Duplicate Elimination in Space-partitioning Tree Indexes [*]

M. Y. Eltabakh[1] Mourad Ouzzani[2] Walid G. Aref[1]
[1]Computer Science Department, Purdue University
[2]Cybr Center, Discovery Park, Purdue University
{meltabak, mourad, aref}@cs.purdue.edu

Abstract

Space-partitioning trees, like the disk-based trie, quadtree, kd-tree and their variants, are a family of access methods that index multi-dimensional objects. In the case of indexing non-zero extent objects, e.g., line segments and rectangles, space-partitioning trees may replicate objects over multiple space partitions, e.g., PMR quadtree, expanded MX-CIF quadtree, and extended kd-tree. As a result, the answer to a query over these indexes may include duplicates that need to be eliminated, i.e., the same object may be reported more than once. In this paper, we propose generic duplicate elimination techniques for the class of space-partitioning trees in the context of SP-GiST; an extensible indexing framework for realizing space-partitioning trees. The proposed techniques are embedded inside the INDEX-SCAN operator. Therefore, duplicate copies of the same object do not propagate in the query plan, and the elimination process is transparent to the end-users. Two cases for the index structures are considered based on whether or not the objects' coordinates are stored inside the index tree. The theoretical and experimental analysis illustrate that the proposed techniques achieve savings in the storage requirements, I/O operations, and processing time when compared to adding a separate duplicate elimination operator in the query plan.

1 Introduction

Space-partitioning trees are a family of access methods that index multi-dimensional objects, e.g., disk-based trie variants [2, 8, 13], quadtree variants [12, 14, 16, 19, 23], and kd-tree variants [6, 7, 17, 20]. Space-partitioning trees are used as supporting structures in a variety of applications such as GIS, data mining, and CAD/CAM applications. In the case of indexing non-zero extent objects, e.g., line segments, and rectangles, space-partitioning trees may replicate the indexed objects over multiple space partitions, e.g., the PMR quadtree [19], expanded MX-CIF quadtree [1, 21], RR quadtrees [22], and extended kd-tree [17]. As a result, the answer to a query over these indexes may include duplicates that need to be eliminated, i.e., the same object may be reported more than once. For example, in Figure 1, we illustrate indexing two line segments, L_1 and L_2, using a PMR quadtree. Each object is attached to all space partitions it intersects with. Given an *intersection* query Q, we want to report all objects that intersect with Q. This will result in reporting L_1 four times because Q intersects with partitions 1, 9, 10, and 11. Moreover, L_2 will be reported twice because Q intersects with partitions 7, and 12.

Eliminating the duplicates of an object can be performed by using the traditional database duplicate elimination techniques i.e., adding the DISTINCT operator to the query plan. However, this approach is not efficient for the following reasons. First, removing the duplicates resulting from the underlying access methods should be transparent to the end-users. End-users do not have to be aware of which access method is used to answer the query and whether or not this access method generates duplicates. Second, the use of a separate operator to eliminate the duplicates is an expensive operation as it involves an additional sorting or hashing phase in the query plan. This overhead can be reduced or completely avoided if these duplicates are eliminated at the INDEX-SCAN operator.

In this paper, we propose generic duplicate elimination techniques for the class of space-partitioning trees in the context of SP-GiST [3, 4]. SP-GiST is an extensible indexing framework for realizing the class of space-partitioning trees inside database engines [11]. The proposed duplicate elimination techniques are embedded inside the SP-GiST INDEX-SCAN operator. They ensure that the INDEX-SCAN operator reports each object that satisfies a given query only once independent of how many times the object is replicated inside the index. We consider two cases for the index structures based on whether or not the objects' coordinates are stored inside the index tree. For indexes that store the objects' coordinates, we propose a technique, termed *Consistency_Reference*, that computes a point *CR*

[*]This work was supported in part by the National Science Foundation under Grants IIS-0093116 and IIS-0209120

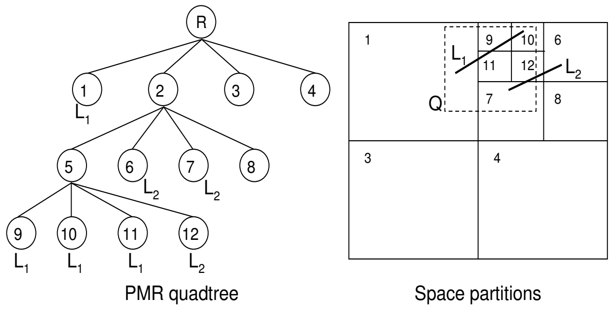

Figure 1. PMR quadtree

for each object satisfying the user's query. The object is reported only when the partition being processed contains CR. For indexes that do not store the objects' coordinates, we use hashing techniques inside the INDEX-SCAN operator to eliminate the duplicates as early as possible, i.e., before retrieving the objects' coordinates to avoid performing I/O operations. The theoretical and experimental analysis illustrate that the proposed approach achieves savings in the storage requirements, I/O operations, and processing time compared to the stand-alone DISTINCT operator approach.

