Evaluation of Iceberg Distance Joins

Yutao Shou[1], Nikos Mamoulis[1], Huiping Cao[1],
Dimitris Papadias[2], and David W. Cheung[1]

[1] Department of Computer Science and Information Systems,
University of Hong Kong,
Pokfulam Road, Hong Kong
{ytshou,nikos,hpcao,dcheung}@csis.hku.hk
[2] Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
dimitris@cs.ust.hk

Abstract. The iceberg distance join returns object pairs within some distance from each other, provided that the first object appears at least a number of times in the result, e.g., "find hotels which are within 1km to at least 10 restaurants". The output of this query is the subset of the corresponding distance join (e.g., "find hotels which are within 1km to some restaurant") that satisfies the additional cardinality constraint. Therefore, it could be processed by using a conventional spatial join algorithm and then filtering-out the non-qualifying pairs. This approach, however, is expensive, especially when the cardinality constraint is highly selective. In this paper, we propose output-sensitive algorithms that prune the search space by integrating the cardinality with the distance constraint. We deal with cases of indexed/non-indexed datasets and evaluate the performance of the proposed techniques with extensive experimental evaluation covering a wide range of problem parameters.

1 Introduction

The most common types of spatial joins involve *intersection* (e.g., "find all pairs of roads and rivers that intersect") or *distance* predicates (e.g., "find all hotels that are within 100 meters from the coastline"). A large amount of research work has been devoted to spatial joins due to their applicability in various GIS operations (e.g., map overlay) and high execution cost, which, in the worst case, is quadratic to the size of the data. Several algorithms have been proposed to minimize the cost, considering cases where both [6], one [18, 21], or neither [23, 19, 2] dataset is indexed. Even though these methods were originally developed for intersection joins, they can be easily adapted for distance joins (i.e., by extending the object boundaries by $\epsilon/2$ [17, 20]). Nevertheless, spatial join operators are optimized for queries that require *all* join results.

For typical geographic layers (e.g., rivers with road-lines), the size of the join result is linear to the size of the data. However, users who want to analyze spatial data often impose additional constraints that restrict the output size.

Such a query is the *iceberg distance join*, which, given two relations R, S, a distance threshold ϵ, and a cardinality threshold t, returns object pairs $\langle r, s \rangle$, ($r \in R$, $s \in S$) within distance ϵ from each other, provided that r appears at least t times in the join result. An example of a semi-join query in this class is "find hotels which are close to *at least* t restaurants". In pseudo-SQL it could be expressed as follows:

```
SELECT H.id
FROM Hotels H, Restaurants R
WHERE dist(H.location,R.location) <= ϵ
GROUP BY H.id
HAVING COUNT(*) >= t ;
```

A spatial DBMS would evaluate this query by (i) processing the spatial join using some existing algorithm and (ii) sorting/hashing the qualifying $\langle r, s \rangle$ pairs by $r.id$ to output all $r \in R$ which appear more than t times in the joined pairs. The spatial join operator itself may produce many results before filtering out the ones that do not qualify the cardinality constraint t. In other words, the methodology above is not *output sensitive*, i.e., its cost does not depend on the result size, which is affected by t.

In this paper we propose output sensitive algorithms, by "pushing" the cardinality constraint t into the spatial join operators in order to filter out large parts of the search space. We study the cases of both indexed and both non-indexed joined datasets, by extending efficient algorithms for each case. The rest of the paper is organized as follows. Section 2 provides background and related work. In Section 3 we describe and optimize techniques for iceberg distance joins. Section 4 evaluates the proposed techniques with comprehensive experiments. Finally, Section 5 concludes the paper with a discussion about related work.

2 Background and Related Work

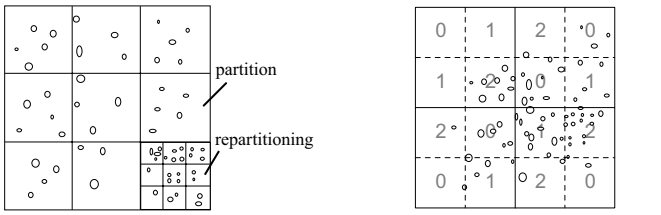
Although our methods are applicable to any space-partitioning access method, we only consider R–trees [11] due to their simplicity and popularity. The R–tree indexes minimum bounding rectangles (MBRs) of objects and processes fast the filter step of the most important query types, i.e., range queries, nearest neighbors [14], and spatial joins. Section 2.1 provides background on intersection join algorithms, while section 2.2 discusses distance join processing. Section 2.3 reviews related work on iceberg queries and motivates the problem studied in this paper.

2.1 Spatial Joins

The R–tree join (RJ) [6] is the most influential algorithm for spatial joins when both datasets are indexed by R–trees. Starting from the two roots, it synchronously traverses the trees, following entry pairs that intersect. Upon reaching the leaves, pairs of intersecting object MBRs are output. RJ employs two heuristics that greatly reduce its computational cost. Given two nodes n_R and n_S to be joined, if an entry e_R in node n_R does not intersect the MBR of node n_S (i.e.,

the MBR of all entries contained in n_S), then there can be no entry $e_S \in n_S$, such that e_R and e_S overlap. Using this observation, RJ performs two linear scans in the entries of both nodes before applying intersection tests, and prunes from each node the entries that do not intersect the MBR of the other node. The second technique (*forward sweep* [2]), is based on plane sweep and applies sorting on one dimension in order to reduce the quadratic number of comparisons for candidate entry pairs. A breadth-first version of RJ with improved I/O cost was proposed in [15].

The Partition Based Spatial Merge join (PBSM) [23] is based on the relational hash join operator and applies on two non-indexed sets. The space is regularly partitioned using an orthogonal grid and objects from both datasets are hashed into the partitions, replicating the ones that span boundaries. Figure 1a illustrates a regular space partitioning incurred by PBSM and some data hashed into the partitions. Data hashed into the same partitions are then joined in memory using plane sweep. If two buckets to be joined do not fit in memory, the algorithm is recursively applied for their contents. Since data from both datasets may be replicated, the simple version of the algorithm may produce duplicates; however, these can be avoided by a simple check [9, 20]. When the data to be joined are skewed, some partitions may contain a large percentage of the hashed objects, whereas others very few objects, rendering the algorithm inefficient. To handle skewed data, the cells of the grid are distributed to partitions according to a hash function and the space covered by a partition is no longer continuous, but consists of a number of scattered tiles. Figure 1b shows such a (round-robin like) spatial hash function, where tiles with the same number are assigned to the same bucket. A parallel, non-blocking version of PBSM was proposed in [20].



