Adaptive and Incremental Processing for Distance Join Queries *

Hyoseop Shin[†] *Bongki Moon*[‡] *Sukho Lee*[†]

[†]*School of Computer Engineering*
Seoul National University
Seoul, Korea
{hsshin@db,shlee@cse}.snu.ac.kr

[‡]*Department of Computer Science*
University of Arizona
Tucson, AZ 85721
bkmoon@cs.arizona.edu

Technical Report 02-03

Abstract

A spatial distance join is a relatively new type of operation introduced for spatial and multimedia database applications. Additional requirements for ranking and stopping cardinality are often combined with the spatial distance join in on-line query processing or internet search environments. These requirements pose new challenges as well as opportunities for more efficient processing of spatial distance join queries. In this paper, we first present an efficient k-distance join algorithm that uses spatial indexes such as R-trees. Bi-directional node expansion and plane-sweeping techniques are used for fast pruning of distant pairs, and the plane-sweeping is further optimized by novel strategies for selecting a sweeping axis and direction. Furthermore, we propose adaptive multi-stage algorithms for k-distance join and incremental distance join operations. Our performance study shows that the proposed adaptive multi-stage algorithms outperform previous work by up to an order of magnitude for both k-distance join and incremental distance join queries, under various operational conditions.

September 2002

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

*This work was sponsored in part by National Science Foundation CAREER Award (IIS-9876037), NSF Grant No. IIS-0100436, and Research Infrastructure program EIA-0080123. It was also supported by Korean Science and Engineering Foundation under Exchange Student Program. The authors assume all responsibility for the contents of the paper.

1 Introduction

A spatial distance join operation was recently introduced to spatial databases to associate one or more sets of spatial data by distances between them [16]. A distance is usually defined in terms of spatial attributes, but it can be defined in many different ways according to various application specific requirements. In multimedia and image database applications, for example, other metrics such as a *similarity distance function* can be used to measure a distance between two objects in a feature space.

In on-line decision support and internet search environments, it is quite common to pose a query that finds the best k matches or reports the results incrementally in the decreasing order of well-matchedness. This type of operations allow users to interact with database systems more effectively and focus on the "best" answers. Since users can say "It is enough already" at any time after obtaining the best answers [9], the waste of system resources can be reduced and thereby delivering the results to users more quickly.

This ranking requirement is often combined with a spatial distance join query, and the ranking requirement provides a new opportunity of optimization for spatial distance join processing [10, 12]. For example, consider a query that retrieves the top k pairs (*i.e.*, the nearest pairs) of hotels and restaurants:

```
SELECT h.name, r.name
FROM Hotel h, Restaurant r
ORDER BY distance(h.location, r.location)
STOP AFTER k;
```

For a relatively small stopping cardinality k, the processing time can be reduced significantly by sorting only a fraction of intermediate results enough to produce the k nearest pairs, instead of sorting an entire set of intermediate results (*i.e.*, a Cartesian product of hotels and restaurants).

A spatial distance join query with a stopping cardinality can be formulated as follows:

$$\sigma_{dist(r,s)<\mathcal{D}_{max}}(R \bowtie S)$$

where $dist(r, s)$ is a distance between two spatial objects $r \in R$ and $s \in S$, and $\mathcal{D}_{max}$ is a cutoff distance that is determined by a stopping cardinality k and the spatial attribute values of two data sets R and S. It may then be argued that a spatial distance join query can be processed by a spatial join operation [1, 7, 8, 18, 19, 23] followed by a sort operation. Specifically, if a $\mathcal{D}_{max}$ value can be predicted precisely for a given stopping cardinality k, we can use a spatial join algorithm with a `within` predicate instead of an `intersect` predicate to find the k nearest pairs of objects. Then, a sort operation will be performed only on the k pairs of objects.

In practice, however, it is almost impossible to estimate an accurate $\mathcal{D}_{max}$ value for a given stopping cardinality k, and, to the best of our knowledge, no method for estimating such a cutoff value has been reported in the literature. If the $\mathcal{D}_{max}$ value is overestimated, then the results from a spatial join operation may contain too many candidate pairs, which may cause a long delay in a subsequent stage to sort all the candidate pairs. On the other hand, if the $\mathcal{D}_{max}$ value is underestimated, a spatial join operation may not return a sufficient number of object pairs. Then, the spatial join operation should be repeated with a new estimate of $\mathcal{D}_{max}$, until k or more pairs are returned. This may cause a significant amount of waste in processing time and resources.

There is another reason that makes it impractical to apply a spatial join algorithm to spatial distance join queries. A spatial join query is typically processed in two steps, *filter* and *refinement*, as proposed in [21]. In a filter step, MBR approximations are used to find pairs of potentially intersected spatial objects. Then, in a refinement step, it is guaranteed that all the qualified (*i.e.*, actually intersected) pairs can be produced from the results returned from the filter step.

In contrast, it is completely unreasonable to process a spatial distance join query in two separate filter and refinement steps, because of the fact that a filtering process is based on MBR approximations. A set of object pairs sorted by distances measured by MBR approximations does not reflect a true order based on actual representations. This is because, for any two pairs of spatial objects $\langle r_1, s_1\rangle$ and $\langle r_2, s_2\rangle$, the fact that $dist(MBR(r_1), MBR(s_1)) < dist(MBR(r_2), MBR(s_2))$ does not necessarily imply that $dist(r_1, s_1) < dist(r_2, s_2)$. Consequently, any processing done in the filter step will be of no use for finding the k nearest object pairs.

In this paper, we propose new strategies for efficiently processing spatial distance join queries combined with ranking requirements. The main contributions of the proposed solutions are:

- New efficient methods are proposed to process distance join queries using spatial index structures such as R-trees. *Bi-directional node expansion* and *optimized plane-sweep* techniques are used for fast pruning of distant pairs, and the plane-sweep is further optimized by novel strategies for selecting a sweeping axis and direction, and by using maximum distance for breaking tied pairs.

- Adaptive multi-stage algorithms are proposed to process distance join queries in a way that the k nearest pairs are returned *incrementally*. When a stopping cardinality is not known a priori (*e.g.*, in on-line query processing environments or a complex query containing a distance join as a sub-query whose results need to be pipelined to the next stage of the complex query), the adaptive multi-stage algorithms can produce pairs of objects in a stepwise manner.

- We provide a systematic approach for *estimating the maximum distance* for a distance join query with a stopping cardinality. This estimated distance allows the adaptive multi-stage algorithms to avoid a *slow start* problem, which may cause a substantial delay in the query processing. This approach for estimating the maximum distance also allows the size of memory to be parameterized into a queue management scheme, so that data movement between memory and disk can be minimized.

The proposed algorithms achieve up to an order of magnitude performance improvement over previous work for both k-distance join and incremental distance join queries, under various operational conditions.

The rest of this paper is organized as follows. Section 2 surveys the background and related work on processing spatial distance join queries. Major limitations of previous work are also discussed in the section. In Section 3, we present a new improved algorithm based on bi-directional expansion and optimized plane-sweep techniques for k-distance join queries. In Section 4, adaptive multi-stage algorithms are presented for k-distance join and incremental distance join queries. A queue management scheme parameterized by memory capacity is also presented. Section 5 presents the results of experimental evaluation of the proposed solutions. Finally, Section 6 summarizes the contributions of this paper.

2 Background and Previous Work

A spatial index structure R-tree and its variants [3, 6, 14] have been widely used to efficiently access multidimensional data – either spatial or point. Like other tree-structured index methods, an R-tree index partitions a multidimensional space by grouping objects in a hierarchical manner. A subspace occupied by a tree node is always contained in the subspace of its parent node. This hierarchy of spatial containment between R-tree nodes is readily used by spatial distance join algorithms as well as spatial join algorithms.

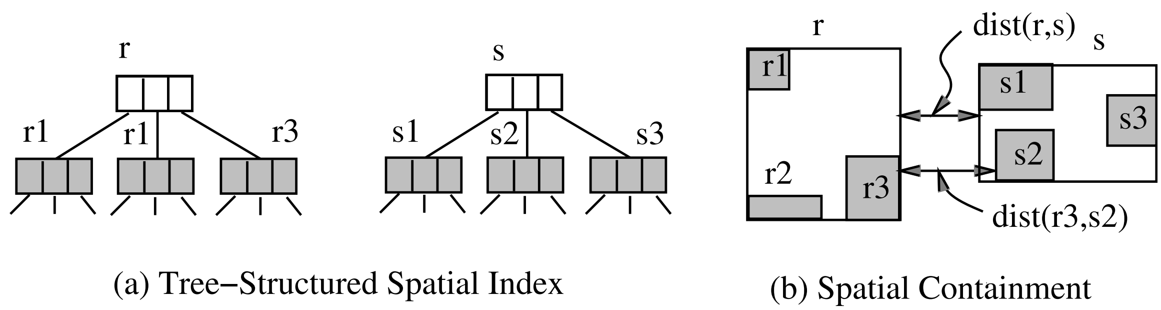


Figure 1: Hierarchy of Spatial Containment of R-Tree Nodes

Suppose r and s are non-leaf nodes of two R-tree indexes R and S, respectively, as in Figure 1. Then, the minimum distance between r and s is always less than or equal to the minimum distance between one of the entries of r and one of the entries of s. Likewise, the maximum distance between r and s is always greater than or equal to the maximum distance between one of the entries of r and one of the entries of s. This observation leads to the following lemma.

Lemma 1 *For two R-tree indexes R and S, if neither $r \in R$ nor $s \in S$ is a root node, then*

$$\begin{aligned} dist(r,s) &\geq dist(parent(r), parent(s)), \\ dist(r,s) &\geq dist(r, parent(s)), \\ dist(r,s) &\geq dist(parent(r), s). \end{aligned} \tag{1}$$

where $dist(r,s)$ is the minimum distance between the MBR representations of r and s.

Proof. From the observation above. □

Lemma 1 allows us to limit the search space, while R-tree indexes are traversed in a top-down manner to process a spatial distance join query. For example, if a pair of non-leaf nodes $\langle r, s\rangle$ turn out to be too far from each other (or their distance is over a certain threshold), then it is not necessary to traverse further down the tree indexes below the nodes r and s. Thus, this lemma provides the key leverage to processing distance join queries efficiently using R-tree indexes.

2.1 Incremental Distance Join and k-Distance Joins

During top-down traversals of R-tree indexes, it is desirable to store examined node pairs in a priority queue, where the node pairs are kept in an increasing order of distances. We call it a *main queue* as opposed to a *distance queue* we will describe later. The main queue initially contains a pair of the root nodes of two R-tree indexes. Each time a pair of non-object nodes are retrieved from the main queue, the entries of one node are paired up with the entries of the other to generate a new set of node pairs, which are then inserted into the main queue. This process that we call *node expansion* is repeated until the main queue becomes empty or until stopped by an interactive user. If an element retrieved from the main queue is a pair of two objects, the pair is returned immediately to the user as a query result. This is how a spatial distance join query is processed *incrementally*. Figure 2 depicts a typical framework of processing an incremental distance join (**IDJ**) query using R-tree indexes.

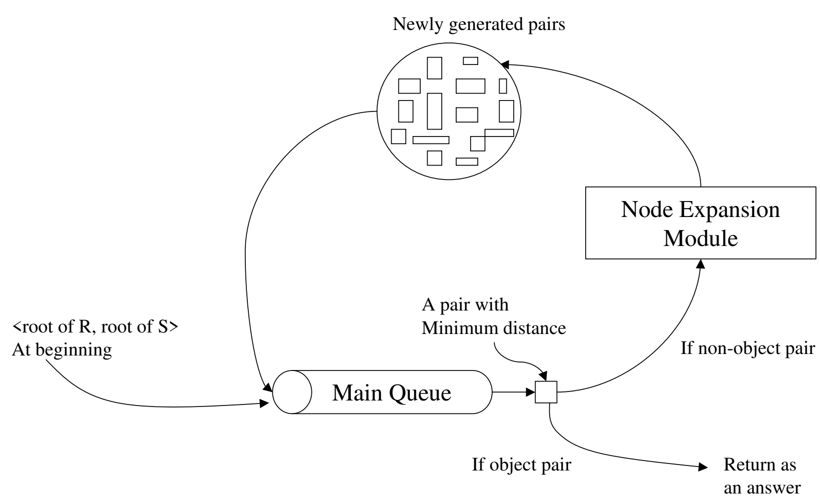


Figure 2: Framework of Incremental Distance Join (**IDJ**) Processing

A distance join query is often given with a stopping cardinality k as in the "stop after" clause of the sample query in Section 1. Since it is known a priori how many object pairs need to be produced for a distance join query, this knowledge can be exploited to improve the performance of the query processing. Suppose a maximum of k nearest pairs of objects are to be retrieved by a query. One plausible approach is to maintain k candidate pairs of objects during the entire course of query processing. As they are the k nearest object pairs known at each stage of query processing, any pair of nodes (and any pair of their entries) whose distance is greater than *all* of the k candidate pairs cannot be qualified as a query result. Thus, we can use another priority queue to store the k minimum distances, and use the queue to avoid having to insert unqualified pairs into the main queue during the node expansions. We call the priority queue a *distance queue*. Figure 3 depicts a typical framework of processing a k-distance join (**KDJ**) query using R-tree indexes and both main and distance queues.

Both main and distance queues can be implemented by heap structures. A main queue is normally implemented as a *min-heap*, because the query results are produced in an increasing order of distances. In contrast, a distance queue should be implemented as a *max-heap* that can store at most k distance values. The cutoff distance is determined by the maximum value among the k distances stored in the distance queue. (When the distance queue contains less than k distances, the cutoff distance is set to an infinity.) Pruning node pairs by the distance queue was shown to be very efficient from our experiments, especially when k was rather small. In the rest of the paper, we use $q\mathcal{D}_{max}$ to denote the cutoff distance from the distance queue.

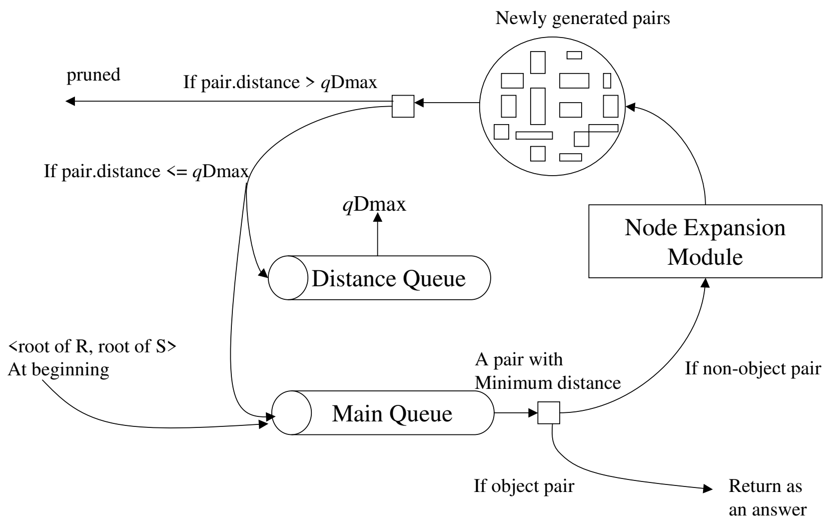


Figure 3: Framework of k-Distance Join (**KDJ**) Processing

2.2 Previous Work

In [16], the authors present both uni-directional and bi-direction node expansion, but conclude based on their experiments that the former provides better performance due to fewer node pairs being produced by their algorithm. When a pair of nodes $\langle r, s\rangle$ are retrieved from a main queue, either node r is paired up with the entries of s, or node s is paired up with the entries of r. None of the pairs are generated from an entry of r and an entry of s. The advantage of the uni-directional expansion is that the number of pairs generated at each expansion step is limited to the fanout of an R-tree index being traversed, and an explosion of the main queue can be avoided. As is acknowledged by the authors of the algorithms, however, the main disadvantage of this approach is that the uni-directional expansion may lead to each node being accessed from disk more than necessary. And also, the uni-directional expansion requires pairing up node r exhaustively with all the entries of node s or vice versa.

