Top-*k* Spatial Joins

Manli Zhu, Dimitris Papadias, Jun Zhang, and Dik Lun Lee

Abstract—Given two spatial data sets A and B, a top-*k* spatial join retrieves the *k* objects from A or B that intersect the largest number of objects from the other data set. Depending on the application requirements, there exist several variations of the problem. For instance, B may be a point data set, and the goal may be to retrieve the regions of A that contain the maximum number of points. The processing of such queries with conventional spatial join algorithms is expensive. However, several improvements are possible based on the fact that we only require a small subset of the result (instead of all intersection/containments pairs). In this paper, we propose output-sensitive algorithms for top-*k* spatial joins that utilize a variety of optimizations for reducing the overhead.

Index Terms—Database, spatial database, spatial joins.

1 Introduction

A spatial join retrieves the pairs of objects that satisfy some spatial predicate (most often intersection). Joins have been studied quite extensively in the spatial database literature because they are involved in several common queries (e.g., map overlays) and they incur high execution cost, which calls for performance optimizations. Consequently, various algorithms have been proposed (see Section 2), covering all possible combinations of indexed and nonindexed data sets. In most cases, the result of a spatial join is rather large (for typical geographic layers, linear to the cardinality of the participating data sets). In several applications, however, the user may be interested in only a small subset of the output. This paper studies such a query type, called the *top-k spatial join* (top-*k* SJ).

Given two data sets A and B, the top-*k* SJ retrieves the k objects in data set A or B that intersect the maximum number of objects from the other data set. For example, in VLSI design, a circuit layer consists of numerous wires, represented as rectangles. When two layers are placed together, the intersection between the wires of different layers may cause electro-magnetic interference. A top-*k* spatial join can be applied to retrieve the wires intersecting the largest number of wires from the other layer. The result indicates the positions where the topology of the circuit can be improved to reduce interference.

Similarly, the *top-k spatial semijoin* returns the k objects of A with the maximum *intersection or containment counts*. For example, consider a traffic supervision system, where data set A represents urban regions and B stands for vehicle locations monitored through sensor networks [7]. In order to balance the traffic, we should have knowledge about the regions with the maximum number of cars. Similar queries are relevant for systems monitoring mobile devices (e.g., find the area with the highest user density) and other spatial decision making applications.

Top-*k* spatial joins have some similarity with multidimensional *range aggregate queries*. Given a set A and a window q in a multidimensional vector space, a range aggregate (RA) query returns a single value that summarizes the subset $R \subseteq A$ of objects intersecting (or inside) q according to an aggregation function (e.g., the number of objects qualifying q instead of their concrete ids). Several techniques [8], [14], [24], [35] have been proposed or the efficient processing of RA queries in spatial databases (see [34] for an overview of related work). However, the application of these techniques to top-*k* spatial (semi) joins would incur large computational overhead because the number of RA queries increases linearly with the cardinalities $|A|$ and $|B|$ of the participating data sets. In particular, the processing of a top-*k* semijoin requires the application of $|A|$ RA queries (one for each object of A), while a full join requires $|A| + |B|$ RA queries.

Top-*k* joins in relational databases retrieve the k tuples of the join result with the highest value of a scoring function. However, existing relational methods (e.g., [13], [22]) are inapplicable to top-*k* SJ for the same reasons that equi-join algorithms are inapplicable to spatial joins (a lack of one-dimensional ordering of tuples, intersection as opposed to equality join condition, etc.). A naïve approach to evaluate top-*k* SJ is to: 1) apply a conventional spatial join algorithm on the two data sets A and B, 2) count the number of output pairs in which each object participates, and 3) return the k objects with the maximum intersection counts. However, this method may cause a significant amount of redundant work, especially if the value of k is small.

Assuming that, in practice, the number k of requested objects is very small compared to the entire spatial join result, we propose a set of output-sensitive algorithms that apply the branch-and-bound framework for pruning the search space. Following most of the related work in the spatial database literature, we consider that the spatial data sets are indexed by R-trees [9] (or their variations [2], [31]), but the general methodology is applicable to any data-partition method (e.g., X-trees [3]). The remainder of the

- *M. Zhu, D. Papadias, and D.L. Lee are with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong. E-mail: {cszhuml, dimitris, dlee}@cs.ust.hk.*
- *J. Zhang is with the Division of Information Systems, Computer Engineering School, Nanyang Technological University, Singapore. E-mail: jzhang@ntu.edu.sg.*

Manuscript received 4 Dec. 2003; revised 9 June 2004; accepted 27 Aug. 2004; published online 17 Feb. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0249-1203.

paper is structured as follows: Section 2 reviews the related work, focusing mostly on spatial joins. Section 3 proposes the basic top-k SJ algorithm and discusses its properties. Section 4 describes several optimizations for reducing the I/O overhead, and studies variations of the problem. Section 5 contains an experimental evaluation of the proposed techniques and Section 6 concludes the paper.

2 Related Work

Early spatial join methods apply transformation of objects in order to overcome the difficulties raised by their spatial extent and dimensionality. The first known algorithm [23] uses a grid to regularly divide the multidimensional space into cells, which are sorted by z-ordering (or any other space-filling curve [21]). Each object is then approximated by the set of cells intersected by its minimum bounding rectangle (MBR), i.e., a set of z-values. Since z-values are 1-dimensional, the objects can be dynamically indexed using relational index structures, like the B+-tree, and the spatial join is performed in a sort-merge join fashion. Rotem [27] proposes an algorithm based on a spatial join index, which partially precomputes the join result and employs grid files to index the objects in space.

The most influential algorithm is the R-tree join (RJ) [5] due to its efficiency and the popularity of R-trees. RJ is based on the *enclosure property* of R-tree nodes: If two intermediate nodes do not intersect, there can be no MBRs below them that intersect. Following this observation, RJ starts from the roots of the trees to be joined and finds pairs of overlapping entries. For each such pair, the algorithm is recursively called, until the leaf levels, where overlapping pairs constitute solutions. Two optimization techniques can be used to improve the CPU speed of RJ [5]. The first, *search space restriction*, reduces the quadratic number of pairs to be evaluated when two nodes n_i, n_j are joined. If an entry $e_i \in n_i$ does not intersect the MBR of n_j, then there can be no entry $e_j \in n_j$, such that e_i and e_j overlap. Using this fact, space restriction performs two linear scans in the entries of both nodes, and prunes out from each node the entries that do not intersect the MBR of the other node. The second technique, based on the *plane sweep* paradigm, applies sorting on one dimension in order to reduce the cost of computing overlapping pairs between the nodes to be joined. Huang et al. [12] propose a breadth-first optimized version of RJ that sorts the output at each level in order to reduce the number of page accesses.

Research after RJ focused mostly on spatial join processing when one or both inputs are nonindexed. Nonindexed inputs are usually intermediate results of a preceding operator. The simplest method to process a pairwise join in the presence of one index, is by applying a window query to the existing R-tree for each object in the nonindexed data set (*index nested loops*). Due to its computational burden, this method is used only when the joined data sets are relatively small. Another approach is to build an R-tree for the nonindexed input using bulk loading [26] and then employ RJ to match the trees. Lo and Ravishankhar [16] use the existing R-tree as a skeleton to build a *seeded tree* for the nonindexed input. The *sort and match* algorithm [25] spatially sorts the nonindexed objects but, instead of building the packed tree, it matches each in-memory created leaf node with the leaf nodes of the existing tree that intersect it. Finally, the *slot index spatial join* (SISJ) [20] applies hash-join, using the structure of the existing R-tree to determine the extents of the spatial partitions.

If no indexes exist, both inputs have to be preprocessed in order to facilitate join processing. Arge et al. [1] propose an algorithm, called *scalable sweeping-based spatial join*, that employs a combination of *plane sweep* and *space partitioning* to join the data sets. Patel and DeWitt [26] describe a hash-join algorithm, *partition-based spatial merge join* (PBSM), that regularly partitions the space, using a rectangular grid, and hashes both inputs into the partitions. It then joins groups of partitions that cover the same area using plane sweep to produce the join results. Some objects from both sets may be assigned in multiple partitions, so the algorithm needs to sort the results in order to remove the duplicate pairs. *Size separation spatial join* [15] is also based on regular space decomposition, but avoids replication of objects during the partitioning phase by introducing more than one partition layers. Spatial *hash-join* [17] avoids duplicate results by performing an irregular decomposition of space, based on the data distribution of the *build* input.

In addition to "conventional" intersection joins, there have been numerous papers on parallel algorithms [18], high-dimensional similarity (distance) joins [4], and multi-way spatial joins involving multiple inputs [19]. In a recent paper, Shou et al. [33] describe several methods for processing *iceberg spatial joins*, which also include a cardinality constraint (e.g., find all regions of A that intersect at least 10 regions of B). In particular, 1) when both A and B are indexed, they apply an optimized version of RJ, 2) when only one data set is indexed, an optimized version of SISJ, and 3) when neither data set is indexed, an optimized version of PBSM. In all cases, the cardinality constraint is utilized to avoid visiting nonqualifying nodes.[1]

Another related problem refers to *closest-pair queries*, which retrieve the k pairs of objects from data sets A and B with the minimum distance. Similar to RJ, algorithms for closest pair queries [6], [10], [32] traverse synchronously the R-trees of the participating data sets, following pairs of entries that can lead to a result (i.e., entries whose minimum distance is smaller than that of the already discovered closest pairs).

