Compact Similarity Joins

Brent Bryan [#1], Frederick Eberhardt [*2], Christos Faloutsos [#3]

[#]*Machine Learning Department, Carnegie Mellon University*
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
[1]`bryanba@cs.cmu.edu`
[3]`christos@cs.cmu.edu`

[*]*Department of Philosophy, University of California, Berkeley*
314 Moses Hall #2390, Berkeley, CA 94720, USA
[2]`fde@berkeley.edu`

Abstract— Similarity joins have attracted significant interest, with applications in Geographical Information Systems, astronomy, marketing analyzes, and anomaly detection. However, all the past algorithms, although highly fine-tuned, suffer an *output explosion* if the query range is even moderately large relative to the local data density. Under such circumstances, the response time and the search effort are both almost quadratic in the database size, which is often prohibitive. We solve this problem by providing two algorithms that find a *compact* representation of the similarity join result, while retaining all the information in the standard join. Our algorithms have the following characteristics: (a) they are at least as fast as the standard similarity join algorithm, and typically much faster, (b) they generate significantly smaller output, (c) they provably lose no information, (d) they scale well to large data sets, and (e) they can be applied to any of the standard tree data structures. Experiments on real and realistic point-sets show that our algorithms are up to several orders of magnitude faster.

I. INTRODUCTION

In numerous applications, such as Geographical Information Systems, office management systems, anomaly detection and astrophysical analyzes, one is often interested in pairs of records that are similar to each other. Such queries can be formulated as similarity joins: queries that ask for pairs of records from a data set that fall within a specified range based on some metric. Checking each pair of entries individually requires $O(n^2)$ work, where n is the size of the database. Numerous algorithms and data structures have been developed to reduce the computational effort of similarity joins. Most of these strategies use distance information to eliminate those tuples that cannot satisfy the query.

Current algorithms are only concerned with the search for pairs of points satisfying the range query and largely disregard whether the produced output is useful. However, when the local data density is large compared to the query range, the output of a similarity join becomes unwieldy. A group of k data points in a locally dense region likely results in $O(k^2)$ pairs satisfying a specified range query. We refer to this situation as an *output explosion*, since the output becomes quadratic rather than linear in the number of data points. We present two similarity join algorithms that control this output explosion while retaining all the information by using a compact representation of the join output.

The literature contains two general approaches to similarity joins. The first assumes the data is stored in a tree structure [1]. The second makes no assumptions about the presence of an index (e.g. the ϵ-grid-order [2]). Our idea of compressing the output can be applied to both cases. Here we focus on the former case, where an index is given. Our approach typically reduces *both the I/O and computation time*. At worst, the algorithm takes the same amount of time as a standard efficient similarity join algorithm.

The compact representation lends itself to two important tasks: storage savings of results and mining of outliers. First note that the compact output of join queries can be stored efficiently, allowing fast further processing later, a common desire. For example, when an astronomer submits a query to the National Virtual Observatory (NVO), the request is forwarded to several services. The results from these services are returned through and combined by NVO servers. Since the different services return asynchronously, the NVO server will need to store the results until the entire query is answered and retrieved by the astronomer. As this process may take several days, it is important for the results to be as small as possible in order to ensure that all users may be served in an efficient manner.

Secondly, a compact representation will highlight unusual pairs. As we explain later, our algorithms produce groups of pairs so that any two members of the group satisfy the query range. Small-size groups could correspond to outliers, drawing further attention upon them. Applications for this sort of analysis exist in areas such as financial analysis (correlating stock trades to detect fraud) as well as astrophysics, (where an unusual pair of galaxies might be of particular interest to further study [3]).

A. The Problem

We propose a solution to the following problem: *Given a set of n data objects in a metric space, stored in a set of metric index tree structures, produce a compact representation of the output of a similarity or spatial join query.* In particular, we desire a solution that requires minimal (and ideally no) changes to the index structure, and avoids the output explosion problem by not explicitly enumerating all qualifying pairs in the output. For ease of exposition we will focus on when the

metric space can be written as a vector space, although we will mention necessary changes for general metric spaces. We call a pair of data points that falls within the query range ϵ a "link" and a set of data points that mutually satisfy the query range a "group". Given the nature of the problem and the computational limitations there are several desiderata for our algorithm:

Speed
> The algorithm should be no slower than a standard similarity join.

Correctness
> The generated compact representation should be provably equivalent to the traditional representation: it should have no missing links, and no extra links.

Scalability
> The algorithm should scale well with the number of records.

Index-Independence
> The algorithm must be able to run on standard data structures, with minor or no changes.

We now discuss different similarity join and clustering approaches in the literature and show why they are insufficient for our problem.

II. RELATED WORK

While there has not been any work that specifically addresses the output explosion problem, one can imagine solving it by first computing the similarity join and subsequently compressing the result. This approach is appealing as there has been a great deal of work on both efficiently processing similarity join queries and clustering data. However, there are a number of difficulties in combining these techniques, which we shall now discuss.

A. Similarity Joins

There is an extensive literature on efficiently performing similarity joins using a variety of data structures and data assumptions. These methods partition the parameter space into smaller (possibly overlapping) sub-regions in the d-dimensional parameter space.

The most popular of these methods use tree based structures to bound the distances of each branch to the target, allowing for pruning. Typical tree structures include R-trees [4], R*-trees [5], M-trees [6] and Filter trees [7], [8], which primarily differ on how they store large objects. Additional speed-ups are obtained by optimally ordering the access of children in branch nodes and the objects in leaf nodes [1]. Other optimizations have been suggested by [9], [10]; [11], [12] present improvements specifically designed for high dimensional data.

Alternatively, methods such as Spatial Hash-Joins [13], Partition Based Spatial Merge Joins [14] and the ϵ-grid-order method [2] do not compute a tree structure, but explicitly or implicitly bucket the data based on a partition of the parameter space. Queries then need only be performed on the relevant buckets allowing the majority of the parameter space to be pruned. However, both these techniques and the tree based methods explicitly enumerate all links of the similarity join, resulting in an output explosion for the cases we are concerned with here.