For a spatial distance join query with a relatively small stopping cardinality k, the use of a distance queue is an effective means to prevent distant pairs from entering a main queue. For a large k value, however, the distance queue may not work well as an effective pruning tool, because the cutoff value stored in the distance queue may remain too high for a long duration. This may in turn lead to a long delay particularly in the early stage of query processing. For these reasons, the previous algorithms suffer from poor performance for a k-distance join query with a large k and an incremental distance join query, for which k is unknown in advance.

Moreover, there is an important issue that was not fully addressed in [16]. A hybrid memory/disk technique was proposed as a queue management scheme, which partitions a queue based on the distance range. This technique keeps a partition in the shortest distance range in memory, while the rest of partitions are stored on disk. However, no mechanism was provided to determine a boundary distance value between the partition in memory and the rest, which may have a crucial impact on the performance of queue management.

Recently, a few recursive and iterative algorithms have been proposed [11]. These algorithms make use of various distance metrics such as $MinMax$, $MinMin$, $MaxMin$ and $MaxMax$ to find k closest pairs. Without using a main queue, the recursive algorithms access R-tree nodes recursively following priorities given to the entry pairs within a pair of the parent nodes. The iterative algorithm (called $heap$ algorithm) is fairly similar to Hjaltason and Samet's distance join algorithm [16] in that both the algorithms use a distance queue to maintain k candidate pairs during node expansion. One notable difference is that $heap$ algorithm does not store object pairs in the main queue to minimize the size of a main queue. Instead, the heap algorithm uses a distance (or candidate) queue to store the k closest pairs of objects. Although this does not guarantee that the main queue always fits in memory, the performance gain by not storing object pairs in the main queue could be non-trivial, given the potentially large number of object pairs produced by node expansion. Since the heap algorithm maintains only the k candidates, the stopping cardinality k must be known a priori. In other words, the heap algorithm cannot be used for incremental distance join queries.

Several closely related studies for nearest neighbor queries have been reported in the literature. Among those are nearest neighbor search algorithms based on Voronoi cells [2, 5] and branch and bound techniques [26, 27], a nearest neighbor search algorithm for ranking requirement [15], and multi-step k-nearest neighbor search algorithms [17, 28].

Another closely related issue is estimating spatial join selectivity. Some estimation techniques proposed to use supplementary structures such as histograms [24] and wavelets [29]; other estimation techniques were based on uniformity assumption [22] and fractal dimensions for self joins [4]. Recently, Faloutsos *et al.* [13] proposed a power law to predict the selectivity of spatial join and to estimate the distance of the k-th closest pair. This power law will be used to estimate cutoff distances for the adaptive distance join algorithm proposed in this paper.

3 Optimized Plane-Sweep for Fast Pruning

In this section, we propose a new distance join algorithm $\mathcal{B}$-**KDJ** (Bidirectional expanding k-Distance Join) using a *bi-directional* node expansion, in an attempt to avoid redundant accesses to R-tree nodes. As is pointed out in Section 2, distance join algorithms based on an uni-directional expansion require accessing an R-tree node more than those based on bi-directional expansions. Under the bidirectional node expansion, for a pair $\langle r, s\rangle$, each of the entries of r is paired up with each of the entries of s. This is essentially a Cartesian product, which may generate more redundant pairs than the uni-directional expansion does. Nonetheless, we will show $\mathcal{B}$-**KDJ** algorithm can effectively avoid generating redundant pairs by a plane sweeping technique [25] and novel strategies for choosing an axis and a direction for sweeping. The $\mathcal{B}$-**KDJ** algorithm is described in Algorithm 1.

3.1 Bidirectional Node Expansion

Like the distance join algorithms proposed in [16], $\mathcal{B}$-**KDJ** algorithm uses $q\mathcal{D}_{max}$ from the distance queue $\mathcal{Q}_D$ as a cutoff value to examine node pairs. If a pair of nodes $\langle r, s\rangle$ removed from the main queue are a pair of objects, then the object pair is returned as a query result. Otherwise, the pair is expanded by the *PlaneSweep* procedure for further processing.

Assume that a sweeping axis (*i.e.*, x or y dimensional axis) and a sweeping direction (*i.e.*, forward or backward) are determined, as we will describe in Sections 3.2 and 3.3. Then, the entries of r and s are sorted by x or y coordinates of one of the corners of their MBRs in an increasing or decreasing order, depending on the choice of sweeping axis and sweeping direction. Each node encountered during a plane sweep is selected as an $anchor$, and it is paired up with entries in the other group. For example, in Figure 4, an entry r_1 of r is selected as an anchor, and the entries s_1, s_2, s_3 and s_4 of s are examined for pairing, as they are within $q\mathcal{D}_{max}$ distance from r_1 along the sweeping axis (lines 11-14 and line 16).

Since an axis distance between any pair $\langle r, s\rangle$ is always smaller than or equal to their real distance (*i.e.*, $axis\_distance(r, s) \leq real\_distance(r, s)$), real distances are computed only for nodes whose axis distances from the anchor are within the current $q\mathcal{D}_{max}$ value (line 17). Given that a real distance is more expensive to compute than an axis distance, it may yield non-trivial performance gain. Then, each pair whose real distance is within $q\mathcal{D}_{max}$ is inserted into the main queue $\mathcal{Q}_M$ (line 18). If it is a pair of objects, then update the current $q\mathcal{D}_{max}$ value by inserting the real distance of the object pair into the distance queue $\mathcal{Q}_D$ (line 19).

Algorithm 1: $\mathcal{B}$**-KDJ**: K-Distance Join Algorithm with Bi-directional Expansion and Plane Sweep

set $AnswerSet \leftarrow$ an empty set;
set $\mathcal{Q}_M$, $\mathcal{Q}_D \leftarrow$ empty main and distance queues;
insert a pair $\langle R.root, S.root\rangle$ into the main queue $\mathcal{Q}_M$;
while $|AnswerSet| < k$ *and* $\mathcal{Q}_M \neq \emptyset$ **do**
    set c $\leftarrow$ dequeue($\mathcal{Q}_M$);
    if c *is an* $\langle object, object\rangle$ **then** $AnswerSet \leftarrow \{c\} \cup AnswerSet$;
    else $PlaneSweep$(c);
end

procedure PlaneSweep($\langle l, r\rangle$)
set L $\leftarrow$ sort_axis($\{$entries of $l\}$); // Sort the entries of l by axis values.
set R $\leftarrow$ sort_axis($\{$entries of $r\}$); // Sort the entries of r by axis values.
while $L \neq \emptyset$ *and* $R \neq \emptyset$ **do**
    $n \leftarrow$ a node with the min axis value $\in L \cup R$; // n becomes an anchor.
    if $n \in L$ **then**
        $L \leftarrow L - \{n\}$; $SweepPruning(n, R)$;
    else
        $R \leftarrow R - \{n\}$; $SweepPruning(n, L)$;
    end
end

procedure SweepPruning($n, List$)
for *each node* $m \in List$ *in an increasing order of axis value* **do**
    if $axis\_distance(n, m) > q\mathcal{D}_{max}$ **then** return; // No more candidates.
    if $real\_distance(n, m) \leq q\mathcal{D}_{max}$ **then**
        insert $\langle n, m\rangle$ into $\mathcal{Q}_M$;
        if $\langle n, m\rangle$ *is an* $\langle object, object\rangle$ **then** insert $real\_distance(n, m)$ into $\mathcal{Q}_D$; // $q\mathcal{D}_{max}$ modified.
    end
end

There are alternatives as to what pairs are to be inserted into a distance queue: (1) any pairs encountered during node expansions, or (2) pairs of objects only. If a pair of non-object R-tree nodes is inserted into a distance queue, then its distance value should be the *maximum* distance (instead of minimum distance), and the minimum number of object pairs that can possibly be generated from the node pair should be maintained in the distance queue, as pointed out in [16]. The minimum number of object pairs can be estimated based on the minimum node occupancy. Since the maximum distance tends to be larger than those of pairs of objects, most of non-object pairs are inserted into a distance queue only to be removed from the distance queue without reducing $q\mathcal{D}_{max}$ value. Consequently, the potential benefit from inserting non-object pairs is expected to be insignificant. More often than not in our experiments, the query processing slowed down slightly due to the overhead of inserting non-object pairs. Thus, we decide to follow the second option in this paper.

For a relatively small $q\mathcal{D}_{max}$ value and two sets of evenly distributed spatial objects, the number of pairs for which $\mathcal{B}$**-KDJ** algorithm computes real distances and performs queue management operations is expected to be roughly $\mathcal{O}(|r|+|s|)$. This justifies the additional cost of sorting entries for plane-sweeping, because the overall cost of $\mathcal{B}$**-KDJ** algorithm would otherwise be $\mathcal{O}(|r| \times |s|)$ by Cartesian products.

3.2 Sweeping Axis

We can improve $\mathcal{B}$**-KDJ** algorithm one step further by deciding the sweeping axis and direction on an individual pair basis. Intuitively, if entries (or data objects) are spread more widely along one dimension (say, x) than the other dimensions, then the bi-directional node expansion is likely to generate a smaller number of node pairs to compute the real distances for by plane-sweeping along the dimension x. This is because, when the nodes are more widely spread along a sweeping axis, the chance that a pair of nodes are within a $q\mathcal{D}_{max}$ distance along the sweeping axis is lower. For a pair of parent nodes shown in Figure 5, as an example, it would be better to choose y-axis as a sweeping axis, as the entries are more widely spread along the y-dimension. On the other hand, if x-axis is chosen as a sweeping axis,

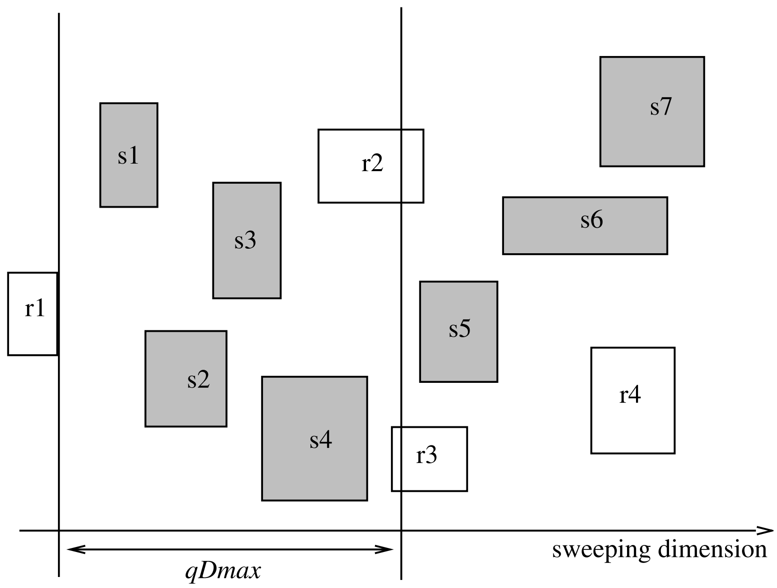


Figure 4: Bidirectional Node Expansion with Plane Sweeping

no pair of the entries will be pruned by x-axis distance comparison with $q\mathcal{D}_{max}$, because the x-axis distance between any pair of the entries is shorter than the $q\mathcal{D}_{max}$ value.

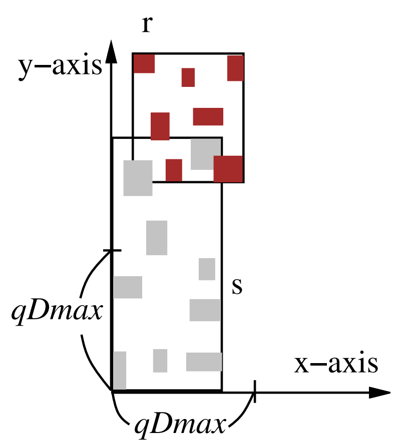


Figure 5: Effect of Right Selection of the Sweeping Axis

Formally, we define a new metric **sweeping index** as follows, and we use the metric to determine which axis a plane-sweep will be performed on. For a given pair $\langle r, s\rangle$ of R-tree nodes and a given $q\mathcal{D}_{max}$ value, we can compute a sweeping index for each dimension. Conceptually, a sweeping index is a normalized estimation of the number of node

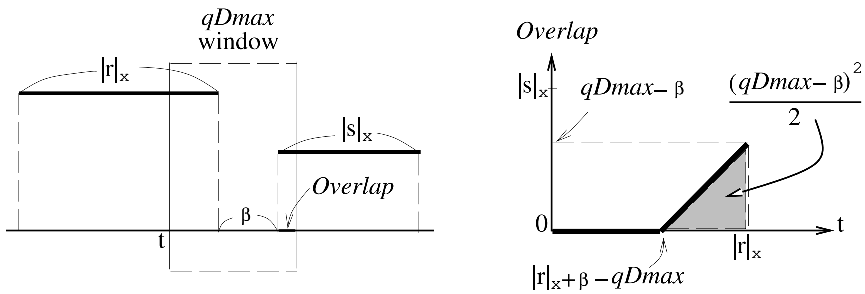


Figure 6: Sweeping Index

pairs we need to compute the real distances for, based on the assumption that data objects are uniformly distributed.[1]

$$\text{Sweeping Index}_x = \int_0^{|r|_x} Overlap(q\mathcal{D}_{max}, r, t)dt + \int_0^{|s|_x} Overlap(q\mathcal{D}_{max}, s, t)dt \quad (2)$$

In the first integral term of the equation above, $|r|_x$ is the side length of node r along the dimension x. The function $Overlap(q\mathcal{D}_{max}, r, t)$ is a portion of the side length of s along the dimension x, overlapped with a window of length $q\mathcal{D}_{max}$ whose left end point is located at a point t within $|r|_x$ (*i.e.*, $0 \le t \le |r|_x$). (See the left diagram in Figure 6.) Thus, $Overlap(q\mathcal{D}_{max}, r, t)/|s|_x$ represents a fraction of s's entries intersected with a window $[t, t + q\mathcal{D}_{max}]$. The value of the function varies as the window moves along the dimension x from [0,$q\mathcal{D}_{max}$] to $[|r|_x, |r|_x + q\mathcal{D}_{max}]$. Therefore, the first integral term represents a relative estimation of the number of s's entries encountered during the plane-sweeps performed for all the entries of r. The second integral term is symmetric with the first integral, and an identical description can be offered by exchanging r and s.