The problem discussed in this paper is related to a certain extent with all the above topics. First, we aim at processing spatial joins, taking, however, into account the number of intersections for each object (similar to iceberg joins). Furthermore, like closest pair queries, the goal is to return the top-k results (and not all objects or pairs that satisfy the cardinality constraint as in the case of iceberg joins). In the next section, we describe the general algorithm for processing top-k spatial joins.

3 Top-k Spatial Join Algorithm

We first present the top-k SJ (TS) algorithm in Section 3.1, followed by a discussion in Section 3.2. For our examples,

1. Shou et al. [33] actually deal with distance joins, but the proposed techniques can also process intersection joins with minor modifications.

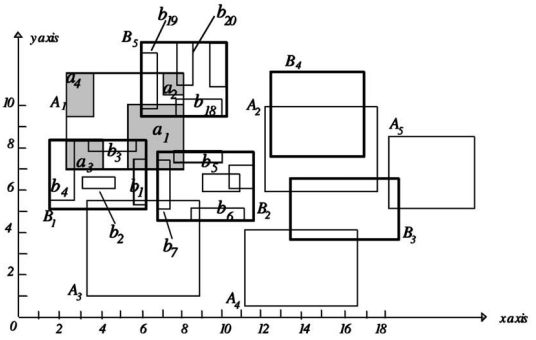


Fig. 1. Example data sets and corresponding R-trees.

we use the two region data sets of Fig. 1 and the corresponding R-trees, assuming a capacity of five entries per node (some objects inside the nodes are omitted for clarity). Both R-trees have height 2 and the leaf nodes are at level 0. The result of a top-1 join in this example is $\{a_1, 6, [b_1\ b_3\ b_5\ b_7\ b_{18}\ b_{19}]\}$ since a_1 intersects these six objects of B.

In the general case, the output of TS is a list with k entries of the form $\langle id, count, IL\rangle$, where *id* is an object ID, *IL* is the list of its intersecting objects (from the other data set), and *count* is the cardinality of *IL*. The result is reported in descending order of *count*. For each two objects i, j, such that i belongs to the result and j does not: $count(j) \leq count(i)$. Some frequently used symbols are described in Table 1.

3.1 Description of the Algorithm

TS is based on the branch-and-bound paradigm. For the effective application of TS (or any branch-and-bound algorithm), we need to define two measures: 1) a *pruning condition*, which excludes subtrees that cannot lead to better results (than the ones already found), and 2) a *key* that determines the order by which the qualifying subtrees are visited so that good solutions are identified as early as possible (increasing the effectiveness of the pruning condition). In some cases, the same metric can be applied for both tasks, e.g., for nearest-neighbor search, the *mindist*[2] of a node to a query point is commonly used to determine the visiting order and to prune nodes (whose *mindist* is larger than the distance of the best neighbor already found). We first provide some definitions, which will be later utilized for determining the visiting order and pruning condition of TS.

Definition 1. *Let e be an intermediate entry of R_a, C the node capacity, and $e.level$ the level of the node that contains e. Then, the upper bound $maxnum(e)$ for the number of objects in the subtree of e is:*

$$maxnum(e) = C^{e.level}.$$

Definition 2. *If e is a leaf entry (i.e., an object) of R_a, we define $count(e)$ as the number of objects of R_b that intersect e. If e is an intermediate entry, $count(e)$ is an upper bound of the actual count of any object in e, which is obtained as:*

$$count(e) = \sum_{e_i \in R_b \text{ and } e_i \text{ intersects } e} maxnum(e_i).$$

In Fig. 1, since A_1 is a level 1 entry, $maxnum(A_1) = 5$, i.e., the maximum number of objects (of R_a) inside A_1 is 5. In order to obtain a value for $count(A_1)$, note that A_1 overlaps three entries B_1, B_2, B_5 of R_b, whose *maxnum* is also 5. Therefore, assuming that there is an object in A_1 intersecting all objects in B_1, B_2, and B_5, $count(A_1) = 3 \cdot 5 = 15$. Although this upper bound seems loose (especially for the high tree levels), as discussed in the Section 3.2, it is the tightest upper bound that can be obtained without any assumptions about the data distribution and object extent. *Count* provides a termination condition (i.e., we do not need to visit entries whose *count* is smaller than that of the top-*k* results already found) and a visiting order (i.e., we first visit entries with high *counts* based on the intuition that they are likely to lead to good results).

Now, we describe the algorithm considering, for simplicity, that the two R-trees have the same height (different heights are discussed in Section 4). First, the roots of the two R-trees are loaded in memory, and TS finds the intersecting entry pairs. Then, it computes, for each root entry e that appears in a pair, its *intersecting list* (*IL*), which contains the entries of the other tree intersecting e: $A_1.IL = [B_1\ B_2\ B_5]$, $B_1.IL = [A_2\ A_3]$, $A_2.IL = [B_3\ B_4]$, $B_2.IL = [A_1\ A_3]$, and so on. TS inserts the root entries, their counts, and their intersecting lists into a heap H, sorted by the counts in descending order. Specifically, each heap element has the form $E : \langle e, count, list\rangle$, where 1) e is the entry (of R_a or R_b), 2) $count(e) = \sum_{e_i \in e.IL} maxnum(e_i)$, and 3) *list* is $e.IL$. Fig. 2 shows the content of H after visiting the root.

Next, TS deheaps the first entry (A_1) of H. The algorithm retrieves, besides node A_1, the nodes pointed by the entries in its IL, i.e., B_1, B_2, B_5 and joins[3] the entries of A_1 with those of B_1, B_2, B_5. The intersection pairs (a_1, b_1), (a_1, b_3), (a_1, b_5), (a_1, b_7), (a_1, b_{18}), (a_1, b_{19}), (a_2, b_{20}), (a_3, b_3), (a_3, b_4) are computed and the *ILs* for the entries (a_1, a_2, a_3) of node A_1 are built: $a_1.IL = [b_{19}\ b_{18}\ b_1\ b_3\ b_7\ b_5]$, $a_2.IL = [b_{20}]$, and $a_3.IL = [b_3\ b_4]$. The entries are then inserted to H, according to their key value ($a_1.key = 6$, $a_3.key = 2$, $a_2.key = 1$), which now corresponds to the actual number of intersections (instead of an upper bound). Fig. 3 shows the contents of H at this point.

2. $Mindist(n, q)$ between node n and q is the minimum distance between any point in the MBR of n and q [28].

3. We apply the *space restriction* and *plane sweep* optimizations introduced in [5].

TABLE 1
Frequently Used Symbols

Symbol	Description
R_a (R_b)	R-tree for dataset A (B)
o_a (o_b)	object from dataset A (B)
RT_a (RT_b)	root node for R_a (R_b)
n	R-tree node
e	R-tree entry (pointing to n)
$e.level$	level of node that contains e

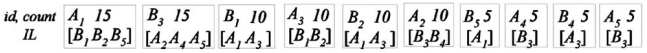


Fig. 2. Contents of heap after visiting the roots.

Note that TS only builds the *intersecting list*s for entries of node A_1 but not for nodes B_1, B_2, and B_5. There are two reasons for this: 1) the deheaped entry is A_1, meaning that we are looking for potential results in node A_1; 2) B_1, B_2, B_5 may also intersect other nodes from R_a. Potential candidates in node B_1 (or B_2, B_5) will be examined when B_1 is deheaped (B_1 is in H), in which case it will be visited again. In Section 4, we explore alternative strategies that expand several nodes simultaneously.

If we only want the top-1 result, a_1 becomes the first candidate and all heap entries with keys smaller than or equal to 6 ($a_1.key$) can be removed since they will not lead to an object with a higher count of intersections (similarly, a_2, a_3 should not be inserted at all in H). Object a_1 is not reported immediately because better objects may exist in nodes preceding it in H. Instead, it is reported when it is deheaped (provided that it has not been pruned before by a better object). Fig. 4 presents the pseudocode of TS.

First, the algorithm joins the roots, computes the intersection lists of their entries, and inserts each entry with its IL and *count* into the heap. When an entry e is deheaped, if it corresponds to an object, it is reported (the algorithm terminates when k objects have been reported). Otherwise, if e points to a node n, n is joined with all nodes pointed by the entries in $e.IL$, and the count of each entry $e' \in n$ is computed. If e' can lead to a better solution (i.e., $e'.count$ exceeds the pruning condition), it is inserted into H. In the case that e' is an actual object (e.g., a_1 in the previous example) the pruning condition is set to the value of $e'.count$. If we want to reduce the heap size requirements, we can remove all entries after e' in H, but this does not affect the performance of TS because the algorithm will terminate before such entries are considered anyway.