(a) data partitioning in PBSM (b) a spatial hash function

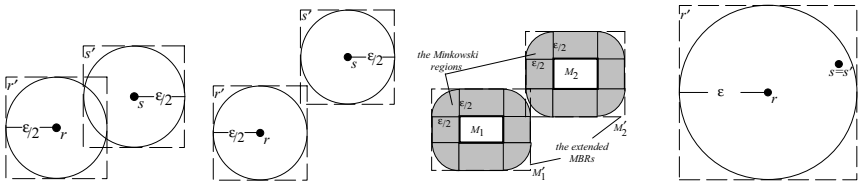
Fig. 1. Example of PBSM

Koudas and Sevcik [16] proposed a hierarchical partitioning that avoids data replication, by assigning each object to the topmost layer where it does not span any grid line. The Spatial Hash Join (SHJ) algorithm [19] defines hash-buckets with irregular extents (which may overlap), such that each object from the inner dataset is hashed to exactly one bucket and only objects from the outer one may be replicated. Finally, the method proposed in [2] applies external plane sweep after sorting both datasets on an axis. This method was later extended in [1] to a unified technique for indexed and non-indexed inputs.

Another class of algorithms aims at joining a non-indexed dataset with an R–tree. The Seeded Tree Join algorithm (STJ) [18] builds a second R–tree using the existing one as a seed and then applies RJ. Slot Index Spatial Join (SISJ) [21] is a hybrid of STJ and SHJ, which uses the existing R–tree in order to determine the bucket extents. If H is the desired number of hash buckets, SISJ finds the topmost level of the tree such that the number of entries is larger or equal to H. These entries are then grouped into H (possibly overlapping) partitions called *slots*, which define the bucket extents. Each slot contains the MBR of the indexed R–tree entries, along with a list of pointers to these entries. The grouping policy used by SISJ is based on the the splitting heuristic of the R*–tree [3]. The hashing and joining phase of SISJ is similar to the corresponding phases of SHJ; all data from the R–tree indexed by a slot are loaded and joined with the corresponding hash-bucket.

2.2 Distance Joins and related queries

Research on the *distance join* operator has mainly focused on point datasets, because they are more relevant in applications of higher dimensionality (e.g., image processing, data mining), where the data are points. If two points r and s are within distance ϵ, then the circles d_r, d_s with centers r, s and radii $\epsilon/2$ intersect, implying that their MBRs r' and s' intersect, as illustrated in Figure 2a. Therefore, given two point sets R and S that fit in memory, we can reduce the $O(|R||S|)$ distance join cost, by applying a plane-sweep algorithm on the MBRs of their circles. Then, we remove any false hits (like the one in Figure 2b) by exact distance calculations. Notice that point extension can be performed dynamically and on-demand for each value of ϵ, without any precomputations. Figure 2c shows how this method can be generalized for R–tree MBRs (i.e., for the RJ algorithm), which are extended to *Minkowsky* regions and approximated by rectangles. An alternative method, which simplifies the join, is to extend by ϵ and approximate only the points (and MBRs) of one dataset. Figure 2d illustrates the extension and approximation of r only, in the configuration of Figure 2a. Distance joins are closely related to *similarity retrieval* in high-



(a) qualifying points (b) a false hit (c) a false hit (MBRs) (d) only r is extended

Fig. 2. Point and MBR extensions for distance join processing

dimensional spaces. [25] present the dynamic construction of a tree that indexes the points using one dimension per level, and can efficiently find point pairs within distance ϵ. [17] develop a generalization of the algorithm in [16] and

compare it with an extension of RJ. Finally, an optimized method for problems of medium dimensionality (e.g., 5–10 dimensions) is proposed in [5]. In summary, these techniques are suitable for high dimensional problems without cardinality constraints and for the case when the qualifying pairs are retrieved from the same dataset.

For the spatial (i.e., 2-dimensional) domain, the algorithms of the previous section are still applicable with some modifications. Furthermore, related work has been proposed in the context of *closest pairs* (CP) queries. Given a threshold k, a CP query retrieves the k closest pairs $\langle r, s \rangle$, $r \in R$, $s \in S$. CP queries have been studied only for the case, where both R and S are indexed by R–trees. [13] present an *incremental* algorithm (for unknown k), which is optimized in [26]. [8] show that a depth-first join algorithm, adapted from RJ, can minimize random I/Os in the presence of a buffer.

2.3 Iceberg queries

The term *iceberg* query was defined in [10] to characterize a class of relational queries that retrieve aggregate values above some specified threshold (defined by a `HAVING` clause). An example of an iceberg query in SQL is shown below:

```
SELECT part, region, sum(quantity)
FROM Sales
GROUP BY part, region
HAVING sum(quantity) >= t;
```

The motivation is that the data analyst is interested in retrieving only exceptional aggregate values that may be helpful for decision support. In this example, we want to find $\langle part, region \rangle$ pairs with many sales, in order to organize advertisement campaigns there. A typical query optimizer would first perform the aggregation for each $\langle part, region \rangle$ group and then find the ones whose aggregate value exceeds the threshold. In order to avoid useless aggregations for the pairs which disqualify the query, [10] present several hash-based methods with output-sensitive cost. Similar techniques for data cubes and On-Line Analytical Processing are proposed by [4, 12].

The efficient processing of iceberg queries is also relevant to Spatial Databases. We focus on iceberg distance joins, since joins are hard by nature and potential improvements are very beneficial. Our proposal includes extensions, which can be easily integrated to existing spatial join algorithms by the Spatial DBMS developer. Notice that the algorithms in [10] are not applicable for our problem, since they consider aggregate queries (not joins) on relational data.

3 Algorithms for iceberg distance joins

Figure 3 illustrates an iceberg distance join example. The user is interested in retrieving hotels and the restaurants within ϵ distance from them, provided that the number of such restaurants is at least $t = 3$. The result of the iceberg join should be $\{\langle h_2, r_2 \rangle, \langle h_2, r_3 \rangle, \langle h_2, r_4 \rangle\}$. On the other hand, h_1 and h_3 are also close

to some restaurants (i.e., they belong to the result of the conventional distance join) but they do not qualify the cardinality threshold t.