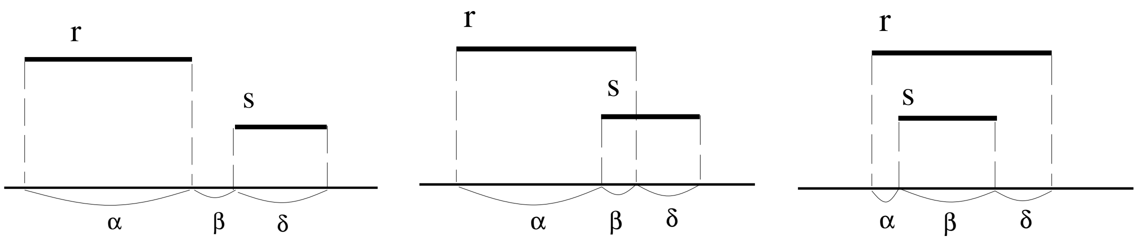
A smaller sweeping index indicates that the bi-directional expansion needs to compute real distances for a smaller number of nodes pairs. For the reason, $\mathcal{B}$**-KDJ** algorithm chooses a dimension with the minimum sweeping index as a sweeping axis.

One thing we may be concerned about is the cost of computing a sweeping index for each dimension. The sweeping index may appear expensive to compute, as it includes two integral terms. For given $|r|_x$ and $|s|_x$ values and the current $q\mathcal{D}_{max}$ value available from the distance queue, however, the sweeping index is reduced to a formula that involves only a few simple arithmetic operations. Suppose nodes r and s are not intersected along a dimension x, the minimum x-axis distance between them is β, and node r appears before node s in the plane-sweep direction along x-axis. (Again, see the left diagram in Figure 6.) Then, the second integral term of Equation (2) becomes zero, because all the entries of r have already been swept when the first entry of s is encountered. The first integral term varies depending on the conditions among $q\mathcal{D}_{max}$, $|r|_x$ and $|s|_x$ values and the proximity (*i.e.*, β) of nodes r and s along a chosen dimension. The right diagram in Figure 6 illustrates how we can compute the first integral term and obtain a simple expression when a condition $\beta \le q\mathcal{D}_{max} \le \beta + \min\{|r|_x, |s|_x\}$ is satisfied.

If nodes r and s are not separated, both the integral terms of Equation (2) become non-zero. By a similar reasoning, each integral term is also transformed into a formula with only a few simple arithmetic operations. Table 1 summaries the formulae of the sweeping index for nodes r and s that are in three different spatial relationships: s is separated from, intersected with, or contained in r. The values of α, β and δ in Table 1 are determined by the side lengths of r and s and their spatial relationship as illustrated in Figure 7.

Considering that each R-tree node may contain hundreds of entries, it will be a trivial cost to compute a sweeping index for each dimension, while the performance gain by the sweeping axis selection is expected to be significant. This is empirically corroborated by our experiments in Section 5.

[1] An actual number of node pairs for which we need to compute the real distances would be computed by counting the number of s's entries within $q\mathcal{D}_{max}$ axis distance from each entry of r, counting the number of r's entries within $q\mathcal{D}_{max}$ axis distance from each entry of s, and then adding all the counts and dividing the count sum by two. However, this process will be very expensive.



(a) Separated (b) Intersected (c) Contained

Figure 7: Spatial relationships between nodes r and s and their projected intervals

r and s	The first integral term of Equation (2)
Separated	$\begin{cases} 0 & \text{if } q\mathcal{D}_{max} < \beta, \\ \frac{(q\mathcal{D}_{max}-\beta)^2}{2\lvert s\rvert_x} & \text{if } \beta \le q\mathcal{D}_{max} < \beta + \min\{\lvert r\rvert_x, \lvert s\rvert_x\}, \\ \frac{2\lvert r\rvert_x(q\mathcal{D}_{max}-\beta)-\lvert r\rvert_x^{2}}{2\lvert s\rvert_x} & \text{if } \lvert r\rvert_x+\beta \le q\mathcal{D}_{max} < \lvert s\rvert_x+\beta, \\ q\mathcal{D}_{max}-\beta-\frac{\lvert s\rvert_x}{2} & \text{if } \lvert s\rvert_x+\beta \le q\mathcal{D}_{max} < \lvert r\rvert_x+\beta, \\ \lvert r\rvert_x - \frac{(\max\{\lvert r\rvert_x+\lvert s\rvert_x+\beta-q\mathcal{D}_{max},0\})^2}{2\lvert s\rvert_x} & \text{if } \max\{\lvert r\rvert_x,\lvert s\rvert_x\}+\beta \le q\mathcal{D}_{max}. \end{cases}$
Intersected	$\begin{cases} q\mathcal{D}_{max} + \frac{q\mathcal{D}_{max}(q\mathcal{D}_{max}-2\delta)}{2\lvert s\rvert_x} & \text{if } q\mathcal{D}_{max} < \min\{\alpha,\delta\}, \\ \frac{2\lvert r\rvert_x q\mathcal{D}_{max}-\alpha^2}{2\lvert s\rvert_x} & \text{if } \alpha \le q\mathcal{D}_{max} < \delta, \\ q\mathcal{D}_{max} - \frac{\delta^2+(\max\{q\mathcal{D}_{max}-\alpha,0\})^2}{2\lvert s\rvert_x} & \text{if } \delta \le q\mathcal{D}_{max} < \lvert s\rvert_x+\alpha, \\ \alpha + \frac{\lvert s\rvert_x^{2}-\delta^2}{2\lvert s\rvert_x} & \text{if } \lvert s\rvert_x+\alpha \le q\mathcal{D}_{max}. \end{cases}$
Contained	$\begin{cases} q\mathcal{D}_{max} & \text{if } q\mathcal{D}_{max} < \alpha, \\ q\mathcal{D}_{max} - \frac{(q\mathcal{D}_{max}-\alpha)^2}{2\lvert s\rvert_x} & \text{if } \alpha \le q\mathcal{D}_{max} < \lvert s\rvert_x+\alpha, \\ \alpha + \frac{\lvert s\rvert_x}{2} & \text{if } \lvert s\rvert_x+\alpha \le q\mathcal{D}_{max}. \end{cases}$

r and s	The second integral term of Equation (2)
Separated	0
Intersected	$\begin{cases} q\mathcal{D}_{max} - \frac{q\mathcal{D}_{max}(2\alpha-q\mathcal{D}_{max})}{2\lvert r\rvert_x} & \text{if } q\mathcal{D}_{max} < \lvert r\rvert_x-\alpha, \\ \frac{(\lvert r\rvert_x-\alpha)^2}{2\lvert r\rvert_x} & \text{if } \lvert r\rvert_x-\alpha \le q\mathcal{D}_{max}. \end{cases}$
Contained	$\begin{cases} \frac{q\mathcal{D}_{max}\lvert s\rvert_x}{\lvert r\rvert_x} & \text{if } q\mathcal{D}_{max} < \delta, \\ \frac{2q\mathcal{D}_{max}\lvert s\rvert_x-(q\mathcal{D}_{max}-\delta)^2}{2\lvert r\rvert_x} & \text{if } \delta \le q\mathcal{D}_{max} < \lvert s\rvert_x+\delta, \\ \frac{\lvert s\rvert_x(\lvert s\rvert_x+2\delta)}{2\lvert r\rvert_x} & \text{if } \lvert s\rvert_x+\delta \le q\mathcal{D}_{max}. \end{cases}$

Table 1: The first and second integral terms of the sweeping index for r and s (the values of α, β and δ are determined by the relative positions of r and s as illustrated in Figure 7.)

3.3 Sweeping Direction

Once a sweeping axis is determined, a sweeping direction can be chosen to be either a *forward* sweep or a *backward* sweep. For a pair of nodes r and s, we can define the forward and backward sweeps as follows.

- A forward plane-sweep scans the entries of r and s in an increasing order of coordinates along the chosen sweeping axis.

- A backward plane-sweep scans the entries of r and s in a decreasing order of coordinates along the chosen sweeping axis.

Consider nodes r and s projected on a sweeping axis. The projected images generate three consecutive closed intervals on the sweeping axis, unless the projected images are completely overlapped. For example, if nodes r and s are intersected as in Figure 7(b), an interval in the left is projected from r, one in the middle from both r and s, and one in the right from s. The interval in the middle may be projected from none of r and s, if r and s are separate as in Figure 7(a). Both the intervals in the left and right may be projected from the same node, if one node is contained in the other as in Figure 7(c).

However, it does not matter which node an interval is projected from, because a sweeping direction is determined solely on the relative length of the intervals in the left and right (*i.e.*, α and δ). A sweeping direction is determined by comparing the length of the left and right intervals: *if the left projected interval is shorter than the right one ($\alpha < \delta$), then a forward direction is chosen. Otherwise, a backward direction is chosen.* By this strategy of choosing a sweeping direction, a pair of nodes closer to each other are likely to be examined earlier than those farther from each other. This in turn allows a pair of closer nodes to be inserted into the main queue (and the distance queue as well if they are an object pair), and helps reduce the $q\mathcal{D}_{max}$ value more rapidly.

In summary, the sweeping axis selection improves the bi-directional node expansion step by pruning more pairs of entries whose axis distances are larger than the $q\mathcal{D}_{max}$ value, while the sweeping direction selection does so by reducing the $q\mathcal{D}_{max}$ value more rapidly.

3.4 Maximum Distance As a Secondary Priority

The main queue maintains node pairs generated by node expansion in an increasing order of their distances. An issue we have not addressed is how we order node pairs of an equal distance in the main queue. This is a non-trivial issue particularly because a pair of intersected nodes are considered to have zero distance between them. Since there may be many pairs of intersected nodes in the main queue, the performance impact by the way of breaking ties is potentially high.

Hjaltason and Samet used the depths of R-tree nodes to break tie [16]. For given two pairs of nodes of an equal distance, they proposed to give preference to a pair that contains a node with the maximum tree depth among the four tree nodes. This approach may assist their distance join algorithms in getting to leaf nodes and data objects as quickly as possible. However, it is not always beneficial to process node pairs in depth-first order, because it does not always accelerate the reduction of $q\mathcal{D}_{max}$ value.

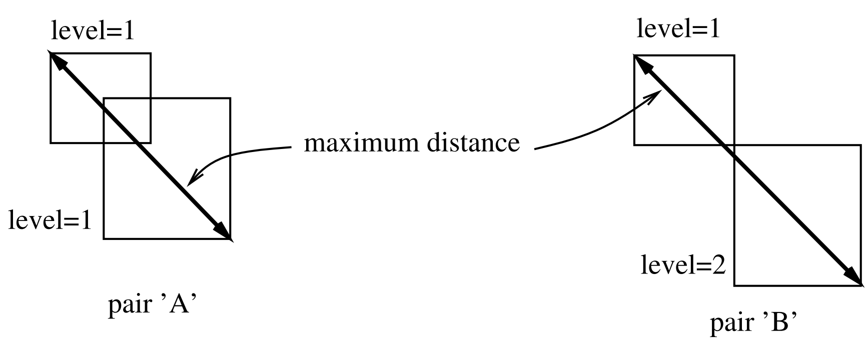


Figure 8: Breaking ties for node pairs of an equal distance

Consider two node pairs A and B in Figure 8 as an example. If we follow the depth-first approach, node pair B will be placed before node pair A in the main queue, because the former contains a node whose depth is deeper than those of the nodes in the latter. However, it is very likely that node pair A contains more pairs of entries with shorter distances than node pair B does, because the maximum distance of node pair A is shorter than that of node pair B. Based on this observation, we propose to use the maximum distance of a node pair to break ties. By choosing the maximum distance of a node pair as a secondary priority of heap (*i.e.*, the main queue), node pairs with shorter distances can be processed in the earlier stage of distance join. Consequently, $q\mathcal{D}_{max}$ value can be reduced more rapidly and the number of distance computations and queue insertions can be reduced.

4 Adaptive Multi-Stage k-Distance Join

In $\mathcal{B}$**-KDJ** algorithm, $q\mathcal{D}_{max}$ value is initially set to an infinity and becomes smaller as the algorithm proceeds. The adaptation of the $q\mathcal{D}_{max}$ value has a crucial impact on the performance of $\mathcal{B}$**-KDJ** algorithm, as $q\mathcal{D}_{max}$ is used as a cutoff to prevent pairs of distant nodes from entering the main queue. If the $q\mathcal{D}_{max}$ value approaches to the real $\mathcal{D}_{max}$ value slowly, the early stage of $\mathcal{B}$**-KDJ** algorithm will be delayed considerably for handling too many pairs of distant nodes. Consequently, at the end of the algorithm processing, the main queue may end up with a large number of distant pairs whose insertions to the main queue were not necessary. The performance effect of *slow start* is more pronounced for a larger k, as the main queue and distance queue tend to grow large for a large k, and thereby increasing the $q\mathcal{D}_{max}$ value. ¿From our experiments with k as high as 100,000, we observed that more than 90 percent of execution time of k-distance join algorithms was spent to produce the first one percent (*i.e.*, 1,000 pairs) of final query results.

In this section, we propose new adaptive multi-stage distance join algorithms $\mathcal{AM}$**-KDJ** and $\mathcal{AM}$**-IDJ** that mitigate the slow start problem by *aggressive pruning* and *compensation*.

4.1 Adaptive Multi-Stage k-Distance Join

The slow start problem is essentially caused by a pruning strategy using $q\mathcal{D}_{max}$, whose value is dynamically updated as tree indexes are traversed and therefore not under direct control of the distance join algorithms. To circumvent this problem, we introduce a new pruning measure $e\mathcal{D}_{max}$, which is an estimated $\mathcal{D}_{max}$ value for a given k. The $e\mathcal{D}_{max}$ value is set to an initial estimation at the beginning and adaptively corrected during the algorithm processing. We will discuss techniques for initial estimation and adaptive correction in Section 4.3.

$\mathcal{AM}$**-KDJ** algorithm is similar to $\mathcal{B}$**-KDJ** algorithm in that both the algorithms use a bi-directional node expansion. However, unlike the single-stage $\mathcal{B}$**-KDJ** algorithm, where only $q\mathcal{D}_{max}$ is used for pruning, both $q\mathcal{D}_{max}$ and $e\mathcal{D}_{max}$ are used as cutoff values for pruning distant pairs in $\mathcal{AM}$**-KDJ** algorithm. Specifically, in the *aggressive pruning* stage (described in Algorithm 2),

- $e\mathcal{D}_{max}$ is used for pruning based on *axis distances* for aggressive pruning and thereby limiting the size of main and distance queues (line 23),
- $q\mathcal{D}_{max}$ is used for further pruning on *real distances* for nodes whose axis distances are within $e\mathcal{D}_{max}$, in the same way as $\mathcal{B}$**-KDJ**.