B. Clustering

Clustering algorithms can broadly be split into four categories: means/mediod-based algorithms, hierarchical clustering algorithms, clustering using specialized structures and graph clustering algorithms. k-means/mediod algorithms and their variants (e.g. CLARANS) choose k initial medoids, calculate the average distance to them and then attempt to sample better means/medoids [15], [16], [17]. Hierarchical clustering algorithms (e.g. CLIQUE [18]) join nearby points into clusters based on a user defined clustering "granularity" Consequently, clustering decisions can be made locally, since no optimization over the whole data is required. The BIRCH algorithm [19] is related to hierarchical clustering methods, but uses its own specialized structure. It is built around a CF-tree which makes maximum use of the available memory while providing the count, center and sum of squares of the data at each node. It produces good clusters in just one pass over the data and was specifically designed to adhere to I/O-limitations. Finally, graph clustering algorithms based on minimum cuts (e.g. CHAMELEON [20]) identify data clusters by connecting nearby data points. Clusters are then separated into groups by cutting the smallest number of links.

C. Limitations of Clustering Methods for Compact Similarity Joins

While post-processing the results of a similarity join using one of the above clustering algorithms would result in compressed output, none of the clustering algorithms are sufficient to solve the compact similarity join problem due to the additional constraints the join problem imposes. Post-processing using clustering will fail to meet our needs due to one or more of the following reasons:

Cluster Shape
> Arbitrary cluster shapes (e.g. k-means/ mediods and sampling based algorithms) cannot guarantee that all points within a cluster mutually satisfy the query range.

Runtime
> Similarity joins are already expensive; adding a clustering algorithm for post-processing will make them even slower. This restriction excludes hierarchical and graphical methods as well as BIRCH[1].

RAM Limitations
> As we are primarily concerned with cases where an output explosion occurs, it is impossible to hold all links in RAM — excluding any graphical approaches based on minimum cuts.

III. METHOD

An output explosion occurs for a data set when multiple points fall within a hypersphere with a radius equal to the

---

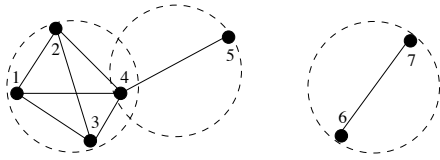[1]The CF-tree would have to be reconstructed each time to be optimal for each new query range.

Fig. 1. An illustration of the output difference for a standard similarity join (solid lines) and our compact similarity join (dotted circles). Note that we reduce the 8 links shown in the diagram to 3 groups ($\{1,2,3,4\}$, $\{4,5\}$ and $\{6,7\}$), a space savings of 50%.
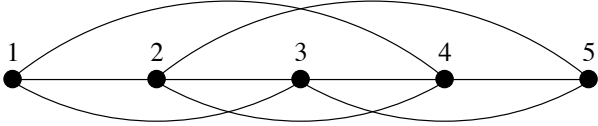

Fig. 2. An example data set with multiple optimal compact representations, when $\varepsilon = 3$. Solid lines indicate the output of a standard similarity join. One optimal compact representation of this join are the three groups: $\{1,2,3,4\}$, $\{2,5\}$ and $\{3,4,5\}$. This example also results in a space savings of 50%.

range query. In these cases, instead of simply returning all pairs of points, we can return groups of points that mutually satisfy the query radius. If a group we return has k points, this implies that all the $\binom{k}{2}$ pairs of data points in the group satisfy the query. For large k the savings are enormous: our approach reports all k points once, whereas other similarity join algorithms return all $O(k^2)$ links individually.

Figure 1 shows 7 data points, with a query range, ε, given by the diameter of one of the dashed circles. The standard similarity join outputs the eight pairs of data points that fall within the query range (represented by the lines between two data points). Instead, a compact representation outputs only 3 groups which contain points that all mutually satisfy the query range; these groups are $\{1,2,3,4\}$, $\{4,5\}$ and $\{6,7\}$. Note that it is possible for points to appear in more than one group, as is the case for point 4. Nevertheless, the output is linear rather than quadratic in the number of points. The circles in Figure 1 show the optimal compact output for the problem.

While the compact representation for the example given in Figure 1 is unique, this does not generally hold. For instance, consider the task of placing the integers on the real line $1, 2, 3, \ldots, 5$ into groups with $\varepsilon = 3$, shown in Figure 2. A standard similarity join would return 9 pairs: $\{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}, \ldots, \{4,5\}$. One optimal approach is to create the groups: $\{1,2,3,4\}, \{2,5\}, \{3,4,5\}$. Similarily the groups $\{1,2,3\}, \{1,4\}, \{2,3,4,5\}$ are also optimal. In both cases, we reduce 9 links to 3 groups, resulting in a 50% output size savings. However, finding an optimum configuration requires a search for the optimal link-graph partition into cliques, which is NP-complete. In this work, we give a near-optimal solution that results in significant time and space savings.

IV. ALGORITHMS

We propose two algorithms to compute the similarity join of a data set in our compact notation. We only assume that the

minimum and maximum distance (similarity) between any two nodes in the tree data structure can be calculated efficiently. This assumption has negligible implications, since most tree structures contain some form of a bounding shape for each node. In particular, these assumptions are upheld by all the commonly used tree structures (e.g. R-trees, R*-trees and Metric trees).

A. Standard Similarity Join Algorithm (SSJ)

A standard similarity join (SSJ) algorithm serves as our benchmark both in terms of runtime and output length. This algorithm recursively descends the tree and outputs all links individually. As such, the number of output links does not depend on the underlying tree structure. However, timing variations are observed due to structure differences in the tree resulting from the order in which data point were inserted.

B. Naive Compact Similarity Join Algorithm (N-CSJ)

Tree structures tend to group nearby points to facilitate quick retrieval of neighbors and executions of range queries. Thus, we can modify the standard similarity join algorithm to exploit the tree's grouping. If the tree structure groups the data well, then the majority of the links will be contained within a single subtree; only a small fraction will cross over between two tree nodes.

Given a query range ε, N-CSJ recursively descends the tree checking whether the maximum distance contained in the bounding shape of each subtree is smaller than ε. If so, the algorithm creates a group that contains all data points in that subtree, using the bounding shape of the node as the group boundary. If not, N-CSJ continues the recursion as before. No effort is made to group links that bridge nodes within the tree structure. In particular, if the bounding shape of a leaf node has a maximum diameter greater than ε then no clusters will be formed and all links will be output in a individually, just as with the SSJ algorithm.

