Slot Index Spatial Join

Nikos Mamoulis and Dimitris Papadias

Abstract—Efficient processing of spatial joins is very important due to their high cost and frequent application in spatial databases and other areas involving multidimensional data. This paper proposes *slot index spatial join* (SISJ), an algorithm that joins a nonindexed data set with one indexed by an R-tree. We explore two optimization techniques that reduce the space requirements and the computational cost of SISJ and we compare it, analytically and experimentally, with other spatial join methods for two cases: 1) when the nonindexed input is read from disk and 2) when it is an intermediate result of a preceding database operator in a complex query plan. The importance of buffer splitting between consecutive join operators is also demonstrated through a two-join case study and a method that estimates the optimal splitting is proposed. Our evaluation shows that SISJ outperforms alternative methods in most cases and is suitable for limited memory conditions.

Index Terms—Spatial databases, query processing, join processing, database index, spatial index, buffer management.

1 Introduction

Spatial database management systems [31] aim at the efficient management of large collections of multidimensional data (e.g., satellite images, molecular structures), found in several fields such as Geographical Information Systems, CAD, Medical Databases, and Multimedia Information Systems. One of the most common spatial query operators is the *spatial join*, which retrieves from two data sets all object pairs that satisfy a spatial predicate, most often *intersect* (also called *overlap*). A spatial join example is "find all cities that are crossed by a river." The high complexity of the objects and the quadratic number of pairs that may qualify the query predicate render efficient processing of spatial joins an important issue.

Spatial objects are usually approximated and indexed by their *Minimum Bounding Rectangle* (MBR). A spatial query is then processed in two steps [25]: First, a *filter step* employs the index to retrieve all MBRs that satisfy the query and possibly some false hits. Then, a *refinement step* uses the exact geometry of the objects to dismiss the false hits. Traditional access methods (e.g., B-trees) are not readily applicable for spatial queries due to the fact that there is no total ordering of objects in space that preserves spatial proximity [12]. As a result, a number of *spatial access methods* (SAMs) have been proposed [9]. The predominant SAM, used in many commercial systems (e.g., Illustra, Informix) is the R-tree [13] and its variations.

Several spatial join algorithms, most of which focus on the filter step, have been proposed during the last decade. Some of these algorithms consider existing indexes for both joined inputs, while others treat data sets with no index, thus providing solutions for the case where at least one input comes as an intermediate result of another database operator. Here, we focus on spatial joins when only one of the joined data sets is indexed by an R-tree. Such situations may arise when the nonindexed input is the result of another operator. For instance, consider the query "find all *cities* with *population over* 5,000, which are crossed by a *river*" and that *cities* and *rivers* are indexed on their spatial extent. If only a few large cities exist, it may be natural to process the selection part of the query before the spatial join. In this case, the resulting cities are nonindexed and some *single-index join* algorithm is required.

Existing single-index join algorithms have certain limitations. For instance, the *indexed nested loop join*, which applies a window query to the R-tree for each object in the nonindexed set can be very expensive in terms of both I/O and computational cost. The *seeded tree join* [22], which creates an R-tree for the nonindexed data, is not appropriate in many cases because of its prohibitive I/O cost. Methods like *bulk loading* and *matching*, *sorting* and *matching* [28] apply external sorting on the nonindexed data and totally or partially build an on-the-fly R-tree in order to join it with the existing one. Therefore, these methods have a disadvantage in cases where the nonindexed input is an intermediate result of an underlying operator because they need to materialize it before processing it.

In this paper, we propose *slot index spatial join* (SISJ), a hash join algorithm that overcomes most of the above deficiencies. SISJ distributes the R-tree entries at a specific level into S partitions, called *slots*, and builds an in-memory index from them. The *slot index* keeps for each slot the identifiers of the nodes pointed to by the corresponding entries along with the MBR of the entries. The nonindexed input is partitioned into S partitions (buckets) with the same spatial extents as the MBR of the slots. The algorithm finally joins each bucket with the R-tree data under the nodes pointed to by the corresponding slot, using plane sweep [29].

In addition to slot index construction heuristics [24], we present two I/O and CPU optimization methods that are applied in the join phase of SISJ and significantly improve its performance. A *bucket ordering* heuristic joins first the

- *N. Mamoulis with the Department of Computer Science and Information Systems, University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: nikos@csis.hku.hk.*
- *D. Papadias is with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: dimitris@cs.ust.hk.*

Manuscript received 8 July 1999; revised 15 Dec. 2000; accepted 14 Dec. 2001. For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 110197.

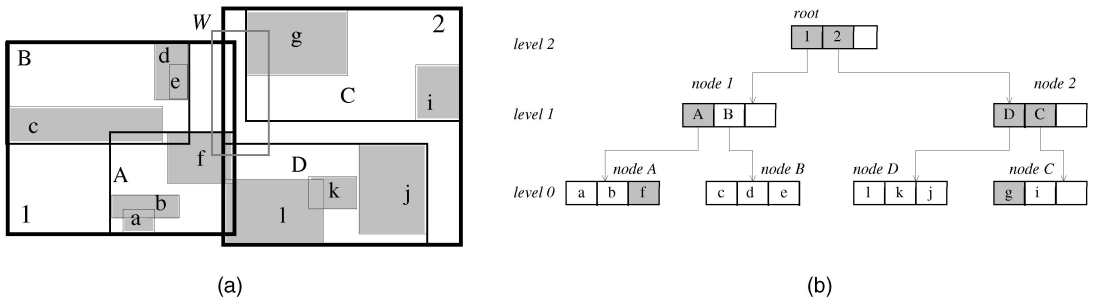

Fig. 1. A set of objects and the corresponding R-tree. (a) A set of objects $\{a, b, c \ldots, i\}$. (b) An R-tree that indexes the objects.

hash buckets with a few pages on disk in order to avoid writing and reading again their in-memory parts. This technique reduces the number of page accesses, especially when the nonindexed input is only slightly larger than the available memory. A *repartitioning heuristic* improves the computational performance of the algorithm and further reduces its space requirements. After the application of these optimization methods, the overall cost of SISJ drops about 35 percent compared to the initial implementation.

We also provide an analytical study about the cost of SISJ and other single-index spatial join algorithms, which is parametric to the characteristics of the joined data sets and the system conditions. All algorithms (with the exception of indexed nested loops) are I/O bound, thus we focus on their I/O cost. Based on the analysis, we present a qualitative comparison, examining both cases where the nonindexed input is read from disk or is produced by another query operator. Finally, the efficiency of the algorithms is experimentally evaluated using real and synthetic data sets of various sizes. The importance of buffer splitting between pipelined operators [10] is demonstrated through a two-join case study, and a method that estimates the optimal splitting is proposed. It turns out that SISJ is the best algorithm in most cases because it is not sensitive to limited memory conditions.

The rest of the paper is organized as follows: Section 2 presents background on R-trees and spatial join algorithms. Section 3 describes SISJ and explores its optimization in terms of both I/O and CPU time. Analytical cost models for SISJ and other single-index join algorithms are provided in Section 4. Section 5 contains an extensive experimental evaluation and Section 6 concludes the paper.

2 Background

During the past two decades, the database community devoted many of its research efforts on the efficient processing of spatial queries. Since most of this work presupposes the existence of R-trees as the underlying access method in spatial database systems, we start with a description of this SAM and its variations. Then, we overview the most important spatial join algorithms focusing on the *seeded tree join* [22] and *spatial hash join* [21] because they motivate the proposed SISJ algorithm and have common modules with it.

2.1 R-Trees

The R-tree [13] is a height-balanced tree, similar to the B^+-tree, that indexes MBRs of objects in the multidimensional space. The nodes of the tree correspond to disk pages and they are at least 40 percent full, ensuring good disk utilization. The entries in leaf nodes are *(MBR, oid)* tuples, containing object MBRs and pointers to their exact representation. Intermediate node entries are *(MBR, ptr)* tuples, where *ptr* points to a lower level node and *MBR* is the MBR of all entries in this node. Fig. 1 shows a set of object MBRs and the graphical representation of an R-tree used to index this data set.

R-trees were originally designed for multidimensional range query processing. Consider the R-tree of Fig. 1 and a query which asks for all objects that intersect window *W*. The entries of the root that intersect *W* are found and, for each such entry, search is applied recursively on the node pointed to by it. The qualifying entries are shown in gray. If an entry at a high R-tree level does not intersect *W*, then there can be no qualifying object in the subtree pointed to by this entry. For instance, *W* does not intersect entry *B*, in node 1, thus the subtree under this entry needs not be accessed.

Guttman [13] proposes an R-tree insertion algorithm that splits the entries in overflowing nodes so that the resulting nodes at the high levels of the tree have as small extents as possible. This is led by the intuition that the smaller the area of R-tree entries, the smaller the "dead space" inside them (i.e., the space not covered by real objects) and, therefore, the lowest the probability that the search algorithm will visit nodes unnecessarily. For instance, observe that *W* in Fig. 1 intersects entry *D*, but there is no rectangle indexed under *D* that has some part in this "dead space." Ideally, if the dead space was zero, search could be optimal.

An alternative insertion algorithm, which, apart from the area of the extents, considers other parameters was proposed in [4]. The resulting access method is called R*-tree. Notice that the only difference between the R-tree and the R*-tree is the insertion algorithm; both access methods have the same structure. The insertion algorithm of the R*-tree can be summarized as follows: A new rectangle *r* is inserted by starting from the root and following entries that fully contain it until a leaf node is reached. In the case where more than one entry at a high-level node contains *r*, the tie is broken by following the one with the minimum area. If no entry contains *r*, the one that is enlarged the least after the insertion is followed (*ChooseSubtree*). If a leaf node overflows for the first time, the algorithm removes 30 percent of its entries and

reinserts them (*Forced Reinsert*). This allows rectangles to move to a more appropriate node dynamically and the structure of the tree to become less sensitive to the insertion order. If a node overflows for a second time, it is split. The splitting algorithm divides the rectangles into two nodes trying to minimize their overlap and their total margin. The margin minimization aims at node MBRs that look more like squares, rather than narrow rectangles, improving query performance.

If all data are available a priori, it is too time consuming to build the R*-tree by consecutive insertions. Several *bulk loading* methods that quickly build an R-tree for a static set of rectangles have been proposed. Most of them sort the data according to some spatial key and then pack each group of consecutive C rectangles in a leaf node (C is the node capacity). The tree is then built from the leaf nodes in a bottom-up fashion and has a minimum number of nodes and height. These methods are also called *R-tree packing* methods.

Several alternative techniques can be used for sorting the rectangles in order to build the packed R-tree. In [30], the data are sorted according to some x-coordinate value and the resulting tree has leaf nodes that are narrow stripes. The resulting R-tree has poor query performance due to the excessive number of leaf nodes intersecting a random window. Sorting the rectangles with respect to the Hilbert value of their center [17] generates nodes with a better shape, but the calculation of the Hilbert value is expensive and this method can be efficient only when the sorting key is precomputed. *Sort tile recursive* (STR) [19] is a simple, yet very effective method, which initially sorts the N rectangles with respect to the x-coordinate of their center. This ordered list is then split into $\lceil\sqrt{P}\rceil$ groups (vertical strips), where $P = N/C$, and each group is sorted using the y-coordinate of the rectangles' center. The output from this sorting formulates the leaves of the resulting packed R-tree. The intuition behind this two-phase sorting is to yield square-like leaf nodes. Yet, another method that bulk loads R-trees that are not packed is [6]. The splitting criteria of the R*-tree insertion algorithm are considered, while dividing the rectangles and building the index in a bottom-up fashion. Finally, the R*-tree bulk loading method proposed in [8] (and used also in [26], [2]) sorts the rectangles by Hilbert value and produces nodes that are not necessarily fully packed, in order to decrease the overlap between them. Recently produced nodes are organized in a cache and the R*-tree split algorithm is used to reorganize their contents and improve the quality of their extents.

2.2 Spatial Join Algorithms

If both data sets, A and B, to be joined are indexed by R-trees R_A and R_B, then a very efficient spatial join algorithm is the *R-tree join* (RJ) [5] (also called *tree matching* or *synchronous traversal*), which synchronously traverses both trees following entry pairs that intersect. Fig. 2 illustrates two data sets indexed by R-trees. The intersecting entry pairs at the root level are (A_1, B_1) and (A_2, B_2). Notice that since A_1 does not intersect B_2, there can be no object pairs under these entries that intersect. RJ is recursively called for the nodes pointed by the qualifying entries until the leaf level is reached, where the intersecting pairs (a_1, b_1) and (a_2, b_2) are output.

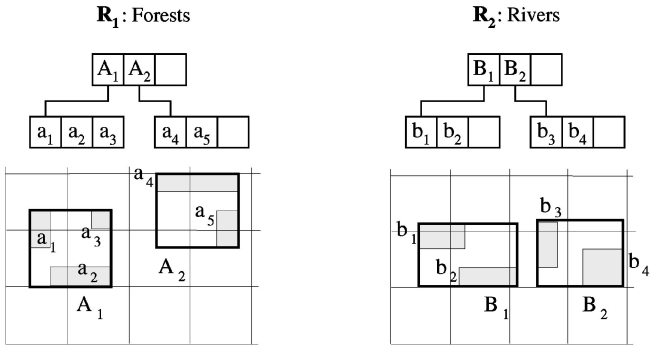

Fig. 2. Two data sets indexed by R-trees.

Two optimization techniques decrease the computational cost of RJ. The first, called *search space restriction* reduces the quadratic number of pairs to be evaluated when two nodes N_A and N_B are joined. If an entry $E_A \in N_A$ does not intersect the MBR of N_B (that is the MBR of all entries contained in N_B), it can be safely ignored. In Fig. 2, for instance, when computing qualifying pairs in nodes A_2 and B_2, entry a_4 does not intersect node B_2, so it cannot intersect any entry inside it. Using this observation, space restriction performs two linear scans in the entries of both nodes before applying entry intersection tests and prunes out from each node the entries that do not intersect the MBR of the other node. The second technique (*forward sweep* [1]) is based on *plane sweep* [29] and applies sorting in one dimension in order to reduce the computation time of the overlapping pairs between the nodes to be joined. A breadth-first search version of RJ with reduced I/O cost is presented in [14].