The contributions of the paper are summarized as follows:

1. We present generic duplicate elimination techniques for the class of space-partitioning trees. The proposed techniques are embedded inside the INDEX-SCAN operator instead of using an expensive stand-alone DISTINCT operator.
2. We implement the proposed techniques inside the PostgreSQL version of SP-GiST. The theoretical and experimental analysis illustrate the efficiency of the proposed techniques with respect to storage requirements, I/O operations, and processing time compared to the stand-alone DISTINCT operator approach.

The rest of the paper proceeds as follows. In Section 2, we present the related work. In Section 3, we overview the SP-GiST framework. The proposed duplicate elimination techniques are presented in Section 4. Section 5 describes the implementation inside SP-GiST. The theoretical and experimental analysis are presented in Sections 6 and 7, respectively. Section 8 contains concluding remarks.

2 Related Work

Duplicate elimination in databases is usually performed by adding a separate DISTINCT operator to the query plan. Several techniques have been proposed to realize the DISTINCT operator [10, 18]. Typically, a non-blocking hashing technique is used when the expected number of distinct tuples can fit into main memory. Otherwise, a blocking duplicate elimination technique is used. Duplicate elimination is known to be an expensive operation as it involves an additional sorting or hashing phase. Therefore, several techniques have been proposed to eliminate duplicates more efficiently in the context of spatial data. In [9], an on-line technique has been proposed to eliminate duplicates over spatial join operations. The technique computes a unique point x for each pair of joined spatial objects. A joined pair is reported from the JOIN operator only when x is inside the space partition being processed. The technique avoids adding a separate DISTINCT operator over the JOIN operator in the query plan. However, the technique proposed in [9] is limited to spatial join operations. In contrast, our proposed techniques can be used to handle duplicate elimination over any spatial operation because they are built inside the INDEX-SCAN operator. For example, if the spatial join is built on top of the proposed INDEX-SCAN operator, then the JOIN operator does not need to handle the duplicates because each object will be reported only once from the INDEX-SCAN operator. Another technique for eliminating duplicates in spatial databases has been proposed in [5], . The technique maintains a data structure, *active border*, that represents the border between the space partitions that have been processed and those that have not. An object is reported only when all the partitions that intersect with the object have been processed, i.e., covered by the *active border*. The technique proposed in [5] is an algorithmic approach that can be used to traverse or retrieve objects from the index tree. However, the technique is not practical to be implemented as an operator inside a database engine because of its complexity.

3 SP-GiST

SP-GiST [3, 4, 11, 15] is an extensible indexing framework that broadens the class of supported indexes to include a wide variety of space-partitioning trees, e.g., disk-based trie variants, quadtree variants, and kd-trees. Space-partitioning trees differ from each other in various ways. For example, tree structures may or may not support *node shrinking* where empty partitions inside each node can be omitted. Other variations include the bucket size of leaf nodes, the support for various data types, and the splitting of nodes (when to trigger a split and how node splitting is performed). As an extensible indexing framework, SP-GiST allows developers to instantiate a variety of index structures in an efficient way through pluggable modules and without modifying the database engine.

SP-GiST provides a set of *internal* methods that are common for all space-partitioning trees, e.g., the *Insert(), Search(),* and *Delete()* methods. The internal methods are the core of SP-GiST. To handle the differences among the various SP-GiST-based indexes, SP-GiST provides a set of *interface* parameters and a set of *external* method interfaces (for the developers).

The interface parameters include:

- *NodePredicate:* specifies the predicate type at the in-

	PMR quadtree	kd-tree

$Parameters$	PathShrink = LeafShrink, NodeShrink = Flase
	BucketSize = B
	NoOfSpacePartitions = 4
	NodePredicate = Quadrant (x1, y1, x2, y2)
	KeyType = Line Segment
		PathShrink = NeverShrink, NodeShrink = False
		BucketSize = 1
		NoOfSpacePartitions = 2
		NodePredicate = “left”, “right”, or blank
		KeyType = Point

Consistent(E,q,level)	If (inserted line segment intersects E.quadrant) Return True Else Return False	If (level is odd AND q.x satisfi es E.p.x) OR (level is even AND q.y satisfi es E.p.y) Return True, Else Return False
PickSplit(P,level)	Decompose the space into four equal partitions Distribute the line segments in P according to their intersection with the new partitions Return False	Put the old point in a child node with predicate “blank” Put the new point in a child node with predicate “left” or “right” Return False

Table 1. Instantiations of the PMR quadtree and kd-tree using SP-GiST.

dex nodes.