The iceberg distance join is an asymmetric operator, since the cardinality constraint t applies only on the occurrences of values from R. In other words, $R \bowtie S \neq S \bowtie R$, in general. Due to this property we have to distinguish between the four cases (i) R and S are not indexed, (ii) R and S are indexed, (iii) only R is indexed, or (iv) only S is indexed. For each of the above cases we discuss how to process the cardinality constraint t together with the join. For joined inputs that fall into case (i), we propose extensions of PBSM [23]. For case (ii), we extend RJ [6] and we discuss adaptations of SISJ [21] for cases (iii) and (iv). Finally, we show how our methods can be applied for variations of the iceberg distance join query.

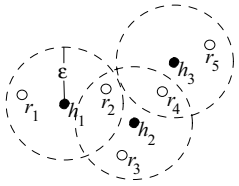


Fig. 3. Iceberg distance join example

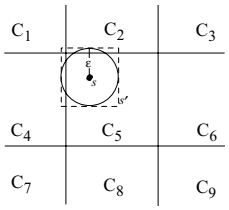


Fig. 4. Replication of points from S

3.1 R and S are not indexed

When neither dataset is indexed, we can process the join, using a hash algorithm based on PBSM [23], called *Partitioned Iceberg Distance Join* (PIDJ). In this section, we present a number of progressively more optimized variations of this method.

Avoiding duplication of results and sorting. In order to avoid sorting the results for validating the cardinality constraint, and at the same time eliminate duplicates, we extend only the objects in S by ϵ. In other words, each point in R is hashed to *exactly one* bucket, based on the cell that contains it. On the other hand, points from S are extended to circles and hashed to multiple partitions (for each cell they intersect). For instance, point $s \in S$ depicted in Figure 4 is hashed to buckets corresponding to cells C_2, C_4, and C_5. Finding these cells is a two-step process. First, we locate the cells, which intersect the circle-bounding rectangle s' (i.e., C_1, C_2, C_4, and C_5). Then, the distance to the qualifying cell boundaries is computed and used to prune false alarms (i.e., C_1).

The second phase of the algorithm, loads for each cell C_x the corresponding buckets R_x and S_x and joins them. If the buckets are too large to be joined in memory, their contents are repartitioned using a finer grid and PIDJ is applied recursively. Here, we have to note that in most cases at least R_x will fit in memory, since only points from S_x are replicated. In such cases repartitioning can be avoided, by building for example a main-memory R–tree for R_x, as suggested by [19, 21]. Because the points from R are not duplicated, we can immediately

count their occurrences in the joined pairs. Thus, after obtaining the results for the bucket pair $\langle R_x, S_x \rangle$, we sort them in memory on $r.id$ and report the pairs for those $r.id$ that appear at least t times.

Filtering out small buckets. If the distance threshold ϵ is small and the cardinality threshold t is large, we can expect that some joined buckets will not return any results. This can happen, when the number of objects in S_x is smaller than t, in which case, it is obvious that no $r \in R_x$ qualifies the query. Therefore, we can use the cardinalities of the buckets, which are already tracked, to prune bucket pairs.

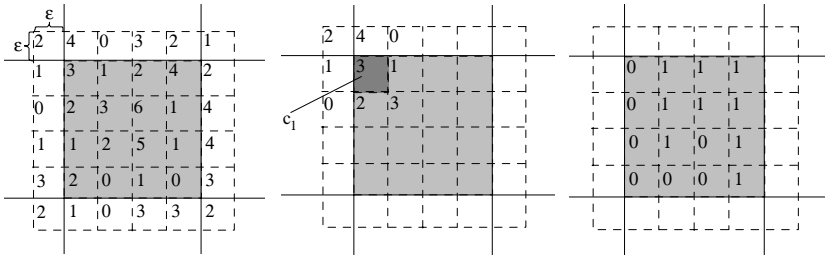
Although this method already avoids loading some bucket pairs, we can still do better by re-scheduling the hashing process. Thus, we first hash S and then R. By doing so, we can immediately spot the cells, which may not contain any results, and mark them. This approach has two benefits. First, all data from R which fall into disqualifying cells are filtered out immediately; hashing them to a bucket (i.e., writing them) is avoided. Second, pages from the memory buffer, which would normally be allocated for disqualified partitions, are now free for use by the remaining partitions.

Filtering out uniform, sparse buckets. Filtering cells that contain fewer than t points from S can reduce the join cost significantly if t is large, but it can still leave many buckets S_x which provide no result. Can we do better? The answer is yes, if we have knowledge about the distribution of points inside each bucket. The rationale is that if the points from S_x are uniformly distributed and the cell extent is much larger than ϵ, we can prune the bucket, or parts of the bucket that may not give any results.

To illustrate the idea, consider the shaded cell C_x of Figure 5a, and assume for simplicity that the length of each side of the cell is $4 \times \epsilon$. While hashing S, we construct a *fine* grid $\mathcal{F}$ with *microcells* of length ϵ at each side, which captures the distribution of objects in S. The dashed cells in Figure 5a show the part of this grid that *influence* cell C_x, i.e., the distribution of points whose extended circle intersects C_x. A counter for each microcell indicates the number of points from S which fall there.

Let us assume that $t = 50$ and have a closer look to the points in S_x that fall into and around the individual microcells of length ϵ in C_x. Observe that the points from R that fall into the microcell c_1 shown in Figure 5b can only join with objects in S_x that fall into c_1 and its surrounding microcells (i.e., the numbered ones in Figure 5b). The total number of objects in these microcells is just $16 < t$, indicating that *no object from R that falls into c_1 can participate in the iceberg join result.* All microcells in C_x are eliminated by this process, thus we can prune the whole C_x. Observe that C_x cannot be pruned by considering only the total number of points in this partition, since $|S_x| = 75 \geq t$.