A number of methods exist for the case where none of the inputs is indexed: some partition the space either regularly [26], [18], or irregularly [21], and distribute the objects into buckets defined by these partitions. The spatial join is then performed in a relational hash join fashion. Although, the above methods work well for uniformly distributed inputs, they cannot guarantee good worst-case performance like the algorithm in [1] which first applies external sorting to both files and then uses an adaptable version of plane sweep, considering that in most cases the "horizon" of the sweep line will fit in main memory. Recently, Arge et al. [2] extended this method to handle both indexed and nonindexed inputs. The basic idea is to use the existing indexes in combination with a heap structure to access the rectangles from the trees that intersect the sweep line.

The simplest method for the case that only one input is indexed is the *indexed nested loop join* (INLJ). In accordance with its relational join counterpart, INLJ applies a window query to the R-tree for every object in the nonindexed data set. The *build and match* join (BaM) [26], [28] builds an R-tree from the raw input using bulk loading and joins it with the existing tree using RJ. *Sort and match*(SaM) [28] employs STR [19] to sort the rectangles from the nonindexed input but, instead of building the packed tree, it directly matches in-memory created leaf nodes. For each produced leaf node, a window query is executed and plane sweep is applied to join it with all leaf nodes from the existing R-tree that

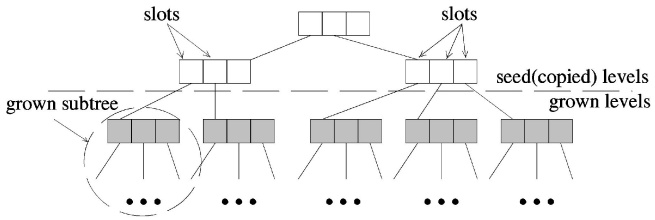

Fig. 3. A seeded tree.

intersect it. Another method that constructs an R-tree and joins it with the existing one is called *seeded tree join* (STJ) [22], described in detail in the next section.

2.3 Seeded Tree Join

STJ [22] builds a second tree using R_A as a *seed* and then applies RJ. The motivation behind creating a seeded tree for the second input, instead of a normal R-tree, is the fact that an R-tree with nodes having similar extents to nodes in R_A will be more efficient during tree matching, as the number of overlapping node pairs between the trees will be smaller. Thus, STJ creates an R-tree that is suitable for the spatial join and not for range searching.

The seeded tree construction is divided in two phases: 1) During the *seeding* phase, the top k levels (k is a parameter of the algorithm) of R_A are copied to formulate the top k levels of the seeded tree R_B. The entries of the lowest level of R_B are called *slots*. After copying, the slots maintain the copied extent, but they point to empty (null) subtrees. 2) During the *growing* phase, all objects from B are inserted into R_B. A rectangle is inserted under the slot that contains it or requires the least area enlargement in order to enclose it. Fig. 3 shows an example of a seeded tree structure. The two topmost levels of the R-tree are copied to guide the insertion of the second data set.

Lo and Ravishankar [22] propose some techniques that optimize the structure of the seeded tree and a filtering mechanism that rejects rectangles from the second set that do not overlap any of the slots. They also present a tree construction technique that reduces I/O page accesses when the size of the tree exceeds the size of the available memory buffer. If this happens, many pages may have to be fetched and written back to disk during a single insertion, resulting in a large I/O cost (*buffer thrashing*). In order to avoid this, the objects to be inserted under a slot are written in a temporary file. After all objects have been assigned to a slot, an R-tree is constructed for each temporary file and is pointed to by the corresponding slot in the seeded tree. To implement this mechanism and minimize random I/O accesses, at least one page is allocated in the buffer for each slot. If the buffer is full, all slots that have more than a constant number of pages flush their data to disk and memory is freed.

In order for the above algorithm to work efficiently, the number of slots S should be smaller than the number of pages M in the memory buffer. In a different case, not every slot can be assigned a page in main memory and pages may have to be temporarily written to and reloaded from the disk during the construction of the seeded tree (another form of buffer thrashing). As a result, the algorithm is inefficient when the fanout of the R-tree nodes is large and the memory buffer is relatively small. Consider, for instance, a data set of 100,000 objects, which are indexed by a 8K page size R-tree. Under the assumption that each node entry is 20 bytes long (16 for the x and y-coordinates, plus four for the object id or block reference), the capacity of a tree node is 409; thus, the data set can be indexed by a 2-level R-tree, with 245 leaf nodes and one root. When applying STJ, we have to copy the root level of the R-tree to the seeded tree, which results in $S = 245$. As a consequence, the algorithm is inefficient for buffers smaller than 1.96 Mbytes.

2.4 Spatial Hash-Join

The *spatial hash-join* (HJ) method [21] is similar to the Grace hash join algorithm used in relational databases [32]. HJ joins two spatial inputs A and B, none of which is indexed. The general idea of the method is to determine a set of rectangular spatial partitions and hash data sets A and B into them. The resulting structures are called *hash buckets*. Each bucket is thus defined by the rectangle of the corresponding partition to which we will refer to as the "extent" of the bucket. After hashing, each pair of buckets from A and B that correspond to the same partition (they have the same extent) are loaded and joined to derive the result of the spatial join.

The extents of the hash partitions are determined by data set A. The goal of HJ is to define partitions with small extents, little overlap, and with approximately the same number of objects from A assigned to them. The number of partitions S is determined, such that the average number of hashed rectangles in a bucket can fit in memory and buffer thrashing while partitioning is avoided (i.e., S should be smaller than the memory buffer). Since no information about the distribution of the rectangles in the data sets is generally available, the initial extents of the partitions are computed in two phases by a sample from A. Initially, S rectangles from the sample are selected and their centers define the partitions. Then, the other rectangles are assigned to the partition closer to their center and the center of this partition is adjusted accordingly. The sample is passed once more to refine the resulting partitions. The result of this process (called *bootstrap seeding* technique [20]) is a set of points that represent the distribution of data set A.

During the hash-phase, each object r_A from A is assigned to exactly one partition, according to the following criteria. If the rectangle is enclosed by one or more partitions, it is assigned to the one with the minimum area (as in the R*-tree insertion algorithm). If r_A is not enclosed by any partition (this case occurs in the beginning, where all partitions are points), it is assigned to the one that needs the least enlargement to contain it. After hashing all rectangles from A, the spatial partitions will have changed and they may overlap with each other. The resulting *hash buckets* from A are stored into sequential files using the same policy as in STJ, in order to reduce random I/Os.

Set B is then hashed into buckets with the same extent as A's buckets, but with a different insertion policy; the extents of the buckets are kept fixed and an object is inserted into all buckets that intersect it. Thus, some objects may go into more than one bucket (*replication*), and some may not be

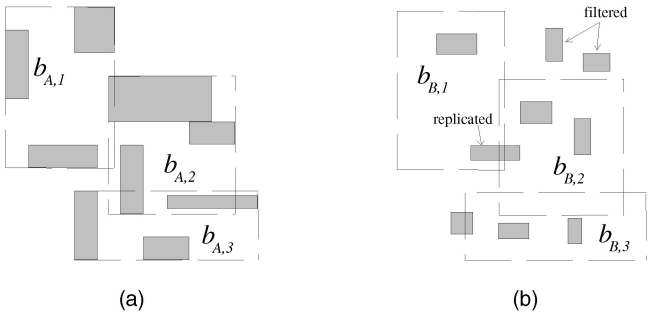

Fig. 4. The partition phase of the HJ algorithm. (a) Objects from set *A* in three partitions. (b) Filtering and replication of objects from set *B*.

inserted at all (*filtering*). The algorithm does not ensure that the rectangles from A are evenly partitioned into the buckets, as sampling cannot guarantee the best possible slots. Buckets of equal size for B cannot be guaranteed in any case, as the distribution of the objects in the two data sets may be totally different. Fig. 4 shows an example of two data sets, partitioned using the HJ algorithm.

After hashing set B, the join is performed by matching each bucket $b_{A,i}$ from A with the corresponding bucket $b_{B,i}$ from B (which has the same extent). If one bucket fits in memory, it is loaded and the objects of the other are prompted against it, thus requiring a single scan and a minimal number of I/Os. Techniques like plane sweep are used to accelerate the join phase if both buckets fit in memory. If none of the buckets fits in memory, an on-the-fly R-tree is built for one of them (e.g., using bulk loading [19]), and the bucket-to-bucket join is executed in an indexed nested-loop fashion by applying a window query for each rectangle from the other bucket.

Experiments in [21] show that HJ is better in terms of I/O than building two seeded trees and joining them. It is also claimed that this algorithm is more efficient than spatial join with precomputed R-tree indexes (RJ) if the difference between sequential and random disk accesses is taken into account. We believe that this comparison of HJ with RJ is unfair. First, as shown in [24], RJ is significantly faster than HJ in terms of CPU-time. Second, as shown in [16], when a R-tree packing method that places sibling nodes in sequence is used, the I/O performance of RJ is significantly improved. In the rest of the paper, we do not consider the difference between random and sequential I/O accesses.

3 Slot Index Spatial Join

In this section, we describe *slot index spatial join* (SISJ), a new algorithm that joins a nonindexed input with an R-tree. SISJ applies hash join using the existing R-tree to guide the hash process. We propose four policies that determine the extents for space partitions used for hashing the nonindexed data set and two optimization techniques that reduce the cost of the algorithm. Finally, we present a series of experiments that evaluate the efficiency of the proposed heuristics.

3.1 Motivation and Description

SISJ combines ideas from STJ and HJ. The key idea is to define the spatial partitions of HJ using the structure of the existing R-tree. Let A be a data set indexed by the R-tree R_A and B be a nonindexed data set. Assume that we attempt to apply HJ to join them. At a specific level of R_A each entry defines a hash bucket that contains the R-tree data indexed by the subtree pointed to by this entry. Thus, we can consider the partition phase of the data set A completed. The partitions are suitable for hash join since they contain approximately the same number of object MBRs from A; their extent is optimized by the R-tree insertion algorithm and each object MBR is indexed by exactly one entry. The extents of these partitions (e.g., the MBRs of the corresponding R-tree entries) can be used to hash the nonindexed data set B. Then each bucket of B can be matched with the subtree indexed by the corresponding entry from R_A.

A problem with the above method is which R-tree level should be chosen for the definition of the spatial partitions. The number of partitions should be small enough to avoid buffer thrashing while hashing B. Recall that hashing algorithms should assign at least one page for each bucket; thus, the number of partitions should be at most $M-1$ (where M is the number of pages in the buffer). There is also a desirable lower bound for this number; we would like the data pointed to by each R-tree entry that defines a partition to fit in memory. If this happens, the match phase of the join could be performed efficiently with a minimal number of I/Os (see Section 2.4). In many cases, finding an R-tree level with an appropriate number of entries is not possible; the difference between the numbers of entries in consecutive levels can be very large, especially for trees of large fanout. In order to overcome the buffer size limitations, (i.e., when the number of entries is larger than the buffer size M), SISJ groups the entries of a tree level to $S(< M)$ possibly overlapping partitions called *slots*. Each slot contains the MBR of the indexed R-tree entries, along with a list of pointers to these entries. The collection of slots (a kind of hash-table) is called *slot index* and it is small enough to be maintained in main memory. Fig. 5a illustrates the middle (level 1) of the three levels of an R-tree where the root contains only two entries. If $M = 10$, SISJ will group the entries at level 1 in $S = 9$ (for this example) slots (Fig. 5b).

After building the slot index, the second set B is hashed into buckets with the same extents as the slots. As in HJ, if an object from B does not intersect any bucket it is filtered; if it intersects more than one bucket, it is replicated. Fig. 5b shows some filtered and replicated objects. The join phase of SISJ is similar to the corresponding phase of HJ. All data from R_A indexed by a slot are joined with the corresponding hash-bucket for set B. When no data from B are inserted into a bucket, the subtrees under the corresponding slot need not be loaded. The upper rightmost slot in Fig. 5b is an example of a *filtered slot*.

When joining a slot with a bucket, four cases may apply:

1. The data from both partitions fit in memory. In this case, the join is performed using forward sweep [1], [5]. This method is simple and its average performance is as good as that of methods with a theoretically better worst-case time (e.g., an *interval-tree* method is theoretically optimal, but found in

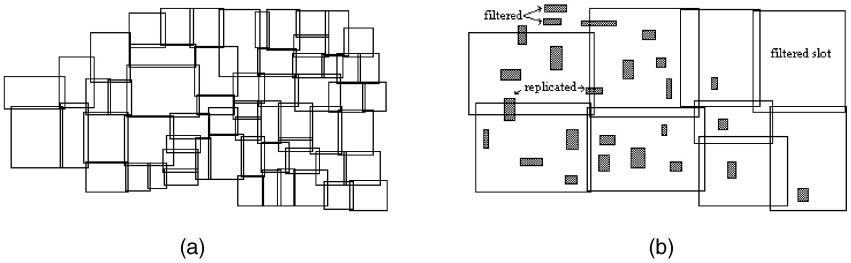

Fig. 5. An R-tree and a slot index built over it. (a) Entries at R-tree level 1 and (b) slot index and hashed data from *B*.

[1] to have similar performance with forward sweep).

2. Only the data under the slot fit in memory. The join is then performed using indexed nested loop join, considering as the root of the R-tree the corresponding slot; for each rectangle in the hash bucket, a window query is applied.
3. Only the bucket from *B* fits in memory. An on-the-fly in-memory R-tree is built for the bucket and the data under the slot are "probed" against it (indexed nested loops).
4. Neither the data under the slot nor the bucket fit in memory. In this case, SISJ is applied recursively in a way similar to recursive hash-join [32]. The slot acts as the virtual root of an R-tree and the hash bucket as the nonindexed data set. The recursion is guaranteed to terminate even in the trivial case where all rectangles from B intersect all rectangles from A; the rectangles from A will be divided in partitions and SISJ will be called until Case 2 applies, where the recursion stops. We do not expect SISJ to be very efficient in such situations, but our aim is to define an algorithm that performs well on the average, without focusing on worst-case settings that do not appear often in practice.

Typically, the values of M and S will be large enough for cases 1, 2, and 3 to hold for most join pairs. Notice that for these cases, the I/O cost of matching is optimal (no disk page is loaded more than once in the join phase). In Section 3.3, we discuss an optimization method that can be applied in cases 1 and 3, significantly improving the CPU performance of the algorithm.