- *KeyType:* specifies the data type stored at the leaf nodes.
- *NumberofSpacePartitions:* specifies the number of disjoint partitions produced at each decomposition.
- *PathShrink:* specifies how the index tree can shrink.
- *NodeShrink:* specifies whether the empty partitions should be kept in the index tree or not.
- *BucketSize:* specifies the maximum number of data items a data node can hold.
- *ObjectReplication:* specifies whether or not the index allows replicating an object over multiple partitions.
- *MaintainCoordinates:* specifies whether or not the index stores the objects' coordinates.

The SP-GiST external methods include the *PickSplit()* method that specifies how the space is decomposed and how the data items are distributed over the new partitions. *PickSplit()* is invoked by the internal method *Insert()* when a node is full and splitting is needed. Another external method is *Consistent()* that specifies how to navigate through the index tree. *Consistent()* is invoked by the internal methods *Insert()* and *Search()* to guide the tree navigation.

In Table 1, we illustrate the instantiation of the PMR quadtree and kd-tree using SP-GiST. Notice that from the developer's point of view, coding of the external methods in Table 1 is all what the developer needs to provide.

We realized the SP-GiST framework [3, 4] inside PostgreSQL to include the family of space-partitioning trees [11]. The proposed duplicate elimination techniques are implemented inside the PostgreSQL version of SP-GiST.

4 Duplicate Elimination in Space-partitioning Trees

Duplicates may occur in space-partitioning trees that have the following two properties: (1) they index non-zero extent objects, and (2) they replicate the indexed objects over multiple space partitions. Examples of such trees include the PMR quadtree [19], the expanded MX-CIF quadtree [1, 21], the RR quadtree [22], and the extended kd-tree [17]. Other index structures such as the kd-tree, point quadtree, and trie do not generate duplicates because they index zero-extent objects, e.g., points, and characters.

Space-partitioning trees that have the problem of generating duplicates may or may not store the objects' coordinates inside the index tree. For example, if the indexed objects are polygons, then storing the objects' coordinates inside the index can be expensive. In this case, only the objects' identifiers are stored inside the index, and the objects' coordinates are stored in a separate table. In Section 4.1, we consider the problem of duplicate elimination in indexes that store the objects' coordinates. In Section 4.2, we consider the complementary case.

4.1 Duplicate Elimination in Coordinate maintained Indexes

Coordinate-maintained indexes have the property that the objects' coordinates can be retrieved without performing any extra I/O operations. We make use of this property to design a duplicate elimination technique, termed *Consistency_Reference*, that requires neither extra space nor extra I/O operations. Consistency_Reference is based on the idea of reporting an object at a certain point that is computed at the query run-time [5, 9]. Consistency_Reference is embedded inside the SP-GiST INDEX-SCAN operator.

Consistency_Reference computes, at the query run-time, a zero-extent object, e.g., Point, called *CR*, for each database object O satisfying a given query Q. CR is computed each time O is encountered, i.e., if O is encountered m times, then CR will be computed m times. O will be reported from the INDEX-SCAN operator to the next operator in the query pipeline only when the space partition being processed contains CR. Since CR is a zero-extent

DB Object Type O	Query Type Q	Operator	CR Definition
Line segment	Line segment	Intersection	CR $= \{point\ P : P \in O\ and\ P \in Q\}$
	Window	Intersection	CR $= \{point\ P \in O' : P.x \geq P'.x\ \forall\ P' \in O'\}$; where O' is the intersected line segment between O and Q
		Contains	
	Point	Contained in	CR $= \{point\ P : P = Q\}$
Rectangle	Line segment	Intersect	CR $= \{point\ P \in O' : P.x \geq P'.x\ \forall\ P' \in O'\}$; where O' is the intersected line segment between O and Q
		Contained in	
	Window	Intersect	CR $= \{point\ P \in O' : P.x \geq P'.x\ and\ P.y \geq P'.y\ \forall\ P' \in O'\}$; where O' is the intersected rectangle between O and Q
		Contains	
		Contained in	
	Point	Contained in	CR $= \{point\ P : P = Q\}$

Table 2. Computing CR for various data types and query operators

	Consistency_Reference	Stand-alone DISTINCT operator approach
Storage	No storage is required	A hash table is used to store the identifier of each reported object
Scalability	The technique scales with the increase of the size of the output relation	If the size of the hash table exceeds the main memory, then a blocking duplicate elimination technique will be used
I/O Operations	No I/O operations are required	If a blocking technique is used, then I/O operations are performed

Table 3. A comparison between the Consistency_Reference technique and the stand-alone DISTINCT operator approach.

object, then it is guaranteed that CR belongs to only one space partition. Hence, O will be reported once.

Definition (Consistency_Reference CR): *For a non-zero extent object O that satisfies a given query Q, CR is a zero-extent object that belongs to O and participates in making O satisfy Q.*

According to the definition, the CR of an object O against query Q has to satisfy two conditions: (1) CR belongs to O, and (2) CR participates in making O satisfies Q. These two conditions ensure that CR belongs to a space partition that will be processed by Q. Therefore O will not be missed. The exact criterion for computing CR depends on the data type of the database objects, e.g., *Line Segment*, or *Rectangle*, and the type of the query, e.g., *intersection*, *overlapping*, or *containment* query. This criterion is provided by the developer of the index structure.