With a properly estimated $e\mathcal{D}_{max}$ value, $\mathcal{AM}$**-KDJ** algorithm can prune a large number of distant pairs in the first stage and avoid a significant portion of delay due to the slow start. However, if $\mathcal{AM}$**-KDJ** algorithm becomes too aggressive by choosing an underestimated $e\mathcal{D}_{max}$ value, even close enough pairs may be discarded incorrectly. To avoid any false dismissals, we introduce another queue called *compensation queue* ($\mathcal{Q}_C$). The compensation queue stores every non-object node pair retrieved from the main queue (line 11), except those for whom all entries have been examined. Note that $q\mathcal{D}_{max}$ *but not* $e\mathcal{D}_{max}$ is used for nodes whose axis distances are within $e\mathcal{D}_{max}$ (line 24). If $e\mathcal{D}_{max}$ values are used instead, this algorithm does not guarantee the correctness due to potential false dismissals. Using $q\mathcal{D}_{max}$ values also makes the performance of $\mathcal{AM}$**-KDJ** fairly insensitive to estimated $e\mathcal{D}_{max}$ values.

For example, in Figure 9 (drawn from Figure 4), an anchor node r_1 is paired up with nodes s_1 and s_2 but not with s_3 and s_4 in the aggressive pruning stage, because only s_1 and s_2 are within $e\mathcal{D}_{max}$ from the anchor node r_1. Thus, $\mathcal{AM}$**-KDJ** algorithm inserts only two pairs ($\langle r_1, s_1\rangle$, $\langle r_1, s_2\rangle$) into a main queue, instead of all four pairs ($\langle r_1, s_1\rangle$, $\langle r_1, s_2\rangle$, $\langle r_1, s_3\rangle$, $\langle r_1, s_4\rangle$) that would be enqueued by $\mathcal{B}$**-KDJ** algorithm. Then, the pair $\langle r, s\rangle$ currently being expanded is inserted into a compensation queue.

The aggressive pruning stage ends when one of the following conditions is satisfied: (1) the main queue becomes empty (line 5), (2) k or more query results have been returned (line 5), or (3) the distance of a node pair retrieved from

Algorithm 2: $\mathcal{AM}$**-KDJ**: Adaptive Multi-Stage K-Distance Join Algorithm (Aggressive Pruning)

1: set $AnswerSet \leftarrow$ an empty set;
2: set $\mathcal{Q}_M$, $\mathcal{Q}_D$, $\mathcal{Q}_C \leftarrow$ empty main, distance and compensation queues ;
3: set $e\mathcal{D}_{max} \leftarrow$ an initial estimated $\mathcal{D}_{max}$;
4: insert a pair $\langle R.root, S.root\rangle$ to the main queue $\mathcal{Q}_M$;
5: **while** $|AnswerSet| < k$ *and* $\mathcal{Q}_M \neq \emptyset$ **do**
6:     set c $\leftarrow$ dequeue($\mathcal{Q}_M$);
7:     **if** c *is an* $\langle object, object\rangle$ **then** $AnswerSet \leftarrow \{c\} \cup AnswerSet$;
    else
8:         **if** $q\mathcal{D}_{max} \leq e\mathcal{D}_{max}$ **then** $e\mathcal{D}_{max} \leftarrow q\mathcal{D}_{max}$; // overestimated $e\mathcal{D}_{max}$
9:         **if** c.$distance > e\mathcal{D}_{max}$ **then**
            reinsert c back into $\mathcal{Q}_M$;
            break; // Terminate the Aggressive Pruning stage.
        end
10:         AggressivePlaneSweep(c);
11:         enqueue($\mathcal{Q}_C$, c);
    end
end
12: **if** $|AnswerSet| < k$ **then** execute Algorithm 3;

procedure AggressivePlaneSweep($\langle l, r\rangle$)
13: set L $\leftarrow$ sort_axis($\{$entries of $l\}$); // Sort the entries of l by axis values.
14: set R $\leftarrow$ sort_axis($\{$entries of $r\}$); // Sort the entries of r by axis values.
15: **while** $L \neq \emptyset$ *and* $R \neq \emptyset$ **do**
16:     $n \leftarrow$ a node with the min axis value $\in L \cup R$; // n becomes an anchor.
17:     **if** $n \in L$ **then**
18:         $L \leftarrow L - \{n\}$; $AggressiveSweepPruning(n, R)$;
19:         $n.compensate \leftarrow$ a node in R with the min axis value and not yet paired with n;
    else
20:         $R \leftarrow R - \{n\}$; $AggressiveSweepPruning(n, L)$;
21:         $n.compensate \leftarrow$ a node in L with the min axis value and not yet paired with n;
    end
end

procedure AggressiveSweepPruning($n, List$)
22: **for** *each node* $m \in List$ *in an increasing order of axis value* **do**
23:     **if** $axis\_distance(n, m) > e\mathcal{D}_{max}$ **then** return; // No more candidates.
24:     **if** $real\_distance(n, m) \leq q\mathcal{D}_{max}$ **then**
25:         insert $\langle n, m\rangle$ into $\mathcal{Q}_M$;
26:         **if** $\langle n, m\rangle$ *is an* $\langle object, object\rangle$ **then** insert $real\_distance(n, m)$ into $\mathcal{Q}_D$; // $q\mathcal{D}_{max}$ modified.
    end
end

the main queue becomes greater than $e\mathcal{D}_{max}$ (line 9). When the condition (2) is satisfied, obviously it is not necessary to execute the compensation stage of the $\mathcal{AM}$**-KDJ** algorithm. (An overestimated $e\mathcal{D}_{max}$ can also be detected by comparing with $q\mathcal{D}_{max}$ value (line 8). In this case, instead of terminating the first stage, $\mathcal{AM}$**-KDJ** behaves exactly the same as $\mathcal{B}$**-KDJ** algorithm by using $q\mathcal{D}_{max}$ alone as a cutoff value.) When the condition (3) is satisfied, $e\mathcal{D}_{max}$ must have been underestimated, because all the object pairs returned after this point will have a greater distance than $e\mathcal{D}_{max}$. Since an object pair with the k-th largest distance has not been obtained by the time when the aggressive pruning stage comes to an end, the compensation stage (described in Algorithm 3) begins its processing by inserting all the pairs stored in the compensation queue to the main queue.

In the compensation stage, the pairs in the main queue are processed in a similar way as $\mathcal{B}$**-KDJ** algorithm, but there are two notable differences from $\mathcal{B}$**-KDJ** algorithm. First, the entries are not sorted again, if they have already been sorted in the first stage. Second, for the pairs already expanded once in the first stage, only child pairs not examined in the first stage are processed by plane sweeping. This is feasible by bookkeeping done in the first stage (lines 19 and 21), which stores the information in an additional field ($n.compensate$) attached to a pair being inserted

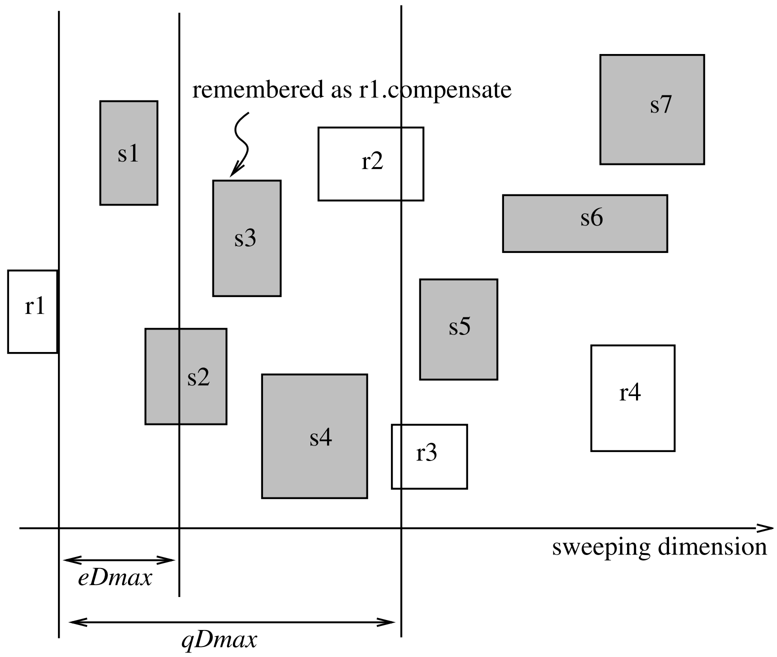


Figure 9: Aggressive pruning with $q\mathcal{D}_{max}$ and $e\mathcal{D}_{max}$

Algorithm 3: $\mathcal{AM}$-KDJ: Adaptive Multi-Stage K-Distance Join Algorithm (Compensation Stage)

insert all elements in $\mathcal{Q}_C$ into $\mathcal{Q}_M$;
while $|\mathcal{A}nswerSet| < k$ *and* $\mathcal{Q}_M \neq \emptyset$ **do**
    set c $\leftarrow$ dequeue($\mathcal{Q}_M$);
    if c *is an* $\langle object, object \rangle$ **then** $\mathcal{A}nswerSet \leftarrow \{c\} \cup \mathcal{A}nswerSet$;
    else CompensatePlaneSweep(c);
end

procedure CompensatePlaneSweep($\langle l, r \rangle$)
L $\leftarrow$ { entries of l sorted in Stage One}; // $\{L[1], L[2], \ldots, L[|L|]\}$
R $\leftarrow$ { entries of r sorted in Stage One}; // $\{R[1], R[2], \ldots, R[|R|]\}$
while $L \neq \emptyset$ *and* $R \neq \emptyset$ **do**
    $n \leftarrow$ a node with the min axis value $\in L \cup R$; // n becomes an anchor.
    if $n \in L$ **then**
        $L \leftarrow L - \{n\}$; $R' \leftarrow$ {node list in R not paired with n in the Stage One };
        // { $R[n.compensate], R[n.compensate + 1], \ldots, R[|R|]$}
        $SweepPruning(n, R')$;
    else
        $R \leftarrow R - \{n\}$; $L' \leftarrow$ {node list in L not paired with n in the Stage One };
        // { $L[n.compensate], L[n.compensate + 1], \ldots, L[|L|]$}
        $SweepPruning(n, L')$;
    end
end

into the compensation queue. For these reasons, the cost of the compensation stage is not considerable compared with the cost of restarting the algorithm. In summary, $\mathcal{AM}$**-KDJ** algorithm uses $e\mathcal{D}_{max}$ to avoid the slow start problem in the aggressive pruning stage and speeds up the query processing.

4.2 Adaptive Multi-stage Incremental Distance Join

Consider on-line query processing and internet database search environments, where users interact with database systems in a way the number of required matches can be determined interactively or changed at any point of query processing. Consider also a complex query that pipelines the results from a spatial distance join to a filter stage. Under these circumstances, the number of pairs (k) that should be returned from a distance join is not known a priori, and hence a k-distance join algorithm proposed in [16] and $\mathcal{B}$**-KDJ** algorithm presented in Section 3 cannot be used directly.

An important advantage of $\mathcal{AM}$**-KDJ** algorithm proposed in the previous section is that $\mathcal{AM}$**-KDJ** algorithm can be extended to an incremental algorithm (we call $\mathcal{AM}$**-IDJ**) to support the interactive applications described above. The main difference between $\mathcal{AM}$**-KDJ** and $\mathcal{AM}$**-IDJ** algorithms is that $\mathcal{AM}$**-IDJ** does not maintain a distance queue. Thus, $\mathcal{AM}$**-IDJ** algorithm uses $e\mathcal{D}_{max}$ alone as a cutoff value for pruning distant pairs, because $q\mathcal{D}_{max}$ would be drawn only from a distance queue.

Without $q\mathcal{D}_{max}$, $\mathcal{AM}$**-IDJ** works as a stepwise incremental algorithm. First, $\mathcal{AM}$**-IDJ** starts by determining an initial value k_1 and estimating an initial $e\mathcal{D}_{max1}$ for k_1. Then, it performs the same way as the first stage of $\mathcal{AM}$**-KDJ** algorithm without $q\mathcal{D}_{max}$. However, the first stage may terminate before producing enough object pairs (*i.e.*, less than k_1), if $e\mathcal{D}_{max}$ is underestimated. If that happens, $\mathcal{AM}$**-IDJ** algorithm estimates $e\mathcal{D}_{max2}$ value for k_2 ($k_2 > k_1$) and initiates a compensation stage.

Even when a sufficient number of object pairs have been returned from the first stage, users may request more answers. Then, $\mathcal{AM}$**-IDJ** initiates a compensation stage by determining k_2 and estimating a new $e\mathcal{D}_{max2}$ accordingly. As shown in Figure 10 (drawn from Figure 4), the compensation stage can initiate another compensation stage at the end of its processing, by choosing k_3 and $e\mathcal{D}_{max3}$. This process continues until users stop requesting more answers. In this way, $\mathcal{AM}$**-IDJ** algorithm can be used to produce query results incrementally without limiting the maximum number of pairs in advance. Except the first stage of $\mathcal{AM}$**-IDJ** algorithm where the $AggresivePlaneSweep$ procedure (in Algorithm 2) is used, the $CompensatePlaneSweep$ procedure (in Algorithm 3) is used to prune distant pairs in the rest of the compensation stages.

4.3 Estimating the Maximum Distance ($e\mathcal{D}_{max}$)

Both $\mathcal{AM}$**-KDJ** and $\mathcal{AM}$**-IDJ** algorithms process a distance join query based on an estimated cutoff value $e\mathcal{D}_{max}$. Thus, there should be a way to obtain an initial estimate and correct the estimate adaptively as the algorithms proceed. Assuming data sets are uniformly distributed, we provide mechanisms to choose an initial estimate of $e\mathcal{D}_{max}$, and to adaptively correct it.

If the distribution of a data set is skewed, then a larger number of close pairs can be found in a smaller dense region of the data space. We expect that the formulae given in this section tend to overestimate $e\mathcal{D}_{max}$ value for non-uniformly distributed data sets, especially when a stopping cardinality k is far smaller than the number of all pairs of objects (*i.e.*, $k \ll |R| \times |S|$). This was corroborated by our experiments as described in Section 5.4.

4.3.1 Initial estimation

Let $|R|$ and $|S|$ be the number of data objects in MBRs R and S, respectively. Suppose that most regions of R and S overlap. Then, for a data object r in R contained in the region shared by R and S, the expected number of objects in S within distance d from r is approximated by $|S| \times \frac{\pi \times d^2}{area(R \cap S)}$, assuming the circle centered at r of radius d is fully contained in the shared region (*i.e.*, $R \cap S$). Thus, by considering all data objects in R, the total number of object pairs within distance d can be approximated by $|R| \times |S| \times \frac{\pi \times d^2}{area(R \cap S)}$.