TS can be applied with minor modifications for incremental processing of top-k spatial joins, where the value of k is not set in advance. The difference with conventional processing is that TS now does not use any pruning condition because all objects may have to be reported. In such cases, the heap may exceed the size of the available main memory, leading to buffer thrashing. Hjaltason and Samet [11] propose a heap management technique to alleviate the problem, which is also applicable to TS. In particular, the content of the heap is organized in a B-tree (where the search key is the *count*). The part of the tree that contains heap elements with small *count* values (which will get deheaped early) is kept in main memory and the rest in the disk.

```
TS (Rtree Ra, Rtree Rb, int k)
1. join RTa and RTb to get intersecting pairs (ea, eb)
2. for each entry e that appears in a pair
3.    build e.IL, compute e.count and insert <e, e.count, e.IL> to a heap H (sorted by e.count)
4. while number of reported objects < k
5.    e = de-heap(H)
6.    if e is a leaf entry // actual object
7.       report (<e.id, e.count, e.IL>)
8.    else // e is an intermediate entry pointing to node n
9.       for each ei∈e.IL
10.         join n and ni // ni is pointed by ei
11.         for each intersecting entry pair (e', e'i) // e'∈n, e'i∈ ni
12.            add e'i to e'.IL
13.      compute e'.count
14.      if e'.count > pruning condition // i.e., count of the k-th best object found so far
15.         insert <e', e'.count, e'.IL> to H
16.         if e' is a leaf entry // object
17.            update pruning condition
18.return
```

Fig. 4. Pseudocode of TS.

3.2 Discussion about Search Order and Pruning

In the previous section, we used the estimated *count* of nodes for determining both the visiting order and the pruning condition. Although *count* in most cases provides only a rough estimation, it turns out that it is the tightest pruning condition that we can achieve given no additional knowledge about the data distribution or extent. In order to comprehend this, consider Fig. 5 where the root entry $e \in RT_a$ intersects all root entries e_1 to $e_5 \in RT_b$, i.e., $count(e) = 25$. Object o_a (shaded) in the subtree of e also has $count(o_a) = 25$. Similar examples can be constructed for any situation when there is no constraint on the object size. Thus, TS has to use *count* in order to avoid false misses. Furthermore, it has been shown in [11] that the best visiting order when using a heap is also according to the pruning condition. Thus, the performance of TS, in terms of node accesses, cannot be improved by simply changing the sorting key in the heap (e.g., using the intersection area of an entry instead of its *count*).

On the other hand, a number of indexes proposed for the efficient processing of range aggregate queries can be utilized to enhance performance of top-k joins. For instance, the aggregate R-tree (aR-tree) [14], [24] stores, for each intermediate entry, the number of objects in its subtree. Having this knowledge, we can replace the upper bound for *maxnum* in the definition of *count* with the actual number, which leads to higher pruning (the algorithm remains the same). In Section 5, we experimentally investigate the performance gains of this approach.

In the next section, we explore several heuristics assuming that the data sets are indexed by conventional

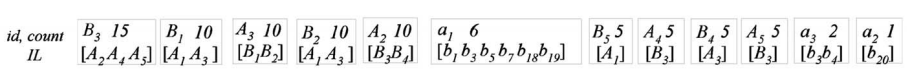


Fig. 3. Contents of heap after deheaping A_1 (A_1, B_1, B_2, B_5 visited).

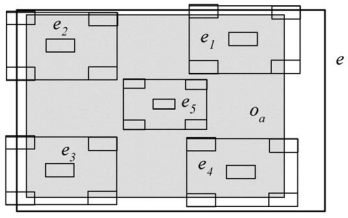


Fig. 5. *Count* gives the actual number of intersections.

R-trees due to their higher generality (compared to aggregate structures). In particular, we first exploit the fact that there are multiple visits to the same node at different phases of the join process, which can be combined in a single access. Furthermore, the existence of a buffer may significantly influence the effectiveness of algorithms that are based on synchronous traversal of two trees; for instance, the evaluation of [6] suggests that the depth-first traversal (despite the larger number of node accesses) is much more efficient than the best-first implementation of closest-pair queries, even if a small buffer is used. Motivated by this observation, we also present alternative visiting order heuristics for reducing the page faults of TS in the presence of an LRU buffer.

4 Optimizations

TS may visit the same node n multiple times: 1) when the entry e (pointing to n) is deheaped; 2) when each entry e', such that $e \in e'.IL$, is deheaped. In the running example, for instance, the node of root entry A_1 will be accessed when B_1, B_2, and B_5 are deheaped (unless the algorithm terminates before). In Sections 4.1, 4.2, and 4.3, we propose optimizations that organize these visits so that the number of node accesses and/or page faults is minimized. Section 4.4 discusses *top-k spatial semijoins*.

4.1 Multiple Expansions Method

The basic idea of the *multiple expansions* (ME) method is, whenever we expand (i.e., visit) an entry, to also update the information about all entries in its intersection list. Consider, for instance, the status of the heap after the processing of root entries shown in the first row of Fig. 6. When A_1 is deheaped, we use its content to update B_1, B_2, and B_5 as follows: In node A_1, the only entries that intersect B_1 are a_1 and a_3; thus, A_1 in $B_1.IL$ is replaced by a_1 and a_3. Accordingly, the contribution of A_1 to the count of B_1 changes from 5 to 2. Similarly, for B_2 : $B_2.IL = [A_3\ a_1]$ and $count(B_2) = 6$, while for $B_5 : B_5.IL = [a_1\ a_2]$ and $count(B_5) = 2$. The second row of Fig. 6 shows the new heap content.

Note that the updated entries B_1, B_2, B_5 have now smaller *counts*, which implies that they may be pruned earlier (i.e., before the expansion of the corresponding nodes). Furthermore, the number of node accesses can be further improved. Recall from Section 3.1 that, in addition to the node of A_1, when A_1 is deheaped, TS also retrieves the nodes of B_1, B_2, and B_5 to compute the actual counts of objects in A_1. Thus, we can use the contents of B_1, B_2, and B_5, to update the other entries in H that intersect them, i.e., all entries in the intersection lists of B_1, B_2, and B_5. For instance, since $B_1.IL = [A_3\ a_1\ a_3]$, B_1 also intersects A_3. In node B_1, the only entries overlapping A_3 are b_7 and b_6. Accordingly, the contribution of B_1 to the count of A_3 changes from 5 to 2. Similarly, $B_2.IL = [A_3\ a_1]$, and we further update A_3. B_5 does not cause any changes because $B_5.IL$ contains only the actual objects a_1 and a_2. The heap after these steps is shown in the third row of Fig. 6 (note the different position and content of the element corresponding to A_3). The above operations update the IL of an entry in H without expanding it, which would cause a significant enlargement of the heap.

A point that needs to be clarified concerns the efficient retrieval of the entries in the intersection list of the expanded node since the heap is organized by the count, whereas we want to retrieve entries according to their IDs. In order to avoid sequential scanning of the heap (which is potentially large), we maintain in memory two binary search trees (BST), each responsible for the entries of one R-tree. Fig. 7 shows the corresponding BST, together with the content of the heap, after the root entries have been traversed (i.e., before visiting A_1 in Fig. 6).

The entries in H are denoted by (*entry.id, treeno*). For example, A_1 is represented as (1,0) meaning that its *entry.id* is 1 and it belongs to R_a; accordingly, B_1 is (1,1). Each BST is built on the object ID (i.e., the IDs of entries in the left subbranch of a BST node are smaller than the ID of the node). The numbers in braces indicate the position of the corresponding element in H. For example, (1,0) (A_1) is the first element. In general, the position of any entry can be found with cost that is logarithmic to the number of elements in the heap.

The maintenance of BST requires insertion, deletion, and update operations. When an element is inserted or removed from H, the pair (*entry.id*, *treeno*) is added to, or deleted from, the corresponding BST. An update occurs when the

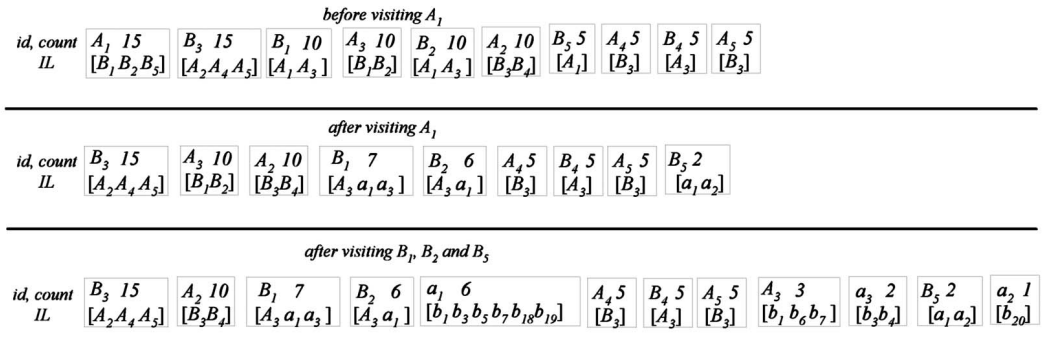


Fig. 6. Example of multiple expansions.

