios that demonstrate the use and practicality of the various proposed semantics for SGB operators. Sections 6 and 7 introduce the algorithmic frameworks for SGB-All and SGB-Any operators, respectively. Section 8 describes the in-database extensions to support the two operators and their performance evaluation from within PostgreSQL. Section 9 concludes the paper.

2 Related Work

Previous work on similarity-aware query processing addressed the theoretical foundation and query optimization issues for similarity-aware query operators [2, 3, 4]. We introduce similarity algebra that extends relational algebra operations, e.g., joins and set operations, with similarity semantics. Similarity queries and their optimizations include algorithms for similarity range search and $K$-Nearest Neighbor (KNN) [5], similarity join [6], and similarity aggregates [7]. Most of work focus on semantic and transformation rules for query optimization purpose independently from actual algorithms to realize similarity-aware operators. In contrast, our focus is on the latter.

Clustering forms groups of similar data for the purpose of learning hidden knowledge. Clustering methods and algorithms have been extensively studied in the literature, e.g., see [8], [11]. The main clustering methods are partitioning, hierarchical, and density-based. $K$-means [9] is a widely used partitioning algorithm that uses several iterations to refine the output clusters. Hierarchical methods build clusters either divisively (i.e., top-down) such as in BIRCH [10], or agglomeratively (i.e., bottom-up) such as in CURE [11]. Density-based methods, e.g., DBSCAN [12], cluster data based on local criteria, e.g., density reachability among data elements. The key differences between our proposed SGB operators and clustering are: (1) the proposed SGB operators are relational operator that are integrated in a relational query evaluation pipeline with various grouping semantics. Hence, they avoid the impedance mismatch experienced by standalone clustering and data mining packages that mandate extracting the data to be clustered out of the DBMS. (2) In contrast to standalone clustering algorithms, the SGB operators can be interleaved with other relational operators. (3) Standard relational query optimization techniques that apply to the standard relational group-by are also applicable to the SGB operators.
as illustrated in [2]. This is not feasible with standalone clustering algorithms. Also, improved performance can be gained by using database access methods that process multi-dimensional data.

An early work on similarity-based grouping appears in [13]. It addresses the inconsistencies and redundancy encountered while integrating information systems with dirty data. However, this work realizes similarity grouping through pairwise comparisons which incur excessive computations in the absence of a proper index. Furthermore, the introduced extensions are not integrated as first class database operators. The work in [14] focuses on overcoming the limitations of the distinct-value group-by operator and introduces the SQL construct “Cluster By” that uses conventional clustering algorithms, e.g., DBSCAN, to realize similarity grouping. Cluster By addresses the impedance mismatch due to the data being outside the DBMS to perform clustering. Our SGB operators are more generic as they use a set of predicates and clauses to refine the grouping semantics, e.g., the distance relationships among the data elements that constitute the group and how inter-group overlaps are dealt with.

Several DBMSs have been extended to support similarity operations. SIREN [13] is a similarity retrieval engine that allows executing similarity queries over a relational DBMS. POSTGRESQL-IE [16] is an image handling extension of PostgreSQL to support content-based image retrieval capabilities, e.g., supporting the image data type and responding to image similarity queries. While these extensions incorporate various notions of similarity into query processing, they focus on the similarity search operation. SimDB [2] is a PostgreSQL extension that supports similarity-based queries and their optimizations. Several similarity operations, e.g., join and group-by, are implemented in as first-class database operators. However, the similarity operators in SimDB focus on one-dimensional data and do not handle multi-dimensional attributes.

3 Preliminaries

In this section, we provide background definitions and formally introduce similarity-based group-by operators.

Definition 1: A metric space is a space $M = (\mathbb{D}, \delta)$ in which the distance between two data points within a domain $\mathbb{D}$ is defined by a function $\delta : \mathbb{D} \times \mathbb{D} \rightarrow \mathbb{R}$ that satisfies the properties of symmetry, non-negativity, and triangular inequality.

We use the Minkowski distance $L_p$ as the distance function $\delta$. We consider the following two Minkowski distance functions. Let $p_x$ be a data point in the multi-dimensional space of the form $p_x \langle x_1, \ldots, x_d \rangle$ and $p_{xy}$ is the value of the $y^{th}$ dimension of $p_x$.

- The Euclidean distance $L_2 : \delta_2(p_i, p_j) = \sqrt{\sum_{y} (p_{iy} - p_{jy})^2}$
- The maximum distance $L_\infty : \delta_\infty(p_i, p_j) = \max_{y} |p_{iy} - p_{jy}|$.

Definition 2: A similarity predicate $\xi_{\delta, \epsilon}$ is a Boolean expression that returns TRUE for two multi-dimensional points, say $p_i$ and $p_j$, if the distance $\delta$ between $p_i$ and $p_j$ is less than or equal to $\epsilon$, i.e., $\xi_{\delta, \epsilon}(p_i, p_j) : \delta(p_i, p_j) \leq \epsilon$. In this case, the two points are said to be similar.

Definition 3: Let $T$ be a relation of tuples, where each tuple, say $t$, is of the form $t = \{GA_1, \ldots, GA_k, NGA_1, \ldots, NGA_l\}$, the subset $GA_c = \{GA_1, \ldots, GA_k\}$ be the grouping attributes, the subset $NGA = \{NGA_1, \ldots, NGA_l\}$ be the non-grouping attributes, and $\xi_{\delta, \epsilon}$ be a similarity predicate. Then, the similarity Group-by operator $\hat{G}( GA_c, (\xi_{\delta, \epsilon}) )(R)$ forms a set of answer groups $G_s$ by applying $\xi_{\delta, \epsilon}$ to the elements of $GA_c$, such that a pair of tuples, say $t_i$ and $t_j$, are in the same group if $\xi_{\delta, \epsilon}(t_i, GA_c, t_j, GA_c)$.

Definition 4: Given a set of groups $G = \{g_1, \ldots, g_m\}$, the Overlap Set $Oset$ is the set of tuples formed by the union of the intersections of all pairs of groups $(g_1, \ldots, g_m)$, i.e., $Oset = \bigcup_{(i,j) \in \{1..m\}} (g_i \cap g_j)$, where $i \neq j$. In other words, $Oset$ contains all the tuples that belong to more than one group.

For simplicity, we study the case where the set of grouping attributes, $GA_c$, contains only two attributes. In this case, we can view tuples as points in the two-dimensional space, each of the form $p(x_1, x_2)$. We enclose each group of points by a bounding rectangle $R(p_1, p_c)$, where points $p_1$ and $p_c$ correspond to the upper-left and bottom-right corners of $R$, respectively.

4 Similarity Group-By Operators

This section discusses the semantics of the two similarity-based group-by operators, namely, SGB-All and SGB-Any.