When the maximum diameter of a tree node N is less than the query range, we need only access each sub-node of N once to write out the group. Moreover, since all pairs of data points in N satisfy the query range, no additional distance computations are necessary. Thus, N-CSJ will be faster than SSJ when there are nodes with diameters less than the query range, otherwise, N-CSJ will reduce to SSJ. The pseudo-code for N-CSJ is given in Figure 3 where `ncsj` is true. The early stopping clauses that differentiate SSJ and N-CSJ are shown in italics.

C. Compact Similarity Join Algorithm (CSJ)

Depending on the query range and how the data is grouped within the tree structure there may be many links that cross the nodes' bounding shapes and/or there may be few tree nodes that fall completely within the query range, resulting in links being returned individually.

Our second algorithm attempts to merge these links into the g most recent groups. Whenever we are confronted with an individual link, we consider the maximum distance the

```
1  SIMJOIN(TreeNode n){
2    if maximum diameter of n < range
3      createNewGroup(n)
4    if n is leaf{
5      for each pair of objects of n : pair
6          if distance between items of pair < range
7            if ncsj
8              output(pair)
9            else // if csj(g)
10             mergeIntoPrevGroup(pair)
11   } else{
12     for each child of n : a
13       simJoin(a)
14       for each child of n with index > a : a'
15         if min distance between a and a' < range
16           simJoin(a, a')
17   }
18 }

19 SIMJOIN(TreeNode n1, TreeNode n2) {
20   if maximum diameter of \{n1, n2\} < range
21     createNewGroup(\{n1, n2\})
22   if n1 and n2 are leaves {
23     for each object of n1 : o1
24       for each object of n2 : o2
25         if distance between o1 and o2 < range
26           if ncsj
27             output(pair(o1, o2))
28           else // if csj(g)
29             mergeIntoPrevGroup(pair(o1,o2))
30   } else if n1 is leaf {
31     for each child of n2 : c2
32       simJoin(n1, c2)
33   } else if n2 is leaf {
34     for each child of n1 : c1
35       simJoin(c1, n2)
36   } else {
37     for each child of n1 : c1
38       for each child of n2 : c2
39         simJoin(c1, c2)
40   }
41 }

42 MERGEINTOPREVGROUP(PAIR P) {
43   for(i in 1 to g of recent groups){
44     extend MBR of group i to include p
45     if (diameter of MBR < range)
46       include p in group i, return;
47     else
48       undo extension of MBR
49   createNewGroup(p)
50 }
```

Fig. 3. High level pseudo-code for Similarity Joins. Lines in italic font indicate differences between standard (SSJ) and compact join algorithms (NCSJ & CSJ(g))

bounding shape *would* contain if it *were* extended to include the link. If that distance is smaller than the query range, the link is merged into the group by extending the bounding shape, if necessary. Due to the inherent grouping of the indexing structure, a link is most likely to fit into one of the *recent* groups since they are likely to be in the proximity of this link. If the algorithm cannot fit a link into one of the g previous groups, a new group containing just that link is created. Results for various values of g are shown in Section VI. Figure 3 provides the pseudo-code for CSJ(g) when `ncsj` is false.

The crucial addition compared to N-CSJ is the merge-into-previous-groups routine. This routine allows CSJ(g) to

incorperate links which cross the subtrees into the compact representation, reducing the overall output size. Thus, the observed output difference between N-CSJ and CSJ(g) yields an estimate of the fraction of output links which cross the subtree structure. As we will see in Section VI-A, this difference is often on the order of a factor of two, indicating that a majority of the clusters include links which cross the subtree structure.

D. Algorithm Extensions

The algorithms are implemented as self-joins, but can easily be adapted for *spatial* join queries, where two trees containing different data sets are joined. The algorithms have two subroutines: one for operating on single nodes for self-self joins, and a second that, given two nodes, performs joins between the nodes. In the case of a spatial join only the latter dual tree join is called using a tree from each data set. In this setting the pruning during the *compact* spatial join allows us to indicate that an entire sub-region from each type of tree is within the query range. If the nodes in the two data sets have a different distribution, the inclusion check will often fail. This indicates that the resulting spatial join is unlikely to contain an output explosion. In general, an output explosion occurs if both data sets have a large number of points in the same regions of parameter space. When this is the case, the indexing trees will place small nodes in these dense areas, to ensure the tree remains balanced. Thus, we expect nodes of the two trees to have similar coverage in the areas where there is a risk of output explosion.

Once the compact output is generated, it allows for efficient storage. Individual links can easily be recovered by expanding the returned groups. However, one could use the compact representation to focus on particular regions of interest, or, pass the compact representation to other algorithms for further savings. We believe this latter approach of *maintaining* the savings is the more interesting. There may be many applications that use similarity join output that (with minor adjustments) could perform their operations on and benefit from the compact representation. For example, if the aim is to search for outliers based on the join output, a compact representation already provides a type of pre-sort. After all, we would expect outliers to be separate from large groups of data, so the focus should be on the small groups returned by the compact similarity join.

E. Algorithm Completeness and Correctness

Theorem 1 (Completeness): Given a query range ε, let p_1 and p_2 be any two data points such that $||p_1 - p_2|| \leq \varepsilon$. The output of N-CSJ and CSJ(g) contains the link $l(p_1, p_2)$.

Proof: (Outline) Let N_i with $i = 1, 2, \ldots$ be the leaf node of the tree data structure storing the points p_1 and p_2, let G_j with $j = 1, 2, \ldots$ be a group created to compact the output of N-CSJ or CSJ(g) and let $\max_{MBR}(O)$ be the maximum diameter of the minimum bounding shape of object O.

Case 1: $p_1, p_2 \in N_1$ and $\max_{MBR}(N_1) \leq \varepsilon$

The early stopping rule will create a group G_1 for all members of N_1 and consequently, p_1 and p_2 will be output in G_1. Hence, the output of N-CSJ and CSJ will implicitly contain $l(p_1, p_2)$ (in G_1).