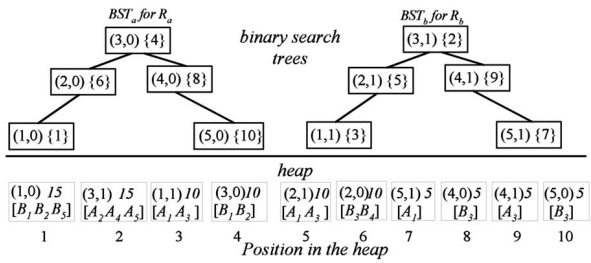


Fig. 7. Example of BST.

```
Multiple_Expansions (entry e)
// e.IL and the contents of n have been retrieved
1.  for each e_i ∈ e.IL
2.      find e_i in H using the BST
3.      if (e ∈ e_i.IL)
4.          remove e from e_i.IL
5.          count(e_i)= count(e_i)- C^e.level
6.          for each e' ∈ n //n is the node pointed by e
7.              if e' intersects e_i
8.                  add e' to e_i.IL
9.                  count(e_i)= count(e_i)+ C^e'.level
10. return
```

Fig. 8. Multiple expansions algorithm.

count of an entry changes, modifying its order in H. In this case, the position field in the BST node is modified accordingly to reflect the change. In our implementation, the construction and update operations of BST are embedded in the heap.

Fig. 8 shows the pseudocode of ME, which is invoked in two cases: 1) when an entry e (e.g., A_1 in the running example) is deheaped, so that its IL is also retrieved (from the top of the heap), and 2) when e is visited because it is in the IL of the deheaped entry (e.g., B_1, B_2, B_5), in which case its $e.IL$ is found using the corresponding BST. The pseudocode assumes that both the intersection list and the contents of the node (n) pointed by e have been retrieved.

It is possible in some cases that, although $e_i \in e.IL$, the opposite is not true, i.e., $e \notin e_i.IL$, even if e and e_i are at the same level (this necessitates line 3 in the pseudocode of Fig. 8). Such a situation is shown in Fig. 9, where each A_i, B_i is a root entry and each a_i, b_i corresponds to a leaf node (the node capacity is 5). After all entries are inserted into the heap, A_1 is deheaped and B_1 is visited because it is in $A_1.IL$. The contents of B_1 are used to update $A_2.IL(B_1 \in A_2.IL = [B_1\ B_3])$, which becomes $A_2.IL = [b_1\ B_3]$ (b_1 is the only child of B_1 that intersects A_2). The heap at this stage is shown in the second row of Fig. 9 (note that B_2 is eliminated because the only entry in $B_2.IL$ is A_1, and no child of A_1 intersects B_2). Subsequently, when B_1 is deheaped, the entries in $B_1.IL$ are examined for possible updates. In this case, although $A_2 \in B_1.IL$, $B_1 \notin A_2.IL$ because of the updates that occurred during the processing of A_1.

ME adds lower level entries, causing a level difference between the heap element and the entries in its IL. On the other hand, the original TS traverses synchronously the two trees (as most conventional spatial join algorithms). In order to integrate ME in TS, we adopt a technique similar to *fix-at-root*[4] [6] for processing node pairs that belong to different levels. The main idea of the technique is that downwards propagation stops in the tree of the lower level node, while propagation in the other tree continues, until a pair of nodes at the same level is reached. Then, the algorithm proceeds in a "synchronous" way.

4. The experimental evaluation presented in [6] suggests that "fix-at-root" is the most efficient method for processing closest-pair queries in trees of different height. The same observation was verified by our experiments for *top-k SJ*.

4.2 Access Locality Method

In this section, we propose an alternative *access locality* (AL) method that utilizes the existence of an LRU buffer. In particular, AL: 1) retrieves a set of top-k candidates (i.e., objects with a large intersection count, but not necessarily actual results) and, then, for every subsequent deheaped entry e (pointing to node n), 2) if e is at level 1 (i.e., pointing to a leaf node), it inserts e in a locality heap (LH) sorted on the lower-left x coordinate, or 3) if e is at a higher level, it performs the join between e and $e.IL$, and inserts the entries of n in the original heap H. Although the total number of node accesses may increase, the number of page faults is expected to decrease due to the locality of subsequent leaf node accesses. Fig. 10 illustrates the general concept of AL.

Fig. 11 presents the pseudocode of the top-k SJ algorithm combined with AL (TS_AL). Initially, TS_AL proceeds in a way similar to TS until it retrieves the first k candidates. After that point, level-1 entries are inserted into LH (instead of H). When the size of LH reaches a threshold T, TS_AL starts to deheap LH and performs the node-pair joins, generating output pairs (of actual objects) in a "batch mode." When the content of LH is exhausted, TS_AL iteratively deheaps H until the number of entries in LH reaches again T, or the last deheaped entry (from H) has a *count* smaller than that of the current result, in which case all entries of H are pruned (but those of LH must be considered). The initial retrieval of the k-candidates (using only the conventional heap) aims at providing a fast bound for subsequent pruning of nonqualifying entries. For our implementation, we set T to $1/3$ of the number of qualifying entries in H.

4.3 Combination of the Optimizations

Multiple expansions and access locality can be integrated in a single algorithm, TS_ME_AL for short, which maintains two heaps H (sorted on *count*) and LH (sorted on lower-left x coordinates). When an intermediate node n (pointed by e) is visited, TS_ME_AL performs the same operations as TS_ME, i.e., upon retrieving the content of n, it updates the entries in H whose intersection lists contain e. On the other hand, if a leaf node n is deheaped, the algorithm does not access it immediately, but inserts it into the local heap LH, so that it will be visited when it is deheaped from LH (i.e., after the size of LH reaches the threshold T). During the expansion of n (from LH), ME is applied only for the entries in its intersection list that also reside in LH. For example, consider that 1) A_1 (pointing to a leaf node) is deheaped from LH, 2) $A_1.IL = [B_1\ B_2]$, and 3) B_1 is in LH, while B_2 is in H. TS_ME_AL will update the entries in the intersection list of B_1 but not of B_2. The reason is that, since B_2 is not deheaped from H yet, and there are still leaf nodes in LH, there is a chance that the algorithm will terminate without having to expand B_2.

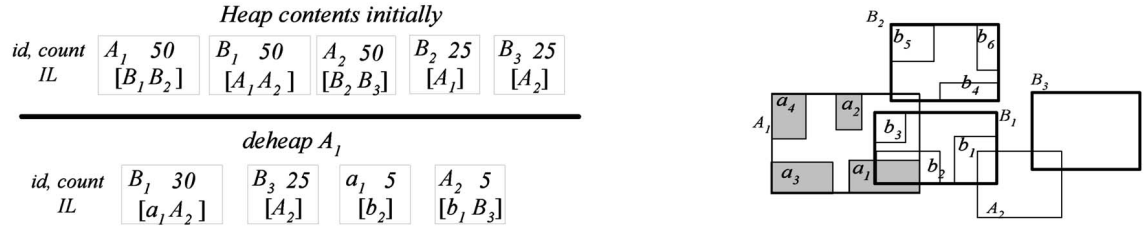


Fig. 9. Example of asymmetric expansion.

4.4 Top-*k* Spatial Semijoins

The top-k spatial semijoin retrieves the k objects from the first data set with the largest intersection count. All the previous algorithms and optimizations can be adapted for this case. For the basic algorithm, the difference is that only entries of the first data set are inserted into the H. Fig. 12 shows the content of the heap using the example of Fig. 1 after visiting the roots (first row—compare with Fig. 2) and after expanding A_1 (second row—compare with Fig. 3). In order to apply multiple expansions, however, we also maintain the intersection lists of entries in R_b, using binary search trees for efficient access on the ID. Finally, the application of the access locality is the same as in the case of full joins.

5 Experiments

In this section, we evaluate the efficiency of the proposed algorithms using a Pentium 4 3.2GHz CPU. In our experiments, we apply the real [29] and synthetic data sets of Fig. 13: 1) MCB with 556,696 census block MBRs, 2) LAP with 1,314,620 street centroids, and 3) SKEW with MBRs[5] whose centroids follow a Zipf distribution (parameter $\alpha = 0.8$) and side lengths range from 0 percent to 1 percent of the data universe. For full join experiments, we convert each point of LAP to an MBR with the same centroid, and size length also in the range 0 percent to 1 perent (we call the resulting data set LA). All the data sets are indexed by R*-trees [2] with page size of 4KBytes, resulting in a capacity of 204 entries per R*-tree node. The tree sizes are 16.2 MBytes for MCB, 37.9 MBytes for LA (LAP), and 80.3 MBytes for SKEW.