In Figure 2, we illustrate an example for the Consistency_Reference technique. We consider an *intersection* query Q over a PMR quadtree that indexes line segments. The criterion for computing CR for a line segment O satisfying Q can be defined as follows:

CR $= \{point\ P \in O' : P.x \geq P'.x\ \forall\ P' \in O'\}$; where O' is the intersected line segment between O and Q.

The criterion selects the point with the largest X-axis value on the intersected line segment between O and Q.

The space partitions in Figure 2 are processed clock-wise in the following order: 1, 9, 10, 11, 12, and 7. Partitions 6, 8, 3, and 4 are skipped because they do not intersect Q. When we process partition 1, we find that L_3 does not satisfy Q, therefore, L_3 will be skipped. However, L_1 satisfies Q, therefore, we compute the CR point for L_1, i.e., L_1-CR. Since L_1-CR is outside partition 1, then partition 1 will not report L_1. Similarly, partitions 9 and 11 will not report L_1 for the same reason. L_1 will be reported only when partition 10 is processed. When we process partition 12, we find that the CR point of L_2, i.e., L_2-CR, belongs to partition 12. Therefore, partition 12 reports L_2. When partition 7 is processed, L_3 and L_4 will be skipped because they do not intersect with Q, and L_2 will not be reported because L_2-CR is outside partition 7.

In Table 2, we present criteria for computing the CR under various objects data types and query predicates. These criteria are not unique and other criteria can be used as long as they satisfy the definition above.

In Table 3, we compare the Consistency_Reference technique against the stand-alone DISTINCT operator ap-

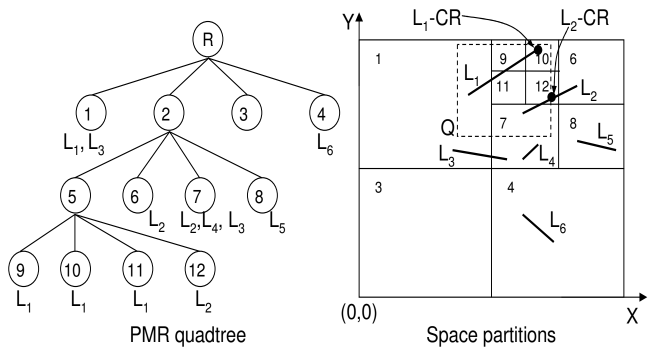

Figure 2. Example of the Consistency_Reference technique

proach. Consistency_Reference does not require any extra storage because CR is computed for each object satisfying the query on the fly, i.e., when the object is encountered by the search algorithm. As a result, Consistency_Reference does not perform any I/O operations. On the other hand, the stand-alone DISTINCT operator approach maintains a hash table to store a copy of each reported object. If the size of the maintained hash table exceeds the available main memory, then a blocking duplicate elimination technique is used that requires performing disk I/O operations.

4.2 Duplicate Elimination in Non-coordinate-maintained Indexes

Non-coordinate-maintained indexes are indexes that store only the objects' identifiers inside the index. The objects' coordinates are stored in a separate table. Hence, each retrieval of an object's coordinates requires performing at least one I/O operation. Applying the Consistency_Reference technique over the non-coordinate-maintained indexes is very expensive because Consistency_Reference retrieves the coordinates of an object at each time this object is encountered during the search.

Our proposed duplicate elimination technique for non-coordinate-maintained indexes is based on hashing techniques. Hashing techniques are well known techniques for eliminating duplicates in database management systems [18]. Our approach, termed *Embedded_Hashing* is based on embedding hashing techniques inside the INDEX-SCAN operator, which has two advantages: (1) it achieves transparency to the end-user, and (2) it saves I/O operations because our approach retrieves the coordinates of each object encountered during the search only once independent of how many times the object is replicated inside the index.

Embedded_Hashing performs in the same way as the standard hashing techniques for eliminating duplicates except that it works inside the INDEX-SCAN operator. Embedded_Hashing stores the identifiers of the objects encountered during the search in a hash table to avoid processing the same object multiple times. When an object O is encountered, we search for O in the hash table. If O is found, then O is skipped because it has been already processed. Otherwise, we perform an I/O operation(s) to retrieve O's coordinates and check whether or not O satisfies the query. If O satisfies the query, then O will be reported. Otherwise, O will be skipped. In both cases, O's identifier will be stored in the hash table.