Case 2: $p_1, p_2 \in N_1$ and $\max_{MBR}(N_1) > \varepsilon$

When N-CSJ or CSJ(g) hit N_1, both algorithms check all pairwise combinations of points for satisfaction of the query range. In N-CSJ, all pairs satisfying the query range are enumerated individually in the output, hence p_1 and p_2 are output as a separate link. CSJ(g), when it finds link $l(p_1, p_2)$ will attempt to merge it into the g most recent groups. A successful merge only occurs if, when $p_1, p_2 \in G_j$ for some $j \leq g$, then $\max_{MBR}(G_j) \leq \varepsilon$. In that case p_1 and p_2 are included in G_j and hence the output of CSJ(g) will implicitly contain $l(p_1, p_2)$ in G_j. If the merge attempts are unsuccessful, CSJ(g) will create a new group G_{g+1} containing p_1 and p_2 defined by $MBR(l(p_1, p_2))$. Hence, the output will implicitly contain $l(p_1, p_2)$ in G_{g+1}.

Case 3: $p_1 \in N_1$ and $p_2 \in N_2$

If $\max_{MBR}(N_1, N_2) \leq \varepsilon$ then, similar to case 1, $l(p_1, p_2)$ is (implicitly) contained in the output due to the early stopping rule. Otherwise, all pairs of data points in the two leaves will be considered and the case is similar to case 2. ∎

Theorem 2 (Correctness): Let p_1 and p_2 be points contained in a group G in the output of N-CSJ or CSJ(g). Then $||p_1 - p_2|| \leq \varepsilon$.

Proof: (Outline) Suppose $||p_1 - p_2|| > \varepsilon$.

Case 1: $p_1, p_2 \in N_1$

Then $\max_{MBR}(N_1) > \varepsilon$. Hence the algorithms will consider each pair individually. p_1 and p_2 will not satisfy the query range, hence N-CSJ will not output them among the individual links, i.e. there is no G in the N-CSJ output that would contain the points. Contradiction.

In the case of CSJ(g) the only further case we have to consider is if there is a point p_3 such that $||p_1 - p_3|| \leq \varepsilon$ and $||p_2 - p_3|| \leq \varepsilon$ and p_1, p_3 are already contained in one of the g most recent groups, say G_{g^*}. When CSJ(g) considers merging the link $l(p_2, p_3)$ into the g most recent groups then the algorithm will attempt to extend G_{g^*} to include p_2, p_3, but then $\max_{MBR}(G_{g^*}) > \varepsilon$ due to $||p_1 - p_2|| > \varepsilon$ and the merge will fail. Hence, there is no G in the CSJ(g) output that would contain the points. Contradiction.

Case 2: $p_1 \in N_1$ and $p_2 \in N_2$

Then $\max_{MBR}(N_1, N_2) > \varepsilon$. So the algorithms will consider each pair of points individually and we have the same situation as in the previous case, again a contradiction of the supposition that $||p_1 - p_2|| > \varepsilon$. ∎

V. FURTHER THEORETICAL NOTES

Given a query range ϵ, our aim is to compact the output of a similarity join query into one or more groups, where each group guarantees that all points it contains mutually satisfy the query range. This problem has a well-defined, though possibly non-unique optimal solution. However, given the complexity of the problem we face many trade-offs between computational tractability and approaching an optimal solution. Here we discuss why we believe the approaches above strike the best balance between near-optimal space and fast response time.

A. Group Shapes: Inclusion and Exclusion

Given a group of data points that mutually satisfy the query range, and a test point, we want to quickly determine whether or not the given point is within the query radius of each point in the group, and hence can be added to the group. The naive approach of computing the distance between between all members of the group and the new point to ensure the query radius is satisfied, becomes prohibitively expensive when group sizes become large — that is, whenever output explosions occur. Thus, to ensure that our compact similarity join is as fast as the standard similarity join for all possible data sets, we restrict grouping operations to those which can be performed in constant time.

One approach to quickly check the group membership is to store a bounding shape for each group. Then, when membership queries are performed, one need only check that the maximum distance between the point and the bounding shape satisfies the query range. In two dimensions the bounding shape that contains the most area such that any pair of points within that area mutually satisfy a query range ε is a circle of diameter ε. Using circles for bounding shapes makes the inclusion check very simple. However, determining the optimum centers for the bounding circles adds a cost that depends exponentially on the number of dimensions [21].

Instead, we approximate groups with a minimum bounding hyper-rectangle, and require that the maximal diagonal of this hyper-rectangle is less than the query range. Membership checks, as well as insertions and updates of the boundary for hyper-rectangles can be computed in constant time. Hyper-rectangles are obviously a conservative estimate of the true group's size and hence do not produce optimal grouping. However, since many tree structures utilize hyper-rectangles as bounding shapes for tree nodes, these shapes can be used directly, eliminating the need to compute bounding shapes in many situations.

B. Insertion Ordering

It is not clear how to choose the best group into which a link should be inserted, such that the sum of sizes of all groups is minimized. Keeping a list of groups (in order of creation) and inserting links one at a time into the first group in which they fit (creating new groups as needed) does not provide the optimally compact solution, as seen in the following example. Take 10 points evenly spaced along a line with values from 1 to 10. Using a query range of

7, add the links which satisfy the query range in sorted order (e.g. $1-2, 1-3, \ldots, 1-10, 2-3, \ldots, 9-10$). The resulting groups are $\{1,2,3,4,5,6,7,8\}$, $\{2,3,4,5,6,7,8,9\}$ and $\{3,4,5,6,7,8,9,10\}$, which implicitly contain many of the same links. Moreover, the links can be represented more concisely by the groups $\{1,2,3,4,5,6,7,8\}$, $\{2,9\}$ and $\{3,4,5,6,7,8,9,10\}$, corresponding to a different insertion order. While the links represented in both sets of groups are the same, we prefer the more compact representation, as it will save disk space and hence algorithm runtime. In this example, the savings are roughly 50%, but in larger data sets we have observed the difference to be much more substantial.