5.1 Comparison on Top-*k* Intersection Joins

The first experiment evaluates the algorithms in terms of node accesses for full intersection joins (i.e., we retrieve the top-k objects from either data set). Joins are performed between 1) $\mathrm{MCB} \times \mathrm{LA}$, and 2) $\mathrm{SKEW} \times \mathrm{LA}$. $\mathrm{MCB} \times \mathrm{LA}$ returns 16,477,244 intersecting pairs and the top-1 object (from MCB) overlaps 9,300 objects in LA. $\mathrm{SKEW} \times \mathrm{LA}$ results in 19,657,973 intersecting pairs and the top-1 object (from SKEW) has intersection count 5,968. Note the large output size of the conventional join, which motivates top-k algorithms.

The proposed methods, TS (basic algorithm), TS_ME (multiple expansions), TS_AL (access locality), and TS_ME_AL (combination of all optimizations), are evaluated in terms of node/page accesses and CPU cost. As a benchmark, we also implemented the conventional R-tree join (RJ) algorithm [5], including the space restriction and plane sweep optimizations (discussed in Section 2). RJ first synchronously traverses both R-trees to find all the intersecting object pairs. It then sorts the pairs on object IDs of one data set (assume A). Due to the large size of the join result, external sorting is usually inevitable, which involves writing and reading the pairs to/from secondary memory. After finding the k objects from A with the largest counts, the pairs are sorted again (externally), this time on the IDs of the objects of B and the objects with the largest number of intersections are also obtained. The final output consists of the k objects with the highest counts among the $2 \cdot k$ candidates. Assuming a buffer of 2,000 pages, the overhead of materialization and external sorting is 257,467 and 307,162 page accesses for $\mathrm{MCB} \times \mathrm{LA}$ and $\mathrm{SKEW} \times \mathrm{LA}$, respectively. We consider that these accesses are sequential and, in the following experiments, when node (page) accesses are evaluated, we normalize 10 sequential accesses to 1 random access.

Fig. 14 shows the number of node accesses as a function of the number k of reported objects (ranging from 1 to 32). The cost of RJ[6] (78,194 for $\mathrm{MCB} \times \mathrm{LA}$, and 103,843 for $\mathrm{SKEW} \times \mathrm{LA}$) is independent of k. On the other hand, all the other algorithms are output sensitive and achieve significant gain for the tested values of k. TS_ME has the best performance because the multiple expansions minimize the number of visits to the same node. However, their combination with access locality (TS_ME_AL) incurs some performance deterioration since, as discussed in Section 4.3, some expansions (for entries that do not reside in the locality heap) are deferred. The basic algorithm (TS) performs slightly better than TS_AL because there is no LRU buffer to utilize access locality and the application of this heuristic may delay the expansion of the most promising node pairs.

Note that the number of node accesses increases faster (as a function of k) in Fig. 14a than in Fig. 14b. This happens because, for $\mathrm{MCB} \times \mathrm{LA}$, the intersection counts of the output objects ranges from 9,300 (for $k = 1$) to 3,244 (for $k = 32$), while for $\mathrm{SKEW} \times \mathrm{LA}$, the counts range from 5,968 to 4,544. The large variation in the first case leads to better pruning for small values on k. In general, the proposed algorithms achieve high performance gains (often an order of magnitude) with respect to RJ when the *count* values for the individual objects are skewed and the value of k is relatively small. If the *counts* are similar and the output size is large, then the proposed algorithms eventually degenerate to RJ with, however, the additional overhead of maintaining the heap.

5. The cardinality of SKEW ranges between 1 and 5 million MBRs, with the default value set to 3 million. We use this synthetic data set in order to evaluate the effect of cardinality.

6. Whenever we evaluate performance with respect to k, we include the cost of RJ in the corresponding figure caption (instead of the diagram) because 1) it is usually very high compared to the proposed algorithms and 2) it is independent of k.

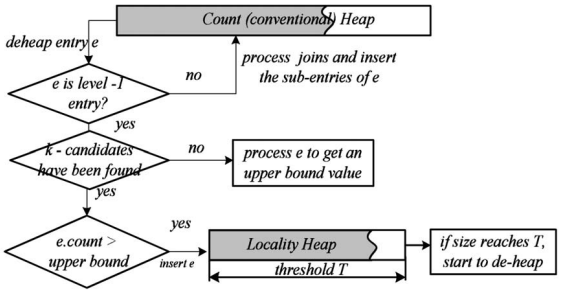


Fig. 10. Concept of access locality optimization.

```
TS_AL (Rtree Ra, Rtree Rb, int k)
1. join RTa and RTb
2. for every e∈RTa∪RTb, insert <e, e.count, e.IL> to a heap H
3. while e.count > results[k].num
4.    e = de-heap(H)
5.    if n is not a leaf node // n is referred by e
6.       for every entry e' in n, insert < e', e'.count, e'.IL> to H
7.    if n is a leaf node and k candidates have not been found completely
8.       join n and the nodes in e.IL
9.       update results[1:k]
10.   if n is a leaf node and k candidates have been found
11.      set threshold T
12.      insert e to LH
13.      if LH.used = T
14.         el = de-heap(LH)
15.         join nl and the nodes in el.IL // nl is referred by el
16.         update results[1:k]
17.return results[1:k]
```

Fig. 11. Pseudocode of TS AL.

Fig. 15 shows the CPU time of the algorithms as a function of k. RJ is again the most expensive algorithm (24.24 and 28.67 seconds) due to the computation of entry intersections in all qualifying node pairs and the overhead of sorting. The proposed methods have to compute fewer intersections but, in addition, they have to maintain the heap. The main difference with respect to Fig. 14 is that TS_ME has the worst performance among the proposed algorithms because multiple expansions involve searching and updating the content of heap, which is a

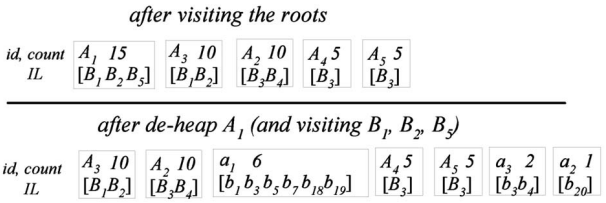


Fig. 12. Application of TS on semijoins.

CPU-demanding operation. Although the method minimizes the number of node accesses, it also performs some redundant work by updating the intersection lists of entries that will be pruned anyway (without having to be expanded). On the other hand, TS_ME_AL has the best CPU performance. This can be explained by the fact that it provides a trade off between TS_ME and TS: It only expands the entries in the locality heap, i.e., the ones that will be considered shortly and may quickly lead to some actual results. The relative performance of TS and TS_AL is the same as in Fig. 14; TS is slightly better as it performs a smaller number of node accesses.

In order to evaluate the total cost of the algorithms, we employ an LRU buffer that accommodates 10 percent of each R-tree. For fairness, we measure the maximum heap size and allocate the same amount of memory to RJ as extra LRU buffer (in addition to the 10 percent cache). In case of $\mathrm{MCB} \times \mathrm{LA}$, the maximum heap consumption occurs for TS_ME and $k = 32$, where there are 7,262 entries in the heap and its size is 3.1 MBytes. For $\mathrm{SKEW} \times \mathrm{LA}$, the maximum heap size (TS_ME and $k = 32$) is also 3.1 Mbytes, but for 9,347 entries (recall that each entry has an intersection list with variable length).

Fig. 16 shows the total cost in seconds (summation of I/O and CPU costs after charging 10ms to each page fault) as a function of k. TS_ME_AL is the winner for both joins because it reduces the node accesses and takes the

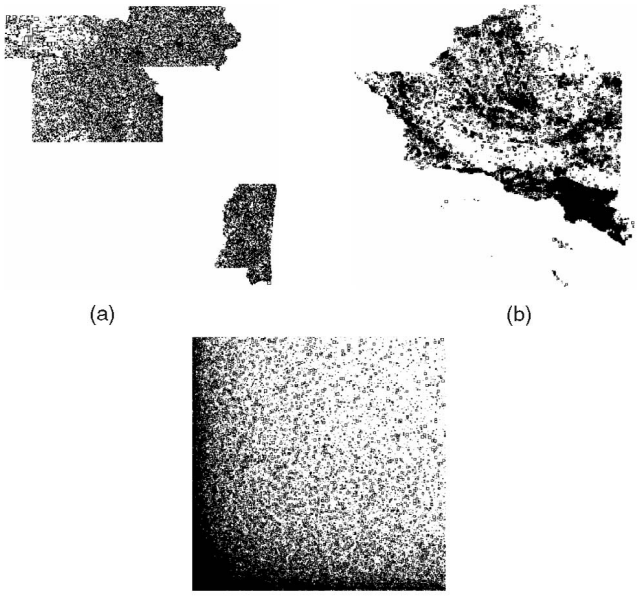


Fig. 13. Three data sets in the experiments. (a) MCB (556,696 MBRs), (b) LA (1,314,620 MBRSs), and (c) SKEW (default cardinality 3,000,000).