Continuing with the example in Figure 2, and assuming that only the objects' identifiers are stored inside the index, the processing of Q is as follows. The processing of partition 1 will report L_1 as it satisfies Q but not L_3. Then, both L_1 and L_3 will be inserted into the hash table. The processing of partitions 9, 10, and 11 will not check L_1 again because L_1 is already in the hash table. The processing of partition 12 will report L_2 because L_2 is not encountered before. Then, L_2 will be inserted into the hash table. Finally, the processing of partition 7 will not check L_2 or L_3 again because both objects are already in the hash table. L_4 will be checked, but it will not be reported because it does not satisfy Q. Then, L_4 will be inserted into the hash table.

If the size of the hash table exceeds the available main memory, we use a blocking hashing technique. In the first phase of the technique, we store the identifiers of the encountered objects during the search in the hash table. In the second phase, we eliminate the duplicates, and then retrieve the coordinates of each of the remaining objects to check them against the query. Since we retrieve the objects' coordinates after eliminating the duplicates, then, Embedded_Hashing guarantees that the coordinates of each encountered object are retrieved only once independent of how many times the object is replicated inside the index.

In Table 4, we compare Embedded_Hashing against the stand-alone DISTINCT operator approach. In Embedded_Hashing, the hash table stores the identifiers of the encountered objects whether or not they satisfy the query. Whereas, the hash table maintained by a separate DISTINCT operator stores only the identifiers of the objects that satisfy the query. In both techniques, the size of the hash table may exceed the available main memory. Therefore, both techniques may use a blocking hashing technique for eliminating the duplicates. The main advantage of Embedded_Hashing over the stand-alone DISTINCT operator approach is in the savings of the I/O operations. Embedded_Hashing retrieves the coordinates of each encountered object, whether or not it satisfies the query, only once. Whereas, the stand-alone DISTINCT operator approach retrieves the coordinates of each encountered object, whether or not it satisfies the query, every time this object appears. The reason is that in the latter case, the INDEX-SCAN operator does not keep track of which objects are already processed. Thus, each time an object is encountered, the

	Embedded_Hashing	Stand-alone DISTINCT operator approach
Storage	A hash table is used to store the identifi ers of the encountered objects including objects that do not satisfy the query	A hash table is used to store the identifi ers of the objects reported from the INDEX-SCAN operator.
Scalability	If the size of the hash table exceeds the main memory then a blocking duplicate elimination technique will be used	If the size of the hash table exceeds the main memory, then a blocking duplicate elimination technique will be used
I/O Operations	Performs one I/O operation for each encountered object independent of the objects' replication. This is the case whether the hash table resides in memory or a blocking technique is used	Performs one I/O operation for each appearance of an encountered object. This is the case whether the hash table resides in memory or a blocking technique is used

Table 4. A comparison between the Embedded_Hashing technique and the stand-alone DISTINCT operator approach.

INDEX-SCAN operator retrieves the coordinates of this object and checks them against the query. The I/O saving of Embedded_Hashing can be significant if the objects' replication factor over the space partitions is large.

5 Implementation Inside SP-GiST

We implement the proposed duplicate elimination techniques inside the PostgreSQL version of SP-GiST [11]. The Consistency_Reference technique is implemented as part of the SP-GiST external methods because it requires that the developer defines and codes the criteria for computing the CR objects (Section 5.1). The Embedded_Hashing technique is implemented as part of the SP-GiST internal methods because they are common for all SP-GiST index structures (Section 5.2).

5.1 Coordinate-maintained Indexes

To implement the Consistency_Reference technique inside SP-GiST, we extend the SP-GiST external methods to include two more functions: *Consistency_Reference()* and *Report_Unique()*. The developer of an index structure will provide these functions among the index's external methods.

Consistency_Reference(Op, E, Q) takes three arguments, where *Op* is the operator type, e.g., *intersection*, *overlapping*, or *containment*, *E* is the operator's left argument which is a database object, and *Q* is the operator's right argument which is the operator predicate. Based on the operator's type and arguments, Consistency_Reference() computes and returns the CR reference of E. *Report_Unique(P, CR)* takes two arguments, where *P* is the partition begin processed, and *CR* is the E's CR computed by Consistency_Reference(). Report_Unique(P, CR) returns True if CR belongs to P, and False otherwise.

Consistency_Reference() and *Report_Unique()* are called from the internal function *Search()* only when the index's interface parameters *ObjectReplication* = True and *MaintainCoordinates* = True (Refer to Section 3). That is, these functions are used only for indexes that store the objects' coordinates and allow objects to be replicated over multiple space partitions. *Search()* is modified to call *Consistency_Reference()* and *Report_Unique()* as follows. When a leaf object *O* is encountered, *Search()* passes *O* to function *Consistent()* to decide whether or not *O* satisfies the query. If *O* does not satisfy the query, then *O* will be skipped. Otherwise, *Search()* calls *Consistency_Reference()* to get *O*'s CR, and then calls *Report_Unique()* to check whether or not *O*'s CR belongs to the partition being processed. If *Report_Unique()* returns True, then *Search()* will report *O*. Otherwise, *O* will be skipped. Notice that, *Consistency_Reference()* and *Report_Unique()* are called only for leaf objects that satisfy the query.