This grouping problem becomes more difficult once it is extended into more dimensions as it is not clear how to arrange the groups to minimize their collective size. Naively, one could check all possible partitions of a set of points to find the optimal grouping. However, the number of partitions of a set of n points is given by the Bell numbers[2]: $B_{N+1} = \sum_{k=0}^{n} B_k \binom{n}{k}$, where $B_0 = 1$. Clearly, even when considering small numbers of data points this is infeasible.

The grouping problem manifests itself in two ways. First, when descending the index tree, one would ideally want to consider all possible partitions of the children for grouping. While this will not guarantee an optimal grouping overall, it will guarantee an optimal grouping of these particular subtrees. However, the branching factor of the underlying tree structure will greatly affect the feasibility of such an approach; since R-trees usually have 50 to 100 children at each node, this sort of computation would be prohibitive. The second way in which we confront the ordering problem is when the algorithm reaches a leaf node that it cannot fit into one group. Ideally one would want to consider all partitions of the data points at this leaf. In the first case, we sidestep the problem by considering all *pairwise* combinations of child nodes or leaves. In the latter case we are forced to add the links sequentially into the previous groups.

VI. Specific Experiments and Results

We tested both algorithms on several different real and synthetic data sets, using computation time and output size as measures of performance. We assume that the data is given in a standard tree data structure (R^*-tree by default). Runtime is measured from the moment the algorithm is started until the last tuple of the complete exact result of the query is written to disk. Consequently, runtime will include all disk accesses. Output size is measured by the size in bytes of the resulting output text file. Each data point is zero-padded to ensure it is represented by the same fixed number of bits. A link is written as a single line in the output file containing the two data points, e.g. `0001 0002`, while a cluster is written as the line `0001 0002 0003...`.

All data sets were normalized to fit into the unit square. We consider 9 different query range values equally spaced

on a log-scale, between 2^{-9} and $\frac{1}{2}$, allowing us to compare the performance of very small query ranges with larger ones. We consider four different data sets, three of which are real world examples. These data sets were chosen as several of them are common in the literature, and their relatively small sizes allowed us to build the corresponding tree structures quickly. However, utilizing small data sets results in output explosions occurring only at relatively large query ranges (as the overall density is low). The synthetic and Pacific NW data sets illustrate the effect of larger data sets.

MG County
    Montgomery County Data with 27K datapoints (2D, real).
LB County
    Long Beach County Data with 36K datapoints (2D, real).
Sierpinski3D
    100,000 datapoints from a Sierpinski pyramid (3D).
Pacific NW
    Road segments in the states of Washington, Oregon and Idaho taken from the U.S. Census Tiger database[3] with 1.5 million datapoints (2D, real).

Scatter plots of the data sets are shown in Figure 4.

Except for experiment 4, we used the Spatial Index Library at UC Riverside, a standard R*-Tree implementation developed by Marios Hadjieleftheriou [4].

A. Experiment 1: Computation Time and Output Reduction

For each data set and for each query range value we compared N-CSJ and CSJ(g) with the standard similarity join (SSJ). In the case of CSJ(g) we varied the number of recent groups that were considered in a merge of a new link with $g \in \{1, 2, 3, 4, 5, 10, 20, 50, 100\}$. For each algorithm and each query range we ran 25 iterations. The results are shown in Figure 5, where we plot time (left) and space (right) as a function of query range. For clarity Figure 5 shows the curves for SSJ, N-CSJ and CSJ(10) only. As we show in Figure 6, the performance difference between CSJ(g) for different values of g is minor in comparison to the difference between any CSJ(g) and SSJ or N-CSJ.

For the Montgomery County data we measured the time and space requirements just among the versions of CSJ(g) algorithm. The results are shown in Figure 6, where g is plotted on the x-axis and the y-axis is linear unlike the previous plots. This figure shows that space savings of about 20% can be achieved by considering the 10 most recent groups with virtually no time expense. For larger values of g there are no additional savings.

Several general trends can be observed:

1) N-CSJ outperforms the SSJ in *both* space and time for all data sets we considered. For large query ranges N-CSJ is strictly better, while for small query ranges it matches the performance of SSJ.

[2] See Eric Weisstein, Mathworld, http://mathworld.wolfram.com/BellNumber.html

[3] See: http://www.census.gov/geo/www/tiger/index.html
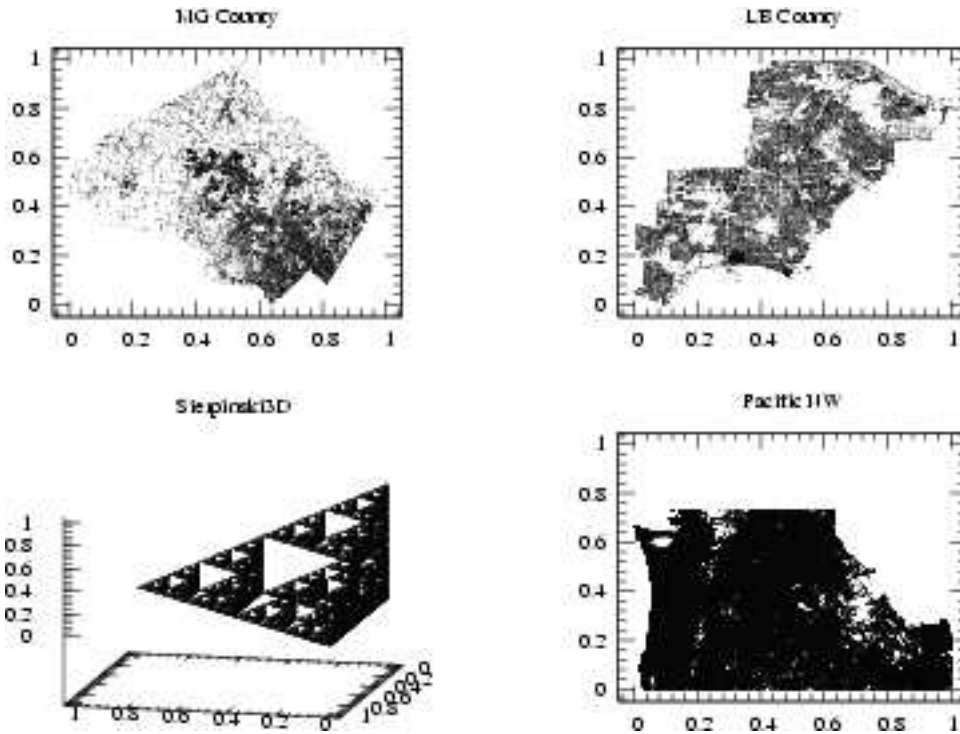[4] See http://www.cs.ucr.edu/~marioh/spatialindex/

Fig. 4. Scatter plots of four data sets we consider. The third is a 3D Sierpinski pyramid, the other three are real world data sets. The Montgomery and Long Beach County data sets are common in the literature. The fourth data set is comprised of all the road segment endpoints in the states of Washington, Oregon, and Idaho (U.S. Census Tiger database).