4.1 Similarity Group-By ALL (SGB-All)

Given a set of tuples whose grouping attributes form a set, say $P$, of two-dimensional points, where $P = \{p_1, \ldots, p_n\}$, the SGB-All operator $\hat{G}_{all}$ forms a set, say $G_m$, of groups of points from $P$ such that $\forall g \in G_m$, the similarity predicate $\xi_{\delta, \epsilon}$ is TRUE for all pairs of points $(p_i, p_j) \in g$, and $g$ is maximal, i.e, there is no group $g'$ such that $g \subseteq g'$. More formally,

$\hat{G}_{all} = \{g \mid \forall p_i, p_j \in g, \xi_{\delta, \epsilon}(p_i, p_j) \wedge g \text{ is maximal}\}$

Figure[1] gives an example of two groups (a-e) and (c,f,g), where all pairs of elements within each group are within a distance $\epsilon \leq 3$. The proposed SQL syntax for the SGB-All operator is as follows:

```sql
SELECT column, aggregate-func(column)
FROM table-name
WHERE condition
GROUP BY column DISTANCE-TO-ALL [L2 | LINF] WITHIN \epsilon
ON-OVERLAP [JOIN-ANY] [ELIMINATE] [FORM-NEW-GROUP]
```

SGB-All uses the following clauses to realize similarity-based grouping:

- DISTANCE-TO-ALL: specifies the distance function to be applied by the similarity predicate when deciding the membership of points within a group.
the SGB-Any operator $\mathcal{G}_\text{any}$ clusters points in $P$ into a set of groups, say $G_m$, such that, for each group $g \in G_m$, the points in $g$ are all connected by edges to form a graph, where an edge connects two points, say $p_i$ and $p_j$, in the graph if they are within Distance $\epsilon$ from each other, i.e., $\xi_{\delta,\epsilon}(p_i, p_j)$. More formally,  

$$
\mathcal{G}_\text{any} = \{ g \mid \forall p_i, p_j \in g, (\xi_{\delta,\epsilon}(p_i, p_j) \lor (\exists p_{k1}, \ldots, p_{kn}, \xi_{\delta,\epsilon}(p_{i}, p_{k1}) \land \ldots \land \xi_{\delta,\epsilon}(p_{k_n}, p_{j})) \land g \text{ is maximal}\}
$$

The notion of distance-to-any between elements within a group is illustrated in Figure 4, where $\epsilon = 3$. All of the points (a-h) form one group. The corresponding SQL syntax of the SGB-Any operator is as follows:

```sql
SELECT column, aggregate-func(column)
FROM table-name
WHERE condition
GROUP BY column
DISTANCE-TO-ANY [L2 | LINF] WITHIN \epsilon
```

SGB-Any uses the DISTANCE-TO-ANY predicate that applies the metric space function while evaluating the distance between adjacent points. When using the semantics for SGB-Any, the case for points overlapping multiple groups does not arise. The reason is that when an input point overlaps multiple groups, the groups merge to form one large group.

**Example 2**: The following query performs the aggregate operation `count` on the groups formed by SGB-Any on the two-dimensional grouping attributes `GPSCoor-lat` and `GPSCoor-long` using the Euclidean distance with $\epsilon = 3$.

```sql
SELECT count(*)
FROM GPSPoints
GROUP BY GPSCoor-lat, GPSCoor-long
DISTANCE-TO-ANY L2 WITHIN 3
```

Consider the example in Figure 3. After processing $a_4$, the following groups are $g_1 \{a_1, a_2\}$ and $g_2 \{a_3, a_4\}$. Since Point $a_5$ is within $\epsilon$ from both $a_1$ and $a_3, a_4$ in $g_2$, the two groups are merged into a single group. Therefore, the output of the query is $\{5\}$. Any overlapping point will cause groups to merge and hence there is no need to add a special clause to handle overlaps.

## 5 Applications

In this section, we present application scenarios that demonstrate the practicality and the use of the various semantics for the proposed Similarity Group-by operators.

**Example 3**: Mobile Ad hoc Network (MANET) is a self-configuring wireless network of mobile devices (e.g., personal digital assistants). A mobile device in a MANET communicates directly with other devices that are within the range of the device’s radio signal or indirectly with distant mobile devices using gateways (i.e., intermediate mobile devices, e.g., $m_1$ and $m_2$ in Figure 3). In a MANET, wireless links among nearby devices are established by broadcasting special messages. Radio signals are likely to overlap. As a result, uncareful broadcasting may result in redundant messages, contention, and collision on communication channels. Consider the Mobile Devices table in...
Figure 3 that maintains the geographic locations of the mobile devices in a MANET. In the following, we give example queries that illustrate how MANETs can tremendously benefit from SGB-All and SGB-Any operators.

**Query 1: Geographic areas that encompass a MANET.** A mobile device, say \( m \), belongs to a MANET if and only if \( m \) is within the signal range from at least one other device mobile. The SGB-ANY semantics identifies a connected group of mobile devices using signal range as a similarity grouping threshold.

```sql
SELECT ST_Polygon(Device-lat, Device-long)
FROM MobileDevices
GROUP BY Device-lat, Device-long
DISTANCE-TO-ALL L2 WITHIN SignalRange
```

Referring to the mobile devices in Figure 3a, the output of Query 1 returns a polygon that encompasses mobile devices \( m_1 \ldots m_6 \).

**Query 2: Candidate gateway mobile devices.** A gateway represents an overlapping mobile device that connects two devices that are not within each other’s signal range. The SGB-All FORM-NEW-GROUP inserts the overlapped devices into a new group. Therefore, those devices in the newly formed group are ideal gateway candidates.

```sql
SELECT COUNT(*)
FROM MobileDevices
GROUP BY Device-lat, Device-long
DISTANCE-TO-ALL L2 WITHIN SignalRange
```

The output of Query 2 returns the number of candidate gateway mobile devices. Along the same line, identifying mobile devices that cannot serve as a gateway is equally important to a MANET. SGB-All ELIMINATE identifies mobile devices that cannot serve as a gateway by discarding the overlapping mobile devices.

**Example 4: Location-based group recommendation in mobile social media.** Several social mobile applications, e.g., WhatsApp and Line, employ the frequent geographical location of users to form groups that members may like to join. For instance, users who reside in a common area (e.g., within a distance threshold) may share similar interests and are inclined to share news. However, members who overlap several groups may disclose information from one group to another and undermine the privacy of the overlapping groups. Query 3 demonstrates how SGB-ALL allows forming location-based groups without compromising privacy.