When the target number of object pairs, k, is given with a query, we can obtain the initial estimation of $\mathcal{D}_{max}$ by

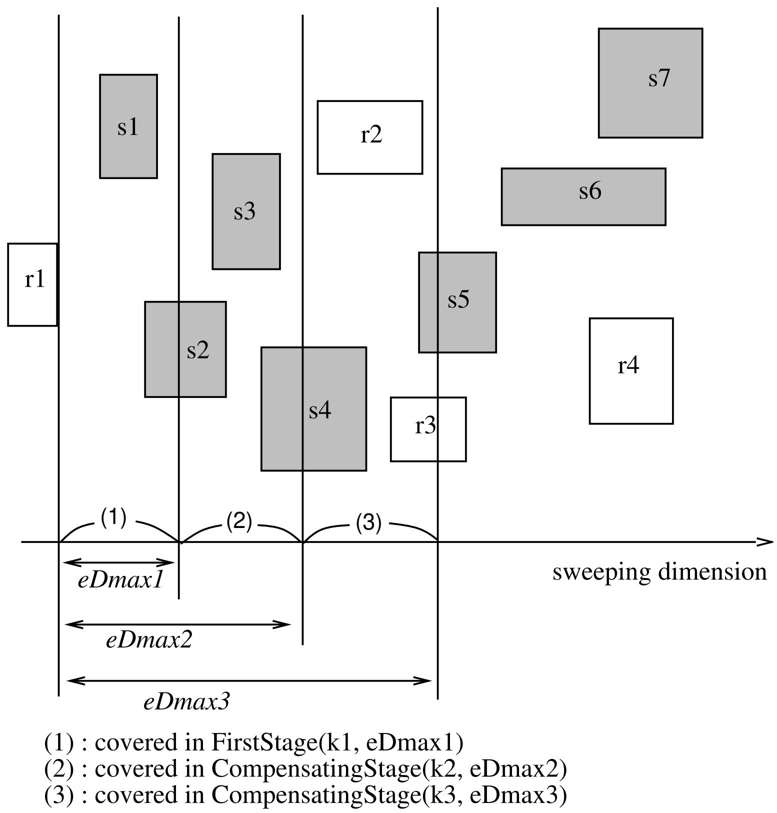


Figure 10: Step-Wise Incremental Distance Join

setting k to the above formula, as follows

$$k = |R| \times |S| \times \frac{\pi \times d^2}{area(R \cap S)},$$

and then by replacing d with $e\mathcal{D}_{max}$. Therefore, for a given stopping cardinality k, the initial estimation of $\mathcal{D}_{max}$ can be obtained by the following equation.

$$e\mathcal{D}_{max} = \sqrt{k \times \rho} \qquad \text{(where } \rho = \frac{area(R \cap S)}{\pi \times |R| \times |S|}\text{)}. \tag{3}$$

Evidently this equation can be applied only when R and S overlap. Nonetheless, it is unlikely this will be a serious limitation, because overlapping node pairs always come before non-overlapping pairs in the main queue $\mathcal{Q}_M$. For two sets of data objects to be joined by distance, the root nodes of two corresponding R-trees are commonly expected to overlap each other under most practical circumstances. We can then make an initial estimation of $\mathcal{D}_{max}$from the pair of root nodes.

4.3.2 Adaptive Correction of Estimated Distance $e\mathcal{D}_{max}$

The performance of $\mathcal{AM}$**-KDJ** and $\mathcal{AM}$**-IDJ** algorithms can be further improved by adaptively adjusting the value of $e\mathcal{D}_{max}$ at runtime. Adaptive correction of $e\mathcal{D}_{max}$ can be done at any point of query processing by estimating a new $e\mathcal{D}_{max}$ from the number of object pairs k_0 ($k_0 < k$) obtained up to the point and the real distance of the k_0-th object pair, $\mathcal{D}_{max(k_0)}$. Specifically, the new estimate $e\mathcal{D}_{max}{}'$ can be computed from Equation (3) as

$$e\mathcal{D}_{max}{}' = \sqrt{\mathcal{D}_{max}{}^2_{(k_0)} + (k - k_0)\rho} \tag{4}$$

by arithmetic correction, or as

$$e\mathcal{D}_{max}{}' = \mathcal{D}_{max(k_0)} \times \sqrt{k/k_0} \quad (5)$$

by geometric correction if $\mathcal{D}_{max}(k_0) \neq 0$. In practice, we propose computing $e\mathcal{D}_{max}{}'$ in both ways, and then choose the minimum if the query processing should be on the aggressive side. Otherwise, the maximum is chosen as $e\mathcal{D}_{max}{}'$.

Note that the new estimate $e\mathcal{D}_{max}{}'$ can sometimes grow beyond the previous estimate. If this happens, some pairs whose distances are larger than the previous estimate but smaller than the new estimate could have already been pruned and will never be examined in the current processing stage under the new estimate. Thus, to guarantee the correctness of the distance join, the algorithm should initiate a compensation stage, as soon as a pair whose distance is smaller than the smallest $e\mathcal{D}_{max}$ is dequeued from the main queue.

4.4 Queue Management

Efficient queue management is one of the key components of the distance join algorithms proposed in this paper. Each of the $\mathcal{B}$**-KDJ**, $\mathcal{AM}$**-KDJ**, and $\mathcal{AM}$**-IDJ** algorithms relies on the use of one or more priority queues for query processing. In particular, the main queue ($\mathcal{Q}_M$) is heavily used by all of the proposed algorithms, and its performance impact is significant. In the worst case, the main queue can grow as large as the product of *all* objects of two R-tree indexes. That is, the size of $\mathcal{Q}_M$ is in $\mathcal{O}(|R_{obj}| \times |S_{obj}|)$, where $|R_{obj}|$ and $|S_{obj}|$ are the number of all objects in R and S, respectively. Thus, it is not always feasible to store the main queue in memory.

It was reported in [16] that a simple memory-based implementation might slow down query processing severely, due to excessive virtual memory thrashing. A hybrid memory/disk scheme [16] and a technique based on range partitioning [10] have been proposed to improve queue management and to avoid wasted sorting I/O operations. We adopt a similar scheme for queue management, which partitions a queue by range based on distances of pairs. A partition in the shortest distance range is kept in memory as a heap structure, while the rest of partitions are stored on disk as merely unsorted piles.

When the in-memory heap becomes full, it is *split* into two parts, and then one in the longer distance range is moved to disk as a new segment. When the in-memory heap becomes empty, a disk-resident segment in the shortest distance range or a part of the segment is *swapped in* to memory to fill up the in-memory heap. Each of the **split** and **swap-in** operations requires $\mathcal{O}(n \log n)$ computational cost for a heap of n elements as well as I/O cost for reading and writing a segment. Thus, it is important to minimize the required number of those operations, which largely depends on the partition boundary values between the in-memory heap and the first disk-resident segment, and between those consecutive segments. However, as it is impossible to predict an exact $\mathcal{D}_{max}$ value for a given k, so is it difficult to determine optimal distance values as segment boundaries.

To address this issue, we use Equation (3) to determine the boundary distance values. Suppose n is the number of elements that can be stored in an in-memory heap. Then, the boundary value between the in-memory heap and the first disk-resident segment is given by $\sqrt{n \times \rho}$, and the boundary value between the first and second segments is given by $\sqrt{(2 \times n) \times \rho}$, and so on.

In addition to a main queue, multi-stage algorithms $\mathcal{AM}$**-KDJ** and $\mathcal{AM}$**-IDJ** use a compensation queue ($\mathcal{Q}_C$) in the compensation stage. Unlike the main queue, a compensation queue does not store any pair of objects. In other words, a compensation queue can store pairs of non-object R-tree nodes only. Thus, the size of $\mathcal{Q}_C$ is in $\mathcal{O}(|R_{node}| \times |S_{node}|)$, where $|R_{node}|$ and $|S_{node}|$ are the number of nodes (both internal and leaf nodes) in R and S, respectively. This is a significantly lower upper-bound than a main queue has. We also observed from our experiments that compensation queues were several orders of magnitude smaller than main queues. As for a distance queue used by $\mathcal{B}$**-KDJ** and $\mathcal{AM}$**-KDJ** algorithms, its size is always bounded by a given k value. For these reasons, under most circumstances, we assume either a compensation queue and a distance queue fits in memory. If any of these queues outgrows memory, the same partitioning technique used for a main queue will be applied.

5 Performance Evaluation

In this section, we evaluate the proposed algorithms empirically and compare with previous work. In particular, the proposed $\mathcal{B}$**-KDJ**, $\mathcal{AM}$**-KDJ** and $\mathcal{AM}$**-IDJ** algorithms were compared with Hjaltason and Samet's k-distance and incremental distance join algorithms (hereinafter denoted as HS-**KDJ** and HS-**IDJ**, respectively) for k-distance join (**KDJ**) and incremental distance join (**IDJ**) queries. We also include the performance of an R-tree based spatial join algorithm [8] combined with a sort operation (denoted as SJ-**SORT**) in most of the experiments. For each distance join query, a spatial join operation was performed with a real $\mathcal{D}_{max}$ value to generate the k nearest pairs. Then, an external sort operation was performed to return the query results in an increasing order of distances. Note that SJ-**SORT** cannot be applied without knowing a real $\mathcal{D}_{max}$ value, and we made a favorable assumption for SJ-**SORT** that the real $\mathcal{D}_{max}$ value was known to SJ-**SORT** a priori. Thus, we conjecture that SJ-**SORT** followed by an external sort yields the best known lower bound performance for distance join processing.

5.1 Experimental Settings

Experiments were performed on a Sun Ultrasparc-II workstation running on Solaris 2.7. This workstation has 256 MBytes of memory and 9 GBytes of disk storage (Seagate ST39140A) with Ultra 10 EIDE interface. The disk is locally attached to the workstation and used to store databases, queues and any temporary results. We used the direct I/O feature of Solaris for all the experiments to avoid operating system's cache effects, and the average disk access bandwidth was about 0.5 MBytes/sec for random accesses and about 5 MBytes/sec for sequential accesses.

Data sets To evaluate distance join algorithms, we used real-world data sets in TIGER/Line97 from the U.S. Bureau of Census [20]. The particular data sets we used were 633,461 streets and 189,642 hydrographic objects from the Arizona state. Throughout the entire set of experiments, the same page size of 4 KBytes was used for disk I/O and R*-tree [3] nodes.

Metrics We measured the performance of various algorithms based on the following metrics to compare the algorithms in different aspects such as computational cost and I/O cost.

1. *number of distance computations*: The cost of computing distances between pairs of nodes (or objects) constitutes a significant portion of the computational cost of a distance join operation. Thus, the total number of distance computations required by a distance join algorithm provides a direct indication of its computational performance.

2. *number of queue insertions*: The task of managing a main queue is largely I/O intensive as well as CPU intensive. Inserting a node pair into the in-memory portion of the queue is CPU intensive, while inserting into the disk resident portion is I/O intensive. We measured the CPU and I/O cost separately for the two different queue insertions.

3. *number of R-tree node accesses*: The number of R-tree nodes accessed during distance join processing is another I/O intensive metric. We measured actual number of nodes fetched from disk with varying R-tree buffer sizes.

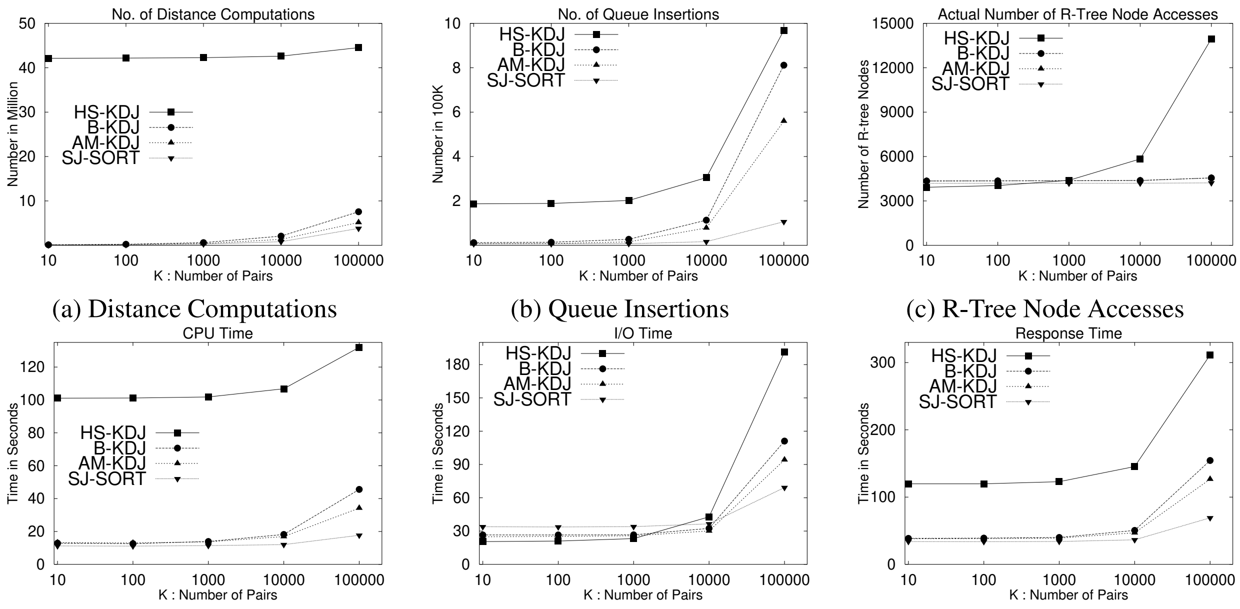
4. *response time*: Actual query response times were measured for overall performance of distance join algorithms. CPU and I/O costs were considered separately in measuring the response times.

5.2 Evaluation of k-Distance Joins

In this set of experiments, we varied a stopping cardinality k from 10 to 100,000 to compare the performance of HS-**KDJ**, $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** algorithms. The sizes of in-memory portion of a main queue and R-tree buffer were fixed to 512 KBytes each. For $\mathcal{AM}$-**KDJ** algorithm, we used Equation (3) to estimate $e\mathcal{D}_{max}$ values, and we observed a tendency for $e\mathcal{D}_{max}$ values to be overestimated with respect to real $\mathcal{D}_{max}$ values. For example, for $k = 100,000$, $e\mathcal{D}_{max}$ was about 2.3 times larger than a real $\mathcal{D}_{max}$.

Figure 11(a) shows that both $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** reduced the number of distance computations significantly. The numbers of distance computations required by the two algorithms were smaller than those required by HS-**KDJ** algorithm by up to two orders of magnitude. $\mathcal{AM}$-**KDJ** was almost identical to SJ-**SORT** by this metric. This demonstrates that the optimized plane-sweep method was very effective in pruning pairs generated by bi-directional expansions. On the other hand, HS-**KDJ** algorithm examines all possible pairs exhaustively in uni-directional expansions.

In Figure 11(b), Both $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** achieved significant reductions in queue insertions for all k values. $\mathcal{AM}$-**KDJ** was always better than $\mathcal{B}$-**KDJ** particularly for large k values. This result confirms our conjecture that the



(a) Distance Computations (b) Queue Insertions (c) R-Tree Node Accesses

(d) CPU Time (e) I/O Time (f) Response Time

Figure 11: Performance of k-Distance Joins

optimized plane-sweep method can prevent an explosion of a main queue that would be caused by bi-directional node expansions without the optimized plane-sweep.