2) CSJ(10) outperforms N-CSJ and SSJ in every case. Again, for small query ranges CSJ(10) does as well as the others while there are significant savings for large query ranges. In particular, it has significant additional space savings compared to N-CSJ, resulting in both time *and* space savings over SSJ.

3) The query range at which SSJ diverges from N-CSJ and CSJ(g) is a function of data set size and density.

4) The results are consistent even for the large Pacific NW data set with 1.5 million data points.

5) The additional gains of CSJ(g) diminish rapidly as the number g of recent groups that are considered for merging a new link increases (see Figure 6).

B. Experiment 2: Scalability (Number of Data Points)

To test the effects of output explosion, we generated data sets with different numbers of data points from a 3D Sierpinski pyramid. For a fixed query range of 0.125, we measured the runtime and output size of the algorithms, shown in Figure 7 plotted in a linear scale. As expected, the SSJ algorithm results in (quadratic) output explosion. For large numbers of data points, we provide estimates (filled as opposed to empty-signs), since they exceeded free disk space on the system. In comparison, the curves of N-CSJ and CSJ(g) have a near linear increase in both performance measures, indicating that the algorithms control the output explosion. This compression of a quadratic to a linear output size was similiar to the optimal output given for the example problem discussed in Section III.

C. Experiment 3: Distribution of Savings

To determine where the observed savings arise in the previous experiments, we measured the computation time of the algorithm with and without disk accesses. Experiments show that there is no significant difference in the number of disk page and cache accesses between the algorithms, regardless of the page and cache sizes. However, Figure 8 illustrates substantial differences in computational and disk write times between the algorithms. To quantify this difference, we measured the computation time of the algorithms both with and without writing the output to the disk. Results depend on the size of the query range, how the data is distributed, and how well the data is grouped by the underlying tree structure. Most of the time savings come from the early stopping criterion, while moderate savings can be attributed to smaller output files, and hence less disk I/O.

D. Experiment 4: Different Tree Structures

We implemented the algorithm for several different underlying tree structures, including R^*-trees, R-trees and Metric trees. Recall that the only requirement the algorithm assumes of the data structures is the ability to easily calculate the maximum and minimum distance between any pair of subtrees. This is satisfied by all data structures we considered since they all contained some form of bounding shapes.