For indexes that have either *ObjectReplication* or *MaintainCoordinates* set to False, only the prototype definition of *Consistency_Reference()* and *Report_Unique()* is required among the index's external methods because *Search()* will not call them in the first place.

5.2 Non-coordinate-maintained Indexes

Unlike the implementation of the *Consistency_Reference* technique, Embedded_Hashing is fully implemented inside the SP-GiST core, i.e., the internal methods. Therefore, no extra coding is required from the index developer side.

The Embedded_Hashing technique is implemented inside the *Search()* internal method. The non-blocking Embedded_Hashing is implemented inside *Search()* as follows. *Search()* maintains a hash table. Whenever a leaf object *O* is encountered, *Search()* looks for *O*'s identifier in the hash table. If *O*'s identifier is found, then *O* is skipped because *O* is already processed. Otherwise, *Search()* retrieves *O*'s coordinates and passes *O* to *Consistent()* to decide whether or not *O* satisfies the query. In both cases, *Search()* adds *O*'s identifier to the hash table.

Parameter	Defi nition
R	The underlying relation
M	The memory size in blocks
B	The memory block size
N	The number of buckets in the hash table
$\lambda(R)$	The number of distinct objects in R that satisfy the query
$\varepsilon(R)$	The number of distinct encountered objects in R that do not satisfy the query
α	The average number of object replications encountered during the search

Table 5. The analysis parameters

The blocking Embedded_Hashing is implemented inside *Search()* as follows. Whenever a leaf object O is encountered, *Search()* inserts O's identifier into the hash table without checking for duplicates. After inserting all the encountered leaf objects into the hash table, *Search()* scans the hash table bucket by bucket to eliminate the duplicates. After eliminating the duplicates of a given bucket, *Search()* retrieves the coordinates of each object in that bucket and passes the object to *Consistent()* to decide whether or not the object satisfies the query. If the object satisfies the query, then the object will be reported. Otherwise, the object will be skipped.

Search() uses Embedded_Hashing only when the index's interface parameters *ObjectReplication* = True and *MaintainCoordinates* = False (Refer to Section 3). That is, Embedded_Hashing is used only for indexes that do not store the objects' coordinates and allow objects to be replicated over multiple space partitions. Deciding whether a non-blocking or blocking hashing will be used is based on the same statistics that the query optimizer uses if a separate DISTINCT operator is used in the query plan.

6 Theoretical Analysis

In this section, we analyze theoretically the proposed duplicate elimination techniques. For each technique, we analyze the memory requirements (in blocks), and the disk I/O operations. The CPU processing time is considered only for techniques in which no I/O operations are performed. The parameters used in the analysis are summarized in Table 5.

We compare the Consistency_Reference technique against the stand-alone DISTINCT operator approach in Table 6. Consistency_Reference is only a memory-based technique, i.e., it has no disk-based analysis. Consistency_Reference does not require any extra storage and does not perform any I/O operations. In Consistency_Reference, each time an object is encountered, the technique checks the object against the query, and if the object satisfies the query, then the object's CR is computed. The number of these encounters is $[\lambda(R) + \varepsilon(R)] * \alpha$. Checking an object against the query and computing the object's CR are assumed to take O(1) processing time. Therefore, the overall CPU processing time is $[\lambda(R) + \varepsilon(R)] * \alpha$.

The memory-based stand-alone DISTINCT operator approach maintains a hash table of size $\lambda(R)/B$ blocks. The technique does not perform any I/O operations. In this technique, the INDEX-SCAN operator checks the encountered objects against the query $[\lambda(R) + \varepsilon(R)] * \alpha$ times. Then, the INDEX-SCAN operator reports $\alpha * \lambda(R)$ objects to the DISTINCT operator. The DISTINCT operator searches for each of the $\alpha * \lambda(R)$ objects in a hash table bucket of average size $\lambda(R)/N$. Therefore, the overall CPU processing time is $[\lambda(R) + \varepsilon(R)] * \alpha + [\alpha * \lambda^2(R)]/N$.

If the size of the hash table exceeds the available main memory, then a disk-based blocking hashing technique is used. The blocking technique uses the entire memory to process the hash table buckets. The size of the hash table is $[\alpha * \lambda(R)]/B$ blocks. The table will be written to and then read from the disk to eliminate the duplicates. Therefore, the total number of I/O operations is $2 * [\alpha * \lambda(R)]/B$.

To summarize the results presented in Table 6, Consistency_Reference involves neither storage overhead nor I/O operations and its CPU processing time is less than that of the stand-alone DISTINCT operator approach. Hence, independently from the objects distribution and the replication factor α, the Consistency_Reference technique outperforms the stand-alone DISTINCT operator approach.