**Query 3: Forming private location-based groups.** The various SGB-All semantics form groups while handling ON-OVERLAP options that restrict members to join multiple groups. In Query 3, we assume that Table UsersFrequentLocation maintains the users’ data, e.g., user-id and frequent location. The user-defined aggregate function List-ID returns a list that contains all the user-ids within a group.

```sql
SELECT List-ID(user-id),
ST_Polygon(User-lat, User-long)
FROM Users – Frequent – Location
GROUP BY User-lat, User-long
DISTANCE-TO-ALL L2 WITHIN Threshold
```

The output of Query 3 returns a list of user-ids for each formed group along with a polygon that encompasses the group’s geographical location. The JOIN-ANY semantics recommends any one arbitrary group for overlapping members in this case will not be able to join multiple groups. The ELIMINATE semantics drops overlapping members from recommendation, while FORM-NEW-GROUP creates dedicated groups for overlapping members.

### 6 Algorithms for SGB-ALL

In this section, we present an extensible algorithmic framework to realize similarity-based grouping using the distance-to-all semantics with the various options to handle the overlapping data among the groups.

#### 6.1 Framework

Procedure 1 illustrates a generic algorithm to realize SGB-All. This generic algorithm supports the various data overlap semantics using one algorithmic framework. The algorithm breaks down the SGB-All operator into procedures that can be optimized independently. For each data point, the algorithm starts by identifying two sets (Line 2). The first set, namely CandidateGroups, consists of groups that \( p_i \) can join. \( p_i \) can join a group, say \( g \), in CandidateGroups if the similarity predicate is true for all pairs \( \langle p_i, p'_j \rangle \) \( \forall p'_j \in g \). The second set, namely OverlapGroups, includes groups that have some (but not all) of its data points satisfying the similarity predicate. A group, say \( g \), is in OverlapGroups if there exist at least two points \( p \) and \( q \) in \( g \) such that the similarity distance between \( p \) and \( q \) holds and the similarity distance between \( p_i \) and \( q \) does not hold. OverlapGroups serves as a preprocessing step required to handle the semantics of ELIMINATE and FORM-NEW-GROUP encountered in later steps. Figure 4 gives four existing groups \( g_1 \ldots g_4 \) while Data-point \( x \) is being processed. In this case, CandidateGroups contains \( \{g_2, g_3\} \) and OverlapGroups contains \( \{g_1\} \).

Procedure ProcessGroupingALL (Line 3 of Procedure 1) uses CandidateGroups and the ON-OVERLAP clause CLS to either (i) place \( p_i \) into a new group, (ii) place \( p_i \) into existing group(s), or (iii) drop \( p_i \) from the output,
in case of an ON-OVERLAP clause. Finally, Procedure \texttt{ProcessOverlap} (Line 5) uses OverlapGroups to verify whether additional processing is needed to fulfill the semantics of SGB-All.

### 6.2 Finding Candidate and Overlap Groups

In this section, we present a straightforward approach to identify CandidateGroups and OverlapGroups. In Section 6.3, we propose a new two-phase filter-refine approach that utilizes a conservative check in the filter phase to efficiently identify the member groups in CandidateGroups. Then, in Section 6.4, we introduce the refine phase that is applied only if \( L_2 \) is used as the distance metric to detect the CandidateGroups that falsely pass the filter step.

Procedure 2 gives the pseudocode for Naive FindCloseGroups that evaluates the distance-to-all similarity predicate between \( p_i \) and all the points that have been previously processed (Lines 6-15). The grouping semantics (Lines 16-20) identify how the two sets CandidateGroups and OverlapGroups are populated.

#### 6.2.1 Processing New Points

The second step of the SGB-All Algorithm in Procedure 1 places \( p_i \), the data point being processed, into a new group or into an existing group, or drops \( p_i \) from the output depending on the semantics of SGB-All specified in the query.

Procedure 3 (ProcessGroupingALL) proceeds as follows. First, it identifies the cases where CandidateGroups is empty or consists of a single group. In these cases, \( p_i \) is inserted into a newly created group or into an existing group depending on \( p_i \)'s distance from the existing group. Procedure \texttt{ProcessInsert} places the data point \( p_i \) into a group. Next, the ON-OVERLAP clause CLS is consulted to determine the proper course of action. The JOIN-ANY clause arbitrates among the overlapping groups by inserting \( p_i \) into a randomly chosen group. The procedure \texttt{ProcessEliminate} (Line 13) handles the details of processing the ELIMINATE clause. Consider the example illustrated in Figure 4, where CandidateGroups consists of \( \{g_2, g_3\} \). \texttt{ProcessEliminate} drops Point \( p_i \).

Finally, Procedure \texttt{ProcessNewGroup} (Line 15) processes the FORM-NEW-GROUP clause. It inserts \( p_i \) into a temporary set termed \( S' \) for further processing. The SGB-All with FORM-NEW-GROUP option forms groups out of \( S' \) by calling SGB-All recursively until \( S' \) is empty.

#### 6.2.2 Handling Overlapped Points

The final step of SGB-All in Procedure 1 processes the groups in the Set OverlapGroups. OverlapGroups consists of groups, where each group has some data points (but not all of them) that satisfy the similarity predicate with the new input point \( p_i \). This step is required by the
In this section, we introduce a Bounds-Checking approach.

### 6.3 The Bounds-Checking Approach

In this section, we introduce a Bounds-Checking approach to optimize over Procedure Naive FindCloseGroups. Consider the data points of Group $g$ illustrated in Figure 5. Procedure Naive FindCloseGroups performs six distance computations to determine whether a new data point $x$ can join Group $g$. To reduce the number of comparisons, we introduce a bounding rectangle for each Group $g$ in conjunction with the similarity threshold $\epsilon$ so that all data points that are bounded by the rectangle satisfy the distance-to-all similarity predicate. For example, Data Element $x$ in Figure 5 is located inside $g$'s bounding rectangle. Therefore, $g$ is a candidate group for $x$.

**Definition 5:** Given a set of multi-dimensional points and a similarity predicate $\xi_{\delta, \epsilon, \cdot}$, the $\epsilon$-All Bounding Rectangle $R_{\epsilon-All}$ is a bounding rectangle such that for any two points $x_i$ and $y_i$ bounded by $R_{\epsilon-All}$, the similarity predicate $\xi_{\delta, \epsilon, \cdot}(x_i, y_i)$ is true.