Preliminary experiments on a variety of data sets found no significant difference in any of the performance measures for any of the algorithms, demonstrating our algorithms can
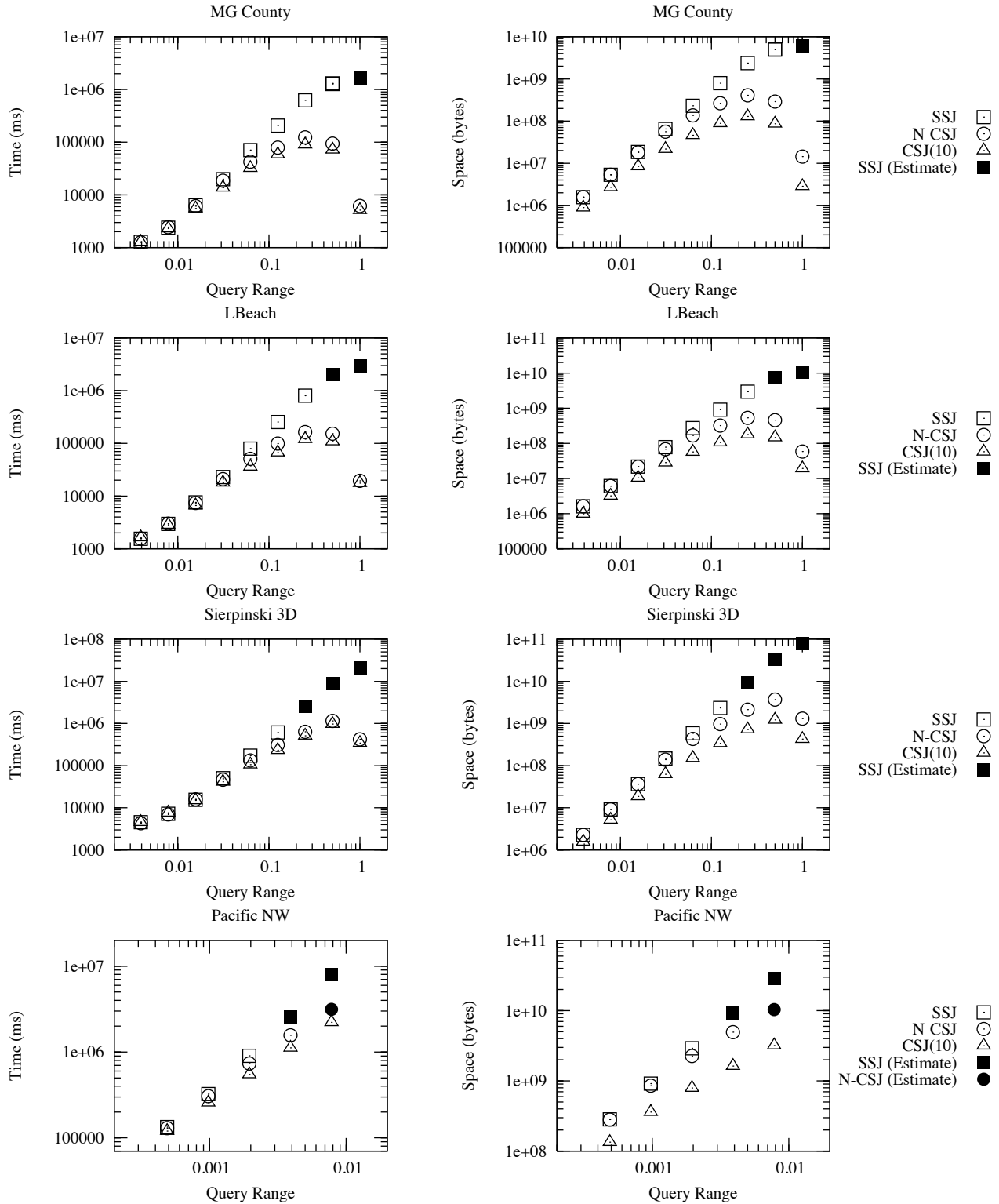
Fig. 5. Results for Experiment 1 on MG County, LB County, Sierpinski3D and Pacific NW, data sets, respectively (rows). All axes *logarithmic*. Left column: time versus query range; right column: output size vs query range. Full, black shapes stand for estimated values, due to *crash*. The traditional SSJ *crashes* or loses consistently, often by *several orders of magnitude*.
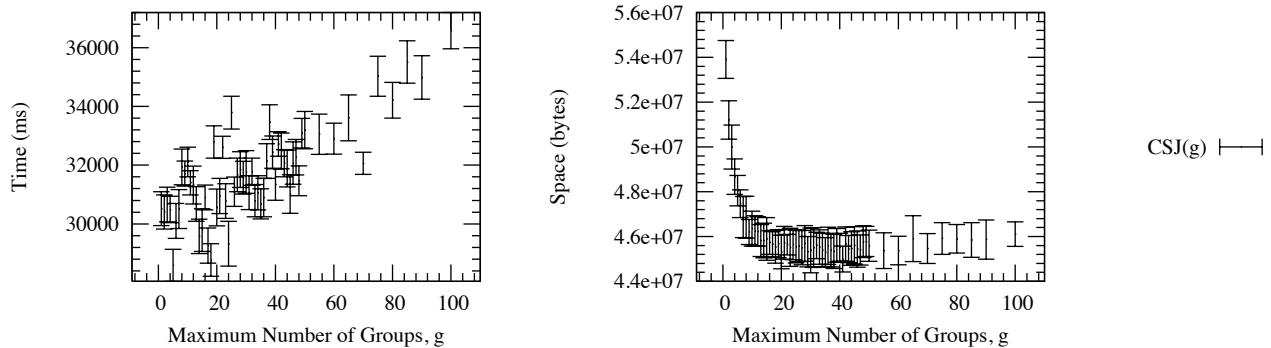
Fig. 6. Results for Experiment 1 on the MG County Data comparing the time savings and output reduction of CSJ(g) for different values of g. There appears to be a linear increase in computation time as g increases. We recommend a choice of $g \simeq 10$, because the output can be reduced by approximately 20% without significant increases in runtime. For larger values of g there is no significant additional output reduction.
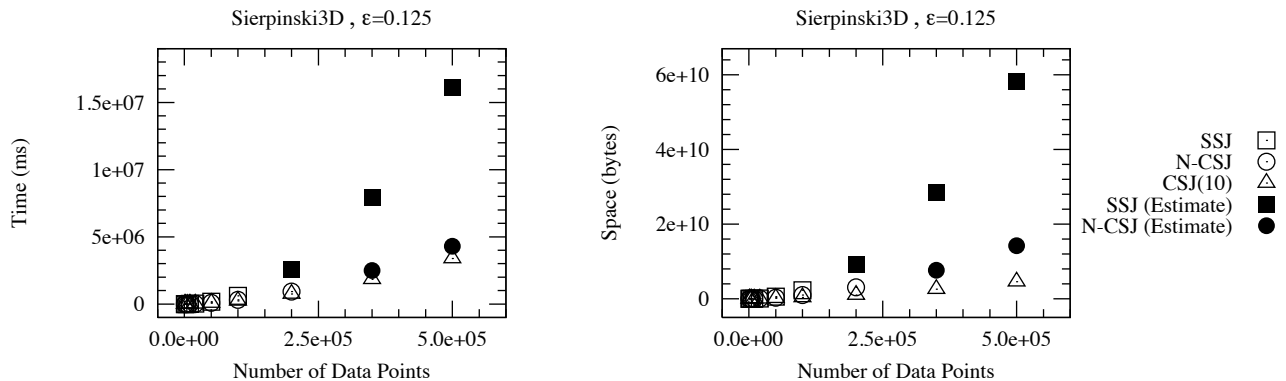

Fig. 7. Results for Experiment 2: Similarity join with varying numbers of data points drawn from a 3D Sierpinski pyramid (in a 3D space). Runtime (left) and output size (right) vs. the number of data points for SSJ, N-CSJ and CSJ(10) with fixed $\varepsilon = 0.125$. Full black shapes stand for estimates, due to *crash*. While SSJ is subject to an output explosion and becomes computationally intractable, N-CSJ and CSJ(g) are able to control this explosion.
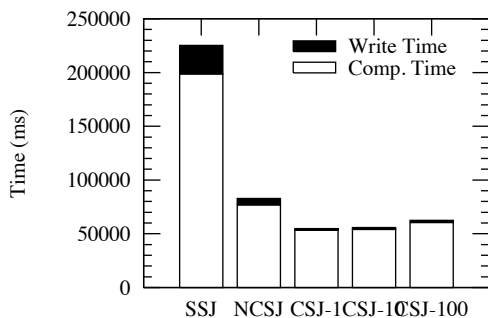

Fig. 8. Comparison of time division between computation and disk I/O in milliseconds for the MG County data where $\varepsilon = 0.1$. The figure shows that most of the savings of N-CSJ and CSJ(g) result from the reduced computation due to the early stopping rule.

be easily applied to any tree structure. Nevertheless, the fact that there were virtually no differences is surprising, as we had expected that the different branching and balance factors, as well as the different techniques for partitioning the data (overlapping or non-overlapping bounding shapes) would have an effect on the gains our algorithm. We plan to pursue this topic in future work.