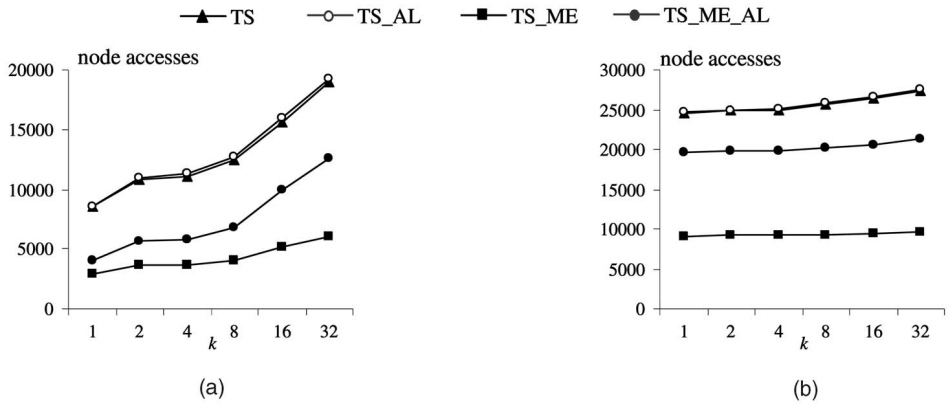


Fig. 14. Node accesses versus k (full join). (a) $\mathrm{MCB} \times \mathrm{LA}$ (RJ accesses = 78,194). (b) $\mathrm{SKEW} \times \mathrm{LA}$ (RJ accesses = 103,843).

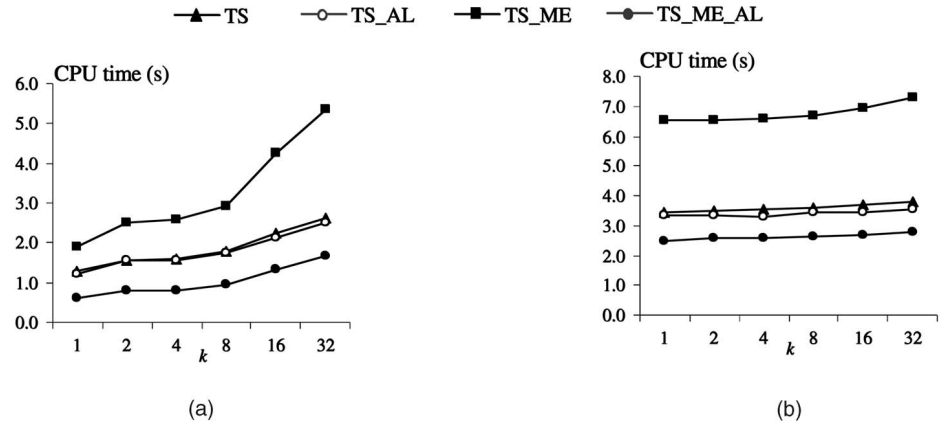


Fig. 15. CPU time versus k (full join). (a) $\mathrm{MCB} \times \mathrm{LA}$ (RJ CPU time = 24.24s). (b) $\mathrm{SKEW} \times \mathrm{LA}$ (RJ CPU time = 28.67s).

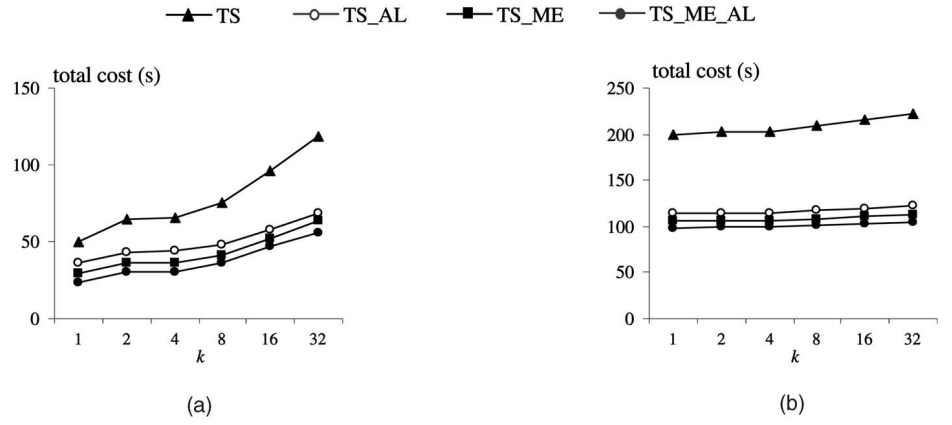


Fig. 16. Total cost versus k (full join, 10 percent cache). (a) $\mathrm{MCB} \times \mathrm{LA}$ (RJ total cost = 357s). (b) $\mathrm{SKEW} \times \mathrm{LA}$ (RJ total cost = 456s).

advantage of the LRU buffer through access locality. TS_ME is the second best (despite its high CPU cost) due to the small number of node accesses. TS_AL consistently outperforms TS because of its visiting order strategy. Finally, although TS has the worst performance among the proposed algorithms, it is better than RJ, which takes around 6 minutes to terminate for $\mathrm{MCB} \times \mathrm{LA}$, and more than 7 minutes for $\mathrm{SKEW} \times \mathrm{LA}$.

To study the scalability of our methods, we measure the total cost of $\mathrm{SKEW} \times \mathrm{LA}$ as a function of the cardinality of SKEW ranging from 1M objects to 5M objects. We fix $k = 4$ and apply the same settings as in Fig. 16 (i.e., we allocate the maximum heap size to RJ as extra LRU buffer). The cost of RJ (included in Fig. 17) grows fast with the cardinality because of the increasing number of intersection pairs (since the distribution remains the same). On the other hand, the performance of the proposed techniques is rather constant and, in some cases, it even improves with the increasing cardinality. This is because the high density leads to large differences between intersection counts and, therefore, to early pruning of the heap entries.

Finally, as discussed in Section 3.2, the proposed algorithms can be easily applied to aggregate structures that maintain summarized information in the intermediate

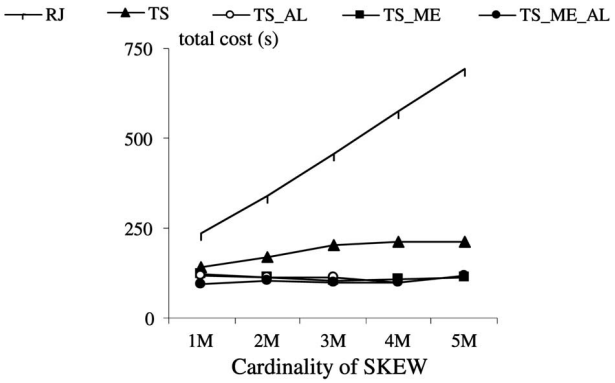


Fig. 17. Total cost versus SKEW cardinality (full join, 10 percent cache, $k = 4$).

entries. Fig. 18 repeats the experiment of Fig. 16 assuming that the data sets are indexed by aR-trees. Because of the additional aggregate information at the intermediate nodes, the capacity of the aR-tree drops to 170. The utilization of the aR-trees reduces the cost of the proposed algorithms, but deteriorates the performance of RJ due to the smaller node capacity. Similar to Fig. 16, TS_ME_AL is still the winner, confirming the advantage of combining *multiple expansion* and *access locality* in full joins.

In summary, even the basic algorithm (TS) achieves significant gains with respect to RJ for practical (i.e., small) values of k. The optimizations lead to further improvements depending on the value of k and the presence of an LRU buffer. Furthermore, our techniques scale well with the data cardinality and can be easily applied with aggregate R-trees.

In the next section, we evaluate the generality of these observations by experimenting with top-k containment semijoins since they are equally important in practice as full joins, and exhibit some different properties (due to the containment relationship and the asymmetry of the join operator).

5.2 Comparison on Top-*k* Containment Semijoins

Semijoins are performed between $\mathrm{MCB} \times \mathrm{LAP}$ and $\mathrm{SKEW} \times \mathrm{LAP}$, i.e., we retrieve the k objects from MCB and SKEW that contain the largest numbers of points from LAP. In this case, RJ sorts the output of the Euclidean join only once (according to the object IDs in the first data set), while the implementation of the proposed algorithms follows the description of Section 4.4. The top-1 object from MCB (SKEW) contains 6,703 (4,398) points of LAP.

Fig. 19 investigates the number of node accesses for different values of k. The relative performance of the algorithms is the same to that of full joins, but the cost is now lower due to the elimination of the second sorting for RJ and the smaller heap size (and, consequently, fewer expansion operations) for the proposed algorithms.