Consider Figure 5: where the bounding rectangle $R_{\epsilon-All}$ is constructed for a group that consists of a single Point $a_1$, where $\epsilon = 2$ and the sides of the rectangle are $2\epsilon$ by $2\epsilon$ centered at $a_1$. After inserting the second Point $a_2$ into $g$, as in Figure 5, $R_{\epsilon-All}$ is shrunken to include the area where the similarity predicate is true for both Points $a_1$ and $a_2$. The invariant that $R_{\epsilon-All}$ maintains varies depending on the distance metric used. For the $L_{\infty}$ distance metric, $R_{\epsilon-All}$ is updated such that if a Point, say $x_i$, is inside $R_{\epsilon-All}$, then $x_i$ is guaranteed to be within Distance $\epsilon$ from all the points that form Group $g$. For the Euclidean distance, the invariant that $R_{\epsilon-All}$ maintains is that if a point, say $x_i$, is outside $R_{\epsilon-All}$, then $x_i$ cannot belong to Group $g$. In this case, if $x_i$ is inside $R_{\epsilon-All}$, it is likely that $x_i$ is within distance $\epsilon$ from all the points inside $R_{\epsilon-All}$. Hence, for the Euclidean distance, $R_{\epsilon-All}$ is a conservative representation of the group $g$ and serves as a filter step to save needless comparisons for points that end up being outside of the group. We illustrate in Figures 5 and how to maintain these invariants when a new point joins the group. We use the case of $L_{\infty}$ for illustration. When a new point $x_i$ is inside the bounding rectangle $R_{\epsilon-All}$ of Group $g$, then $x_i$ is within Distance $\epsilon$ from all the points in the group, and hence will join Group $g$. Once $x_i$ joins Group $g$, the bounds of Rectangle $R_{\epsilon-All}$ are updated to retain the truth of $R_{\epsilon-All}$'s invariant. The sides of $R_{\epsilon-All}$ will need to shrink and will be updated as illustrated in Figures 5.

Notice that deciding membership of a point into the group requires a constant number of comparisons regardless of the number of points inside Group $g$. Furthermore, the maintenance of the bounding rectangle of the group takes constant time for every inserted point into $g$. Also, notice that $R_{\epsilon-All}$ stops shrinking if its dimensions reach $\epsilon \times \epsilon$, which is a lower-bound on the size of $R_{\epsilon-All}$. Figure 5 gives the updated $R_{\epsilon-All}$ after Point $a_3$ is inserted into the group.

Procedure 4 gives the pseudocode for Bounds-Checking FindCloseGroups. The procedure uses the $\epsilon$-All bounding rectangle to reduce the number of distance computations needed to realize FindCloseGroups using the $L_{\infty}$ distance metric. Procedure PointInRectangleTest (Line 4) uses the $\epsilon$-All rectangle to determine in constant time whether $g_j$ is a candidate group for the input point. Procedure OverlapRectangleTest (Line 6) tests whether the $\epsilon$-All rectangle of $p_i$ overlaps Group $g_j$'s bounding rectangle. In case of an overlap, all data points in $g_j$ are inspected to verify whether the overlap is nonempty. The correctness of the $\epsilon$-All bounding rectangle for the $L_{\infty}$ distance metric
follows from the fact that the rectangles are closed under intersection, i.e., the intersection of two rectangles is also a rectangle.

A major bottleneck of the bounding rectangles approach is in the need to linearly scan all existing bounding rectangles that represent the groups to identify sets CandidateGroups and OverlapGroups, which is costly. To speedup Procedure Bounds-Checking FindCloseGroups, we use a spatial access method (e.g., an R-tree [17]), to index the $R_{e-All}$ bounding rectangles of the existing groups.

Procedure 5 gives the pseudocode for Index Bounds-Checking FindCloseGroups. The procedure performs a window query on the index Groups_IX (Line 4) to retrieve the set $GSet$ of all groups that intersect the bounding rectangle $R_{p_i}$ for the newly inserted point $p_i$. Next, it iterates over $GSet$ (Lines 4-11) and executes PointInRectangleTest to determine whether the inspected group belongs to either one of the two sets CandidateGroups or OverlapGroups. Finally, the elements of OverlapGroups are inspected to retrieve the subset of elements that satisfy the similarity predicate.

Refer to Figure 6 for illustration. An R-tree index, termed Groups_IX, is used to index the bounding rectangles of the groups discovered so far. In this case, Groups_IX contains bounding rectangles for Groups $g_1$-$g_4$. Given the newly arriving Point $x$, a window query of the $e$-All rectangle for $x$ is performed on Groups_IX that returns all the intersecting rectangles; in this case, $g_1$, $g_2$, and $g_3$. The outcome of the query is used to construct the sets CandidateGroups and OverlapGroups.

6.4 Handling False Positives $L_2$

In this section, we study the effect of using $L_2$ as a similarity distance function on the SGB-All operator. Refer to Figure 7a for illustration. In contrast to the $L_\infty$ distance, the set of points that are exactly $\epsilon$ away from $a_1$ in the $L_2$ metric space form a circle. Inserting $a_2$ (Figure 7b) is correct using the $L_\infty$ distance since $a_2$ is inside the $\epsilon$-All rectangle of $a_1$’s group. However, under the $L_2$ distance, $a_2$ is more than $\epsilon$ away from $a_1$ since $a_2$ lies outside $a_1$’s $\epsilon$-circle. As a result, all points that are inside $a_1$’s $\epsilon$-All group rectangle but are outside the $\epsilon$-circle (i.e., the grey-shaded area in Figure 7c) falsely pass the bounding rectangle test.

Procedure Naive FindCloseGroups in (Procedure 2) inspects all input data points. Therefore, the problem of false-positive points does not occur. On the other hand, the Bounds-Checking approach introduced in Procedures 4 and 5 uses the $\epsilon$-All rectangle technique to identify the sets CandidateGroups and OverlapGroups and hence must address the issue of false-positive points for the $L_2$ distance metric.

We introduce a Convex Hull Test to refine the data points that pass the Bounds-Checking filter step. Given a group of points, a convex hull [18] is the smallest convex set of points within a group. In Figure 7c, the points $a_1$-$a_5$ form the convex hull set for Group $g$. Based on the SGB-All semantics, the diameter of the convex hull (i.e., the
Fig. 7: (a) The $\epsilon$-radius circle in $L_2$; (b) The problem of false positive for $L_2$; (c) The $\epsilon$-convex hull test for $\epsilon = 6$.

Procedure 6: Convex Hull Test

```
Input: $p_i$: data point, $g$: existing group
Output: True if $p_i$ is not false positive, False otherwise
1 ConvexHullSet ← getConvexHull($g$)
2 if $p_i$ inside convex hull then
3 return True
4 else
5 farthestPoint ← getMaxDistElem(ConvexHullSet, $p_i$)
6 if distance(farthestDistPoint, $p_i$) <= $\epsilon$ then
7 return True
8 end
9 end
10 return False
```

The Convex Hull Test, illustrated in Procedure 6, verifies whether a point is a false-positive. This additional test can be inserted immediately after (Line 4) in Procedure 4 or immediately after (Line 6) in Procedure 5. Consequently, any new point that lies inside a group’s convex hull (e.g., Point $y$ in Figure 7b) satisfies the similarity predicate. In addition, in order to verify points that are outside the convex hull (e.g., Point $x$ in Figure 7b), it is enough to evaluate the similarity predicate between $p_i$ and the convex hull. The correctness of the convex hull test follows from the fact that the convex hull set contains the farthest point from $p_i$, say $p_f$. Therefore, it is sufficient to evaluate the similarity predicate between $p_i$ and $p_f$ (e.g., Point $x$ and Point $a_3$ in Figure 7). Section 8.1 discusses the complexity of the convex hull approach.