VII. DISCUSSION

The experiments in Section VI show that our algorithms solve the first problem we posed: compaction of results in the case of an output explosion with no additional computation requirements. However, since both of our algorithms only assume the inclusion property – parent minimum bounding regions (MBRs) include children MBRs – it follows that these results are not restricted to data sets comprised of data points. In particular, the algorithms are equally applicable to metric space, and the gains carry over. Hence, problem 2 is solved as a corollary of the solution we provide for problem 1. The inclusion property is the only essential requirement any index structure must satisfy.

In this work, we have focused on instances where an index structure is given by some tree that stores the data. While this situation is natural with many database applications (as databases often use R-tree variants to store spatial data), it is clearly not universal. Without an index, one would have to build a tree structure on top of the data in order to use the algorithms we presented. Bulk loading algorithms exist that speed up the process [22], [23], [24]. However, tree creation is expensive in computation time and memory. As a result, several techniques for similarity joins have been developed for situations where one does not have the tree structure at

one's disposal. These techniques include the powerful ϵ-grid-ordering, [2] that extend similarity joins to massive multi-dimensional regimes. While not the focus of this paper, we note that the ideas we present can be extended to the ϵ-grid-ordering algorithm. In particular, one need only modify the JoinBuffer function in [2]'s work to add the early termination-as-a-group case.

VIII. CONCLUSION

We have addressed the problem of *output explosion* in similarity joins. We have shown that a win-win is possible for a *compact similarity join*: We can vastly reduce the size of the output of a similarity join *and* at the same time achieve a faster runtime. This is a huge benefit given how expensive similarity joins are in the first place.

The contributions of this work are the following:

- Identification of the *output explosion* issue.
- Proposal of a family of solutions, the *compact similarity join* algorithms ("naive" and with windows).
- Proofs that our algorithms report exactly the same information as the traditional similarity joins do.
- General algorithmic design which can be applied directly to *any* indexing tree. Algorithms can be applied to both vector and metric data sets, as they only require that parent nodes completely cover their children.
- Guidelines on parameter tuning: Even the "naive" N-CSJ achieves significant performance savings; the more elaborate CSJ(g) is even better, achieving a "sweet-spot" when the window size g is ~ 10.
- Extensive experiments on real and synthetic data, illustrating (a) the scalability of our methods with respect to database size N, and (b) dramatic savings over the traditional similarity joins, by up to several orders of magnitude, with respect to both space and time.

A promising future research problem is the analysis of the response time of the methods as a function of the query range ϵ, and also as a function of the intrinsic ("fractal") dimensionality of the input data set.

ACKNOWLEDGMENT

REFERENCES

[1] T. Brinkhoff, H.-P. Kriegel, and B. Seeger, "Efficient processing of spatial joins using r-trees," in *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*. ACM Press, 1993.

[2] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel, "Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data," in *SIGMOD Conference*, 2001.

[3] T. Malik, R. Burns, and A. Chaudary, "Bypass caching: Making scientific databases good network citizens," in *ICDE 05: Proceedings of the International Conference on Data Engineering*. ACM Press, 2005.

[4] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. ACM Press, 1984.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," *SIGMOD Rec.*, vol. 19, no. 2, 1990.

[6] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in metric spaces," in *The VLDB Journal*, 1997.

[7] K. C. Sevcik and N. Koudas, "Filter trees for managing spatial data over a range of size granularities," in *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1996.

[8] N. Koudas and K. C. Sevcik, "Size separation spatial join," in *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*. ACM Press, 1997.

[9] C. Böhm, "A cost model for query processing in high dimensional data spaces," *ACM Trans. Database Syst.*, vol. 25, no. 2, 2000.

[10] J.-P. Dittrich, B. Seeger, D. S. Taylor, and P. Widmayer, "On producing join results early," in *PODS '03: Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM Press, 2003.

[11] J.-P. Dittrich and B. Seeger, "Gess: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces," in *KDD '01: Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM Press, 2001.

[12] K. Shim, R. Srikant, and R. Agrawal, "High-dimensional similarity joins," *Knowledge and Data Engineering*, vol. 14, no. 1, 2002.

[13] M.-L. Lo and C. V. Ravishankar, "Spatial hash-joins," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. ACM Press, 1996.

[14] J. M. Patel and D. J. DeWitt, "Partition based spatial-merge join," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. ACM Press, 1996.

[15] R. T. Ng and J. Han, "Efficient and effective clustering methods for spatial data mining," in *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, J. Bocca, M. Jarke, and C. Zaniolo, Eds. Los Altos, CA 94022, USA: Morgan Kaufmann Publishers, 1994.

[16] M. Ester, H.-P. Kriegel, and X. Xu, "A database interface for clustering in large spatial databases," in *Proceedings of 1st International Conference on Knowledge Discovery & Data Mining, Montreal, Canada, 1995*, 1995.

[17] P. S. Bradley, U. M. Fayyad, and C. Reina, "Scaling clustering algorithms to large databases," in *Knowledge Discovery and Data Mining*, 1998.

[18] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic subspace clustering of high dimensional data for data mining applications," in *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1998, pp. 94–105.

[19] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM Press, 1996, pp. 103–114.

[20] G. Karypis, E.-H. S. Han, and V. Kumar, "Chameleon: Hierarchical clustering using dynamic modeling," *Computer*, vol. 32, no. 8, pp. 68–75, 1999.

[21] J. Matousek, M. Sharir, and E. Welzl, "A subexponential bound for linear programming," *Algorithmica*, vol. 16, no. 4/5, 1996.

[22] Y. J. García, M. A. López, and S. T. Leutenegger, "A greedy algorithm for bulk loading r-trees," in *GIS '98: Proceedings of the 6th ACM international symposium on Advances in geographic information systems*. New York, NY, USA: ACM Press, 1998.

[23] S. Berchtold, C. Böhm, and H.-P. Kriegel, "Improving the query performance of high-dimensional index structures by bulk load operations," *Lecture Notes in Computer Science*, vol. 1377, 1998.

[24] T. Lee and S. Lee, "Omt: Overlap minimizing top-down bulk loading algorithm for r-tree," in *The 15th Conference on Advanced Information Systems Engineering (CAiSE '03)*, Austria, June 2003.