3.2 Slot Index Construction Policies

As stated above, S should be smaller than M in order to avoid buffer thrashing. The lower limit of S is such that the expected number of data from set A in each slot will fit in memory. If P_A is the number of pages that fit for the first data set (i.e., the number of R_A leaf nodes assuming that R_A is packed), then the number of blocks that fit for the data under a slot is:

$$size(slot_A) = \lceil P_A/S \rceil. \quad (1)$$

In order for the data under each slot to fit in memory ($\lceil P_A/S \rceil < M$) and to avoid buffer thrashing, the value of S should be restricted by the following inequalities:

$$\lceil P_A/M \rceil < S < M. \quad (2)$$

There exist some cases (when M is very small compared to P_A) where the lower limit P_A/M should be ignored because it is larger or equal to M. Consider, for instance, that the page size is 8K, the buffer size is 128K, and set A consists of 100,000 objects (=2Mbytes); then $M = 128/8 = 16$ and $P_A = 245$. Equation (2) results in $16 < S < 16$, which does not provide a valid value for S. Thus, the lower limit is ignored and the data under each slot are not guaranteed to fit in memory. Notice that the upper limit cannot be ignored because this would lead to buffer thrashing. A study about selecting S from a valid range of values is performed in Section 3.5.

The topmost tree level k with total number of entries $n_{E,k} > P_A/M$ is the level where grouping will take place[1] (*slot level*). If $n_{E,k}$ is within the valid range for S, i.e., $P_A/M < n_{E,k} < M$, S is exactly $n_{E,k}$ and the slots will have as extents the MBRs of these entries. If $n_{E.k} \geq M$, we cannot directly use the entries $n_{E,k}$ to partition and the slot index should be built. A good slot index construction mechanism will minimize the total area and overlap between the slots and create slots with the same number of entries. We consider four policies for creating the desired number of slots from the $n_{E,k}$ entries at the slot level of R_A:

1. *SplitXL*. Sort entry MBRs with respect to their lower x-bound and divide them into S equal sized groups. This method is motivated by [30].
2. *SplitHC*. Sort entry MBRs with respect to the Hilbert value of their center and divide them into S equal sized groups. SplitHC is motivated by [17].
3. *SplitSTR*. Sort entry MBRs using the *sort tile recursive* algorithm [19] and divide them into S equal sized groups.
4. *IRS*. Insert the entries into S slots using the R*-tree insertion algorithm [4].

From the above grouping methods, the first three include just sorting and splitting. IRS (*insert*, *reinsert*, and *split*) is more sophisticated. Starting from a single empty slot, for each entry *e*, the insertion algorithm of Fig. 6 is called.

The first part of IRS is equivalent to the *ChooseSubtree* R*-tree algorithm that determines the best leaf node when inserting a rectangle (see Section 2.1). Each slot can be assigned a maximum number of entries (*maximum slot capacity*). If the entries in a slot are more than this number, an overflow occurs. Part *2a* is equivalent to the *Forced Reinsert*, whereas *2b* is the R*-tree *split*

1. Under typical system conditions (e.g., page size 4K-8K, M in the order of 100) and for typical size of data set A (e.g. 10K-200K rectangles) usually k will be the root level, or the level under the root.

```
Algorithm IRS(RTreeEntry e)

1. choose a slot s, such that e.MBR is contained into s.MBR;

1a. If more than one such slots exist, choose the one with the smallest area;

1b. If no such slot exists, choose the one that causes the minimum overlap
enlargement (between the slots) after e has been inserted to it;

2. insert e into s and update its MBR;

2a. If s overflows, and no other overflow has occurred during this insertion:

• sort the entries in s according to the distance of their centers to the center
  of s.MBR;

• delete from s the 30% last (furthest) entries and update s.MBR;

• re-insert the entries into the slots;

2b. If s overflows, and overflow has occurred again during this insertion:

• apply the R*-tree split algorithm to split s into 2 slots;
```

Fig. 6. The IRS slot index construction algorithm.

algorithm. IRS does not guarantee slots of equal size; the equal size splitting criterion is not considered in order to favor the good shape criterion. To ensure that the final number of slots after IRS will be "around S" and considering that slot utilization is around 70 percent [4] (given that each slot is at least 40 percent full), we set the maximum slot capacity to $(10/7) \cdot (n_{E,k}/S)$, so that the average number of entries in a slot will be $n_{E,k}/S$. The final number of groups may not be S but will definitely be between $(7/10) \cdot S$ (if all groups are full) and $(7/4) \cdot S$ (if they are 40 percent full). If these limits are out of the valid range, the maximum slot capacity should be tuned correspondingly.

Notice that $n_{E,k}$ cannot exceed $M \cdot C$, where C is the node capacity (i.e., maximum fanout) of R_A; otherwise, the upper tree level $k-1$ should be used for grouping (it would contain at least M entries to be partitioned into $S < M$ slots). Therefore, all four policies can take place in main memory ($M \cdot C$ entries occupy exactly M pages and this is a worst case) with small computational cost. As an example of slot index construction, consider a real data set with 131K rectangles (T1) stored in an 8K page size R*-tree. The tree has three levels: the entries of level 0 (i.e., object MBRs) are shown in Fig. 7a, the 466 entries of level 1 in Fig. 7b and the two root entries in Fig. 7c.

If $S = 20$, grouping will take place at level 1 where the 466 entries will be arranged into 20 slots. The extents of the slots using the four different policies are shown in Fig. 8. In Section 3.5, we empirically compare the effect of the different policies on the performance of SISJ.

3.3 Optimizing the Bucket Join Order

After the creation of the slots, objects from set B are hashed using the slot index. A naive implementation of SISJ would then flush all in-memory partitions to disk in order to free space for the join phase. Instead of flushing all blocks, we can perform the join trying to maintain as many nonflushed blocks in memory, as possible (*warm buffer*). For instance, if set B is smaller than the buffer, we can avoid writing anything to disk since hashing can take place in memory and, provided that there is enough space for the data indexed by a slot, the hash-join can also be performed without any extra I/O accesses. For typical cases, when set B does not fit in memory, we have to consider an ordering of the slot-bucket pairs that would minimize the I/O cost by writing as few blocks as possible. The required space to join $slot_A$ and $bucket_B$ in memory is:

$$\text{join\_mem}(slot_A, bucket_B) = \text{size}(slot_A) + \text{size\_mem}(bucket_B) + \text{size\_disk}(bucket_B), \quad (3)$$

where $\text{size}(slot_A)$ is the number of blocks needed to fit the R-tree data under $slot_A$, $\text{size\_mem}(bucket_B)$ is the number of blocks in memory for $bucket_B$ and $\text{size\_disk}(bucket_B)$ is the number of flushed blocks.

If there is not enough space in the buffer, $\text{size}(slot_A) + \text{size\_disk}(bucket_B)$ blocks have to be loaded from disk and an equal number of blocks have to be written from some other hash buckets. After joining the first pair, there will be more available memory for the second one, due to the release of the $\text{size\_mem}(bucket_B)$ pages, in addition to the pages loaded from disk. Thus, the early join pairs will cause more page faults than the latter ones. Since $\text{size}(slot_A)$ is expected to be approximately the same for all partitions (i.e., P_A/S), it is natural to join the pair involving the bucket form set B that has the minimum $\text{size\_disk}(bucket_B)$. This will minimize the number of pages that must be flushed to disk (and later be reloaded) in order to make space for loading $\text{size\_disk}(bucket_B)$ pages. A pseudocode for the bucket join ordering algorithm is given in Fig. 9. In some extreme cases involving large pairs of slots-buckets, we cannot avoid flushing everything to disk. The bucket join ordering heuristic has the advantage that it delays such a situation because it fetches the smallest partitions first and utilizes best the in-memory parts. The I/O savings yielded by this method are demonstrated in Section 3.5.

Fig. 7. Entries of T1 R*-tree. (a) Level 0 (leaf) entries. (b) Level 1 entries. (c) Level 2 (root) entries.

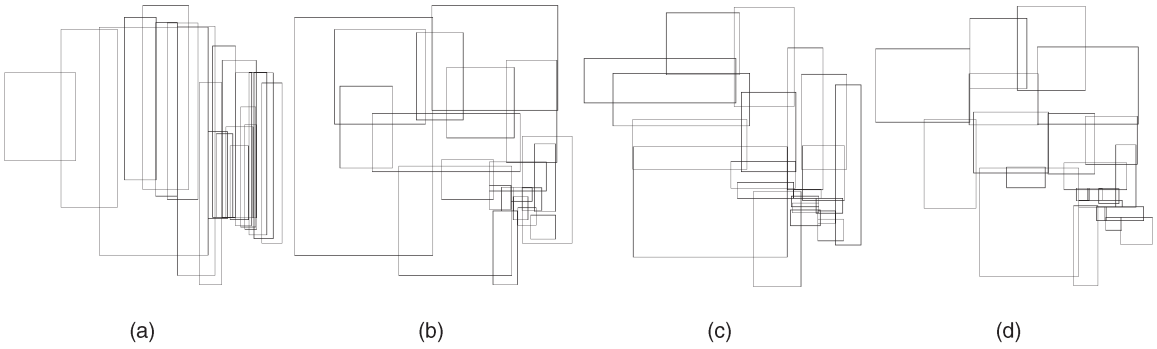

Fig. 8. A slot index for data set T1 using four grouping policies. (a) SplitXL. (b) SplitHC. (c) SplitSTR. (d) IRS.

```
Algorithm bucket_join_order()

1. Choose from set B the bucket_B, not processed yet, with minimum
size_disk(bucket_B);

2. If P_A/S + size_disk(bucket_B)  > fmb (# free memory blocks)

       then flush P_A/S + size_disk(bucket_B) - fmb blocks;

3. Load slot_A; Load size_disk(bucket_B);

4. Join(slot_A, bucket_B);

5. Free slot_A; Free bucket_B;

6. If there are more slot-bucket pairs to be joined go to 1;
```

Fig. 9. An algorithm that determines a good bucket join order.

3.4 Repartitioning before Plane Sweep

The natural method of joining a slot with a bucket is to load the contents of both into memory and apply plane sweep. When the contents are large this method has two disadvantages. First, the available memory can be limited for all data to fit in the buffer. In Cases 2 and 3 discussed in Section 3.1, indexed nested loops may have a large CPU time overhead comparing to plane sweep, and in Case 4, recursive application of SISJ may lead to a nontrivial I/O penalty. Second, as shown in [1], the *forward sweep* heuristic and plane sweep methods in general, are CPU-intensive for large data sets; for each rectangle in $slot_A$ or $bucket_B$, forward sweep may need to trace $\sqrt{N}$ rectangles (where N is the total number of rectangles in a bucket/ under a slot) in order to compute the intersection pairs that include this rectangle (*square root rule* [11]). When N is large this operation is rather expensive.

Arge et al. [1] propose a *striped sweep* algorithm that regularly divides the space in s horizontal[2] stripes and partitions both data sets using them. Then, plane sweep is used to calculate the results in each stripe. In this way, the number of rectangles intersected by the sweep line is at most $\sqrt{N}/s$ at each stripe and the total CPU cost is greatly reduced. Nevertheless, this method has three drawbacks. First, both data sets have to be partitioned. Second, since each stripe has fixed bounds, rectangles of both sets are replicated during partitioning (one rectangle may intersect more than one stripe). Therefore, it is possible for one

2. In fact, the authors use *vertical stripes* because the sweep line in their implementation is horizontal.

```
Algorithm RLNP_join(slot_A, bucket_B)
1. Load subtree_A under slot_A, except from the leaf nodes;
2. Hash all rectangles in bucket_B into RLNP partitions guided by subtree_A;
3. Free subtree_A except from the leaf node block pointers of all RLNP partitions;
4. For each partition p_B in RLNP do:
   4a. Load corresponding leaf_node_A;
   4b. Join leaf_node_A with p_B using forward_sweep;
   4c. Free leaf_node_A;
5. Free bucket_B;
```

Fig. 10. An improved method for joining a slot with a bucket.

intersection pair to be reported in more than two stripes and a method that removes duplicate results has to be engaged (with a an extra computational overhead). Third, both data sets have to fit in memory; otherwise, extra I/O accesses for writing the partitioned results are required.

Instead of striped sweep, we propose a method that only requires $bucket_B$ to fit in main memory. The R-tree nodes under the $slot_A$ except from the leaves are loaded into the buffer. Then, all rectangles of $bucket_B$ are hashed into main memory partitions with the same extents as the leaf node MBRs under the slot, guided by the loaded subtree structure. Hashing is done in the same way as rectangles from B are partitioned into buckets, i.e., each rectangle is hashed into all partitions it intersects. We call this method *R-tree leaf node partitioning* (RLNP). After RLNP, the $subtree_A$ structure under the slot is removed from memory and each leaf node is loaded and matched with the corresponding RLNP partition. Thus, each node in R_A is matched only with relevant rectangles from B and the main memory join is performed efficiently in small areas of the data space. The pseudocode for this slot-bucket join method is given in Fig. 10.

Since rectangles from R_A reside in exactly one leaf node, no duplicate solutions are generated. Replication occurs, like in the hash-phase of SISJ, but the extra memory required to store the replicated rectangles is much less than the extra disk space required at the replication during the hash-phase of SISJ. This is because 1) the partitions defined by the leaves of the R*-tree usually have a smaller overlap than the slot extents and 2) only the ids of the rectangles need to be replicated since the coordinates reside in memory and can be easily accessed. Therefore, replication introduces no extra or trivial I/O overhead when clever caching is used.

Notice that apart from the CPU time gain due to the small number of rectangles in the partitions joined during plane sweep, the repartitioning method also saves I/O accesses. While $\text{size}(slot_A) + \text{size}(bucket_B)$ pages were required to perform the join in memory before, now we need $\text{size}(subtree_A) + \text{size}(bucket_B)$ pages, where $\text{size}(subtree_A)$ is the number of blocks needed to fit the subtree under $slot_A$, excluding the leaf level. Provided that in typical cases of 4K and 8K page sizes, the slot index level is over the leaves, $\text{size}(subtree_A)$ will be just the size of the corresponding slot in the slot index, i.e., trivial. Furthermore, while joining the RLNP partitions with the leaf nodes, only one leaf node is needed at a time. The required memory of a slot-bucket join using this method is:

$$\text{join\_mem}_{\text{RLNP}}(slot_A, bucket_B) = \text{size}(subtree_A) + \text{size\_mem}(bucket_B) + \text{size\_disk}(bucket_B). \quad (4)$$

Due to the fact that

$$\text{join\_mem}_{\text{RLNP}}(slot_A, bucket_B) \leq \text{join\_mem}(slot_A, bucket_B),$$

fewer pages will be flushed to disk, thus reducing the I/O cost of SISJ. Implementing this optimization method is rather simple because partitioning modules are already used by SISJ. The algorithm is not expected to profit by bucket join ordering and repartitioning optimization techniques, if no $bucket_B$ fits in memory. However, for typical data sets and buffer sizes, such situations seldom arise. This is demonstrated in the experiments of the next section.