7 Algorithms for SGB-Any

In this section, we present an algorithmic framework to realize similarity-based grouping using the distance-to-any semantics. The generic SGB-Any framework in Procedure 7 proceeds as follows. For each data point, say $p_i$, Procedure FindCandidateGroups (Line 2) uses the distance-to-any similarity predicate to identify the set CandidateGroups that consists of all the existing groups that $p_i$ can join. In contrast to SGB-All, in the distance-to-any semantics, a point, say $p_i$, can join a candidate group, say $g$, when $p_i$ is within a predefined similarity threshold from at least one another point in $g$. Procedure ProcessGroupingANY (Line 3) inserts $p_i$ into a new or an existing group.

Procedure 7: Similarity Group-By ANY Framework

```
Input: $P$: set of data points, $\epsilon$: similarity threshold, $\delta$: distance function, Points\_IX: spatial index
Output: Set of groups $G$
1 for each data element $p_i$ in $P$ do
2 CandidateGroups ← FindCandidateGroups($p_i$, Points\_IX, $\epsilon$, $\delta$)
3 ProcessGroupingANY($p_i$, CandidateGroups)
4 end
```

7.1 Finding Candidate Groups

A Naive FindCandidateGroups approach similar to Procedure 2 can identify the set CandidateGroups. However, this solution incurs many distance computations, and brings the upper-bound time complexity of the SGB-Any framework to $O(n^2)$. The filter-refine paradigm using an $\epsilon$-group bounds-checking approach while applying a distance-to-any predicate (i.e., similar to Procedures 4-6) suffers from two main challenges. By drawing squares of size $\epsilon \times \epsilon$ around the input point and forming a bounding rectangle that encloses all these squares results in a consecutive chain-like region and the area of false-positive progressively increases in size as we add new data points. Furthermore, the convex hull approach to test for false-positive points cannot be applied in SGB-Any as it suffers from false-negatives caused by the fact that the length of the diameter of the convex hull can actually be more than $\epsilon$ in the case of SGB-Any. Details are omitted here for brevity.

Consequently, FindCandidateGroups in Procedure 8 uses an R-tree index, termed Points\_IX. Points\_IX maintains the previously processed data points to efficiently find CandidateGroups. Refer to Figure 8 for illustration. For an incoming point, say Point $x$, an $\epsilon$-rectangle (Line 2 of Procedure 8) is created to perform a window query on Points\_IX to retrieve PointsSet (Line 3). PointsSet corresponds to the points that are within epsilon from $x$, e.g., $\{a_3, c_1, c_2, c_3, b_1, b_2\}$. Based on the semantics of SGB-Any, CandidateGroups contains the groups that cover the points in PointsSet. For instance, point $a_3$ belongs to $g_1$, points $\{c_1, c_3\}$ belong to $g_2$, and points $\{b_1, b_2\}$ belong to group $g_3$. Hence, CandidateGroups = $\{g_1, g_2, g_3\}$. Procedure GetGroups (Line 7) employs a Union-Find data structure [19] to keep track of existing, newly created, and merged groups (see Figure 9) to efficiently construct CandidateGroups given PointsSet.

7.2 Processing New Points

Procedure 9 gives the pseudocode for ProcessGroupingANY. Lines 1-6 identify the cases when CandidateGroups is empty, or when it consists of one group. In these cases, $p_i$ is inserted into a newly created
Procedure 8: FindCandidateGroups

Input: $p_i$: data point, $Points_{IX}$: spatial index, $\delta$: distance function, $\epsilon$: similarity threshold

Output: CandidateGroups

1 $CandidateGroups \leftarrow \text{NULL}$
2 $R_{ps} \leftarrow \text{CreateBoundingRectangle}(p_i, \epsilon)$
3 $PointsSet \leftarrow \text{WindowQuery}(p_i, R_{ps}, Points_{IX})$
4 if $\delta$ is $L_2$ then
5 \quad $PointsSet \leftarrow \text{VerifyPoints}(Points_{IX}, \delta, \epsilon)$
6 end
7 $CandidateGroups \leftarrow \text{GetGroups}(PointsSet)$
8 insert $p_i$ into $Points_{IX}$

Fig. 8: (a) SGB-Any: Performing a window query on $Points_{IX}\epsilon = 4$ using $L_{\infty}$ (b) The disjoint data structure Union-Find is used to maintain existing groups.

using the index is $O(n \log n)$. The worst-case and best-case running times, and detailed analysis are given in the Appendix.

Table 1: SGB-All Complexity for the $L_{\infty}$ distance

| JOIN-ANY | ELIMINATE | FORM-NEW GROUP |
|----------|-----------|----------------|
| $O(n^2)$ | $O(n^2)$ | $O(n^3)$ |
| Bounds-Checking | $O(n(G))$ | $O(n(G))$ | $O(mn(G))$ |
| on-the-fly Index | $O(n \log(n))$ | $O(n \log(n))$ | $O(m \log(n))$ |

Table 2: Performance Evaluation Queries on TPC-H

8.2 Implementation

We realize the proposed SGB operators inside PostgreSQL. In the parser, the grammar rules, and actions related to

8.3 Evaluation

Table [1] summarizes the average-case running time of SGB-All using the proposed optimizations for the $L_{\infty}$ distance metric. The All-Pairs algorithm corresponds to naive FindClosestGroups in Procedure [1]. Similarly, Bounds-Checking and On-the-fly Indexing corresponds to the Bounds-Checking and Index Bounds-Checking optimizations, where $|G|$ is the number of output Groups and $m$ is the recursion depth for the ON-OVERLAP FORM-NEW. In addition, the average-case running time of SGB-Any when
the “SELECT” statement syntax are updated with similarity keywords (e.g., DISTANCE-TO-ALL and DISTANCE-TO-ANY) to support the SGB query syntax. The parse and query trees are augmented with parameters that contain the similarity semantics (e.g., the threshold value and the overlap action). The Planner and Optimizer routines use the extended query-tree to create a similarity-aware plan-tree. In this extension, the optimizer is manipulated to choose a hash-based SGB plan.