In addition, this method can also save us I/Os even when C_x is not pruned. Assume for example that $t = 20$. Figure 5c shows a bitmap for C_x, indicating which microcells may contain qualifying points from R. The bitmap can be constructed by adding to a microcell's counter the counters of the surrounding ones. We use this (lightweight) bitmap as a replacement of the fine grid $\mathcal{F}$, to



(a) a fine statistical grid (b) the influence cells of c_1 (c) the filtering bitmap

Fig. 5. Pruning buckets or data space using an ϵ-length grid

prune points from R, while hashing the dataset; if a point $r \in R$ falls into a microcell with a 0, it can be immediately eliminated, reducing the size of R_x on disk. The generation, computation and management of $\mathcal{F}$ and its associated bitmap comes with little overhead, assuming that $|S|$ is much larger than the total number of microcells. For instance, if $\epsilon = 1\%$ of the map's length, we need space for 10,000 integers, or just 40Kb of memory. If ϵ is much smaller, we can use as length a multiple of ϵ, trading memory space with accuracy. The speedup improvement due to this technique is significant, as demonstrated in Section 4.

Figure 6 shows a pseudocode of PIDJ with all optimizations we have discussed. Before join processing, the space is partitioned at two levels, by the regular grid $\mathcal{G}$ which defines the buckets and by the fine grid $\mathcal{F}$ (lines 1–2). A counter for the number of points from S that fall in each microcell, is initialized (lines 3–4). Then, S is read and each $s \in S$ is hashed to the buckets that correspond to the cells intersected by its extended area (lines 5–9). After hashing S, cells with fewer objects than t are pruned. For each cell C_x that is not pruned, a bitmap like the one in Figure 5c is defined (using $\mathcal{F}$) to mark regions, in which points from R can potentially participate in the result. If there are no such regions, we can prune the whole cell (lines 10–16). The next phase of the algorithm hashes each point from R to exactly one cell C_r that contains it, provided that neither this cell nor the corresponding microcell c_r from $\mathcal{F}$ have been pruned (lines 17–21). Finally, the algorithm loads all bucket pairs that correspond to active cells, performs the distance join in memory, and applies the cardinality constraint t to output the qualifying pairs (lines 22–27).

3.2 *R* and *S* are indexed

In this section we propose extensions of the R–tree join algorithm [6] for the iceberg join query. The join can be evaluated by (i) applying RJ to get all object pairs $\langle r, s \rangle$ within distance ϵ, and (ii) sorting them by $r.id$ to bring together pairs with the same $r.id$ in order to validate the threshold constraint t. Sorting (or hashing) in step (ii) is essential, since RJ does not produce the pairs clustered by $r.id$. In the next section we propose a modification of RJ that employs a priority queue to produce the join results clustered. In section 3.2 we propose an

```
Algorithm PIDJ(Objectset R, Objectset S, real e, int t)
1.  define the regular grid G according to the available memory; /* like PBSM */
2.  define the fine grid F according to ε;
3.  for each microcell c_x ∈ F
4.      c_x.count := 0;
5.  for each object s ∈ S
6.      extend s to a circle s_c with center s and radius ε;
7.      for each cell C_s ∈ G intersected by s_c
8.          hash s to the corresponding partition S_s;
9.      c_s := microcell in F containing s; c_s.count++;
10. for each cell C_x ∈ G
11.     if |S_x| < t /*size of corresponding partition of S is small */
12.         prune C_x;
13.     else
14.         use fine grid F to compute c_y.bit, ∀c_y ∈ C_x (and C_x.bitmap);
15.         if C_x.bitmap has no 1's
16.             prune C_x;
17. for each object r ∈ R
18.     C_r := cell in G containing r;
19.     c_r := microcell in F containing r;
20.     if C_r has not been pruned and c_r.bit = 1
21.         hash r into partition R_r that corresponds to C_r;
22. for each cell C_x ∈ G
23.     if C_x has not been pruned
24.         load R_x and S_x and perform the distance join in memory;
25.         for each r ∈ R_x in the join result
26.             if r appears at least t times in the result;
27.                 output all join pairs that contain r;
```

Fig. 6. Partitioned Iceberg Distance Join

improved version of this algorithm that can save I/Os by pruning nodes in the R–tree of R, which may not point to qualifying $r \in R$.

Avoiding sorting. RJ is a *depth-first* search algorithm, since after joining a pair of nodes, it solves the subproblems created by their intersecting entries one by one. Thus it holds in memory only two paths p_R and p_S from the R–trees that index R and S, respectively. On the other hand, our adapted *R–tree based Iceberg Distance Join* (RIDJ) algorithm traverses the trees in a fashion between *depth-first* search and *breadth-first* search. We use a priority queue PQ, which organizes qualifying entry pairs $\langle e_R, e_S \rangle$ with respect to the lower x-coordinate e_R's MBR. Ties are broken by using e_R's node level as a second priority key; higher level entries get higher priority. In case of an additional tie, we use $e_R.id$ as a second key to prevent interleaving of entry pairs at the same level with the same lower x-coordinate. Initially, PQ contains the roots of the trees (i.e., their MBRs and pointers to them). At each step of the algorithm, the first pair is fetched

from PQ and the distance join algorithm is applied for the corresponding nodes. When joining a pair of nodes n_R and n_S, the qualifying entry pairs $\langle e_R, e_S \rangle$ are enqueued in PQ. By continuing this way, the algorithm guarantees that the distance join results $\langle r, s \rangle$ will be clustered by $r.id$. The trade-off is that more memory might be required compared to the original RJ. A similar technique that employs a priority queue to traverse data in sorted order was proposed in [1].

A pseudocode of the algorithm is shown in Figure 7. At each step it gets the next pair $\langle e_i, e_j \rangle$ from PQ, which can be either a pair of intermediate node entries or a pair of objects. In the first case, it applies the main-memory distance join algorithm for the R–tree nodes pointed by e_i and e_j and enqueues the qualifying entry pairs (lines 7–8). In the second case, it continuously removes pairs from PQ as long as they contain the same entry e_i. These correspond to the results of the distance join algorithm. Since they are clustered by $r.id$, we can count them immediately in order to validate the cardinality constraint t (lines 10–14) and output potential pairs that qualify the iceberg join. The pseudocode assumes that the two R–trees have the same height, but it can be easily extended for the general case. Details are omitted for the sake of readability.

```
Algorithm RIDJ(Objectset R, Objectset S, real e, int t)
1.    e_R := ⟨MBR of R, ptr to root node of R's R–tree ⟩;
2.    e_S := ⟨MBR of S, ptr to root node of S's R–tree ⟩;
3.    Initialize PQ and enqueue ⟨e_R, e_S⟩;
4.    while PQ is not empty
5.       get next ⟨e_i, e_j⟩ from PQ;
6.       if e_i and e_j are non-leaf node entries
7.          distance-join nodes n_i, n_j pointed by e_i and e_j; /* same as in RJ */
8.          add qualifying entry pairs ⟨e_R, e_S⟩, e_R ∈ n_R, e_S ∈ n_S to PQ;
9.       else /* e_i and e_j are leaf node entries */
10.         counter := 1;
11.         while e_i.id = e_R.id of the first pair ⟨e_R, e_S⟩ in PQ
12.            get next ⟨e_R, e_S⟩ from PQ and put it in a memory-buffer B;
13.            counter := counter + 1;
14.         if counter > t output ⟨e_i, e_j⟩ and the results in B; /* found results */
```