3.5 Experimental Evaluation

Three sets of experiments were conducted in order to evaluate the relative performance of SISJ after the use of slot index construction and slot-bucket join heuristics. We first compare the quality of the four grouping policies (SplitXL, SplitHC, SplitSTR, and IRS), then test the effect of the selection of S in the performance of SISJ and, finally, evaluate the gains of the two slot-bucket join optimization methods. Table 1 describes several real and synthetic data files used in the experiments. GS and GR are highly skewed data sets containing MBRs of Greek roads and rivers. Files AS, AL (*http://www.maproom.psu.edu/dcw/*) comprise street and railroad segments, respectively, of Germany. T1 contains streets and T2 river and railroad segments of California [7]; both files are commonly used to benchmark spatial join algorithms [5], [21], [14], [16]. The synthetic files G1 and G2 were created according to a Gaussian distribution with 16 clusters. The centers of the clusters were randomly generated, and the sigma value of the data distribution around the clusters followed a random value between 1/20 and 1/10 of the map size. In addition to the cardinality of the data sets, Table 1 shows their *density*,

TABLE 1
Characteristics of the Data Sets Used at the Experiments

Set	*Description*	*Size*	*Density*	*h*	*P*	*T*
GS	Greek roads	23268	0.33	2	57	88
GR	Greek rivers	24650	0.39	2	61	93
AS	German roads	30674	0.08	2	75	113
AL	German railroads	36334	0.07	2	89	129
T1	California roads	131461	0.05	3	322	469
T2	California rivers + railroads	128971	0.39	3	316	428
U1	Uniformly distributed MBRs	100000	0.5	2	245	324
U2	Uniformly distributed MBRs	100000	1	2	245	321
G1	Gaussian distributed MBRs	100000	0.5	2	245	328
G2	Gaussian distributed MBRs	100000	1	2	245	329

defined as the total area of the rectangles divided by the total area of the workspace.[3]

Whenever a data set was indexed, an R*-tree with node size 8K (equal to the system page size) was built. The buffer size was set to 512K ($M = 64$).[4] Table 1 also shows the height h of the corresponding R*-trees, the number P of pages that fit the data sets in a sequential file and the number T of tree nodes. An UltraSparc2 workstation (200 MHz) with 256MB of main memory was used for the experimental evaluation.

Fig. 11 presents the effect of the four slot index construction policies on the performance of SISJ for various join pairs, letting $S = 20$. The (normalized) overall cost was computed here and in the rest of the paper after charging 10ms for each page access; a typical value for modern disks[5] [32]. In all cases, IRS is better than the other policies, with SplitXL performing very poorly because the slots have large overlapping areas introducing extensive replication at the partitioning of set B. The inferior performance of SplitHC and SplitSTR compared to IRS is due to the fact that the entries to be split have large spatial extents which do not favor the application of packing algorithms [6]. The slot extents created by these methods are thus worse than the ones produced by IRS. This can also be evaluated by some quality metrics of the partitions. For example, the area covered by more than one partition (overlap area) as a percentage of the total area of the slots was 142 percent, 53 percent, 34 percent, and 14 percent on the average for the four partitioning policies. Other quality metrics like the minimization of margins and replication ratio also prove the superiority of IRS. In the rest of the paper, we adopt IRS as the standard slot index construction policy.

The number of slots S may take a wide range of values according to inequality (2). In the next experiment, we test the effect of S on the performance of SISJ. The overall costs of the three joins that involve real data sets are split to hashing and join cost (Fig. 12). Notice that there is no significant difference in performance for the various choices. The hashing cost grows slightly with S, as more bucket extents have to be tested and more replication is introduced. The join time is larger for small S when the data sets are large (e.g., $T1 \bowtie T2$) because the chance that the contents of some slot-bucket pair will not fit in memory increases. In general, the minimum value of S that can lead to pairs fitting in memory is a good choice.

In the final set of experiments, we evaluate the join ordering and repartitioning optimization methods for SISJ. The number of slots S in the experiments was in the order of 10-25, depending on the size of the data sets. Fig. 13 presents the performance of three versions of SISJ for various join pairs. The first version of the algorithm does not utilize bucket join ordering or repartitioning. The second version uses only bucket join ordering and the last both heuristics. The bucket join ordering method has effects only on the I/O time of the algorithm due to the delay of page flushing. The gain is larger for the small data sets, where few pages have been flushed before the join phase and the partitions are small enough to avoid writing many pages when loading the first join pairs.

The repartitioning method (applied in combination with bucket join ordering) has a large impact on the CPU-time performance of SISJ. The deterioration in the performance of forward sweep is avoided due to the small number of rectangles in each joined pair after the repartitioning. The performance gain grows with the size of data sets because the size of each hash-bucket increases. Another advantage of repartitioning is that it introduces extra filtering; the R-tree leaf nodes have less dead space than the slots and many hashed rectangles are filtered. There is also a small I/O gain due to the decrease of memory demands during the slot-bucket join. In the current experimental settings, the buckets from set B always fit in memory and repartitioning was applicable. In the sequel, these two methods are considered in the standard implementation of SISJ.

4 Analysis of Single-Index Spatial Join Algorithms

In addition to SISJ, several other algorithms can be used for processing spatial joins when only one input is indexed. This section provides analytical formulae for their expected I/O performance. The analysis is provided for completeness since to our knowledge, no other work provides cost formulae for these methods (except from indexed nested loops) that can be used by query optimizers. Moreover, it helps to comprehend the functionality and the pros and cons of the methods. However, it cannot be used for a

3. Given a series of different layers of the same region (e.g. rivers, streets, forests), its *workspace* is defined as the total area (not necessarily rectangular) covered by all layers including holes, if any.

4. In all experiments throughout the paper, the data sets used have sizes between 500Kb-3Mb. We also consider buffer sizes one order of magnitude smaller than the data sets because we believe that they match real-life situations best, e.g., a system that joins data sets with sizes in the order of 1Gb is expected to use a memory buffer of at least 100Mb. Thus, we also expect that the relative differences in performance to be representative for larger scales. The results of some experiments we did with synethic data sets on larger scales justify this assumption.

5. Only CPU clock time could be accurately measured in the system where the experiments were run; we had to estimate the I/O cost using information from disk benchmarks. At the time this paper is being published, we expect the average page access cost (10ms) to have dropped, but so will have the computational power of modern computers, in a similar rate. Thus, our results can be used to draw general conclusions for other system settings.

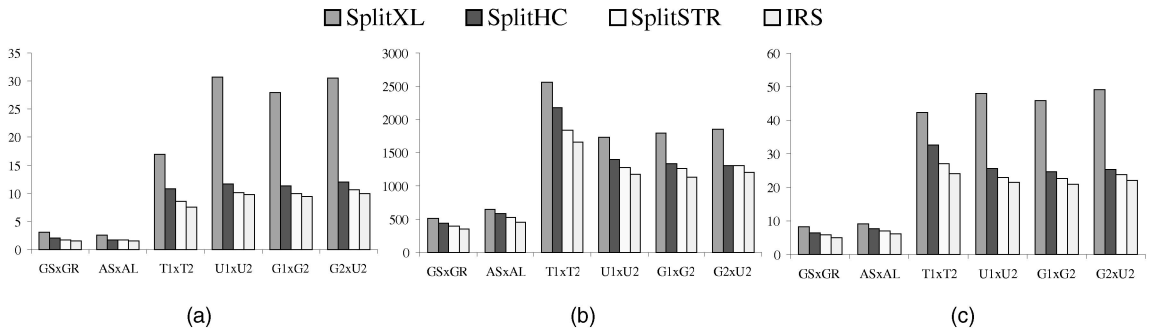

Fig. 11. Performance of SISJ with the four partitioning policies. (a) CPU-time (sec.), (b) page accesses, and (c) overall cost (sec.).

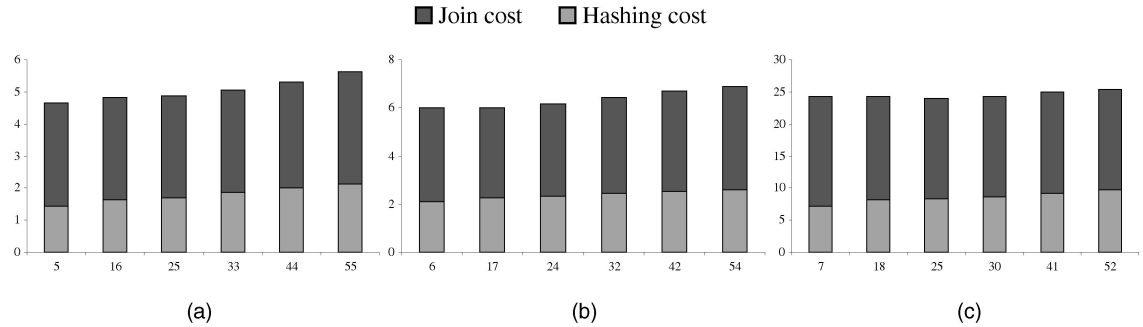

Fig. 12. Overall partition and join cost (in seconds) of SISJ for various values of S. (a) $\mathrm{GS} \bowtie \mathrm{GR}$. (b) $\mathrm{AS} \bowtie \mathrm{AL}$. (c) $\mathrm{T1} \bowtie \mathrm{T2}$.

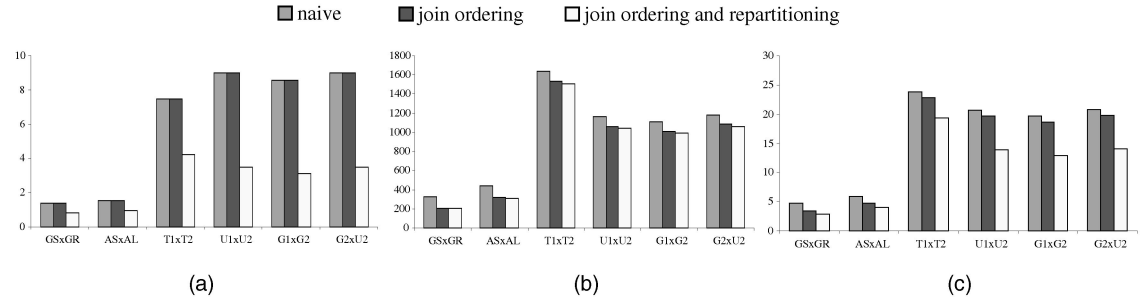

Fig. 13. The effects of the two optimization methods on the performance of SISJ. (a) CPU-time (sec.), (b) page accesses, and (c) overall cost (sec.).

quantitative comparison between these single-index spatial join algorithms because their performance depends on different system and data parameters. Instead, we make a qualitative comparison of the algorithms' performance, which is parametric to the size of the memory buffer for both cases where the nonindexed input is read from disk or it is an intermediate result of another operator. We also discuss the importance of splitting the memory buffer between different join operators in the same query. A method that estimates a good buffer splitting in a query with two spatial joins is provided.

In the analysis, we consider that set A is indexed by R-tree R_A and B contains object identifiers and MBRs, but not whole tuples, to include the case that it may come as a result of another operator's filter step. For simplicity, we assume that the joined data sets contain uniformly distributed rectangles, whose coordinates are normalized to take values from the unit space [0,1). The above is a typical assumption in analytical studies of spatial queries [33], [15], [34]. Table 2 contains a list of symbols to be used in the sequel. In the analytical cost formulae of the algorithms, we assume that the nonindexed input is materialized and read from disk. Notice that we have made the same assumption in the experiments of the previous section. If the nonindexed input is available from an underlying operator, the cost of reading it should be deducted from the corresponding formulae.

4.1 Slot Index Spatial Join

SISJ initially loads the top k levels of R_A in order to find the appropriate slot level. Let $seed_A$ be the number of nodes in R_A from the root until k. The slot index is built in memory, thus no additional I/O is required. Then, the loaded $seed_A$ pages are not needed and may be removed from the buffer. Set B is hashed into the slots requiring P_B accesses for reading. After hashing, the size of B becomes $P_B + r_B P_B - f_B P_B$, where r_B is the fraction of replicated data and f_B is the fraction of filtered data. If we keep the buffer warm, $(P_B + r_B P_B - f_B P_B) - M$ pages have to be written to disk.[6] Thus, the cost of SISJ's partition phase is:

6. When the partitioned set is smaller than M no data are flushed to disk, but let us assume for simplicity that $P_B + r_B P_B - f_B P_B > M$.