The executor modifies the hash-based aggregate group-by routine. Typically, an aggregate operation is carried out by the incremental evaluation of the aggregate function on the processed data. However, the semantics of ON-OVERLAP ELIMINATE and ON-OVERLAP FORM-NEW-GROUP can realize final groupings only after processing the complete dataset. Therefore, the aggregate hash table keeps track of the existing groups in the following way. First, the aggregate hash table entry (AggHashEntry) is extended with a TupleStore data structure that serves as a temporary storage for the previously processed data points. Next, referring to the Bounds-Checking FindCloseGroups presented in Procedure 4, each group’s bounding rectangle is mapped into an entry inside the hash directory. Bounds-Checking FindCloseGroups linearly iterates over the hash table directory to build the sets CandidateGroups and OverlapGroups. The Index Bounds-Checking in Procedure 5 employs a spatial index to efficiently look up all existing groups a data point can join. Consequently, we extend the executor with an in-memory R-tree that efficiently indexes the existing groups’ bounding rectangles.

In the implementation of FindCloseGroupsAny in Procedure 8, a spatial index is created to maintain the set of points that have been processed and assigned to groups. Moreover, we extend the executor with the Union-Find data structure Disjoint-set forest to support the operations GetGroups and MergeGroupsInsert.

8.3 Datasets

The goal of the experimental study is to validate the effectiveness of the proposed SGB-All and SGB-Any operators using the optimization methods discussed in Sections 6 and 7. The datasets used in the experiments are based on the TPC-H benchmark [20], and two real-world social checking datasets, namely Brightite [2] and Gowalla [21].

Table 2 shows the queries used for performance evaluation experiments on TPC-H data. The multi-dimensional attribute is the combination of different tables. For example, SGB queries, i.e., SGB1/SGB2, are combination of Customer and Order Table, and the number of tuples in the Customer and Order tables is 1500K × SF and 1500 K × SF, respectively, where the scale factor SF ranges from 1 to 60. For Brightite and Gowalla data, SGB queries follow Queries 1 and 3 to cluster users into groups by the corresponding users’ check-in information (i.e., latitude and longitude).

The experiments are performed on an Intel(R) Xeon(R) E5320 1.86 GHz 4-core processor with 8G memory running Linux, and using the default configuration parameters in PostgreSQL. At first, we focus on the time taken by SGB and hence disregard the data preprocessing time, (e.g., the inner join and filter predicates in Query 18). Furthermore, to understand the overhead of new SGB query, we calculate SGB response time with complicated queries (e.g., the SGB Query 3 to 6). In the paper, we only give the execution time of the $L_2$ distance metric because the performance when using the $L_\infty$ distance metric exhibits a similar behavior.

8.4 Effect of similarity threshold $\epsilon$

The effect of the similarity threshold $\epsilon$ on the query runtime is given in Figure 9 for SGB-Any and all three overlap variants of SGB-All; JOIN-ANY, ELIMINATE and FORM-NEW-GROUP. The experimental data consists of 0.5 million records. The similarity threshold $\epsilon$ varies from 0.1 to 0.9.

Consider an unskewed dataset, performing SGB-All using a smaller value of $\epsilon$ (e.g., 0.1 or 0.2) forms too many output groups because the similarity predicate evaluates to true on small groups of the data. Increasing the value of $\epsilon$ forms large groups that decreases the expected number of output groups. Thus, we observe in Figure 9a, 9b, and 9c that the runtime of SGB-All using the various semantics decreases as the value of $\epsilon$ approaches 0.9 with the exception of $\epsilon$ of value 0.7. The slight increase in runtime in the JOIN-ANY and FORM-NEW-GROUP semantics can be attributed to the distribution of the experimental data.

The runtime and speedup in Figure 9a, 9b, and 9c validate the advantage of the optimizations for Bounds-Checking and on-the-fly Index over All-Pairs. The on-the-fly Index approach shows two orders of magnitude speedup over All-Pairs, and Bounds-Checking approach wins one order magnitude faster than that of All-Pairs. The reason is that All-Pairs realizes similarity grouping by inspecting all pairs of data points in the input, and its runtime is bounded by the input size. In contrast, Bounds-Checking defines group bounds in conjunction with the similarity threshold to avoid excessive runtime while grouping. Therefore, the runtime of Bounds-Checking is bounded by the number of output groups. Lastly, indexing output groups using on-the-fly Index alleviates the effect of the number of output groups on the overall runtime and makes it steady across the various ON-OVERLAP options.

The effect of the similarity threshold $\epsilon$ on the query runtime for the SGB-Any query is given in Figure 9d. The experiment illustrates that the runtime for All-Pairs SGB-Any decreases as the value of $\epsilon$ increases. Furthermore, the runtime of the on-the-fly Index method slightly changes. As a result, the speedup between the All Pairs and the on-the-fly Index methods slightly decreases. The runtime result validates that the performance of the on-the-fly Index method is stable as we vary the value of $\epsilon$. The reason

1. http://www.tpc.org/tpch/
2. https://snap.stanford.edu/data/loc-brightkite.html
3. https://snap.stanford.edu/data/loc-gowalla.html
is that the Union-Find data structure efficiently finds and merges the candidate groups. Figure 9 verifies that, for all values of $\epsilon$, the runtime performance of the on-the-fly Index method for SGB-Any is two orders of magnitude faster than the All-Pairs SGB-Any.

### 8.5 Speedup

Figure 10a, 10b, and 10c give the performance and speedup of the Bounds-Checking and on-the-fly Index methods for large datasets with scale factor up to 60. The similarity threshold $\epsilon$ is fixed to 0.2. We do not show the results for the naive approach All-Pairs because its runtime increases quadratically as the data size increases. From Figure 10a, 10b, and 10c we observe that the runtime of the Bounds-Checking method increases as the number and size of groups increases. The on-the-fly Index Bounds-Checking method finds the sets CandidateGroups and OverlapGroups efficiently using the R-tree index, and the runtime of on-the-fly Index Bounds-Checking method increases steadily and is consistently lower than the Bounds-Checking methods. We observe that the speedup of the on-the-fly Index Bounds-Checking method is one order of magnitude better than that of Bounds-Checking.

Figure 10d gives the effect of varying the data size on the runtime of SGB-Any when $\epsilon$ is fixed to 0.2. The TPC-H scale factor (SF) ranges from 1 to 32. We observe that, as the data size increases, the runtime of the All-Pairs method increases quadratically, while the runtime of the on-the-fly Index method has a linear speedup. Moreover, the speedup results in the figure demonstrate that the on-the-fly Index method is approximately three orders of magnitude faster than All-Pairs SGB-Any as the data size increases.