The comparison between the Embedded_Hashing technique and the stand-alone DISTINCT operator approach for non-coordinate-maintained indexes is presented in Table 7. Both techniques may use a blocking hashing technique if the hash table cannot fit entirely into memory. The memory-based version of Embedded_Hashing stores in the hash table the identifier of each encountered object including objects that do not satisfy the query. Hence, the size of the hash table is $[\lambda(R)+\varepsilon(R)]/B$. Embedded_Hashing retrieves the coordinates of each of the encountered objects only once. Assuming that each retrieval requires one I/O, then the total number of I/O operations performed by the technique is $\lambda(R) + \varepsilon(R)$. On the other hand, the memory-based stand-alone DISTINCT operator approach stores in the hash table the identifiers of the objects that satisfy the query. Hence, the size of the hash table is $\lambda(R)/B$. In the stand-alone DISTINCT operator approach, the INDEX-SCAN operator retrieves the coordinates of an object each time the object appears. Therefore, the number of I/O operations is $[\lambda(R) + \varepsilon(R)] * \alpha$.

With respect to the disk-based processing, both techniques will use the entire memory to process the hash table buckets. In Embedded_Hashing, the size of the hash table to be written to and read from the disk is $[\lambda(R)+\varepsilon(R)]*\alpha$. After eliminating the duplicates from the hash table, Embedded_Hashing retrieves the coordinates of each of the remaining objects to check them against the query which re-

	Consistency_Reference	Stand-alone DISTINCT operator	
	Memory-based	Memory-based	Disk-based
Memory requirements	0	$\lambda(R)/B$	M
Disk I/O	0	0	$[2 * \alpha * \lambda(R)]/B$
CPU time	$[\lambda(R) + \varepsilon(R)] * \alpha$	$[\lambda(R) + \varepsilon(R)] * \alpha + [\alpha * \lambda^2(R)]/N$	

Table 6. Analysis of the coordinate-maintained duplicate elimination techniques

	Embedded_Hashing		Stand-alone DISTINCT operator	
	Memory-based	Disk-based	Memory-based	Disk-based
Memory requirements	$[\lambda(R) + \varepsilon(R)]/B$	M	$\lambda(R)/B$	M
Disk I/O	$\lambda(R) + \varepsilon(R)$	$[\lambda(R) + \varepsilon(R)] * [(2 * \alpha)/B + 1]$	$[\lambda(R) + \varepsilon(R)] * \alpha$	$[\lambda(R) + \varepsilon(R)] * \alpha + [2 * \alpha * \lambda(R)]/B$

Table 7. Analysis of the non-coordinate-maintained duplicate elimination techniques

quires $[\lambda(R) + \varepsilon(R)]$ I/O operations. Therefore, the total number of I/O operations is $[(\lambda(R) + \varepsilon(R)) * \alpha * 2]/B$ + $[\lambda(R)+\varepsilon(R)]$ = $[\lambda(R)+\varepsilon(R)]*[(2*\alpha)/B+1]$. In the stand-alone DISTINCT operator technique, the INDEX-SCAN operator retrieves the coordinates of an object each time the object is encountered which requires $[\lambda(R) + \varepsilon(R)] * \alpha$ I/O operations. Then, the INDEX-SCAN operator reports $\alpha * \lambda(R)$ objects to the DISTINCT operator which stores these objects in the hash table. The hash table will be written to and then read from the disk. Therefore, the total number of I/O operations is $[\lambda(R)+\varepsilon(R)]*\alpha$ + $[2*\alpha*\lambda(R)]/B$.

It is worth noting that Embedded_Hashing may use the disk-based version while the stand-alone DISTINCT operator technique uses the memory-based version. The reason is that the size of the hash table maintained by Embedded_Hashing is larger than that of the other technique. However, as illustrated in Table 7, the number of I/O operations performed by the disk-based Embedded_Hashing is less than the number of I/O operations performed by the memory-based version of the stand-alone DISTINCT operator technique, assuming that $(2 * \alpha)/B + 1 < \alpha$.

To summarize the results presented in Table 7, Embedded_Hashing requires larger hash table than the stand-alone DISTINCT operator technique. However, the number of I/O operations performed by Embedded_Hashing is around factor of α less than that performed by the stand-alone DISTINCT operator technique.

Notice that, in the case where no duplicates are present, i.e., each object is encountered only once during the search, the cost of the above techniques can be easily derived by setting the replication factor (α) to 1. In this case, the Consistency_Reference technique still outperforms the stand-alone technique with respect to memory, disk, and CPU requirements. For the hashing techniques, if the search results can fit into memory, then the stand-alone operator is preferred as it requires less memory. Otherwise, the Embedded_Hashing technique is preferred as it requires less I/Os.