TABLE 2
Symbols Used throughout the Analysis of Spatial Join Algorithms

Symbol	*Interpretation*
R_A	R-tree that indexes A
R_B	R-tree constructed for B, by STJ or BaM
$h(R_A)$	height of R_A
C	R-tree node capacity
N_B	number of rectangles in B
M	number of pages in the system buffer
T_A	number of nodes in R_A
P_B	number of sequential pages that fit B ($P_B = \lceil N_B/C \rceil$)
T_B	number of nodes in R_B

$$\mathrm{C}_{\text{SISJ-part}} = seed_A + (2 + r_B - f_B) \cdot P_B - M. \quad (5)$$

Next, the algorithm joins the contents of the slot-bucket pairs. The pages from set A that have to be fetched during the join phase are the remaining $T_A - seed_A$ since the pointers to the slot entries are kept in the slot index and need not be loaded again from the top levels of the R-tree. Moreover, some of these will not be fetched at all if a slot is filtered. We consider the worst case and ignore the possibility of filtered slots for data set A. We also assume that each partition from set B fits in memory, thus the SISJ optimization techniques can be employed. This assumption is driven from our observations on the experiments of Section 3.5, where only in exceptions some bucket from B did not fit in memory; the replication ratio for SISJ when IRS was used was 25 percent on the average. By using the bucket join order heuristic, the partitions with the smallest flushed part will be joined first. If F_B is the number of blocks that are flushed to disk during the join phase of SISJ, the cost of this phase is:

$$\mathrm{C}_{\text{SISJ}-\text{join}} = T_A - seed_A + (1 + r_B - f_B) \cdot P_B + 2 \cdot F_B - M, \quad (6)$$

as all pages flushed before or during the join phase have to be reloaded. Summarizing, the total cost of SISJ is:

$$\begin{aligned} \mathrm{C}_{\text{SISJ}} &= \mathrm{C}_{\text{SISJ}-\text{part}} + \mathrm{C}_{\text{SISJ}-\text{join}} \\ &= T_A + (3 + 2r_B - 2f_B) \cdot P_B + 2F_B - 2 \cdot M. \end{aligned} \quad (7)$$

4.2 Indexed Nested Loops Join

The indexed nested loop (INLJ) algorithm applies N_B window queries to R_A, where N_B is the cardinality of B. The analysis of INLJ presented here is based on cost analysis for window queries and provided in other papers [17], [33], [15]. We assume that the memory buffer organizes pages using an LRU policy. In addition to the size of the joined data sets and the available buffer, the performance of the algorithm also depends on whether data set B is *clustered*, i.e., whether two consecutive window queries are close to each other. For a typical buffer size and when R_A is shallow (the node size is large), a requested intermediate R_A node will be in the buffer with high probability, and most page faults will occur due to fetching of leaf pages. The number of leaf nodes that intersect a rectangle r_B from B depends on the average area of r_B and the average MBR area of a leaf node in R_A [17]. Since for typical R-tree structures (due to the large fanout), the number of leaf nodes is almost equal to T_A (i.e., the total number of R_A nodes), the latter parameter can be estimated by the size of a cell in a grid that uniformly divides the unit space into T_A parts [33]. Let s_B be the average length of an 1-dimensional projection of r_B, and let l_A be the average length of an 1-dimensional projection of a leaf node MBR in R_A; the number of leaf nodes in R_A that intersect r_B is [17]:

$$\mathrm{N}_{\text{intsect(r)}} = T_A \cdot (l_A + s_B)^2 = T_A \left(1/\sqrt{T_A} + s_B\right)^2. \quad (8)$$

Equation (8) assumes that both A and B contain uniformly distributed rectangles in a unit workspace $[0,1)^2$. The number of page faults $\mathrm{C}_{\text{query(r)}}$ generated by a window query is the expected fraction of the $\mathrm{N}_{\text{intsect(r)}}$ nodes that will not reside in the buffer. If the data in B are nonclustered (i.e., two consecutive window queries are not close to each other with high probability), then:

$$\mathrm{C}_{\text{query(r)}} = \mathrm{N}_{\text{intsect(r)}}(1 - \min\{1, M/T_A\}) \quad (9)$$

because the probability of a page that is requested for the second and subsequent times to be in the buffer is $\min\{1, M/T_A\}$. Assuming that the two data sets cover the same area and $N_B \gg T_A$, the first T_A window queries will read all tree pages and the cost of the subsequent ones will be given by (9) [15]. Thus, the total I/O cost of INLJ is:

$$\begin{aligned} \mathrm{C}_{\text{INLJ}} &= P_B + T_A + (N_B - T_A) \cdot \mathrm{C}_{\text{query(r)}} \\ &= P_B + T_A + (N_B - T_A) \cdot T_A \cdot (1/\sqrt{T_A} + s_B)^2 \\ &\quad \cdot (1 - \min\{1, M/T_A\}). \end{aligned} \quad (10)$$

When B is clustered, two consecutive window queries are close to each other with high probability and the last factor of (10) will be close to zero for a nontrivially small memory buffer. In this case, the computational cost $\mathrm{C}_{\text{CPU-INLJ}}$ of the algorithm usually dominates the I/O cost and should be used by a query optimizer. Typically, the rectangles in B will be small enough so that out of the C entries of an intermediate node to be checked for intersection, only one will be followed at the next level. Let $h(R_A)$ be the height of R_A and $\mathrm{C}_{\text{intersection}}$ the cost of a rectangle intersection test.[7] The CPU cost of INLJ is then:

$$\mathrm{C}_{\text{CPU-INLJ}} = N_B \cdot h(R_A) \cdot C \cdot \mathrm{C}_{\text{intersection}}. \quad (11)$$

4.3 Seeded Tree Join

The seeded tree join (STJ) algorithm has common modules with SISJ and their analysis is similar. We charge the same I/O accesses for copying the seed levels, as for determining the slots in SISJ, i.e., $seed_A$. Inserting the data from B under the slots costs $P_B - (M\text{-}seed_A)$ I/Os, as the last $M\text{-}seed_A$ pages will remain in the buffer. For fairness to STJ (since we have assumed that the buckets of B in SISJ fit in memory), we assume that each grown subtree fits in memory. Thus, the growing phase of a subtree costs only reading its parts

7. Four number comparisons are required to verify whether two rectangles intersect.

that are in disk, constructing it in memory, and then writing it back. A similar assumption was done for SISJ, where the hashed data under a slot, or the indexed data from R_A by a slot were assumed to fit in memory. If we follow a similar policy as the bucket join ordering in SISJ (smallest flushed partition first), the number of the blocks F_B in the buffer that need to be flushed while constructing the grown subtrees will be minimized. For instance, if no data under a slot are flushed to disk, the corresponding subtree is constructed first. After constructing the subtree and writing it to disk, more space will be available for the second one. Summarizing the seeded tree construction phase of STJ costs:

$$\begin{aligned} \mathrm{C}_{\text{STJ-construction}} = seed_A + P_B + (P_B - (M\text{-}seed_A)) \\ + 2 \cdot F_B + T_B. \end{aligned} \quad (12)$$

The I/O cost of RJ, i.e., the matching phase of STJ, has been studied in [15], where the following formula has been proposed:

$$\mathrm{C}_{\mathrm{RJ}} = T_A + T_B + (NA(R_A, R_B) - T_A - T_B) \cdot \mathrm{Prob}(node, M). \quad (13)$$

$\mathrm{NA}(R_A, R_B)$ is the total number of R-tree nodes accessed by RJ, and $\mathrm{Prob}(node, M)$ is the probability that a requested R-tree node will not be in memory and will result in a page fault. For typical data set densities $\mathrm{NA}(R_A, R_B)$ depends solely on T_A and T_B [34] and the value $\mathrm{Prob}(node, M)$ can be estimated using an exponential distribution function determined by M. For a nontrivial memory buffer, C_{RJ} is slightly larger than $T_A + T_B$.

In our case, however, we have to consider the last $M\text{-}seed_A$ pages of the constructed subtrees, as well as the $seed_A$ pages of the top k levels of R_B, which will remain in memory (*write-reflect policy* [22]) after the growing phase and some of them may not incur page faults when requested during RJ. The probability that a page is in the buffer when requested is $\min\{1, M/(T_A + T_B)\}$. Thus, the total savings of not reading pages from R_B are $T_B \cdot \min\{1, M/(T_A + T_B)\}$, and the cost of STJ matching phase is:

$$\mathrm{C}_{\text{STJ-matching}} = T_A + T_B \cdot (1 - \min\{1, M/(T_A + T_B)\}). \quad (14)$$

The total I/O cost of STJ from (12) and (14) is thus:

$$\begin{aligned} \mathrm{C}_{\mathrm{STJ}} = 2 \cdot seed_A + 2 \cdot P_B - M + 2 \cdot F_B \\ + T_A + T_B \cdot (2 - \min\{1, M/(T_A + T_B)\}). \end{aligned} \quad (15)$$

4.4 Build and Match

Build and match (BaM) involves the bulk loading of a normal R-tree for the nonindexed input B (not a seeded one, like in STJ). Several methods can be used for sorting the rectangles in order to build the packed R-tree, as discussed in Section 2.1. In this analysis and in the implementation of BaM, we consider *sort tile recursive* (STR) because it is a simple method, which as shown in [19], builds packed R-trees of good quality. Methods proposed in [8], [6] are more complicated and it is questionable whether they produce better indexes. An extensive comparison of bulk loading techniques is out of the scope of this paper.

The cost of BaM is rather simple to estimate. It includes the cost of external sorting and the cost of tree matching. The cost of external sorting is $P_B \cdot (2 \cdot \lceil log_{M-1}(P_B/M) \rceil + 1)$ [32]. Typically, each group of $\lceil \sqrt{P_B} \rceil$ pages formulated after the last x-merge fit in memory and y-tile sorting can be performed without extra I/Os. After building R_B, if a write-reflect policy is followed, like in STJ, the last M tree pages will remain in the buffer. The I/O cost of BaM is then:

$$\begin{aligned} \mathrm{C}_{\mathrm{BaM}} = P_B \cdot (2 \cdot \lceil \log_{M-1}(P_B/M) \rceil + 1) + T_A + T_B \\ \cdot (2 - \min\{1, M/(T_A + T_B)\}). \end{aligned} \quad (16)$$

4.5 Sort and Match

Sort and Match (SaM) sorts the rectangles from B using STR and applies a window query to R_A for each page of consecutive rectangles in the sorted output. If B fits in memory, sorting can be performed without any extra I/Os and matching is cheap because after matching one page, the space allocated for it can be appended to the LRU buffer. However, if B does not fit in memory and external sorting is required, SaM needs to allocate one page for each run prior to the final merge plus $\lfloor \sqrt{P_B} \rfloor$ pages from the join buffer for the y-tile sorting required at STR, limiting thus the memory dedicated for matching to $M - \lceil P_B/M \rceil - \lfloor \sqrt{P_B} \rfloor$. The cost of SaM's match phase can be estimated using the same analysis as in INLJ, but considering as rectangles of B the extents of the P_B pages. More specifically, P_B window queries are applied to R_A, each with an average 1D projection of length $1/\sqrt{P_B}$. Assuming uniform distribution of these queries and that P_B is in the order of T_A, the whole R_A will be read. If P_B is larger than T_A, additional page faults will occur. In summary, the I/O cost of SaM is the cost of external sorting plus the cost of the match phase, derived after changing M, s_B, and N_B:

$$\begin{aligned} CSaM = P_B \cdot (2 \cdot \lceil \log_{M-1}(P_B/M) \rceil + 1) + T_A \\ + \min(0, P_B - T_A) \cdot \left(1/\sqrt{T_A} + 1/\sqrt{P_B}\right)^2 \\ \cdot \left(1 - \min\{1, \left(M - \lceil P_B/M \rceil - \left\lfloor \sqrt{P_B} \right\rfloor\right)/T_A\}\right). \end{aligned} \quad (17)$$

Due to STR sorting, two consecutive window queries will be close to each other with high probability [28] (i.e., the queries are clustered). Therefore, the probability that an incoming window query causes a page fault will be close to zero if a nontrivial memory part is allocated for matching. In this case, the cost of SaM drops to the cost of external sorting and reading R_A and the algorithm becomes very efficient in terms of I/O, assuming that not many levels of sorting and merging are required.

4.6 A Qualitative Comparison of the Algorithms

Comparing the algorithms from the above analysis is not straightforward due to the different parameters in the cost formulae. We will make a qualitative comparison for two cases: 1) when the nonindexed file is read from the disk and 2) when it comes as a result of another operator.

Let us first assume that the nonindexed input is read from disk. The I/O performance of INLJ is more sensitive to the sizes of the data sets than to the buffer size; INLJ can be

the most efficient algorithm if N_B and T_A are small, but in general (i.e., if $T_A > M$) its cost depends on the clustering of B and is expected to be very high. Furthermore, as discussed in Section 4.2, the computational cost of the algorithm is very high for large data sets, making its application prohibitive. For the rest of the algorithms, we distinguish two cases, depending on the relative sizes of the nonindexed input and the available buffer.

First, let $P_B < M$, i.e., the nonindexed input fits in memory and external sorting can be avoided for BaM and SaM. The replication factor of SISJ decides the dominant algorithm. If r_B is small enough for $P_B + r_B P_B - f_B P_B$ to fit in memory, then no materialization is required for SISJ and both SISJ and SaM have optimal I/O cost ($C_{opt} = P_B + T_A$). If $P_B < M < P_B + r_B P_B - f_B P_B$, SaM is expected to outperform SISJ. BaM will be more costly than SaM due to the tree construction, except for the case where both data sets fit in memory and the tree can be constructed and maintained without extra I/O. The same applies for STJ; in general, the algorithms that build trees (STJ and BaM) have optimal cost when $T_A + T_B < M$. Otherwise, they should be outperformed by SaM and SISJ, given a typical replication factor (20 percent - 40 percent).

When $P_B \geq M$, sorting algorithms like SaM and BaM cannot avoid external sorting, which may lead to a large overhead if multiple passes of the data are required. We consider the case when M is large enough for only one merge to be required. The sorting cost is then $3 \cdot P_B$. Even for a large replication ratio this is usually larger than $(3 + 2r_B - 2f_B) \cdot P_B + 2 \cdot F_B - 2 \cdot M$, i.e., the hashing and join cost of SISJ, when P_B is not much larger than M. In this case, the flushed parts in each partition will be small and so will be F_B during join. The cost of SISJ is expected to increase linearly as the buffer size decreases because of the linear decrease of the in-memory partitions. On the other hand, since two consecutive matches at SaM are close to each other with high probability and due to the LRU buffer dedicated for matching, the cost of SaM will be the cost of external sorting plus T_A for a wide range of memory values. Only when the number of pages, $M - \lceil P_B/M \rceil - \lfloor \sqrt{P_B} \rfloor$, available for matching becomes very small, the join cost ceases to be trivial and many I/Os during matching will occur, rendering the algorithm expensive. If P_B is much larger than M, hash-based algorithms (i.e., SISJ, STJ) have an advantage in terms of I/O over ones that use external sorting (i.e., SaM, BaM) mainly because the latter methods have the additional overhead of y-tile sorting, which reduces the available memory for sorting and causes multiple passes of the files. It must be noted than STJ could outperform SISJ if P_B is much larger than M and extensive replication occurs in SISJ.