Fig. 7. R–tree based Iceberg Distance Join

Pruning R–tree nodes early. The only benefit of the RIDJ algorithm described above compared to RJ, is that it avoids sorting the distance join results for validating the cardinality constraint t. However, it does not consider t during the R–tree join process. In this section, we propose an improved version that prunes R–tree nodes early, considering t. The motivation is that, due to PQ, the node pairs to be joined are produced clustered on R. In other words, if $\langle e_i, e_j \rangle$ is the pair currently first in PQ, we know that if there are any other pairs containing e_i, these will be the next ones in the queue.

We can exploit this property to prune node pairs as follows. Whenever we retrieve a pair $\langle e_i, e_j \rangle$ from PQ, we continue fetching more pairs $\langle e_R, e_S \rangle$ as long as $e_i.id = e_R.id$ (just like we do for the leaf level). Now for the current $e_i \in R$ we have a list $L(e_i) = \{e_{S1}, e_{S2}, \ldots, e_{Sk}\}$ that join with it and we know that these are the only ones (at the same level in the R–tree indexing S) that join with e_i. If the total number of points indexed by entries in $L(e_i)$ is smaller than t, no object in R indexed by the subtree rooted at e_i can participate in the result. Thus we do not have to join the node pairs indexed by e_i and the entries in $L(e_i)$. Using this observation, we can prune early many R–tree node pairs if t is large enough. Since the exact number of objects indexed by an intermediate entry is not known, we use an upper bound based on the fanout of the tree nodes.[1] Alternatively we could employ the *aggregate R–tree* [22], which stores for each intermediate entry the number of objects in its sub-tree.

3.3 Only one dataset is indexed

In this section, we discuss an adaptation of SISJ that borrows ideas from PIDJ (discussed in Section 3.1). Let us first assume that only dataset R is indexed. While hashing S, we construct the fine grid $\mathcal{F}$ and create a bitmap for each slot, similar to the one depicted in Figure 5c. If all bits in this bitmap are 0, we can prune the corresponding slot, which means (a) we avoid loading the part of the tree indexed by this slot, (b) we avoid loading the corresponding bucket from S. Naturally, we expect this technique to work best for small values of ϵ, large values of t, and uniform data from S in the pruned buckets.

The case where S is indexed is harder to handle. The reason is that objects $r \in R$ are now replicated to multiple buckets. Therefore it is possible that some r within distance ϵ of objects $s \in S$ may belong to different slots. In that case, it will be impossible to prune a slot, unless it does not overlap with any other slots. Therefore, we can only process the iceberg join using the conventional two-step processing, i.e., (i) apply SISJ to evaluate the distance join, (ii) sort the pairs $\langle r, s \rangle$ to count the occurrences of each r and report the iceberg join result. Alternatively, we can ignore the R–tree for S and employ PIDJ. Since PIDJ does not require sorting, it can be faster than the two-step, conventional technique.

3.4 Adaptation to special types of iceberg distance joins

Our algorithms can be easily adapted for other forms of iceberg join queries. One is the iceberg semi-join, discussed in the introduction. In this case, we are

[1] When e_i points to a leaf node, we perform an additional optimization. The entries in $L(e_i)$ are clustered to sets, such that no two entries in different sets are within distance ϵ from each other. Then, we initially use an upper bound for the total number of objects in each set using the fanout of the nodes. Finally, we visit the nodes pointed by the entries one-by-one, and replace the contribution of each of them to the upper bound by the actual number of objects in them. If at some point the upper bound becomes smaller than t, we can prune this set of entries. We do the same for the remaining sets, until e_i is pruned, or we are forced to perform the distance join.

interested only in the objects $r \in R$, which qualify the iceberg distance join constraints. A straightforward way to process the semi-join is to evaluate the iceberg distance join, and output only the distinct r which appear in the resulting pairs $\langle r, s \rangle$. However, we can do better by exploiting statistical information. Consider for instance the grid $\mathcal{F}$, as depicted in Figure 5a. This grid has granularity ϵ and captures the number of $s \in S$ that fall into each cell, as explained in Section 3.1. Observe that while reading (and hashing) R, if point $r \in R$ falls into a microcell c_x with $c_x.count \geq t$, we can *immediately* output r, without hashing or joining it, because we know for sure that it will participate in the semi-join result (since there are at least t objects from S in the same microcell, i.e., within distance ϵ from it[2]).

In another iceberg distance join query, the cardinality constraint t may not provide a lower threshold for the occurrences of $r \in R$ in the distance join pairs, but an upper threshold. For instance, in this case we might be interested in hotels which are close to *at most* t restaurants. Again, for this query we can use the fine grid $\mathcal{F}$ to detect early pruned regions for R; if $c_x.count \geq t$ for a microcell c_x, we can prune the microcell, since we know that every r that falls there joins with at least t $s \in S$. Therefore, the bit-mapping technique now uses the counters in the microcells instead of adding to them the ones of the influence microcells around them. Finally, the proposed techniques can be easily adapted for the iceberg join, when the number of appearances should be in the range between t_{low} and t_{up} times. For each c_x, its counter provides a *lower* bound of the join results and the counters in its influence region an *upper bound*. These bounds can be used in combination with t_{low} and t_{up} to prune disqualifying cells.