### 8.6 Runtime Comparison with Clustering Algorithms

We compared the runtime of our SGB operators with three clustering algorithms, namely, K-means [9], DBSCAN [10], and BIRCH [12]. Specifically, we use the state-of-the-art implementation of DBSCAN with an R-tree from [22], the similarity threshold $\epsilon$ for both DBSCAN and SGB is set to 0.2, and the parameter $K$ of K-means is set to 20 and 40, respectively. Figure 11 shows the proposed SGB operations significantly outperform DBSCAN, BIRCH and K-means by 1 to 3 order of magnitude on the real-world data respectively. The main reason is that the clustering algorithms scan the data more than once for convergence. On the contrary, SGB operations compute groups on-the-fly, and use group bounda and a spatial index to reduce the overhead of distance computation with processed tuples. In addition, clustering algorithms have to read data from the database system making them slower than our built-in SGB operations.

### 8.7 Overhead of SGB

Figure 12 illustrates the effect of the various data sizes on the runtime of similarity-based groupings and traditional Group-By queries while varying the scale factor from 1G to 20G. The similarity threshold $\epsilon$ is fixed to 0.2. The semantics of the ON-OVERLAP clause plays a key role on the runtime of SGB-All. For instance, the JOIN-ANY variant achieves the best runtime among the SGB-All variants as it places overlapped elements into arbitrarily chosen groups. On the contrary, the FORM-NEW-GROUP incurs additional runtime cost while placing overlapped elements into new groups. The ELIMINATE semantics drops all overlapped elements causing the size of the output groups to shrink. Furthermore, the performance of
traditional Group-by operator is comparable to the SGB-All and SGB-Any variants when using the on-the-fly Index. For instance, the SGB-All ON-OVERLAP JOIN-ANY shows better performance than that of traditional Group-By. The SGB-All ON-OVERLAP ELIMINATE, SGB-All ON-OVERLAP FORM-NEW and SGB-Any shows 15 percent, 40 percent and 20 percent overhead than the traditional Group-By, respectively.

9 Conclusion
In this paper, we address the problem of similarity-based grouping over multi-dimensional data. We define new similarity grouping operators with a variety of practical and useful semantics to handle overlap. We provide an extensible algorithmic framework to efficiently implement these operators inside a relational database management system under a variety of semantic flavors. The performance of SGB-All performs up to three orders of magnitude better than the naive All-Pairs grouping method. Moreover, the performance of the optimized SGB-Any performs more than three orders of magnitude better than the naive approach. Finally, the performance of the proposed SGB operators is comparable to that of standard relational Group-by.

References
[1] J. Han, M. Kamber, and J. Pei, Data mining: concepts and techniques. Morgan kauffmann, 2006.
[2] Y. N. Silva, W. G. Aref, P.-A. Larson, S. S. Pearson, and M. H. Ali, “Similarity queries: their conceptual evaluation, transformations, and processing,” The VLDB Journal, vol. 22, no. 3, pp. 395–420, 2013.
[3] S. Adali, P. Bonatti, M. L. Sapino, and V. Subrahmanian, “A multi-similarity algebra,” in ACM Sigmod Record, vol. 27, no. 2. ACM, 1998, pp. 402–413.
[4] S. Attanu, L. Brunie, and H. Kosch, “Similarity-based operators and query optimization for multimedia database systems,” in IDEAS, 2001, pp. 346–355.
[5] B. Braunmuller, M. Ester, H.-P. Kriegel, and J. Sander, “Multiple similarity queries: A basic dbms operation for mining in metric databases,” KDE, vol. 13, no. 1, pp. 79–95, 2001.
[6] J. Y. Chen and J. V. Carlis, “Similar_join: extending dbms with a bio-specific operator,” in SAC, 2003, pp. 109–114.
[7] H. L. Razente, M. C. N. Barioni, A. J. Traina, and C. Traina Jr, “Aggregate similarity queries in relevance feedback methods for content-based image retrieval,” in SAC, 2008, pp. 869–874.
[8] P. Berkhin, “A survey of clustering data mining techniques,” in Grouping multidimensional data. Springer, 2006, pp. 25–71.
[9] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, “An efficient k-means clustering algorithm: Analysis and implementation,” PAMI, vol. 24, no. 7, pp. 881–892, 2002.
[10] T. Zhang, R. Ramakrishnan, and M. Livny, “Birch: an efficient data clustering method for very large databases,” in ACM SIGMOD Record, vol. 25, no. 2. 1996, pp. 103–114.
[11] S. Guha, R. Rastogi, and K. Shim, “Cure: an efficient clustering algorithm for large databases,” in SIGMOD, vol. 27, no. 2. ACM, 1998, pp. 73–84.
[12] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in KDD, vol. 96, 1996, pp. 226–231.
[13] E. Schallehn, K.-U. Satler, and G. Saake, “Efficient similarity-based operations for data integration,” Data & Knowledge Engineering, vol. 48, no. 3, pp. 361–387, 2004.
[14] C. Zhang and Y. Huang, “Cluster by: a new sql extension for spatial data aggregation,” in GIS, 2007, p. 53.
[15] M. C. N. Barioni, H. Razente, A. Traina, and C. Traina Jr, “Siren: A similarity retrieval engine for complex data,” Vldb, pp. 1155–1158, 2006.

Appendix

We analyze the runtime of SGB-All and SGB-Any. Let \( n, k, |G|, |G_c|, |G_e| \) be the data cardinality, the expected number of points per group, the number of existing groups, the size CandidateGroups, and the size of OverlapGroups, respectively, where \( k \leq n \) and \( |G| \leq n \) as each point can belong to only one group.

1. SGB-All
The runtime for SGB-All is output-sensitive and is influenced by several factors e.g., the ON-OVERLAP options, and the runtimes of FindCloseGroups and Process-Overlap. These factors vary with \( \epsilon \) and with the data distribution. For instance, the number of Groups \( |G| \) can vary from 1 to \( n \) depending on the value of \( \epsilon \). For example, when \( \epsilon \) is very small, \( |G| = n \). Next, we analyze the runtime complexity for Bounds-Checking and, the on-the-fly index
for Bounds-Checking using the various ON-OVERLAP options.

**SGB-All Join-Any.** Refer to Procedure 3 Bound-
Checking. It finds the groups CandidateGroups by lin-
early testing all existing groups (Lines 4-6) to determine
if point $p_i$ can join Group $g_j$. Each test takes constant
time. Thus, the runtime of ON-OVERLAP JOIN-ANY is
bounded by the number of groups, i.e., $O(n |G|)$.

Refer to Procedure 5, Groups $1X$ is an on-the-fly R-tree
that indexes the bounding rectangles of all existing groups.
Given a new data point, say $p_i$, a window query of size $2\epsilon$
on Groups $1X$ finds the groups CandidateGroups that
$p_i$ can join. Thus, the runtime for Procedure 5 (Line 4)
is $O(|G|)$ and the overall runtime of ON-OVERLAP
JOIN-ANY is $O(n \log |G|)$. When $|G| = n$ (the number of
inputs tuples), the worst-case runtime of the on-the-fly
Index for Bounds-Checking ON-OVERLAP JOIN-ANY is
no better than $O(n \log n)$. In contrast, when $|G|$ is constant,
e.g., 1, the best-case runtime is $O(n)$. Finally, the average-
case runtime of the on-the-fly Index for Bounds-Checking
is $O(n \log |G|)$.