When data set B is a result of another subquery, the performance of the algorithms decreases for two reasons. First, the buffer needs to be shared between query operators and, therefore, the available memory M for the spatial join may be limited. Second, algorithms that require the whole input B to be available in the beginning cannot be directly applied without materialization of B (*blocking* operators). The most suitable algorithm for pipelining intermediate results is INLJ. The buffer can be shared between any number of operators and multiple joins can be executed without any materialization of intermediate results.[8] Nevertheless, INLJ is too expensive when the joined data sets are large. In case of SISJ and STJ, the slots can be determined prior to the existence of B and the intermediate results can be partitioned while produced. SISJ maintains the advantage of being fast while demanding a flexible amount of memory for pipelining since only S pages need to be allocated for the partitioning phase. Although STJ can also follow the same procedure, it does not have the flexibility of choosing the number of slots, which can lead to a bad memory and disk space utilization. BaM and SaM apply external sorting before matching, requiring the whole input B to be available. This situation can be avoided when a fraction of the memory buffer is dedicated to create sorted runs for a part of intermediate results. If the number of runs written to disk is smaller than M, no extra I/Os occur due to the unavailability of the results. In the next section, we present a method that approximates a good buffer sharing when three indexed inputs are joined using pipelining between two join operators.

TABLE 3
Cost of INLJ Depending on the Nature of the Nonindexed Input

INLJ	*clustered data*		*non-clustered data*	
size of B (in tuples)	CPU	I/O	CPU	I/O
very small (e.g., 1K)	low	low	low	high
moderate, large (>10K)	high	low	high	high

As a general conclusion, SISJ and SaM will typically outperform STJ and BaM because they avoid the I/O consuming tree construction. Whether SISJ is better than SaM depends on the available memory and the characteristics of the data sets, which affect the replication introduced by SISJ. INLJ is a good option only if its CPU cost, as estimated by (11), is smaller than the cost of the other alternatives and either input B is clustered or $T_A < M$. Finally, SISJ and STJ are more suitable than SaM and BaM when the nonindexed input comes from another operator because they can hash the intermediate results immediately at the time they are produced. The validity of this qualitative comparison is evaluated in the next section through experiments. Table 3 and Table 4 summarize the conclusions of this analysis.

4.7 Analysis of a 2-Join Query Case

In this section, we study in detail the issue of optimal sharing the memory buffer between a single-index algorithm and an underlying operator that provides the nonindexed input. We deal with the special case where three data sets A, B, and C indexed by the R-trees R_A, R_B, and R_C are joined in a centralized uniprocessor environment that uses pipelining. Let A *overlap* $B$and B *overlap* C be the join conditions. A possible way[9] to execute the query is to first execute $B \bowtie C$, and then join the intermediate

8. INLJ can be very efficient when a small number of query results is required because it never blocks the data flow between pipelined operators.

9. Query optimization algorithms that detrmine the best execution plan for a multiway spatial join are presented in [23], [24].

TABLE 4
Performance of Single-Index Algorithms in Various Conditions

Condition	STJ	BaM	SaM	SISJ
$T_A + T_B < M$	optimal	optimal	optimal	optimal
$P_B < M < P_B + r_B P_B - f_B P_B$	average	average	optimal	good
$P_B + r_B P_B - f_B P_B < M$	average	average	optimal	optimal
$P_B \geq M$	bad	bad	average	good
$P_B >> M$	average	bad	bad	good
B comes from another operator	average	bad	bad	good

results with A. RJ is employed for $B \bowtie C$ but, since the intermediate results are not indexed, one method from the single-index join class has to be applied for the final join. STJ and BaM are omitted for simplicity because they can be considered similar to SISJ and SaM, respectively.

The performance of the 2-join query is crucially affected by how the available memory is shared between the operations. INLJ is more flexible in the sense that an intermediate result of $B \bowtie C$ can immediately be probed against R_A and no materialization is required. Thus, the buffer can be shared between RJ and INLJ with various schemes, including a unified one where both operators share the same buffer. On the other hand, SaM and SISJ require the whole intermediate result $B \bowtie C$ to be sorted or hashed before the final join. When SaM is considered for the final join, while executing $B \bowtie C$ it is natural to dedicate one part of the buffer for the join and the rest for maintaining in memory and sorting large parts of intermediate results in order to avoid reading them twice during external sorting. After $B \bowtie C$ has finished, the sorted runs are loaded, merged, and probed against R_A, provided that there is enough space for this operation. If there is not enough space, SaM may require an extra level of sorting and merging. When SISJ is considered for the final join, at least S pages have to be allocated for hashing the intermediate results of $B \bowtie C$ and at most $M - S$ pages can be dedicated to RJ. If B and C are very small, but expected to produce many results, more than S pages should be dedicated for hashing the results, as this would minimize the number of flushed pages while hashing. The whole buffer can be dedicated to the join phase of SISJ. Fig. 14 summarizes the buffer allocation scheme for the query when each of the three algorithms is applied at the final join.

The buffer splitting for the 2-join query is determined by three factors; the sizes of B and C, the size of the intermediate result, and the size of A. In the sequel, we describe how these factors affect the performance of the three methods and provide a methodology that estimates a good buffer splitting between the operators.

The cost of RJ is affected by the sizes of B and C and the number P_{RJ} of buffer pages allocated for the operator (i.e., M in (13)). The size of the intermediate result is important for all three algorithms for different reasons. Let P_{INT} be the number of pages that can fit the intermediate result of $B \bowtie C$, and N_{INT} the number of tuples in it. INLJ applies N_{INT} window queries at R_A, and this number mainly affects the CPU cost of the algorithm since the intermediate results are clustered. When RJ-SaM is applied, P_{INT} will determine the number of sorted runs produced from the results of RJ. A large P_{INT} in combination with a large P_{RJ} will create many short runs, which will occupy a significant part of the buffer during merging and matching. Moreover, a large P_{INT} will require many buffer pages to be dedicated for the y-tile sorting prior to matching, rendering the algorithm inefficient. SISJ is only affected when P_{INT} is very large and no bucket from $B \bowtie C$ can fit in memory in order to be repartitioned. In this case, the subtrees of R_A under the slots must be small enough so that nested loops for each join pair can be efficiently applied; otherwise, SISJ should be called recursively.

N_{INT} (and P_{INT}) can be estimated using catalog information from the joined data sets. Following the analysis in [34] and [15], the number of output tuples when joining data sets B and C is:

$$N_{INT} = N_B \cdot N_C \cdot (s_B + s_C)^2, \tag{18}$$

where s_X is the average length of a 1-dimensional projection of a rectangle in a data set X, and the rectangle coordinates are normalized to take values from [0,1). In other words, the size of $B \bowtie C$ is the number of rectangles in B intersected by an average rectangle in C, multiplied by the number of rectangles in C. Given the density D_B of set B, s_B can be calculated from:

$$s_B = \sqrt{D_B / N_B}. \tag{19}$$

When joining files with nonuniform distributions, (18) does not provide accurate join size estimation. In such cases, a method that divides the workspace into a grid of

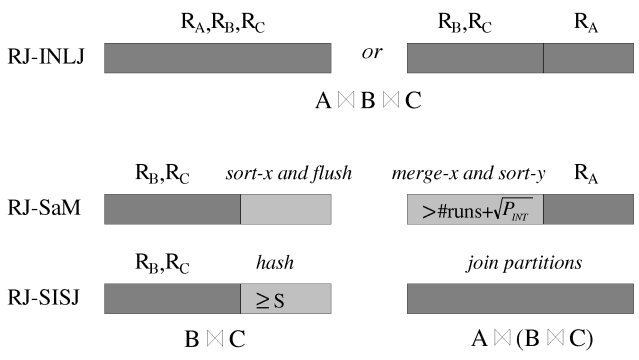

Fig. 14. Buffer splitting policies for cascading joins.

equal sized cells and uses statistical information for each cell to compute the size of the spatial join [24] can be employed. Achaya et al. [3] propose a method for estimating the selectivity of window queries, which can be more accurate of the one in [24]. However, it cannot be directly applied for spatial joins since the data space is decomposed irregularly and the decomposition of the two data sets may vary significantly.

Finally, the size of R_A affects the buffer splitting mostly in INLJ, where the intermediate results are immediately probed against R_A. If R_A is small, not much memory is required for the window queries, even when P_{INT} is large. Estimating the optimal buffer splitting at RJ-INLJ is not easy due to the clustering of intermediate results. SaM is also affected by the size of R_A, but not as much as by the size of the intermediate result. The cost of SISJ depends on R_A only when P_{INT} is very large, as discussed above.

When using the above analysis, a good buffer split can be determined by the characteristics of the three joined data sets. We propose an optimal buffer split estimation method as follows: The splitting space is approximated by a discrete number of splits and, for each one, the I/O cost is estimated for the chosen algorithm (RJ-INLJ, RJ-SaM, or RJ-SISJ). The approximation that gives the smallest cost according to (13) and (18) and the corresponding single-index join cost formula for the final join method is then chosen for execution. In general, for arbitrarily complex queries, the optimal buffer splitting between pipelined join operators should be applied to the execution plan determined by the query optimization algorithm.

Another optimization issue is when to apply the refinement step of the query [27]. In the 2-join case analyzed above, the refinement could be applied in two steps: once after RJ and once after the single-index join algorithm, or only once for each triple that qualifies the filter step. If the final result is estimated to be much larger than the intermediate one and there is only one join predicate from *A* to *B* or *C*, the refinement should be performed in two steps; otherwise, it should be performed only once. In the future, we plan to investigate on optimizing the order of the filter and refinement step in complex spatial queries.

5 Experimental Evaluation

We implemented all single-index join algorithms, namely, INLJ, STJ, BaM, and SaM and employed the data sets presented in Table 3 for their experimental comparison. STJ was implemented in a slightly different way than the one proposed in [22]. We do not incrementally insert the rectangles into each grown subtree, but build the trees bottom-up using the STR bulk loading method [19]. In this way, 1) the resulting grown subtrees are smaller (as they are packed), 2) the memory requirements at construction are minimal (only *h* blocks have to be maintained in memory, where *h* is the height of the grown subtree), and 3) each subtree is expected to fit in memory and external sorting is required in few cases. The algorithms are compared in both cases where the nonindexed input is read from disk or it is a result of another operator.

5.1 Input B is Read from Disk

In most experimental studies that include single-index spatial join algorithms [18], [21], [22], [26], the nonindexed inputs are read from sequential files. In order to comply with this typical method, we first assume that input *B* is read from the disk. In the first set of experiments, we keep the memory buffer fixed to 512K and evaluate the algorithms using different page sizes. Fig. 15 illustrates their performance in terms of CPU time, I/O accesses, and normalized overall cost for various join pairs and page sizes. For each joined pair, the nonindexed data set is the second one. In most cases, the cost of INLJ was too high and its value is shown on the top of the corresponding bar. Wherever there is no value for STJ, the algorithm was inappropriate due to buffer limitations (i.e., the seeded tree could not be constructed without buffer thrashing). This problem could be fixed by changing the original algorithm to group the entries at the seed levels into a smaller number of slots. However, this would change the nature of the algorithm since the seeded R-tree would have a different structure than the existing one. Moreover, the height of the seeded tree would grow and, as suggested in [5], large differences in the heights of joined R-trees increase the overhead of RJ (see also the performance of BaM in some cases of Fig. 15).

SISJ outperforms the rest of the algorithms in most cases, while SaM is a close second. In general, SISJ is better than SaM in terms of CPU time because the number of matches between R-tree leaf nodes and repartitioned bucket parts is minimal (as many as the number of leaf nodes). On the other hand, SaM matches each produced leaf node of R_B with every R_A leaf node that intersects it. The performance of SISJ degrades only when the replication factor is too large and the size of *B* increases significantly after hashing. In $\mathrm{GS} \bowtie \mathrm{GR}$, for instance, SISJ is outperformed by SaM because GR fits in memory ($P_{GR} < M$), but due to the replication factor ($r_{GR} = 23.8\%$) the $(1 + r_{GR} - f_{GR}) \cdot P_{GR}$ pages required for the partitioned data in SISJ do not. Therefore, SISJ flushes pages to disk having larger overhead than SaM. In $\mathrm{T1} \bowtie \mathrm{T2}$ and for small pages, the replication factor of SISJ is also large (for 1K, 2K, 4K $r_{T2} > 40\%$, for 8K $r_{T2} = 30\%$), and the algorithm is again outperformed by SaM.

STJ is never better than SISJ and BaM is never better than SaM. In most cases, STJ outperforms BaM in terms of I/O, but it is slightly worse in terms of CPU time. The bulk loading version of STJ overcomes the large CPU time overhead observed in [28]. Notice, however, that STJ cannot be applied in half of the experiments involving 4K or 8K pages. This is a great drawback of the algorithm as large page sizes are common in database systems. BaM behaves very poorly in the $\mathrm{AS} \bowtie \mathrm{AL}$, 4K, and $\mathrm{T1} \bowtie \mathrm{T2}$, 8K cases because the resulting packed trees have smaller height than R_A, producing a very large number of CPU intensive window queries.