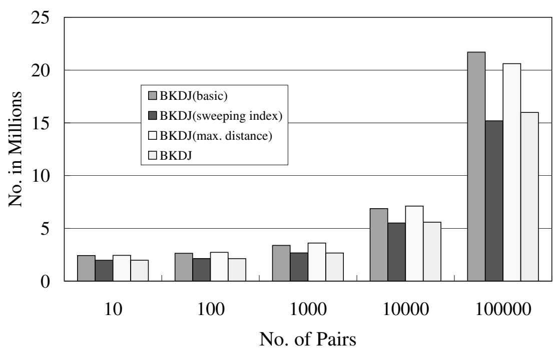
Figure 11(c) shows the number of R-tree nodes fetched from disk for distance join processing by each algorithm. For large k values, the proposed $\mathcal{B}$**-KDJ** and $\mathcal{AM}$**-KDJ** algorithms required a far smaller number of R-tree node accesses than HS**-KDJ** algorithm. For small k values, on the other hand, HS**-KDJ** algorithm was slightly better than the other algorithms, due to its more localized node access patterns for small k. Table 2 compares the number of R-tree nodes that would be fetched from disk with R-tree buffer size set to zero. Apparently, the bi-directional node expansion used by $\mathcal{B}$**-KDJ** and $\mathcal{AM}$**-KDJ** algorithms requires much less number of R-tree node accesses than uni-directional node expansion used by HS**-KDJ** algorithm. It should be noted that the number of R-tree node accesses for $\mathcal{B}$**-KDJ**, $\mathcal{AM}$**-KDJ** and SJ**-SORT** algorithms are all identical in Table 2. This is because these algorithms use the same bi-directional node expansion and access the same collection of R-tree nodes, though they may traverse an R-tree index in different orders.

The total CPU time spent on executing each algorithm is shown in Figure 11(d). The $\mathcal{B}$**-KDJ** and $\mathcal{AM}$**-KDJ** algorithms consistently outperformed HS**-KDJ** up to an order of magnitude. This significant improvement in computational cost is due mainly to the reduced number of distance computations. Recall that the uni-directional expansion requires distance computations for an exhaustive set of node pairs, while bi-directional node expansion with plane sweeping requires distance computations only for node pairs whose axis distances are smaller than $q\mathcal{D}_{max}$ value at the top of the distance queue. Additionally, the proposed algorithms are further optimized by techniques for selecting sweeping axis and direction and by using maximum distance as a secondary priority for the main queue.

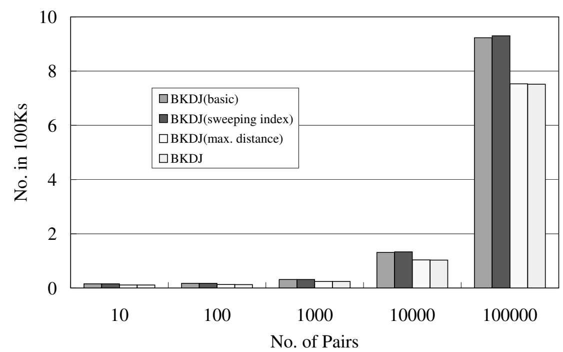
The total I/O time shown in Figure 11(e) reflects mostly the combined effects of queue insertions and R-tree node accesses in Figure 11(b) and Figure 11(c), respectively. Figure 11(f) shows the response time of each algorithm with the CPU and I/O times combined together. Both $\mathcal{B}$**-KDJ** and $\mathcal{AM}$**-KDJ** algorithms outperformed HS**-KDJ** algorithm by a factor of two or three in response times. $\mathcal{AM}$**-KDJ** performed better than $\mathcal{B}$**-KDJ** for large k values, demonstrating that $\mathcal{AM}$**-KDJ** deals with the slow start problem better than $\mathcal{B}$**-KDJ** does. For small k values, both $\mathcal{B}$**-KDJ** and $\mathcal{AM}$**-KDJ** were comparable with SJ**-SORT**. Even for large k values, the response time of $\mathcal{AM}$**-KDJ** was within about 80 percent above that of SJ**-SORT**, which we conjecture yields the best known lower bound performance.

KDJ Algorithms	Stopping cardinality k				
	10	100	1,000	10,000	100,000
$\mathcal{HS}$-**KDJ**	186,184	186,403	186,801	188,354	197,113
$\mathcal{B}$-**KDJ**	12,652	12,660	12,672	12,688	12,916
$\mathcal{AM}$-**KDJ**	12,652	12,660	12,672	12,688	12,916
$\mathcal{SJ}$-**SORT**	12,652	12,660	12,672	12,688	12,916

Table 2: No. of R-tree Node Accesses for k-Distance Joins



(a) Distance Computations



(b) Queue Insertions

Figure 12: Improvements by Optimized Plane Sweep for $\mathcal{B}$-**KDJ**

5.3 Impact of Optimized Plane-Sweep and Secondary Priority

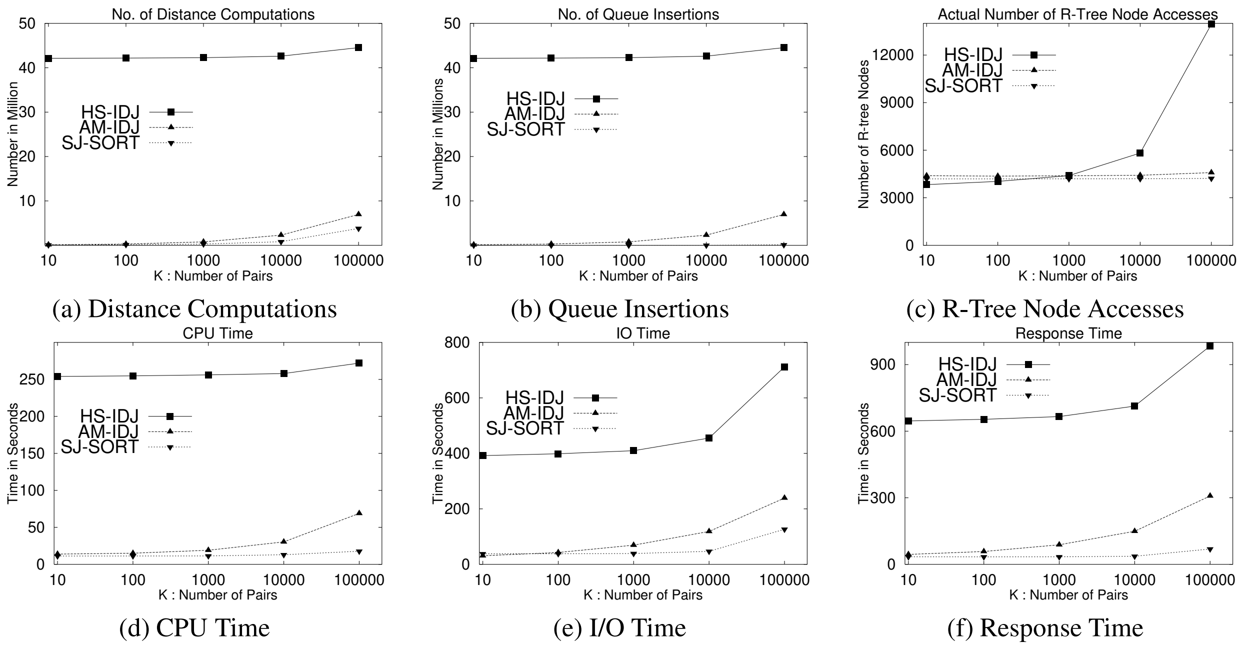
We have proposed optimization techniques for $\mathcal{B}$-**KDJ** in Section 3. One is for selecting sweeping axis and direction, which is mainly aimed at reducing the number of distance computations. The other is using the maximum distance between node pairs as a secondary priority of the main queues, which is mainly aimed at reducing the number of queue insertions. To further analyze the performance impacts of the optimization techniques, we measured the performance of $\mathcal{B}$-**KDJ** (1) with both optimizations turned on, (2) with the sweeping index only, (3) with the secondary priority only, (4) with both optimizations turned off. For the cases with sweeping index turned off, the sweeping index and direction were fixed to x-axis and forward direction.

The sweeping index method alone reduced the number of distance computations by up to 20 percent as shown in Figure 12(a). The use of the maximum distance as a secondary priority alone reduced the number of queue insertions by up to 15 percent as shown in Figure 12(b). The use of the maximum distance also helped decrease the $q\mathcal{D}_{max}$ value more quickly and reduce the number of distance computations slightly as shown in Figure 12(a). However, the synergistic effect of the two optimization techniques was rather insignificant. As they improve the performance of distance join processing largely independently in two different aspects, we recommend that both the optimization techniques be used together.

5.4 Evaluation of Incremental Distance Joins

As in the previous section, we varied a stopping cardinality k from 10 to 100,000 to compare the performance of incremental distance join algorithms $\mathcal{HS}$-**IDJ** and $\mathcal{AM}$-**IDJ**. Like the previous experiments for k-distance joins, the sizes of in-memory portion of a main queue and R-tree buffer were fixed to 512 KBytes.

In Figures 13(a) and 13(b), $\mathcal{AM}$-**IDJ** algorithm required 75 to 98 percent less distance computations and queue insertions than $\mathcal{HS}$-**IDJ** algorithm did. For large k values, as shown in Figure 13(c), $\mathcal{AM}$-**IDJ** algorithm required a much smaller number of disk accesses than $\mathcal{HS}$-**IDJ** algorithm. This is because $\mathcal{AM}$-**IDJ** accesses R-tree nodes using bi-directional node expansion, in the same way as $\mathcal{AM}$-**KDJ** does. The significant improvement in these three metrics in turn led to improvement in response time by an order of magnitude in Figure 13(f). Specifically, the improvement in CPU time (Figure 13(d)) is attributed to the reduction in distance computations and queue insertions, and the



(a) Distance Computations (b) Queue Insertions (c) R-Tree Node Accesses

(d) CPU Time (e) I/O Time (f) Response Time

Figure 13: Performance of Incremental Distance Joins

improvement in I/O time (Figure 13(e)) is attributed to the reduction in queue insertions and R-tree node accesses. Like $\mathcal{AM}$-**KDJ** algorithm, Equation (3) in Section 4.3.1 was used to estimate $e\mathcal{D}_{max}$ values for $\mathcal{AM}$-**IDJ** algorithm.

Now it is well worthwhile investigating the performance impact of the stopping cardinality k. Generally, **KDJ** algorithms make use of the apriori knowledge of the k value to minimize the distance computations and the number of queue insertions. Thus, **KDJ** algorithms are expected to be much faster than **IDJ** algorithms. From our experiments, however, HS-**KDJ** required almost as many distance computations as HS-**IDJ** did. This indicates that HS-**KDJ** does not take advantage of the stopping cardinality enough to achieve performance gain in the distance computations.

In contrast, $\mathcal{AM}$-**KDJ** required only about 70 percent of distance computations that $\mathcal{AM}$-**IDJ** did (Figure 13(a) and Figure 11(a)), and required only 8 percent of queue insertions that $\mathcal{AM}$-**IDJ** did (Figure 13(b) and Figure 11(b)). This is because **KDJ** algorithms need not insert a node pair into main queue if its distance is greater than $q\mathcal{D}_{max}$ value. The number of queue insertions has direct impact on both CPU and I/O times. The response time of $\mathcal{AM}$-**KDJ** algorithm was about 60 percent less than that of $\mathcal{AM}$-**IDJ** algorithm (see Figure 13(f) and Figure 11(f)).

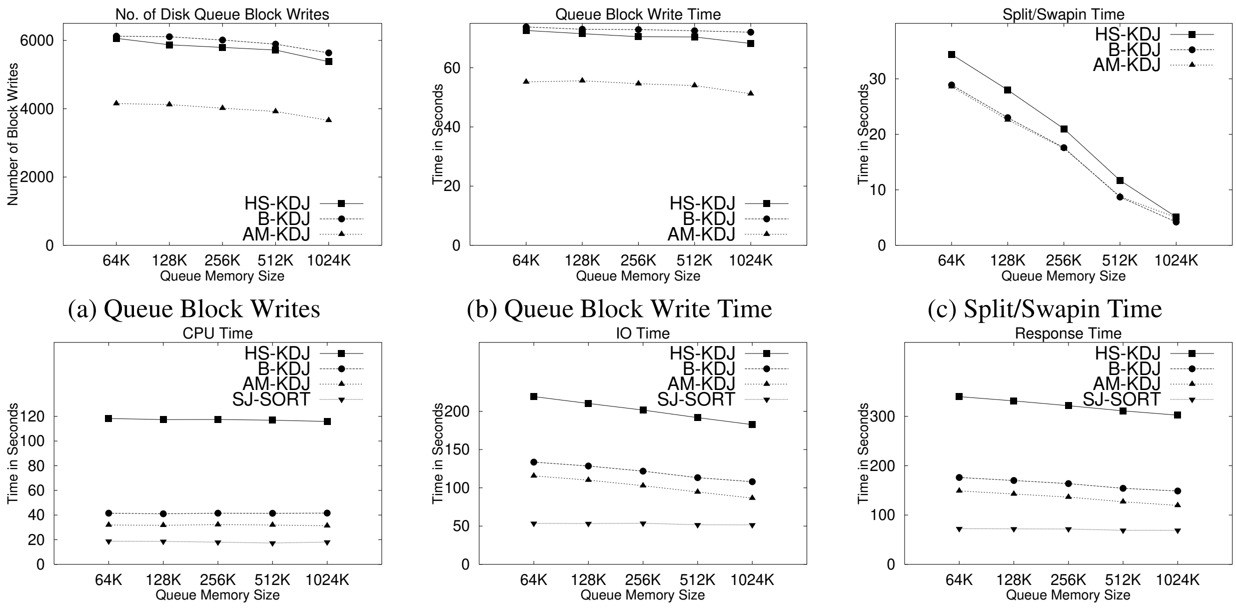
5.5 Impact of Memory Size

In this set of experiments, we examined the performance impact of memory constraint on queue management and R-tree access. The sizes of in-memory portion of a main queue and R-tree buffer were varied from 64 KBytes to 1024 KBytes. We measured the response time of HS-**KDJ**, $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** algorithms for a fixed stopping cardinality $k = 100,000$.

5.5.1 Buffer Size for Main Queue

No measurement for SJ-**SORT** algorithm appears in Figures 14(a) through 14(c), because SJ-**SORT** algorithm need not use the main queue for distance join processing. As we expected, in Figures 14(a) and 14(b), the cost of queue management decreased in terms of both the number of required write operations and time spent on the write operations. More noticeable improvement was observed in handling the overflow and underflow of the in-memory portion of queue, by *split* and *swap-in* operations respectively. (The *split* and *swap-in* operations are descibed in Section 4.4.) The time spent on the *split* and *swap-in* operations was improved substantially for all three algorithms in Figure 14(c).

It should be noted that the cost of queue management can be further reduced by not storing object pairs in the main queue, as proposed in the recent work by Corral *et al.* [11]. It is straightforward to modify the distance queue



(a) Queue Block Writes (b) Queue Block Write Time (c) Split/Swapin Time

(d) CPU Time (e) I/O Time (f) Response Time

Figure 14: Impact of Queue Buffer Size

to store k object pairs for $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** algorithms. The reason we did not use the optimization was that the optimization cannot be applied to incremental distance join queries and it was desired to evaluate the performance of **KDJ** and **IDJ** algorithms on the same basis.