4 Experimental evaluation

We evaluated the performance of the proposed techniques for a wide range of settings, using synthetic and real data of various cardinalities and skew. The real datasets, which were originally line segments, were converted to points, by taking the center of the segments. Files T1 and T2 [7] contain 131,461 roads and 128,971 rivers, respectively, from an area in California. AS and AL [24] contain 30,674 roads and 36,334 railroads, respectively, from Germany. The buffer size was set to 20% of the total size both joined datasets occupy on disk. As a measure of performance, we counted random I/Os. The sequential I/Os were normalized to random ones according to current disk benchmarks (e.g., a random page access costs as much as 10 sequential ones, if the page size is 8Kb). The experiments were run using a 700MHz Pentium III processor.

4.1 Evaluation of PIDJ

In this section we evaluate the performance of PIDJ under various experimental settings. We compare the three progressively more optimized versions of the algorithm. The baseline implementation (referred to as PIDJ_1) avoids duplication

[2] Actually for defining the microcells here we use a value smaller than ϵ, which is the side of the *minimum enclosed square* in a circle with radius $\epsilon/2$. We skip the details for the sake of readability.

and sorting. In addition to these, PIDJ$_2$ filters out buckets using their cardinalities without applying the fine grid $\mathcal{F}$ refinement. Finally PIDJ$_3$ includes all the optimizations described in Section 3.1. For these experiments the page size was set to 8Kb.

In the first experiment, we compare the I/O performance of the three PIDJ versions for several pairs of joined datasets, by setting the distance threshold ϵ to 2% of the data-space projection, and varying the value of the cardinality threshold t. Figures 8 and 9 plot the I/O cost of the algorithms as a function of t for the joins AS $\bowtie$ AL and AL $\bowtie$ AS, respectively; in the first case the relation R where t applies is AS and in the second it is AL (notice that the join is asymmetric, therefore it is possible to have different results and performance in these two cases). Observe that the improvement of PIDJ$_2$ over PIDJ$_1$ is marginal for the tested values of t. This is because the number of buckets in this case is small (only 25 in this example) and the regions that they cover are quite large. PIDJ$_2$ cannot prune buckets, since all of them contain a large number of points from R. On the other hand, the fine grid $\mathcal{F}$ employed by PIDJ$_3$ manages to prune a large percentage of R; as t increases more cells are pruned by the bit-mapping method and less data from R are hashed.

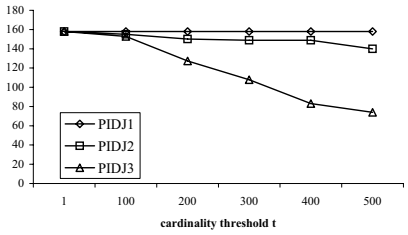


Fig. 8. I/O cost, AS $\bowtie$ AL, $\epsilon = 2\%$

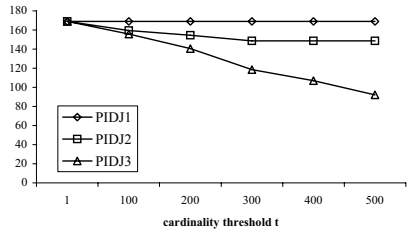


Fig. 9. I/O cost, AL $\bowtie$ AS, $\epsilon = 2\%$

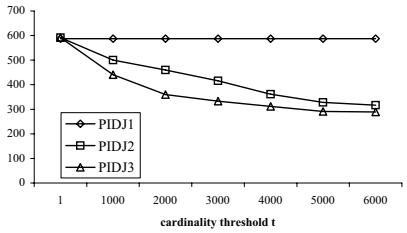


Fig. 10. I/O cost, T1 $\bowtie$ T2, $\epsilon = 2\%$

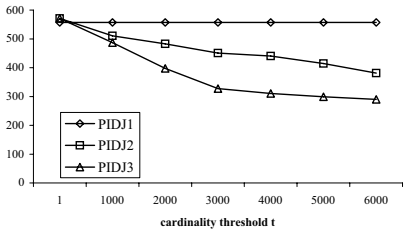


Fig. 11. I/O cost, T2 $\bowtie$ T1, $\epsilon = 2\%$

For the Tiger files T1 and T2, on the other hand, PIDJ$_2$ reduces significantly the cost of the baseline implementation (especially for the T1 $\bowtie$ T2 pair). Figures 10 and 11 plot the I/O cost of the three versions of PIDJ. The improvement of PIDJ$_3$ is significant also in this case. The large improvement of PIDJ$_2$ is due to the large values of t tested, and due to the fact that now grid $\mathcal{G}$ is finer than for

the AS $\bowtie$ AL joins.[3] The sparse areas of the map are usually pruned using $\mathcal{G}$ and the rest are very dense this is why PIDJ$_3$ adds little improvement.

In the next set of experiments we compare the three versions of PIDJ for joins on synthetic data. For this purpose we generated four synthetic datasets G1, G2, U1, and U2, as follows. All datasets contain 200K points. G1 and G2 were created according to a Gaussian distribution with 10 clusters. The centers of the clusters for both files were randomly generated. The sigma value of the data distribution around the clusters in G1 was set to a random value between 1/5 and 1/10 of the data space. Thus, all clusters are similar, they spread around the whole data-space, and the resulting dataset has little skew. On the other hand, the sigma value of the clusters in G2 has high variance; half of the clusters are very skewed, with sigma between 1/10 and 1/20 of the data space, and the other half have little skew (sigma is between 1/3 and 1/5). In this way we wanted to simulate the case where both datasets are non-uniform, but one of them has higher skew in some regions. U1 and U2 contain uniformly distributed points.

Figures 12 and 13 plot the I/O cost for G1 $\bowtie$ G2 and G2 $\bowtie$ G1, respectively. In both joins, PIDJ$_3$ achieves significant improvement over the baseline version of the algorithm. Notice that for G2 $\bowtie$ G1 the improvement is larger. This is due to the fact that G1 is more uniform than G2, so if it is taken as S, the probability that PIDJ$_3$ prunes a microcell increases. We also experimented with other combinations of skewed data with similar results. In general, the improvement of PIDJ$_3$ over PIDJ$_2$ increases with the uniformity of the data, since more regions corresponding to the microcells $c_i \in \mathcal{F}$ are pruned. The experiment plotted in Figure 14 validates this argument. This time we joined two uniform datasets U1 and U2. Note that uniformity is normally the worst setting for an iceberg join algorithm, since buckets cannot be pruned using their global statistics. As expected, here PIDJ$_2$ has no improvement over PIDJ$_1$, since the number of data from U2 in each bucket is the same and larger than t (unless t becomes very large). On the other hand, PIDJ$_3$ for $t > 200$ starts pruning many cells in $\mathcal{F}$, sharply until it converges to the cost of just reading and hashing S.