Although INLJ has very good I/O performance when the size of the data sets is smaller or slightly larger than *M*, it is always much worse than the other alternatives in terms of CPU time. For the large data sets, INLJ is I/O bound,

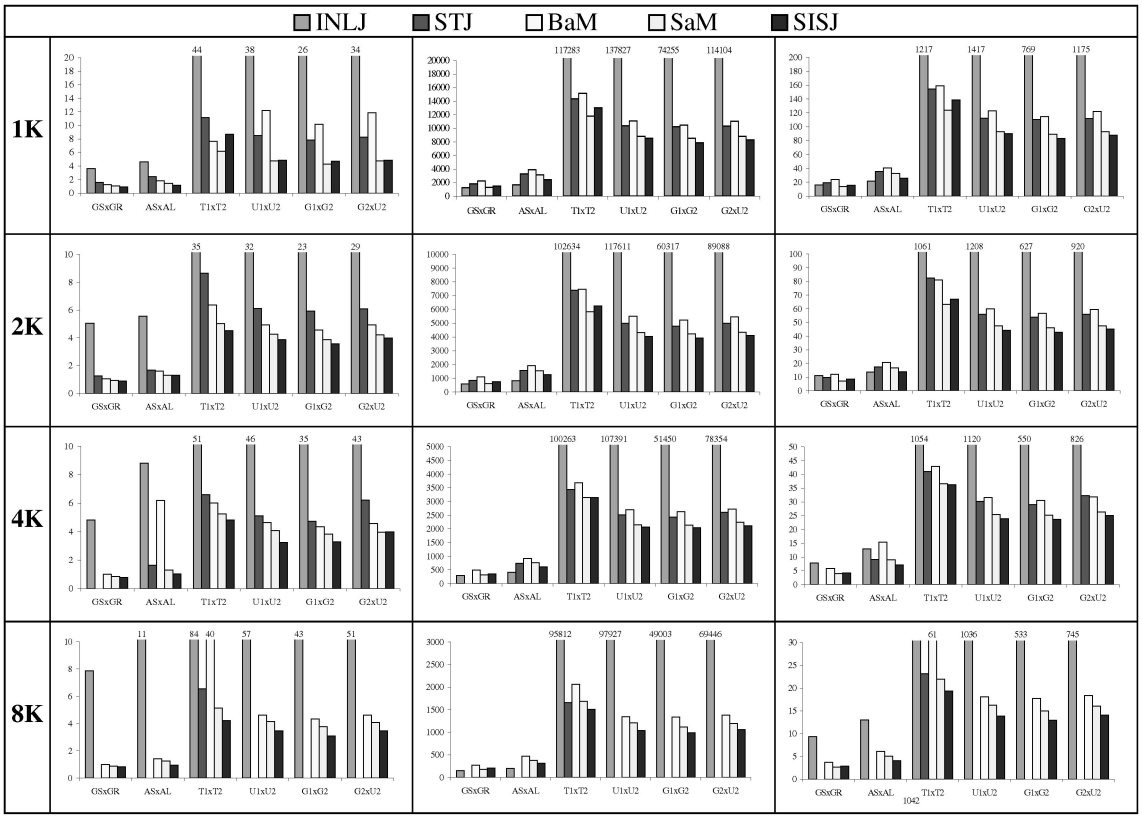

Fig. 15. Experimental comparison of the single-index, join algorithms for a memory buffer of 512K. (a) CPU time (sec.). (b) I/O accesses. (c) Overall cost (sec.).

whereas at GS $\bowtie$ GR and AS $\bowtie$ AL it becomes CPU-bound for large page sizes due to the good utilization of the LRU buffer. The I/O cost decreases as the page size increases, because R_A becomes shallower. On the other hand, the computational cost increases with the page size as opposed to the other algorithms, which are favored by large pages. Since the extents of the window queries are small, a single path from the root to the leaves is followed by most queries. The CPU cost of the query, which is proportional to the node capacity (see (11)), is not counterbalanced by the height of the tree, which does not decrease with the same rate as the node capacity.

In order to test the I/O behavior of the algorithms, we also conducted experiments using various buffer sizes. Fig. 16 illustrates the I/O cost for four join pairs, and a 4K page size. The replication factor r_B at SISJ for each pair is also given. The I/O cost of INLJ for T1 $\bowtie$ T2 and U1 $\bowtie$ U2 is presented separately in Table 5 due to its very large values. At GS $\bowtie$ GR, STJ was only applicable for buffers larger than 640K.

The main observation is that the cost of STJ and SISJ decreases almost linearly with the buffer size. On the other hand, BaM and SaM have constant performance for small ranges of buffer sizes. The effects are larger for SaM, which after a given buffer threshold causes extra page faults only due to external sorting. As soon as set B fits in memory, the cost of SaM falls to almost its optimal value. On the other hand, SISJ, due to replication, does not converge to the optimal cost as fast as SaM. Nevertheless, the stable performance of SISJ makes it the best algorithm when B is larger than the available memory, with the exception of the case when the replication factor is very large (e.g., T1 $\bowtie$ T2), where it loses the lead from SaM for some buffer values. It is worth noticing that when the nonindexed input is slightly larger than the available buffer, STJ outperforms SaM. This can be explained by the fact that SaM does a bad memory management by sorting (externally) the whole data set. If, for example, set B were broken in two parts and SaM were applied for each part (avoiding external sorting), the cost of the algorithm would possibly decrease. For very small buffers, BaM and SaM behave very badly due to the introduction of extra sorting and merging levels.

The I/O cost of INLJ depends mainly on T_A and P_B. If T_A is smaller than the buffer (e.g., U1 $\bowtie$ U2, buffer size > 2816K) no extra I/Os will occur and the algorithm has optimal performance. When T_A is larger than the buffer the cost depends on the size and the distribution of the set B; when P_B is small, as in GS $\bowtie$ GR and AS $\bowtie$ AL, the I/O cost converges to its optimal value very fast. On the other hand (see Table 5), when the size of P_B is large and the data are not highly clustered, the I/O cost converges to its optimal value slowly. However, even for very clustered data, the CPU cost of INLJ is so high (see Fig. 15) that renders it inapplicable for joining large data volumes.

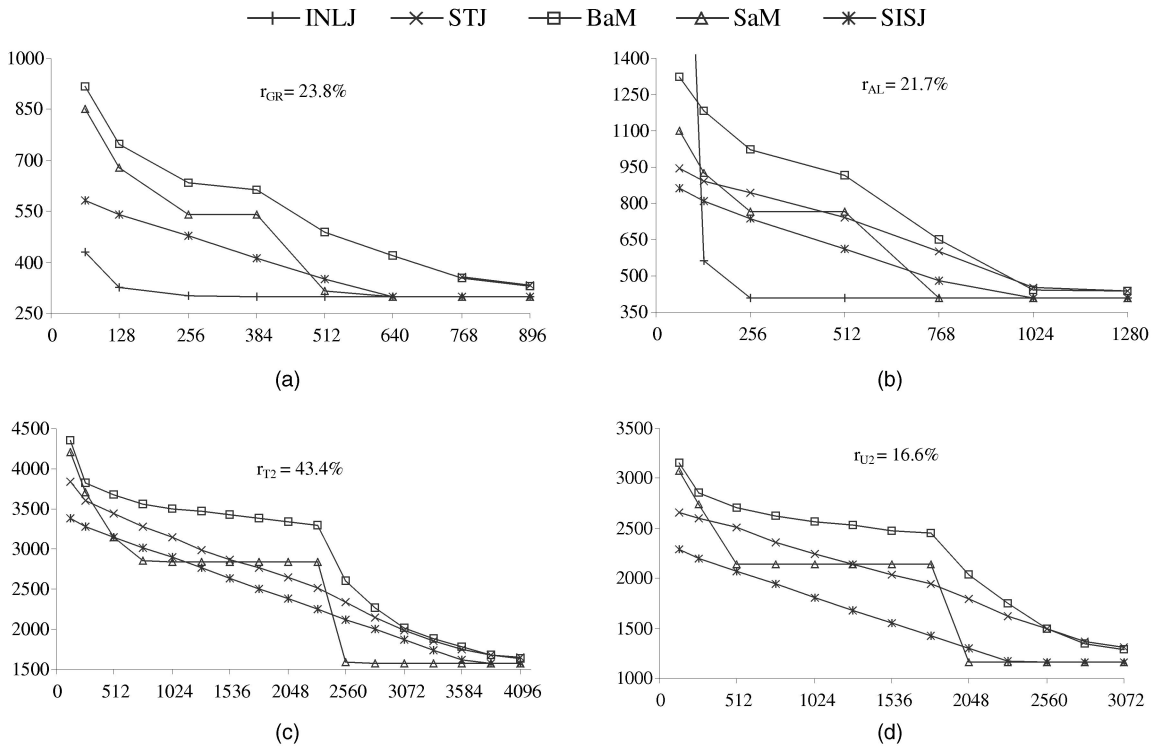

Fig. 16. Page accesses as a function of the buffer size (in Kbytes). (a) $\mathrm{GS} \bowtie \mathrm{GR}$. (b) $\mathrm{AS} \bowtie \mathrm{AL}$. (c) $\mathrm{T1} \bowtie \mathrm{T2}$. (d) $\mathrm{U1} \bowtie \mathrm{U2}$.

5.2 Input B comes from an Underlying Join Operator

We tested the relative performance of the three methods by running the algorithms for 2-join queries $A \bowtie (B \bowtie C)$, varying the size of the intermediate result. We experimented with RJ-INLJ, RJ-SaM, and RJ-SISJ, skipping STJ and BaM because they are inferior to SISJ and SaM in all tested cases of the previous section. A series of data sets with 100,000 uniformly distributed rectangles was generated for the experiments. The densitiy of A was set to 0.5 and the density of B to 0.1. Four versions of C were used with densities 0.04, 0.1, 0.2, and 0.33 so that the number of tuples in the intermediate result $B \bowtie C$ is 25,000, 50,000, 100,000, and 200,000, respectively. During the experiments, the buffer size was set to 512K and the page size to 8K. Fig. 17 shows the I/O cost of the three methods for the four versions of data set C, as a function of the buffer percentage allocated for RJ (P_{RJ}/M).

In the current experimental settings, RJ-INLJ achieves the best I/O performance when the buffer is evenly shared between RJ and INLJ. For small values of P_{RJ}, the I/O cost of RJ increases exponentially, and the same applies for the window queries when P_{RJ} is large. Surprisingly, the cost of the algorithm for the 25K intermediate result was larger than that of the 50K. This shows that when the intermediate result is clustered the I/O cost of the algorithm is not significantly affected by its size. If both RJ and INLJ used a unified memory buffer, the cost would be 1,886, 1,880, 2,043, and 2,155 page accesses for the four versions of C. These values are slightly worse than those of the best buffer splitting (around 50 percent in Fig. 17a).

RJ-SaM is not affected by the size allocated for sorting the runs, except from the case where one page for each run plus the amount of data needed for the y-tile sorting do not fit in memory. In this case, an extra level of merging, which greatly degrades the performance of the algorithm, is required (see $P_{RJ} \geq 30\%M$ and $N_{INT} = 200\mathrm{K}$, $P_{RJ} \geq 80\%M$, and $N_{INT} = 100\mathrm{K}$ in Fig. 17b). In general, extra merges should be avoided wherever possible by tuning the buffer split correspondingly. This tuning can be done following the methodology proposed in Section 4.7. In the current experimental setting, the largest possible P_{RJ} that avoids extra merges achieves the best performance. Even though a simpler sorting scheme [30] that avoids the memory demanding y-sorting could be engaged for SaM, it is expected to be worse due to the bad quality of the produced packed nodes [19]. We also experimented by applying SaM using Hilbert values for

TABLE 5
Page Accesses of INLJ as a Function of the Buffer Size (in Kbytes)

buffer(Kb)	256	768	1280	1792	2304	2816	3328	3840
T1 ⋈ T2	124449	81507	54221	35214	21474	11771	4865	1579
U1 ⋈ U2	120852	94064	67587	41482	16604	1156	1156	1156

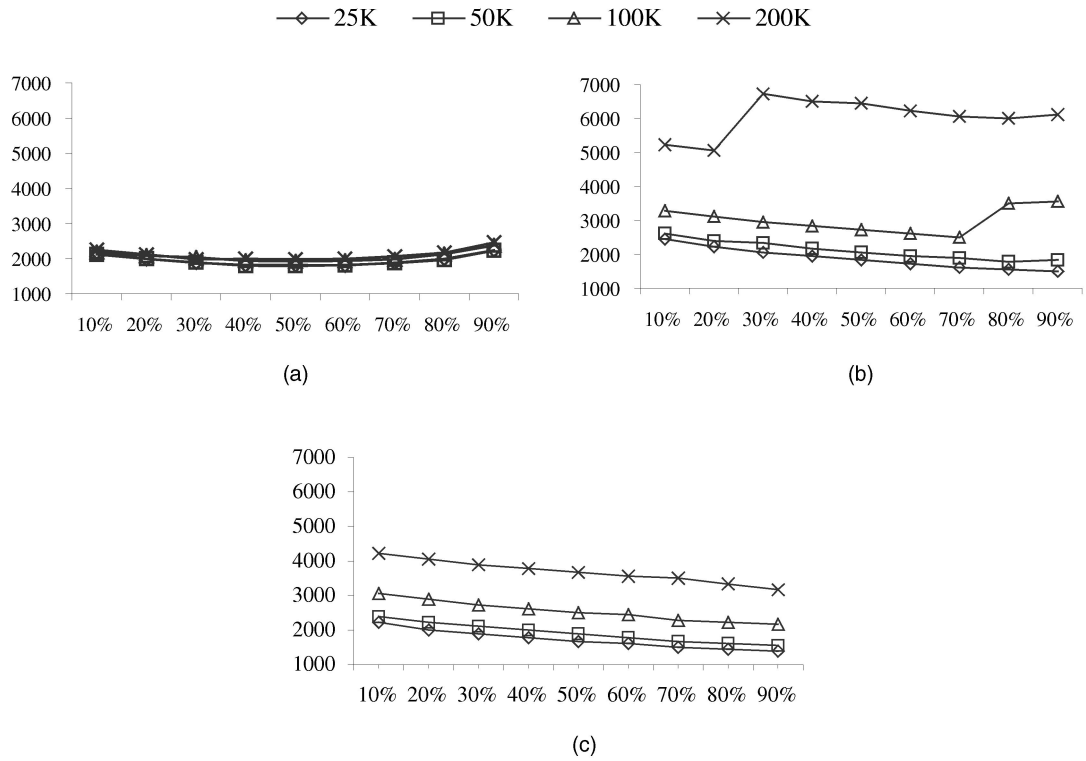

Fig. 17. I/O cost of the three alternatives as a function of the memory percentage allocated for RJ. (a) RJ-INLJ. (b) RJ-SaM. (c) RJ-SISJ.

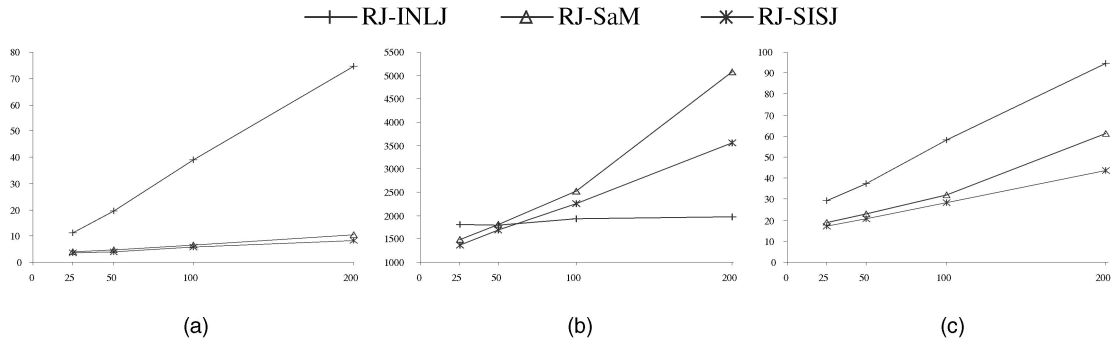

Fig. 18. Cost of the three methods when optimal splitting is chosen as a funstion of N_{INT} (in thousands). (a) CPU time (sec.). (b) I/O accesses. (c) Overall cost (sec.).

sorting [17], but found it to be very expensive due to the CPU-intensive calculation of the sorting key.