While the CPU time remained almost unchanged in Figure 14(d), the I/O time was improved with more memory for all the algorithms shown in Figure 14(e). The improved response time was mainly attributed to the improved I/O time. The proposed $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** algorithms showed consistently better performance in queue management than HS-**KDJ** all over the examined range of memory size. This is because $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** algorithms reduced the number of required queue insertions and queue write operations.

5.5.2 Buffer Size for R-Tree

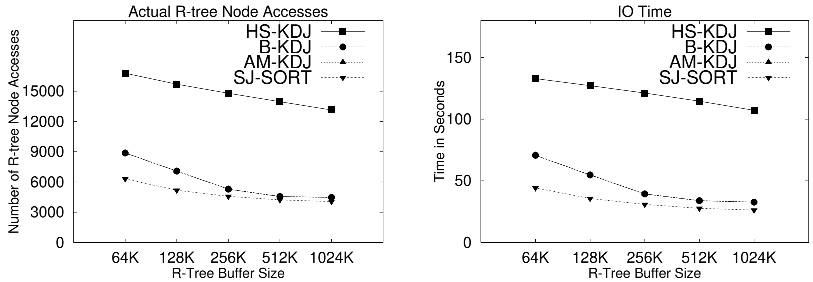
As shown in Figures 15(a) and 15(b), a considerable amount of improvement in R-tree accesses was observed by increasing the size of buffer for R-tree. For example, by increasing the buffer size from 64 KBytes to 1024 KBytes, the R-tree access time was reduced by 46 percent for $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** algorithms. Recall that $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ** algorithms, which are based on bi-directional node expansion, show the same behavior in R-tree access.

Like the queue management in the previous section, the CPU time spent on R-tree accesses remained almost unchanged, as shown in Figures 15(c). It was again the I/O time that affected the response time most in Figures 15(d) and 15(e).

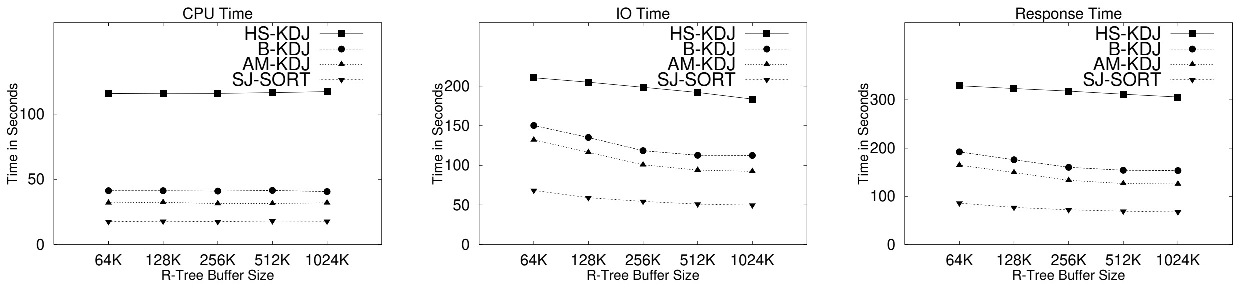
5.6 Impact of Duplicates and Zero-Distance Pairs

In real-world applications, spatial data sets often contain duplicates (*i.e.*, different objects with identical spatial extents or positions). These duplicates may cause query processing procedures to behave differently than normally expected. To evaluate the performance impact of duplicates for distance join processing, we carried out another set of experiments with slightly different data sets. Specifically, several thousands of data objects were added to the hydragraphic data set and the street data set, so that about 10,000 pairs of hydragraphic objects and streets are intersected (*i.e.*, within zero distance).

Figure 16 shows the performances of k-distance joins for the datasets with duplicates. For all the k values, the proposed algorithms, $\mathcal{B}$-**KDJ** and $\mathcal{AM}$-**KDJ**, outperformed HS-**KDJ**, and $\mathcal{AM}$-**KDJ** was better than $\mathcal{B}$-**KDJ** for large k values due to the slow-start problem of $\mathcal{B}$-**KDJ**. However, for small k values, the performance gap between

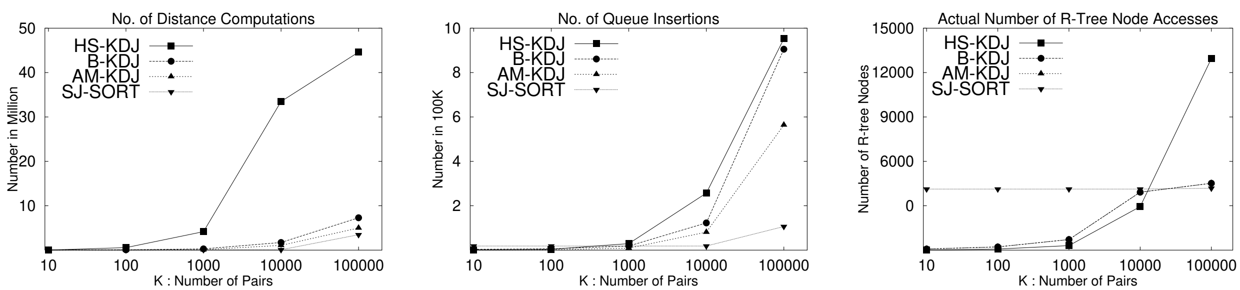


(a) R-Tree Node Accesses (b) R-Tree Node Access Time

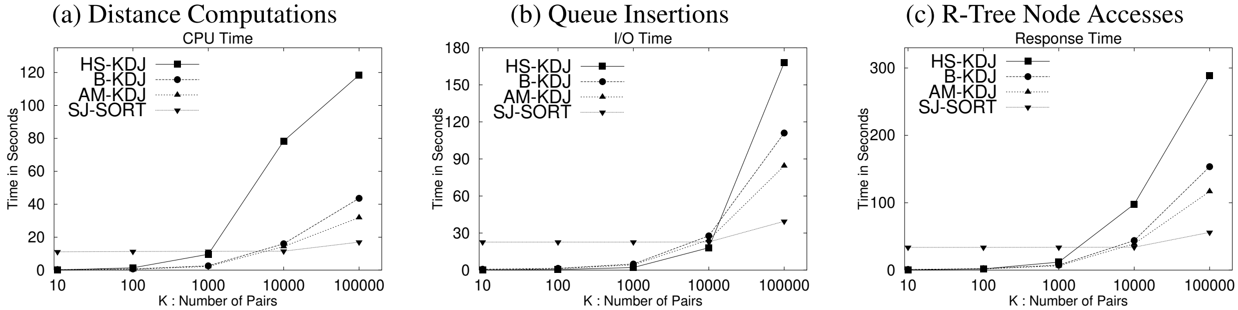


(c) CPU Time (d) I/O Time (e) Response Time

Figure 15: Impact of R-Tree Buffer Size



(a) Distance Computations (b) Queue Insertions (c) R-Tree Node Accesses



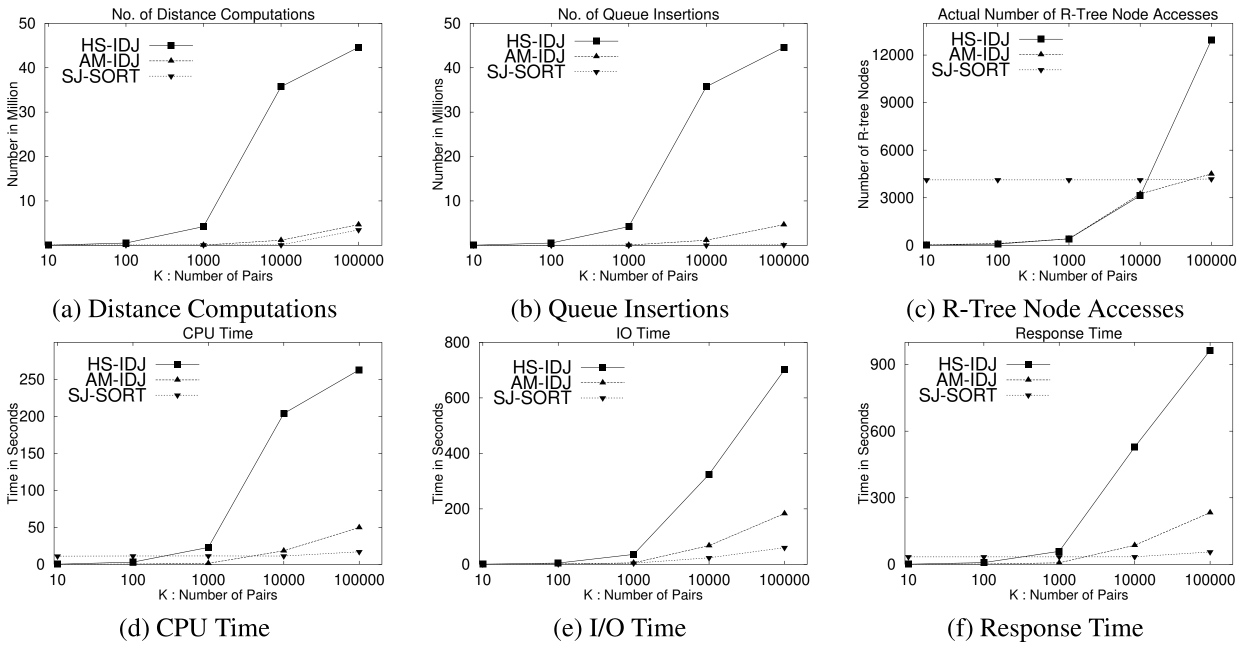
(d) CPU Time (e) I/O Time (f) Response Time

Figure 16: Performance of k-Distance Joins for Datasets with Duplicates

these algorithms was rather small compared with the case for data sets without duplicates. (See Figure 11.) This is because a large number of zero-distance pairs diminishes the distinctions among different **KDJ** algorithms.

Contrary to our conjecture, SJ-**SORT** was worse than all three k-distance join algorithms in response times for small k values ($k \leq 10,000$). This is again due to the fact that there were about 10,000 pairs of zero distance. No matter what distance cutoff was provided for the SJ-**SORT** algorithm, an exhaustive set of zero-distance pairs were returned as a distance join query results, which turned out a significant overhead for small k values.

For the incremental distance join algorithms HS-**IDJ**, $\mathcal{AM}$-**IDJ** and SJ-**SORT**, we observed the same trend in



(a) Distance Computations (b) Queue Insertions (c) R-Tree Node Accesses

(d) CPU Time (e) I/O Time (f) Response Time

Figure 17: Performance of Incremental Distance Joins for Datasets with Duplicates

the performance from the data sets with duplicates. (See Figure 17.) $\mathcal{AM}$**-IDJ** was always better than HS**-IDJ** for all k values, and $\mathcal{AM}$**-IDJ** yielded better response time than SJ**-SORT** in small k values($k \leq 1000$).

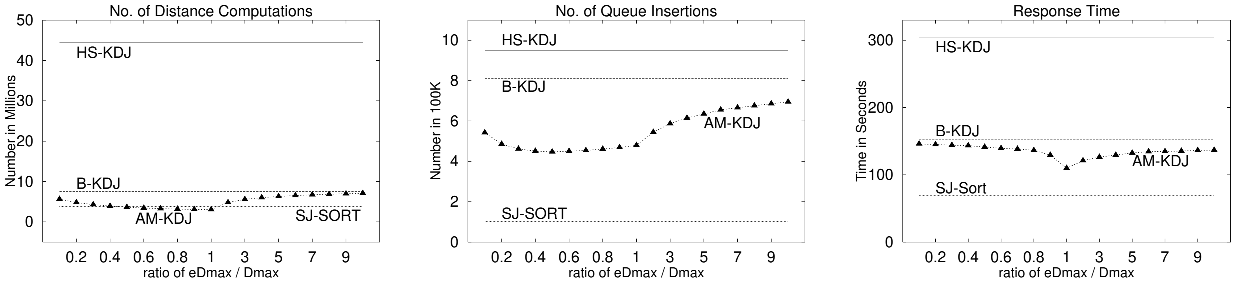
5.7 Impact of $e\mathcal{D}_{max}$ Estimation on $\mathcal{AM}$-KDJ Performance

We designed two sets of experiments to characterize the performance of $\mathcal{AM}$**-KDJ** algorithm with respect to the accuracy of estimated $e\mathcal{D}_{max}$ values. In Section 5.7.1, instead of using Equation (3) to estimate $e\mathcal{D}_{max}$, we varied the $e\mathcal{D}_{max}$ value from $0.1 \times \mathcal{D}_{max}$ to $10 \times \mathcal{D}_{max}$. Recall that $\mathcal{D}_{max}$ is a real distance between the k-th nearest pair of objects. In Section 5.7.2, we used Equation (3) and *power law* proposed in [13] to compute $e\mathcal{D}_{max}$values. Again, the sizes of in-memory portion of a main queue and R-tree buffer were fixed to 512 KBytes.

5.7.1 Robustness of $\mathcal{AM}$-KDJ

While fixing a stopping cardinality k to 100,000, we varied the $e\mathcal{D}_{max}$ value from $0.1 \times \mathcal{D}_{max}$ to $10 \times \mathcal{D}_{max}$. When $e\mathcal{D}_{max}$ is overestimated ($e\mathcal{D}_{max} > \mathcal{D}_{max}$), the compensation stage of $\mathcal{AM}$**-KDJ** algorithm is not necessary, because all the k nearest pairs will be produced in the first (aggressive pruning) stage. Even when $e\mathcal{D}_{max}$ is overestimated, $\mathcal{AM}$**-KDJ** guarantees that $e\mathcal{D}_{max}$ is always smaller than or equal to $q\mathcal{D}_{max}$ (obtained from a distance queue) throughout the first stage. Thus, $\mathcal{AM}$**-KDJ** always requires no more distance computation and queue insertion operations than $\mathcal{B}$**-KDJ** algorithm does.

On the other hand, if $e\mathcal{D}_{max}$ is underestimated ($e\mathcal{D}_{max} < \mathcal{D}_{max}$), the node pairs in the compensation queue will be revisited in the compensation stage. Thus, the cost of tree traversals will increase, but it will be bounded by twice the cost of $\mathcal{B}$**-KDJ** algorithm. Although there is no such a bound on the cost of queue management, we observed in most of our experiments that the cost of queue management was lower than that of $\mathcal{B}$**-KDJ** algorithm. This is because a large number of insertions to a compensation queue were prevented by aggresive pruning, and the compensation queue was several orders of magnitude smaller than the main queue. As discussed in Section 4.1, for a pair already expanded once in the first stage, only child pairs not examined in the first stage are paired up in the compensation stage and thereby wasting no time for redundant work. The value of $q\mathcal{D}_{max}$ is likely to have become quite close to a real $\mathcal{D}_{max}$ value in the compensation stage. So, $\mathcal{AM}$**-KDJ** algorithm usually *prunes distant pairs much more efficiently in the compensation stage* than $\mathcal{B}$**-KDJ** algorithm would do in a single stage. Therefore, $\mathcal{AM}$**-KDJ** outperforms the k-distance join algorithms HS**-KDJ** and $\mathcal{B}$**-KDJ**, despite the additional cost of compensation stage.