We have also tracked the computational cost of the three versions and compared it with the I/O difference. Figure 15 plots these costs for G2 $\bowtie$ G1. Observe that the costs of PIDJ$_1$ and PIDJ$_2$ are almost identical since they essentially perform the same computations. PIDJ$_2$ can only avoid hashing some objects from R, but this affects mainly the I/O cost. On the other hand, PIDJ$_3$ prunes more buckets and many $r \in R$ using $\mathcal{F}$. In effect, it computes much fewer joined pairs in its hash-join part, which is reflected in the CPU cost. For $t = 1$ and $t = 100$ the overhead of the fine grid $\mathcal{F}$ does not pay-off, this is why PIDJ$_3$ is slightly slower than PIDJ$_1$ and PIDJ$_2$. The computational costs of the methods show similar trends for the other joined pairs and are therefore omitted.

In the next experiment, we study the effects of ϵ in the three versions of PIDJ. Figure 16 plots the I/O cost of AL $\bowtie$ AS as a function of ϵ when $t = 400$. The improvement of PIDJ$_3$ over the simpler versions of the algorithm decreases with

[3] The datasets are larger, thus more memory (20% of the total size) is allocated for the join and more buckets are defined.

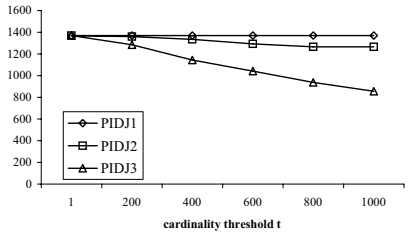


Fig. 12. I/O cost, G1 ⋈ G2, $\epsilon = 2\%$

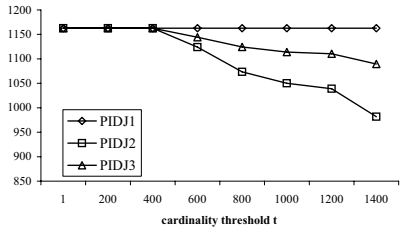


Fig. 13. I/O cost, G2 ⋈ G1, $\epsilon = 2\%$

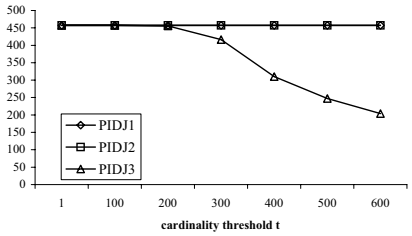


Fig. 14. I/O cost, U1 ⋈ U2, $\epsilon = 2\%$

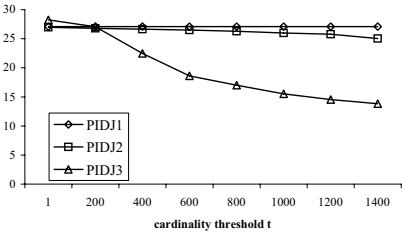


Fig. 15. CPU time (sec), G2 ⋈ G1, $\epsilon = 2\%$

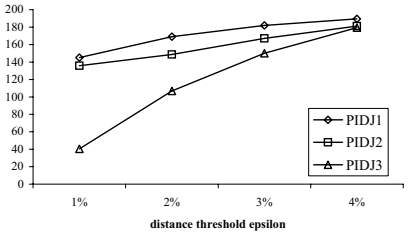


Fig. 16. I/O cost, AL ⋈ AS, $t = 400$

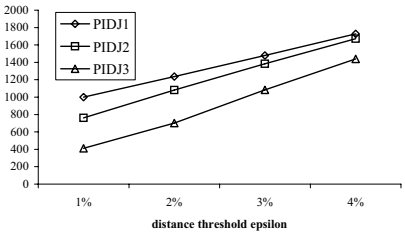


Fig. 17. I/O cost, G2 ⋈ G1, $t = 1400$

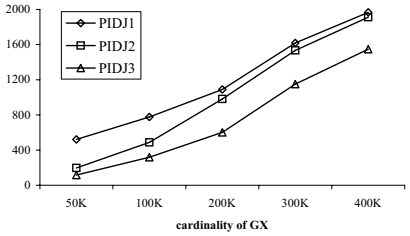


Fig. 18. I/O cost, G2 ⋈ GX, $\epsilon = 2\%$, $t = 1400$

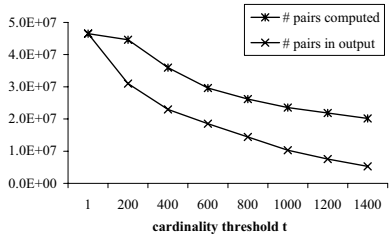


Fig. 19. I/O cost, G2 ⋈ G1, $\epsilon = 2\%$

ϵ because of two reasons. First, the number of results increases, therefore much fewer cells are pruned. Second, the effectiveness of $\mathcal{F}$ decreases significantly with ϵ, since the grid becomes much coarser (recall that $\mathcal{F}$ is defined based on ϵ). In practice, we expect iceberg joins with small ϵ for more selective and more useful result. Figure 17 shows a similar trend for query G2 $\bowtie$ G1, after fixing $t = 1400$.

The next experiment tests the scalability of the algorithm to the problem size. We joined G2 with many synthetic files GX, with the same cluster settings as G1, but varying size from 50K objects to 400K objects. In these experiments $\epsilon = 2\%$ and $t = 1400$. Figure 18 shows the performance of PIDJ as a function of the size of GX. Observe that the improvement of PIDJ_3 decreases with the problem size, but this can be explained by the increase of the query output. On the other hand, the performance of PIDJ_2 decreases more rapidly. This is due to the fact that as the cardinality increases, fewer buckets can be pruned by the global constraint.

Finally, we test how PIDJ_3 adapts to the output size. Figure 19 shows the number of pairs within distance ϵ which have been computed by the algorithm (i.e., not pruned by $\mathcal{F}$) and the number of pairs in the iceberg join output. The upper line reflects the cost of the algorithm. The joined pair is G2 $\bowtie$ G1, $\epsilon = 2\%$, and t varies. Indeed, the cost of the algorithm drops as its output decreases, however, not at the same rate, since pruning a large percentage of false pairs becomes harder as t increases. In summary, PIDJ, after employed with all heuristics, is a robust, output sensitive algorithm for computing iceberg distance joins. Its efficiency is mainly due to the introduction of the fine grid $\mathcal{F}$ and the corresponding bit-mapping technique.