SISJ shows the most robust behavior to buffer splitting. The largest P_{RJ} gives the best I/O performance in all cases. This is due to the fact that the profits in RJ are larger than the gain from avoiding flushing extra pages in the partitions to disk. A small S achieves the best I/O performance. There were no cases where the some bucket or slot data did not fit in memory at a match, even when $P_{RJ} = 90\%M$, where S took a small value. However, for $P_{RJ} \geq 70\%M$ and $N_{INT} = 200\text{K}$ and $P_{RJ} \geq 80\%M$ and $N_{INT} = 100\text{K}$, many buckets from set B did not fit in memory and the repartitioning heuristic could not be applied. For these buckets, the join was performed by applying indexed nested loops using the subtree under the corresponding slot, resulting in considerable CPU overhead. In these cases, the best overall cost is achieved when $P_{RJ} = 60\%M$ and $70\%M$, respectively. As a general conclusion for RJ-SISJ, RJ should be given the largest possible buffer part which leads to a number of slots S formulating buckets whose average size is smaller than the available buffer. The optimal buffer splitting for this case can also be approximated using the methodology proposed in Section 4.7.

Fig. 18 presents the cost for each algorithm, when the optimal buffer split is chosen, as a function of the intermediate result size. Due to the clustering of the intermediate results, the CPU overhead of INLJ dominates the I/O cost and is analogous to N_{INT}, i.e., the number of window queries to be executed. The computational costs of SaM and SISJ are almost the same and do not increase as fast as that of INLJ due to the efficiency of the matching method (forward sweep). SISJ retains an advantage over SaM in I/Os, which becomes larger when P_{INT} is much larger than the available buffer; SaM, in this case, cannot dedicate a large part of the buffer for RJ because it requires fewer large sorted runs in order to apply the y-tile sorting in

memory (see Fig. 17b). The I/O cost of both algorithms exceeds the one of INLJ for large intermediate results because they cannot avoid materializing them. Nevertheless, the materialization is worth, especially for SISJ, which as shown in Fig. 18c, preserves a constant advantage over INLJ.

As a general conclusion, SISJ is more appropriate than SaM for pipelined joins because of its large flexibility to the buffer size. INLJ is good only for relatively small data sets and intermediate results (<30,000 objects), due to its high computational cost. It should be used when 1) R_A is not much larger than the memory dedicated for the window queries, 2) the probed input is clustered, and 3) its computational cost is comparable to the I/O cost of the best alternative (e.g., SISJ). Nevertheless, INLJ is a fully pipelined and parallelizable algorithm that avoids materialization and it is the most suitable when a small part of join results are required. Other schemes that may perform well under conditions include partial sorting of the intermediate results with respect to the key of the rectangle to be joined next, in order to avoid extra window queries by INLJ and application of SaM for each part of intermediate result that fits in memory, in order to avoid external sorting.

6 Conclusions

Like its relational counterpart, the spatial join is a time and resource demanding operation, which calls for efficient processing methods. This paper explores the application of *slot index spatial join* (SISJ), a hash join algorithm that achieves very good performance when computing joins in the presence of a single R-tree. SISJ builds a structure, called *slot index*, over the existing R-tree by grouping the entries at a specific level into S groups called slots. All data from the nonindexed input are hashed into buckets with same extents as the slot MBRs. The hash-buckets are finally joined with the R-tree data under the corresponding slot. We propose two optimization methods that achieve overall performance improvement up to 35 percent compared to the first implementation of SISJ [24].

SISJ is compared both analytically and experimentally with four alternative methods that can be applied for processing spatial joins in the presence of one R-tree, namely, the *indexed nested loop join*, the *seeded tree join*, the *build and match* join, and the *sort and match* method. The results show that SISJ achieves constantly good performance and outperforms other methods in most cases. INLJ is good only when the joined data sets are relatively small where its computational cost does not explode to high levels. SaM is the best alternative when the replication introduced by SISJ is large and when the nonindexed input is slightly smaller than the available buffer. On-the-fly building of an R-tree by using either bulk loading or the seeded tree method is not recommended in any case. The spatial join algorithms are also compared when used as operators at complex spatial query processing, through a case study where three inputs indexed by R-trees are joined. SISJ is the most suitable method for this case because it can immediately hash intermediate results, whereas SaM needs to materialize them first, in order to sort them. The experiments show that good buffer splitting between cascading query operators is crucial for the I/O cost of a complex query and a method that estimates an optimal buffer splitting for an execution plan is presented.

Throughout the experiments and discussions in this paper, we did not consider the difference between random and sequential I/O accesses. We have also normalized the CPU and I/O cost into the same scale, providing thus a uniform "overall" cost. The reason for these assumptions is that we wanted to keep the comparison model of the algorithms simple without confusing the reader with extra parameters about the results. In any case, the relevant performance of the algorithms and the optimization methods does not change after the distinction of random and sequential I/Os. A typical search in the R-tree would result in mostly random I/Os and a sorting or hashing operator in mostly sequential I/Os. All methods include a sorting or hashing module (i.e., the hash phase of SISJ, the seed phase of STJ and the sort phase of SaM and BaM) and a tree matching or tree searching module (i.e., the join phase of SISJ, the RJ phase of STJ and BaM, and the match phase of SaM), which cause the percentage of random and sequential I/Os to be about the same. The only exception is INLJ, which performs mostly random I/Os, but as shown in Sections 4 and 5, the algorithm already presents a totally different behavior from the other methods, which would not change after the consideration of sequential I/Os. It is an issue of future work to test the relative performance between SISJ and the algorithm in [2]. Intuitively, the algorithm presented there may perform better than SISJ when the nonindexed data are read from disk because of its powerful sorting modules and the careful trace of the R-tree. However, this algorithm loses power when combined with other operators (i.e., when the nonindexed input is an intermediate result) because of the general weakness of sorting-based methods to utilize the nonindexed input at the time it is produced.

Summarizing, SISJ is a robust spatial join algorithm that achieves good performance, based on the following properties:

1. it avoids the expensive on-the-fly building of an R-tree,
2. the partitions of the indexed data set are decided upon the tree structure and hashing it is avoided,
3. the partitions of the build input are guaranteed to have, approximately, the same number of objects, as they point to almost the same number of R-tree entries; skewed data are thus handled very efficiently,
4. it adapts to limited memory resources, and
5. it is suitable as a module of an execution engine that uses pipelining because it can immediately hash produced intermediate results.

Acknowledgments

This research was performed while Nikos Mamoulis was with the Hong Kong University of Science and Technology and revised while he was with CWI, the Netherlands. It was supported by grants HKUST 6070/00E and HKUST 6090/99E from Hong Kong RGC.

REFERENCES

[1] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J.S. Vitter, "Scalable Sweeping-Based Spatial Join," *Proc. Very Large Data Base Conf.*, pp. 570-581, Aug. 1998.

[2] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J. Vahrenhold, and J.S. Vitter, "A Unified Approach for Indexed and Non-Indexed Spatial Joins," *Proc. EDBT Conf.*, pp. 413-429, Mar. 2000.

[3] S. Achaya, V. Poosala, and S. Ramaswamy, "Selectivity Estimation in Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 13-24, June 1999.

[4] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 322-331, May 1990.

[5] T. Brinkhoff, H.P. Kriegel, and B. Seeger, "Efficient Processing of Spatial Joins Using R-Trees," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 237-246, May 1993.

[6] J. van der Bercken, B. Seeger, and P. Widmayer, "A Generic Approach to Bulk Loading Multidimensional Index Structures," *Proc. Very Large Data Base Conf.*, pp. 406-415, Aug. 1997.

[7] Bureau of the Census, TIGER/Line Precensus Files: 1990 Technical Documentation, Washington DC, 1989.

[8] D.J. DeWitt, N. Kabra, J. Luo, J.M. Patel, and J.B. Yu, "Client-Server Paradise," *Proc. Very Lareg Data Base Conf.*, pp. 558-569, Sept. 1994.

[9] V. Gaede and O. Günther, "Multidimensional Access Methods," *ACM Computing Surveys,* vol. 30, no. 2, pp. 170-230, 1998.

[10] G. Graefe, "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys,* vol. 25, no. 2, pp. 73-170, 1993.

[11] R.H. Güting and W. Schilling, "A Practical Divide-and-Conquer Algorithm for the Rectangle Intersection Problem," *Information Sciences,* vol. 42, pp. 95-112, 1987.

[12] O. Günther, "Efficient Computation of Spatial Joins," *Proc. Int'l Conf. Data Eng.*, pp. 50-59, Apr. 1993.

[13] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. of ACM SIGMOD Int'l Conf. Management of Data,* pp. 47-57, June 1984.

[14] Y.W. Huang, N. Jing, and E.A. Rundensteiner, "Spatial Joins Using R-Trees: Breadth First Traversal with Global Optimizations," *Proc. Very Large Data Base Conf.*, pp. 395-405, Aug. 1997.

[15] Y.W. Huang, N. Jing, and E.A. Rundensteiner, "A Cost Model for Estimating the Performance of Spatial Joins Using R-Trees," *Proc. Int'l Conf. Scientific and Statistical Database Management,* pp. 30-38, Aug. 1997.

[16] K. Kim and S.K. Cha, "Sibling Clustering of Tree-Based Spatial Indexes for Efficient Spatial Query Processing," *Proc. ACM Int'l Conf. Information and Knowledge Management,* pp. 398-405, Nov. 1998.

[17] I. Kamel and C. Faloutsos, "On Packing R-Trees," *Proc. ACM Int'l Conf. Information and Knowledge Management,* pp. 490-499, Nov. 1993.

[18] N. Koudas and K. Sevcik, "Size Separation Spatial Join," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 324-335, May 1997.

[19] S.T. Leutenegger, J. Edgington, and M.A. Lopez, "STR: A Simple and Efficient Algorithm for R-Tree Packing," *Proc. Int'l Conf. Data Eng.*, pp. 497-506, Apr. 1997.

[20] M.L. Lo and C.V. Ravishankar, "Generating Seeded Trees from Datasets," *Proc. Int'l Symp. Large Spatial Databases (Advances in Spatial Databases: SSD '95),* pp. 328-347, Aug. 1995.

[21] M.L. Lo and C.V. Ravishankar, "Spatial Hash-Joins," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 209-220, May 1996.

[22] M.L. Lo and C.V. Ravishankar, "The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins," *IEEE Trans. Knowledge and Data Eng.*, vol. 10, no. 1, pp. 136-151, 1998.

[23] N. Mamoulis, "Algorithms and Optimization Techniques for Complex Spatial Queries," doctoral dissertation, Hong Kong Univ. of Science and Technology, Dept. Computer Science, 2000.

[24] N. Mamoulis and D. Papadias, "Integration of Spatial Join Algorithms for Processing Multiple Inputs," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 1-12, June 1999.

[25] J.A. Orenstein, "Redundancy in Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 294-305, June 1989.

[26] J.M. Patel and D.J. DeWitt, "Partition Based Spatial-Merge Join," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 259-270, June 1996.

[27] H.H. Park, C.G. Lee, Y.J. Lee, and C.W. Chung, "Early Separation of Filter and Refinement Steps in Spatial Query Optimization," *Proc. NATO Advanced Research Workshop (ARW) Confluence of Computer Vision and Computer Graphics (DASFAA '99),* pp. 161-168, Apr. 1999.

[28] A.N. Papadopoulos, P. Rigaux, and M. Scholl, "A Performance Evaluation of Spatial Join Processing Strategies," *Proc. Int'l Symp. Large Spatial Databases (Advances in Spatial Databases: SSD '99),* pp. 286-307, July 1999.

[29] F.P. Preparata and M.I. Shamos, *Computational Geometry—An Introduction.* Springer, 1985.

[30] N. Roussopoulos and D. Leifker, "Direct Spatial Search on Pictorial Databases Using Packed R-Trees," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 17-31, May 1985.

[31] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu, and C. Lu, "Spatial Databases—Accomplishments and Research Needs," *IEEE Trans. Knowledge and Data Eng.*, vol. 11, no. 1, pp. 45-55, 1999.

[32] A. Silberschatz, H.F. Korth, and S. Sudarshan, *Database System Concepts.* McGraw-Hill, 1997.

[33] Y. Theodoridis and T. Sellis, "A Model for the Prediction of R-Tree Performance," *Proc. ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems (PODS),* pp. 161-171, June 1996.

[34] Y. Theodoridis, E. Stefanakis, and T. Sellis, "Cost Models for Join Queries in Spatial Databases," *Proc. Int'l Conf. Data Eng.*, pp. 476-483, Feb. 1998.

Nikos Mamoulis received the diploma in computer engineering and Informatics in 1995 from the University of Patras, Greece, and the PhD degree in computer science in 2000 from Hong Kong University of Science and Technology. From September 2000 to September 2001, he was a postdoctoral researcher at Centrum voor Wiskunde en Informatica (CWI), the Netherlands. He is currently an assistant professor in the Department of Computer Science and Information Systems, University of Hong Kong. His research interests include spatial and multimedia databases, query processing and optimization, online analytical processing, and constraint satisfaction problems.

Dimitris Papadias is an associate professor in the Department of Computer Science at the Hong Kong University of Science and Technology (HKUST). Before joining HKUST, he worked at various places including, the Data and Knowledge Base Systems Laboratory—National Technical University of Athens (Greece), the Department of Geoinformation—Technical University of Vienna (Austria), the Department of Computer Science and Engineering—University of California at San Diego, the National Center for Geographic Information and Analysis—University of Maine, and the Artificial Intelligence Research Division—German National Research Center for Information Technology (GMD). His research interests include data and knowledge base systems, spatial databases, geographic information systems, query languages, constraint satisfaction algorithms, WWW collaborative systems, and data warehousing.