**SGB-All Eliminate.** The semantics of ON-OVERLAP
ELIMINATE incurs additional $(k |G_c|)$ time while in-
specting Set OverlapGroups to retrieve the subset that
satisfies the similarity predicate (Lines 8-10) in Proce-
dure 3 and (Lines 10-12) in Procedure 5. In addition,
ProcessEliminate (Line 13) in Procedure 3 incurs addi-
tional cost of $|G_c|$ to update the bounds of the candidates
groups after removing the overlapped points. Thus, the
runtime of Bounds-Checking ON-OVERLAP ELIMINATE is
$O(n (|G| + |G_c| + |G_v|) k)$ while the runtime of on-the-fly
Index for Bounds-Checking ON-OVERLAP ELIMINATE is
$O(n (\log |G| + |G_c| + |G_v|) k)$. Naturally, $k = n/|G|$, so
the runtime of on-the-fly Index for Bounds-Checking
ON-OVERLAP ELIMINATE is $O(n (\log |G| + |G_c| +
n |G_v|/|G|))$. In the worst-case, $|G| = n, |G_c| = |G|
and $|G_v|/|G| = constant$, and the corresponding runtime
of on-the-fly Index for Bounds-Checking ON-OVERLAP
ELIMINATE is $O(n^2 \log n)$. In contrast, the best-case runtime
is $O(n)$ when the sizes $|G| = |G_v| = |G_c| = 1$. The
average-case runtime is $O(n \log |G|)$ when the sizes
of OverlapGroups $|G_v| \ll n$ and CandidateGroups
$|G_c| \ll n$.

**SGB-All FORM-NEW-GROUP.** Procedures Process-
NewGroups and ProcessOverlapNewGroup insert the
overlapped points into a temporary set $S'$. Upon finding
all points in $S'$, SGB-All recursively performs a new
round of Form-NEW-GROUP while grouping the contents
of $S'$ until $S'$ is empty. Let $m$ be the recursion counter
that is initially 0, and $S_m'$ be the set $S'$ at recursion
stage $m$. Then, $S_0'$ is the input dataset where the size of
$S_0'$ i.e., $|S_0'| = n$. The time cost for each round is
$t_m = O(|S_m'| \leftarrow O(FindCloseGroupsALL) + O(Process-
Overlap))$ that $t_m = O(|S_m'| (|G^m| + |G^m_c| + |G^m_v| k^m))$,
where $|G^m|$, $|G^m_c|$ and $|G^m_v|$ are the number of existing
groups, CandidateGroups, and OverlapGroups at each
round $m$, respectively. Thus, the overall runtime of SGB-
All FORM-NEW-GROUP is the sum of $t_m$ from recursion
depth 0 to $DP$, where $t_m$ is the cost at Recursion Depth
$m$. Then, the complexity of Bounds-Checking is $\sum_{m=0}^{DP} t_m$
where $\sum_{m=0}^{DP} t_m = O(|S_m'| (|G^m| + |G^m_c| + |G^m_v| k^m))$. Similarly, the
time complexity of the on-the-fly index for Bounds-
Checking is $\sum_{m=0}^{DP} O(S_m' (\log |G^m| + |G^m_c| + |G^m_v| k^m))$.
The best-case behavior of Index Bounds-Checking for
FORM-NEW-GROUP occurs when set OverlapGroups is
empty and the size of CandidateGroups is constant. Then,
the best-case runtime is $O(n)$. In contrast, if the recursion
depth is almost $n$, the worst-case runtime is $O(n^3)$. On
average, the recursion counter $m = constant \ll n$ and
$|S_m'| \ll n$, and the complexity is $O(n \log (|G|))$.

The **Convex Hull Test** in Section 6 forms a convex hull
for each group $g_j$ to filter out the false-positive points.
The expected size of the convex hull for one group $g_j$ is
$h$, where $h = \log k$ [23], where $k$ is the expected number
points in $g_j$. Refer to Procedure 6. It takes $O(|g_j|)$ to
test if a point is inside the convex hull (Line 2). Moreover,
given a point, say $p_i$, located outside the convex hull, it
takes $O(|g_j|)$ to obtain the farthest point from $p_i$ (Line
5). Thus, for a group of points, $g_j$, the time to test if $p_i$
can join $g_j$ is $O(\log h + \log k)$; that is $O(\log k \log k)$.
ConvexHullTest is performed for each group that passes the
PointInRectangle test with $O(\log k)$ cost (using $L_\infty$). Thus, the computation cost to extend Procedures 4
and 5 with ConvexHullTest is $O(n |G| \log k)$ for Bounds-
Checking and $O(n \log |G| \log k)$ for the on-the-fly Index
for Bounds-Checking. Finally, the average-case runtime of
the on-the-fly Index for Bounds-Checking when using $L_2$
is $O(n \log |G| \log k)$. Notice that the actual running time
is faster than the average-case because the convex hull test
is executed only if a new point has passed the Group $g_j$’s
rectangle test.

**2 SGB-Any**

Refer to Procedure 8. For each new input point $p_i$, the
window query returns the processed points that are within
$\epsilon$ from $p_i$. Given a set of $n$ points, the complexity of
the window query is $O(n \log n)$. Moreover, Procedures
getGroups and MergeGroupsInsert use Union-Find to
keep track of new, existing, and merged groups. The amor-
tized runtime of Union-Find for $n$ points is $O(m'\alpha(n))$
[19], where $m'$ is the total operations to build new groups,
$m' = |G|$, $\alpha(n)$ is a very slowly growing function, and
$\alpha(n) \leq 4$. Therefore, the average case of Union-Find
running time is $O(n)$, where $m' \leq n$. Hence, the average-
case runtime of SGB-Any using an on-the-fly index is
$O(n \log n) + O(n)$, that is $O(n \log n)$. Also, using $L_2$
requires an additional step (verifyPoints) to filter out the
points that do not satisfy the similarity predicate in Over-
lapGroups (Line 7) with a cost $k'$ per point, where $k'$
is the expected number of points within a window query.
Consequently, the runtime cost of SGB-Any using $L_2$ is
$O(n \log n + n k')$. $k'$ is influenced by $\epsilon$. Thus, the worst-
case runtime when using $L_2$ is $n^2$, when $k' \approx n$. If $k'$
is constant, the average-case runtime is $O(n \log n)$. The
average-case runtime of the on-the-fly Index for SGB-Any
is $O(n \log n)$ for both $L_\infty$ and $L_2$. 