4.2 Evaluation of RIDJ

In this section we evaluate the performance of the RIDJ algorithm for iceberg distance join queries. For this, we built two R–trees for the real datasets AS and AL, with node size 1Kb. We used a small node size, since this facilitates pruning in RIDJ; the fanout of the nodes is not very large and for values of t which return results (i.e., in the range 100–1000) it becomes possible for the algorithm to prune R–tree node pairs early.

The following experiment validates this assertion. We performed the join AL $\bowtie$ AS for different values of t and ϵ and compared the performance of three algorithms; RJ, RIDJ and RIDJ employed with the pruning heuristic. Figures 20 shows the performance of the algorithms for $\epsilon = 2\%$ and various values of t. Observe that the cost difference between RJ and the basic version of RIDJ is the same for all values of t. It translates to the overhead of RJ to sort a large number of pairs that qualify the distance join (around 3.6 million for this join) in order to validate the cardinality threshold t. Most page accesses are sequential, this is why the normalized cost difference is not extreme. On the other hand, the version of RIDJ with pruning is output sensitive. Many node pairs are pruned as t increases (and the result decreases).

In the experiment of Figure 21, the cardinality threshold is fixed to $t = 400$ and ϵ varies. Observe that the cost of RJ increases with ϵ, since more pairs qualify the distance join as ϵ is relaxed. On the other hand, the cost of the ba-

sic RIDJ algorithm remains constant, since the algorithm exploits the buffer to avoid incurring more I/O accesses than the total number of nodes in both trees. The efficiency of the pruning heuristic diminishes with the increase of ϵ, because the lists $L(e_i)$ increase significantly due to the increase of the extended MBRs by $\epsilon/2$ while processing the distance join. Nevertheless, at the same time, the number of the iceberg query results increases significantly with ϵ. In summary, RIDJ, when employed with the pruning heuristic, is an output sensitive algorithm that manages to prune the search space, by exploiting gracefully the query constraints.

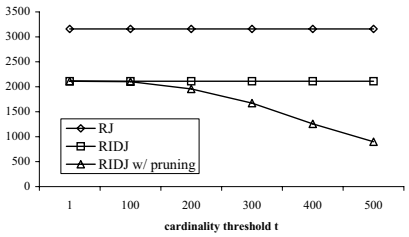


Fig. 20. I/O cost, AL $\bowtie$ AS, $\epsilon = 2\%$

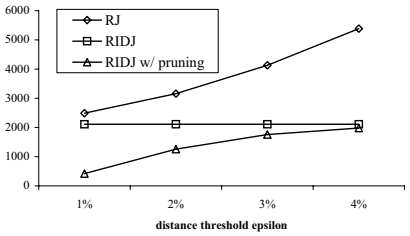


Fig. 21. I/O cost, AL $\bowtie$ AS, $t = 400$

5 Conclusions

In this paper we have shown how spatial join algorithms can be adapted for the interesting class of iceberg distance join queries. The proposed methods exploit data density and distribution statistics, obtained while processing the query, to shrink the search space according to the cardinality constraint t. In an attempt to cover all possible indexing presumptions for the joined data we have extended a hash-based algorithm (PBSM) and a index-based method (RJ). We have also described adaptations for single-index algorithms. Finally, we have discussed how our methods can be used to process special types of iceberg distance joins.

We conducted a comprehensive experimental evaluation, which demonstrates that the proposed techniques are indeed output sensitive; their cost steadily decreases with t. The current study considers iceberg distance joins between point-sets. In the future, we will study their application to joins of datasets containing objects with spatial extent. It would also be interesting to see how they can be adapted for iceberg intersection joins, where, instead of the distance threshold ϵ, intersection is considered. Finally, an interesting issue for future work is how *ranking* predicates can be embedded in distance join algorithms (e.g., output the top-k hotels with the largest number of nearby restaurants).

References

1. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J. S. Vitter. A unified approach for indexed and non-indexed spatial joins. In *Proc. of EDBT Conference*, 2000.
2. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proc. of VLDB Conference*, 1998.

3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of ACM SIGMOD Int'l Conference*, 1990.
4. K. S. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *Proc. of ACM SIGMOD Int'l Conference*, 1999.
5. 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 *Proc. of ACM SIGMOD Int'l Conference*, 2001.
6. T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R–trees. In *Proc. of ACM SIGMOD Int'l Conference*, 1993.
7. Bureau of the Census, TIGER/Line Precensus files: 1990 Technical Documentation, 1989.
8. A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest pair queries in spatial databases. In *Proc. of ACM SIGMOD Int'l Conference*, 2000.
9. J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2000.
10. M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. of VLDB Conference*, 1998.
11. A. Guttman. R–trees: a dynamical index structure for spatial searching. In *Proc. of ACM SIGMOD Int'l Conference*, 1984.
12. J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proc. of ACM SIGMOD Int'l Conference*, 2001.
13. G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *Proc. of ACM SIGMOD Int'l Conference*, 1998.
14. G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.
15. Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *Proc. of VLDB Conference*, 1997.
16. N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proc. of ACM SIGMOD Int'l Conference*, 1997.
17. N. Koudas and K. C. Sevcik. High dimensional similarity joins: Algorithms and performance evaluation. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 1998.
18. M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proc. of ACM SIGMOD Int'l Conference*, 1994.
19. M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proc. of ACM SIGMOD Int'l Conference*, 1996.
20. G. Luo, J. F. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 2002.
21. N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *Proc. of ACM SIGMOD Int'l Conference*, 1999.
22. D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *Proc. of SSTD*, 2001.
23. J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proc. of ACM SIGMOD Int'l Conference*, 1996.
24. Penn State University Libraries, Digital Chart of the World, http://www.maproom.psu.edu/dcw/, 1997.
25. K. Shim, R. Srikant, and R. Agrawal. High-dimensional similarity joins. In *Proc. of Int'l Conf. on Data Engineering (ICDE)*, 1997.
26. H. Shin, B. Moon, and S. Lee. Adaptive multi-stage distance join processing. In *Proc. of ACM SIGMOD Int'l Conference*, 2000.