Fig. 20 presents the CPU cost as a function of k. Compared to the full join case (Fig. 15), TS_ME is still the most CPU-intensive algorithm, but the difference with respect to the rest is reduced because of the smaller number of expansions that are performed. On the other hand,

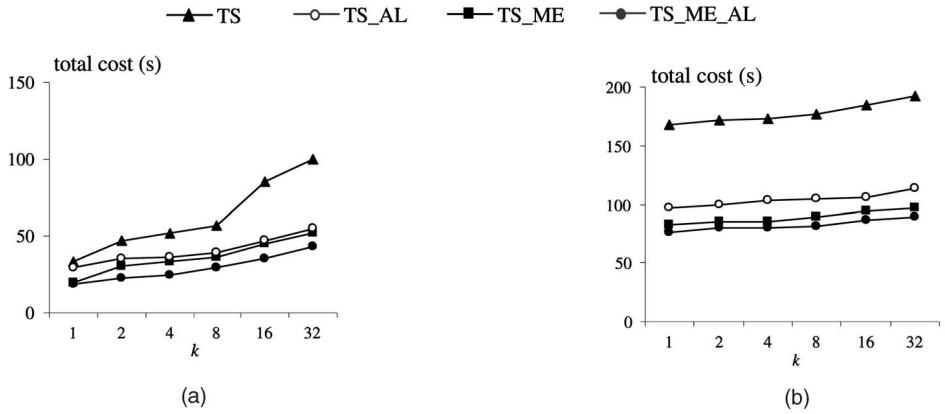


Fig. 18. Total cost versus k (aR-tree, full join, 10 percent cache). (b) $\mathrm{MCB} \times \mathrm{LA}$ (RJ total cost = 373s). (b) $\mathrm{SKEW} \times \mathrm{LA}$ (RJ total cost = 482s).

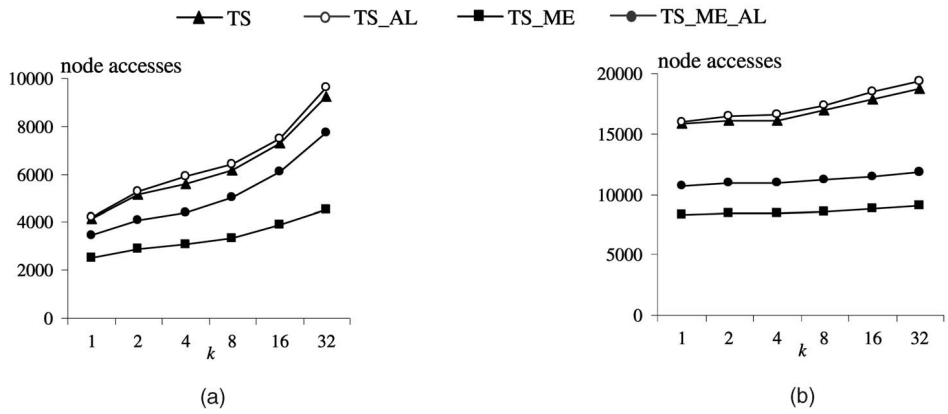


Fig. 19. Node accesses versus k (semijoin). (a) $\mathrm{MCB} \times \mathrm{LAP}$ (RJ accesses = 35,383). (b) $\mathrm{SKEW} \times \mathrm{LAP}$ (RJ accesses = 66,208).

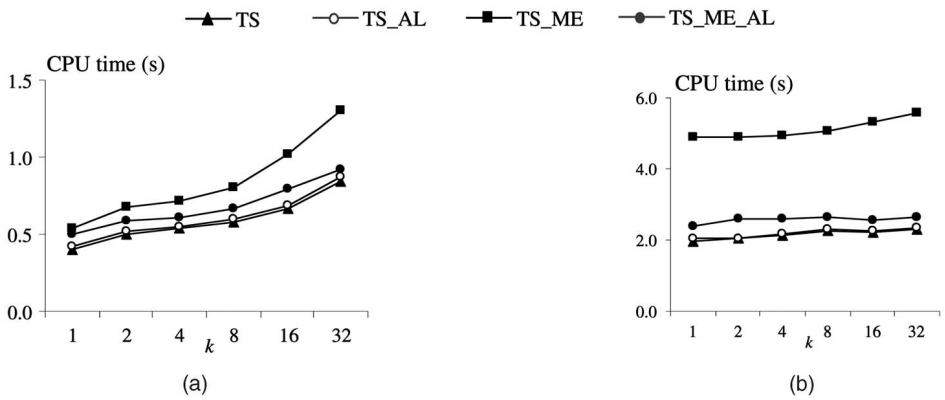


Fig. 20. CPU time versus k (semijoin). (a) $\mathrm{MCB} \times \mathrm{LAP}$ (RJ CPU cost = 5.5s). (b) $\mathrm{SKEW} \times \mathrm{LAP}$.

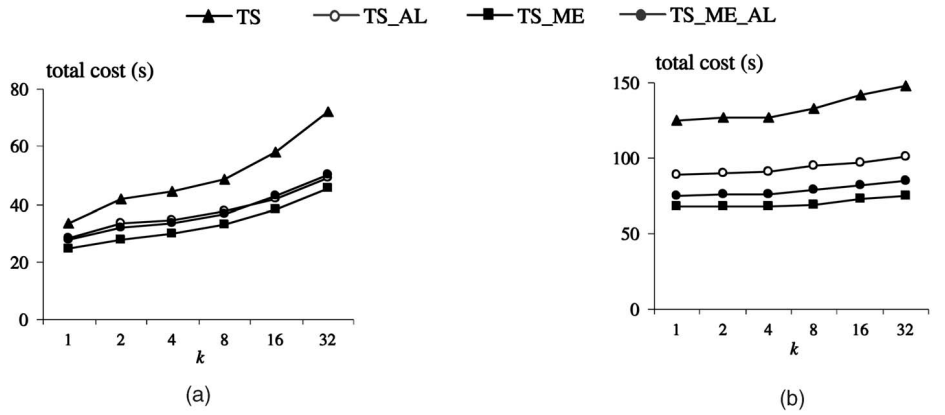


Fig. 21. Total cost versus k (semijoin, 10 percent cache). (a) $\mathrm{MCB} \times \mathrm{LAP}$ (RJ total cost = 102s). (b) $\mathrm{SKEW} \times \mathrm{LAP}$ (RJ total cost = 197s).

TS_ME_AL is slightly better, but does not exhibit the large gains as in the case of full joins. The best algorithm in terms of CPU performance is TS, followed by TS_AL.

In Fig. 21, we evaluate the total cost of the algorithms as a function of k by fixing the cache size to 10 percent of the two R-trees. Similar to full joins, we measure the maximum heap size of the proposed algorithms and allocate the same amount of memory to RJ in addition to the 10 percent cache size. TS_ME now outperforms all the other algorithms because: 1) it has the smallest number of node accesses; 2) the LRU buffer is more effective for semijoins. The second factor is analyzed as follows: The node accesses consist of visits to the deheaped nodes (n) and the nodes in the intersection list of n. Since in semijoins the deheaped nodes are only from the first data set, it is more likely that subsequent nodes are close in space and intersect similar nodes from the second data set (so that these nodes are more likely to be in the buffer). On the other hand, in full joins, the deheaped nodes alternate from both data sets and may be scattered in the data universe. The order of the other algorithms is the same as full joins (i.e., TS_ME_AL outperforms TS_AL, which outperforms TS). We also repeated the experiment for the case that the data sets are indexed by aR-trees; the costs of the proposed algorithms decrease (similar to Fig. 18), but their relative performance remains as shown in Fig. 21 (and the diagrams are omitted).

Finally, in order to evaluate the effect of cardinality, we apply the same settings (i.e., including the LRU buffer with the extra allocation to RJ) and measure the cost of $\mathrm{SKEW} \times \mathrm{LAP}$ as a function of the cardinality of SKEW ($k = 4$) in Fig. 22. Similar to Fig. 17 (for full joins), with the exception of TS, the proposed methods are insensitive to the cardinality, whereas the cost of RJ increases linearly.

6 Conclusions

This paper proposes the top-k spatial (semi) join, a query type which is interesting from a research point of view and has practical relevance for several multidimensional applications. The processing of such queries by conventional spatial join algorithms incurs high overhead because we only require a small percentage of the output (as opposed to

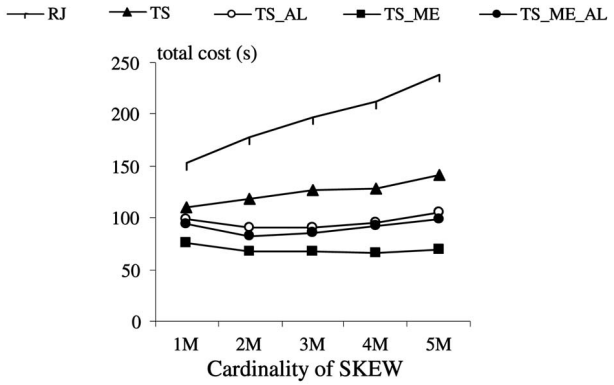


Fig. 22. Total cost versus SKEW cardinality (semijoin, 10 percent cache, $k = 4$).

the entire set of intersection pairs). Instead, we develop a basic algorithm (TS) and two optimizations based on the concepts of multiple expansions (ME) and access locality (AL). TS_ME_AL (the combination of the two optimizations) is the best method for full joins since it utilizes the LRU buffer, while TS_ME is preferable for semijoins (because the visited nodes are more likely to be already clustered). Our algorithms can also process bottom-k queries (retrieving the k objects with the smallest intersection count), by simply sorting the entries in the heap with respect to the minimum possible count. We chose not to experiment with this version, since for most geographic joins there are numerous objects that do not intersect any other object (and, therefore, the output is not very interesting). There may exist, however, applications where such queries are important.