(a) Distance Computations (b) Queue Insertions (c) Response Time

Figure 18: Performance Impact of $e\mathcal{D}_{max}$

Figure 18 shows that as $e\mathcal{D}_{max}$ approaches to a real $\mathcal{D}_{max}$ value, the performance of $\mathcal{AM}$-**KDJ** improves consistently in all three metrics. When $e\mathcal{D}_{max}$ increases far beyond the real $\mathcal{D}_{max}$ value, the performance of $\mathcal{AM}$-**KDJ** converges to that of $\mathcal{B}$-**KDJ** algorithm. More importantly, however, $\mathcal{AM}$-**KDJ** always outperformed $\mathcal{B}$-**KDJ**, not to mention HS-**KDJ**, with $e\mathcal{D}_{max}$ in a wide spectrum of estimated value range.

We have not measured the cost of compensation queue management. A compensation queue contains pairs of non-object R-tree nodes. During the first (aggressive pruning) stage of $\mathcal{AM}$-**KDJ** algorithm, The number of pruned pairs is far larger than the number of non-object pairs inserted into a compensation queue. In most of our experiments, the size of a compensation queue was *less than 0.5 percent* of the size of a main queue. Thus, the additional cost required for the compensation queue was almost negligible. This is one of the reasons why $\mathcal{AM}$-**KDJ** algorithm always outperformed $\mathcal{B}$-**KDJ**, which does not need a compensation queue.

5.7.2 Uniformity Assumption and Power Law

Faloutsos *et al.* [13] proposed a power law to predict the selectivity of spatial join and to estimate the distance of the k-th closest pair. We used the *box-occupancy-product-sum* method as proposed in [13] to determine the values of coefficient (C) and slope (s) of the power law. Then, we used the following formula to estimate $e\mathcal{D}_{max}$ for different k values.

$$e\mathcal{D}_{max} = L \times \left(\frac{k}{C}\right)^{1/s} \tag{6}$$

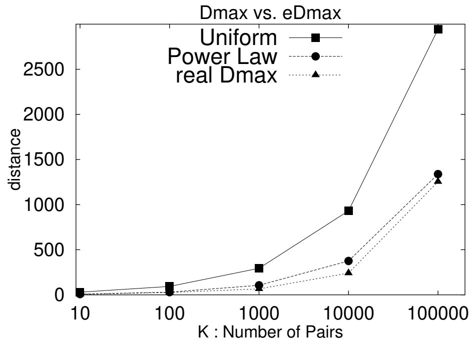
Here L is the maximum length of a data domain along x or y axis. For the data sets without duplicates, C was 3.85894×10^{11} and s was 1.812235. For the data sets with duplicates, C was 3.96152×10^{11} and s was 1.806431. L was 5766370 in both cases.

Figures 19(a) and 19(b) show real $\mathcal{D}_{max}$ and estimated $e\mathcal{D}_{max}$ values for k values varying from 10 to 100,000. $e\mathcal{D}_{max}$ values estimated by Equation 3 are labeled `Uniform`; those estimated by the power law are labeled `Power Law`. For the data sets without duplicates, the $e\mathcal{D}_{max}$ estimation by the power law was very accurate, while Equation 3 consistently overestimated. However, for data sets with duplicates, even the power law was not as accurate and it overestimated for small k values.

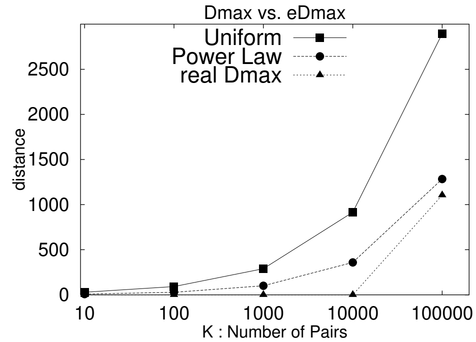
In the experiments, both Equation 3 (uniformity assumption) and Equation 6 (power law) overestimated $\mathcal{D}_{max}$ values. As we discussed in the previous section, the compensation stage of $\mathcal{AM}$-**KDJ** is not necessary when $e\mathcal{D}_{max}$ is overestimated. To demonstrate the performance impact of $e\mathcal{D}_{max}$ estimation, we measured response times of $\mathcal{AM}$-**KDJ** algorithm using real $\mathcal{D}_{max}$ values and estimated $e\mathcal{D}_{max}$ values in Figures 19(c) and 19(d). Evidently, the response times of $\mathcal{AM}$-**KDJ** were not so affected by $e\mathcal{D}_{max}$ estimation for both data sets with and without duplicates. This is another evidence that $\mathcal{AM}$-**KDJ** yields very stable performance under various circumstances, and Equation 3 based on uniformity assumption is a viable method to estimate $e\mathcal{D}_{max}$ values for real-world data sets.

5.8 Stepwise Incremental Execution of $\mathcal{AM}$-IDJ

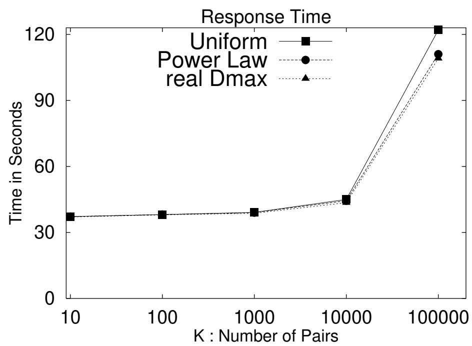
Incremental distance join algorithms do not require a preset stopping cardinality k. Thus, in this set of experiments, we simulated a situation where users repeatedly requested a set of 10,000 nearest pairs at a time until a total of 100,000



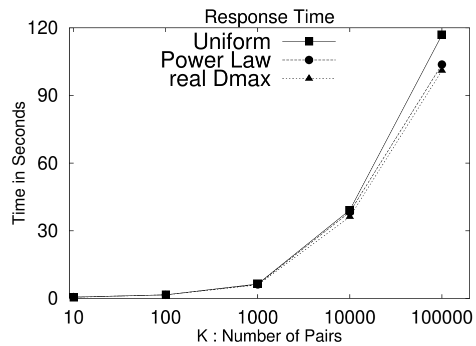
(a) $e\mathcal{D}_{max}$ for data sets without duplicates



(b) $e\mathcal{D}_{max}$ data sets with duplicates



(c) Response time for data sets without duplicates



(d) Response time for data sets with duplicates

Figure 19: $e\mathcal{D}_{max}$ estimation based on uniform distribution and Power Law

nearest pairs were generated. Incremental algorithms HS-**IDJ** and $\mathcal{AM}$-**IDJ** each were executed once in a single experiment run, until a total of 100,000 nearest pairs were generated. The sizes of in-memory portion of a main queue and R-tree buffer were fixed to 512 KBytes both for HS-**IDJ** and $\mathcal{AM}$-**IDJ**.

For SJ-**SORT**, which is not an incremental algorithm, we restarted its processing each time $i \times 10,000$ nearest pairs were generated for i $(1 \leq i \leq 9)$. Thus, the performance measurements of SJ-**SORT** presented in Figure 20 are cumulative. For example, the response time of SJ-**SORT** for $k = 20,000$ includes the times spent on executing SJ-**SORT** twice, once for $k = 10,000$ and another for $k = 20,000$. For each run of SJ-**SORT**, we used a real $\mathcal{D}_{max}$ value for each of different stopping cardinalities.

In Figure 20, we measured the response time of $\mathcal{AM}$-**IDJ** algorithm in two different ways: (i) with $e\mathcal{D}_{max}$ values estimated by Equation (3), and (ii) with real $\mathcal{D}_{max}$ values provided for 10 different k values. When estimated $e\mathcal{D}_{max}$ values were provided, $\mathcal{AM}$-**IDJ** needed compensation processing only after generating 30,000 pairs and 90,000 pairs, due to overestimated $e\mathcal{D}_{max}$ values. In the second case (denoted by `AM-IDJ (Dmax)` in Figure 20), a real $\mathcal{D}_{max}$ value was provided for each of k values from 10,000 through 100,000, to simulate a situation where the next set of 10,000 pairs of objects were repeatedly requested by a user. Consequently, $\mathcal{AM}$-**IDJ** was forced to initiate a compensation stage, each time the next set was requested. This overhead slowed down the processing due mainly to redundant R-tree node accesses. Overall, $\mathcal{AM}$-**IDJ** showed a fairly consistent performance over varying $e\mathcal{D}_{max}$ estimates, as $\mathcal{AM}$-**KDJ** did in Section 5.7. For all the k values, $\mathcal{AM}$-**IDJ** with estimated $e\mathcal{D}_{max}$ improved the response time by a factor of two to four, when compared with HS-**IDJ**.

6 Conclusions

We have developed new distance join algorithms for spatial databases. The proposed algorithms provide significant performance improvement over previous work. The plane-sweep technique optimized by novel strategies for selecting a sweeping axis and direction minimizes the computational overhead incurred by bi-directional node expansions. The

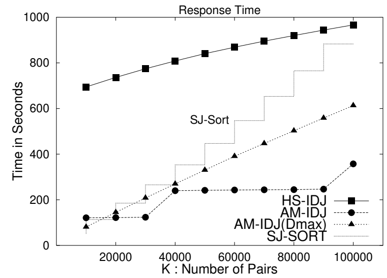


Figure 20: Step-Wise Incremental Execution

node expansions are further optimized by using maximum distance for breaking tied pairs. We have shown that this optimized plane-sweep technique alone improves processing of a k-distance join query considerably.

The adaptive multi-stage algorithms employ aggressive pruning and compensation methods to further optimize the distance join processing. These algorithms address a slow start problem by using estimated maximum distances as cutoff values for pruning distant pairs. Our experimental study shows that the proposed algorithms outperformed previous work significantly and consistently for all the stopping cardinalities over a wide spectrum of estimated maximum distances. Ample evidence was observed that the adaptive algorithm yielded significant improvement in query processing time regardless of the techniques used for maximum distance estimations. For a relatively small stopping cardinality, the proposed algorithms achieved up to an order of magnitude improvement over previous work. Assuming data objects are uniformly distributed, we have developed strategies to choose an initial estimate and to correct the estimate adaptively during the query processing.

When the stopping cardinality of a distance join query is unknown (as in on-line query processing environments or a complex query that contains a distance join as a sub-query), the adaptive multi-stage algorithms process the query in a stepwise manner so that the query results can be returned incrementally.

References

[1] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the 24th VLDB Conference*, pages 259–270, New York, USA, June 1998.

[2] Sunil Arya, David M. Mount, and Onuttom Narayan. Accounting for boundary effects in nearest neighbor searching. In *Proc. 11th Annual Symp. on Computational Geometry*, pages 336–344, Vancouver, Canada, 1995.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM-SIGMOD Conference*, pages 322–331, Atlantic City, NJ, May 1990.

[4] Alberto Belussi and Christos Faloutsos. Estimating the selectivity of spatial queries using the correlation fractal dimension. In *Proceedings of the 21st VLDB Conference*, pages 299–310, Zurich, Switzerland, September 1995.

[5] Stefan Berchtold, Bernhard Ertl, Daniel Keim, Hans-Peter Kriegel, and T. Seidl. Fast nearest neighbor search in high-dimensional spaces. In *Proceedings of the 14th International Conference on Data Engineering*, Orlando, Florida, September 1998.

[6] Stefan Berchtold, Daniel A. Keim, and Hans-Peter. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd VLDB Conference*, Bombay, India, September 1996.

[7] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-step processing of spatial joins. In *Proceedings of the 1994 ACM-SIGMOD Conference*, pages 197–208, Minneapolis, Minnesota, May 1994.

[8] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient processing of spatial joins using R-Trees. In *Proceedings of the 1993 ACM-SIGMOD Conference*, pages 237–246, Washington, DC, May 1993.

[9] Michael J. Carey and Donald Kossmann. On saying "enough already!" in SQL. In *Proceedings of the 1997 ACM-SIGMOD Conference*, pages 219–230, Tucson, AZ, May 1997.

[10] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *Proceedings of the 24th VLDB Conference*, pages 158–169, New York, NY, August 1998.

[11] Antonio Corral, Yannis Manolopoulos, Yannis Theodoridis, and Michael Vassilakopoulos. Closest pair queries in spatial databases. In *Proceedings of the 2000 ACM-SIGMOD Conference*, pages 189–200, Dallas, TX, May 2000.

[12] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top N queries. In *Proceedings of the 25th VLDB Conference*, Edinburgh, Scotland, September 1999.

[13] Christos Faloutsos, Bernhard Seeger, Agma Traina, and Caetano Traina Jr. Spatial join selectivity using power laws. In *Proceedings of the 2000 ACM-SIGMOD Conference*, pages 177–188, Dallas, TX, May 2000.

[14] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, Boston, MA, June 1984.

[15] Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In *Proc. of 4th Intl. Symposium on Large Spatial Databases(SSD'95)*, pages 83–95, September 1995.

[16] Gisli R. Hjaltason and Hanan Samet. Incremental distance join algorithms for spatial databases. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 237–248, Seattle, WA, June 1998.

[17] Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, Eliot Siegel, and Zenon Protopapas. Fast nearest neighbor search in medical image databases. In *Proceedings of the 22nd VLDB Conference*, pages 215–226, June 1996.

[18] Ming-Ling Lo and Chinya V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM-SIGMOD Conference*, pages 209–220, Minneapolis, Minnesota, May 1994.

[19] Ming-Ling Lo and Chinya V. Ravishankar. Spatial hash-join. In *Proceedings of the 1996 ACM-SIGMOD Conference*, pages 247–258, Montreal, Canada, June 1996.

[20] Bureau of the Census. *Tiger/Line Precensus Files: 1997 technical documentation*. Washington, DC, 1997.

[21] Jack A. Orenstein. A comparison of spatial query processing techniques for native and parameter spaces. In *Proceedings of the 1990 ACM-SIGMOD Conference*, pages 343–352, Atlantic City, New Jersey, May 1990.

[22] Dimitris Papadias, Nikos Mamoulis, and Yannis Theodoridis. Processing and optimization of multiway spatial joins using r-trees. In *Proceedings of the 1999 ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 44–55, June 1999.

[23] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. In *Proceedings of the 1996 ACM-SIGMOD Conference*, pages 259–270, Montreal, Canada, June 1996.

[24] Viswanath Poosala. *Histogram-based Estimation Techniques in Databases*. PhD thesis, University of Wisconsin-Madison, 1997.

[25] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry: An Introdution*. Springer-Verlag, New York, NY, 1985.

[26] V. Ramasubramanian and K. K. Paliwal. Fast k-dimensional tree algorithms for nearest neighbor search with application to vector quantization encoding. *IEEE Trans. on Signal Processing*, 40(3):518–531, March 1992.

[27] Nick Roussopoulos, Stephen Kelley, and Frederic Vincent. Nearest neighbor queries. In *Proceedings of the 1995 ACM-SIGMOD Conference*, pages 71–79, San Jose, CA, May 1995.

[28] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 154–165, Seattle, Washington, June 1998.

[29] Jeffrey S. Vitter and Min Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proceedings of the 1999 ACM-SIGMOD Conference*, pages 193–204, June 1999.