Directions for future work include several variations of the problem. For instance, the proposed algorithms can be easily extended for the top-k distance (semi) join, which given two data sets A, B and a range e, it returns the k objects of A that are within distance e from the largest number of objects of B. Moreover, since our methods are independent of the underlying data structure, they can also be used to process moving objects indexed by TPR-trees [30], or other spatio-temporal access methods. Another interesting problem concerns top-k nearest-neighbor (semi) joins. In this case, the goal is to return the k objects of A that are the closest neighbors to the largest number of objects of B. Consider, for instance, that A is a set of facilities, B is a set of clients, and that each client is serviced by the closest facility. The output of the top-k nearest-neighbor semijoin would represent the k facilities that serve the highest number of customers. A possible processing method involves 1) computing the nearest neighbors (in A) of all objects of B, 2) sorting the resulting pairs (o_b, o_a), where $o_a \in A$ is the NN of $o_b \in B$, with respect to o_a, and 3) reporting the top-k objects of A. This technique is expected to be very expensive because the number of NN queries equals the cardinality of B. Alternative algorithms are possible, based on a methodology similar to that of top-k spatial joins.

ACKNOWLEDGEMENTS

This work was supported by grants HKUST 6179/03E and HKUST 6180/03E from Hong Kong RGC. The authors would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. Vitter, "Scalable Sweeping-Based Spatial Join," *Proc. 24th Int'l Conf. Very Large Data Bases (VLDB),* 1998.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD,* 1990.

[3] S. Berchtold, D. Keim, and H. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data," *Proc. 22nd Int'l Conf. Very Large Databases (VLDB),* 1996.

[4] B. Boehm, B. Braunmuller, F. Krebs, and H. Kriegel, "Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High Dimensional Data," *Proc. ACM SIGMOD,* 2001.

[5] T. Brinkhoff, H.P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-Trees," *Proc. ACM SIGMOD,* 1993.

[6] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos, "Closest Pair Queries in Spatial Databases," *Proc. ACM SIGMOD,* 2000.

[7] M. Denny, M. Franklin, P. Castro, and A. Purakayastha, "Mobiscope: A Scalable Spatial Discovery Service for Mobile Network Resources," *Proc. Fourth Int'l Conf. Mobile Data Management (MDM),* 2003.

[8] S. Govindarajan, P. Agarwal, and L. Arge, "CRB-Tree: An Efficient Indexing Scheme for Range Aggregate Queries," *Proc. Int'l Conf. Database Theory (ICDT),* 2003.

[9] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD,* 1984.

[10] G. Hjaltason and H. Samet, "Incremental Distance Join Algorithms for Spatial Databases," *Proc. ACM SIGMOD,* 1998.

[11] G. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems,* vol. 24, no. 2, pp. 265-318, 1999.

[12] Y. Huang, N. Jing, and E. Rundensteiner, "Spatial Joins Using R-Trees: Breadth First Traversal with Global Optimizations," *Proc. 23rd Int'l Conf. Very Large Data Bases,* 1997.

[13] I. Ilyas, W. Aref, and A. Elmagarmid, "Supporting Top-k Join Queries in Relational Databases," *Proc. 29th Int'l Conf. Very Large Data Bases,* 2003.

[14] M. Jurgens and H. Lenz, "PISA: Performance Models for Index Structures with and without Aggregated Data," *Proc. 11th Int'l Conf. Scientific and Statistical Database Management (SSDBM),* 1999.

[15] N. Koudas and K. Sevcik, "Size Separation Spatial Join," *Proc. ACM SIGMOD,* 1997.

[16] M.-L. Lo and C. Ravishankar, "Spatial Joins Using Seeded Trees," *Proc. ACM SIGMOD,* 1994.

[17] M.-L. Lo and C. Ravishankar, "Spatial Hash-Joins," *Proc. ACM SIGMOD,* 1996.

[18] G. Luo, J. Naughton, and C. Ellman, "A Non-Blocking Parallel Spatial Join Algorithm," *Proc. IEEE Int'l Conf. Data Eng.,* 2002.

[19] N. Mamoulis and D. Papadias, "Multiway Spatial Joins," *ACM Trans. Database Systems,* vol. 26, no. 4, pp. 424-475, 2001.

[20] N. Mamoulis and D. Papadias, "Slot Index Spatial Join," *IEEE Trans. Knowledge and Data Eng.,* vol. 15, no. 1, pp. 211-231, 2003.

[21] M. Mokbel, W. Aref, and I. Kamel, "Performance of Multi-dimensional Space-Filling Curves," *Proc. 10th ACM Int'l Symp. Advances in Geographic Information Systems,* 2002.

[22] A. Natsev, Y. Chang, J. Smith, C. Li, and J. Vitter, "Supporting Incremental Join Queries on Ranked Inputs," *Proc. 27th Int'l Conf. Very Large Data Bases,* 2001.

[23] J. Orenstein, "Spatial Query Processing in an Object-Oriented Database System," *Proc. ACM SIGMOD,* 1986.

[24] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient OLAP Operations in Spatial Data Warehouses," *Proc. Seventh Int'l Symp. Spatial and Temporal Databases (SSTD),* 2001.

[25] A.N. Papadopoulos, P. Rigaux, and M. Scholl, "A Performance Evaluation of Spatial Join Processing Strategies," *Proc. Sixth Int'l Symp. Spatial Databases,* 1999.

[26] J.M. Patel and D.J. DeWitt, "Partition Based Spatial-Merge Join," *Proc. ACM SIGMOD,* 1996.

[27] D. Rotem, "Spatial Join Indices," *Proc. IEEE Int'l Conf. Data Eng.,* 1991.

[28] N. Roussopoulos, S. Kelly, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD,* 1995.

[29] Rtree portal, http://www.rtreeportal.org, 2003.

[30] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez, "Indexing the Positions of Continuously Moving Objects," *Proc. ACM SIGMOD,* 2000.

[31] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-tree: A Dynamic Index for Multidimensional Objects," *Proc. 13th Int'l Conf. Very Large Data Bases,* 1987.

[32] J. Shan, D. Zhang, and B. Salzberg, "On Spatial-Range Closest-Pair Query," *Proc. Ninth Int'l Symp. Spatial and Temporal Databases,* 2003.

[33] S.Y. Shou, N. Mamoulis, H. Cao, D. Papadias, and D. Cheung, "Evaluation of Iceberg Distance Joins," *Proc. Ninth Int'l Symp. Spatial and Temporal Databases,* 2003.

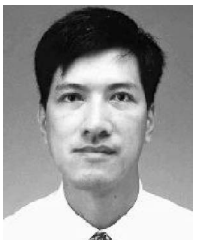
[34] Y. Tao and D. Papadias, "Range Aggregate Processing in Spatial Databases," *IEEE Trans. Knowledge and Data Eng.,* to appear and available at: www.cs.ust.hk/~dimitris/.

[35] D. Zhang, V. Tsotras, and D. Gunopulos, "Efficient Aggregation over Objects with Extent," *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems,* 2002.

Manli Zhu received the master's degree from the Department of Computer Science and Engineering, South China University of Technology (SCUT) in 1999. She is currently a PhD candidate in the Department of Computer Science at the Hong Kong University of Science and Technology. Before joining HKUST, She worked at Hewlett-Packard, China, for one year as a management trainee. Her research interests include spatial queries in mobile environment and location-based information systems.

Dimitris Papadias is an associate professor in the Computer Science Department, Hong Kong University of Science and Technology. Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and the University of Patras (Greece). He has published extensively and been involved in the program committees of all major Database Conferences, including SIGMOD, VLDB, and ICDE.

Jun Zhang received the bachelor's degree from the Department of Computer Science and Engineering, South China University of Technology (SCUT) in July 2000 and the PhD degree from the Computer Science Department, Hong Kong University of Science and Technology (HKUST) in January 2004. He is an assistant professor in the Division of Information Systems, Computer Engineering School, Nanyang Technological University (NTU), Singapore. His research interests include indexing techniques and query optimization in spatial and spatio-temporal databases and data warehouses.



Dik Lun Lee received the BSc degree in electronics from the Chinese University of Hong Kong and the MS and PhD degrees in computer science from the University of Toronto in 1981 and 1985, respectively. He is a professor in the Department of Computer Science at the Hong Kong University of Science and Technology and was an associate professor in the Department of Computer and Information Science at the Ohio State University, Columbus. He served as guest editor for several journals and as a program committee member and chair for many international conferences. He was the founding conference chair for the International Conference on Mobile Data Management. His research interests include document retrieval and management, search engines, mobile computing, and pervasive computing. He was the chairman of the ACM Hong Kong Chapter.