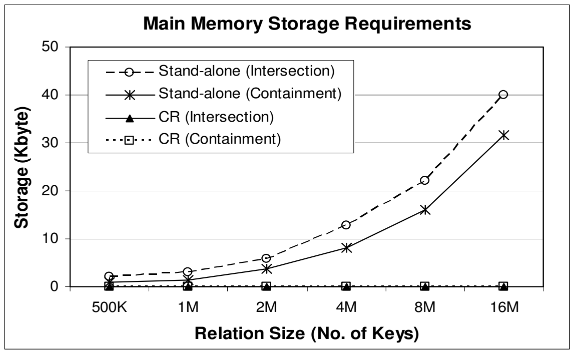

Figure 3. Coordinate-maintained indexes: Storage requirements

7 Experimental Analysis

In this section, we study experimentally the performance of the proposed duplicate elimination techniques against the stand-alone DISTINCT operator approach. We perform the experiments from within the PostgreSQL version of SP-GiST [11]. We consider two types of queries: an *intersection* query (find all objects that intersect the query box), and a *containment* query (find all objects that are contained

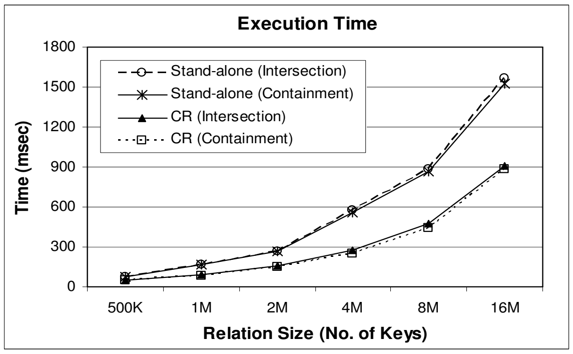

Figure 4. Coordinate-maintained indexes: Execution time

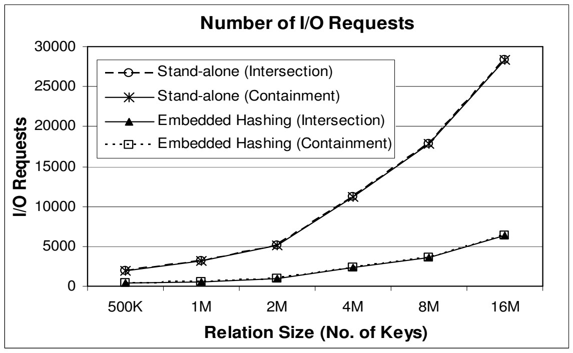

Figure 6. Non-coordinate-maintained indexes: I/O Requests

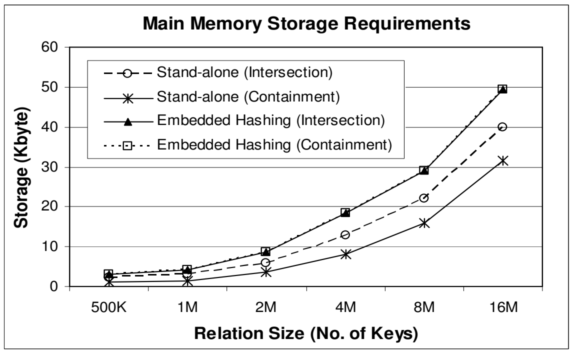

Figure 5. Non-coordinate-maintained indexes: Storage requirements

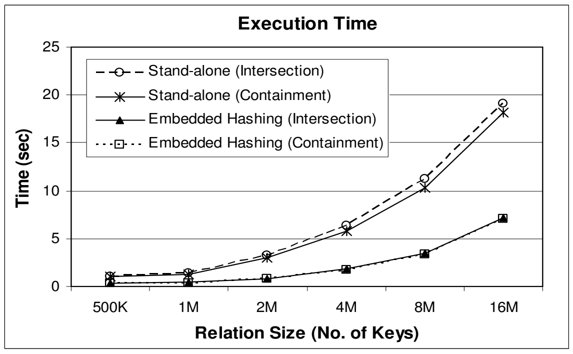

Figure 7. Non-coordinate-maintained indexes: Execution time

inside the query box). Both queries have the same coordinates, i.e., the same query box. We run the experiments over a PMR quadtree that indexes line segments and an extended kd-tree that indexes rectangles. The results from both indexes show similar behavior. We present only the results obtained from the PMR quadtree because of the space limitations. The generated line segments are uniformly distributed over the two dimensional space. Half of the line segments are very small in length, and hence, their replication factor is very small, while the other half are large in length, and hence, their replication factor is relatively large.

Figures 3 and 4 depict the performance of the Consistency_Reference technique against the stand-alone DISTINCT operator approach. Figure 3 illustrates that Consistency_Reference does not require any additional storage, whereas the stand-alone DISTINCT operator approach requires an additional storage to maintain a hash table. The figure illustrates that the *intersection* query requires more storage because its answer set is larger than that of the *containment* query. With respect to the execution time, Figure 4 illustrates that Consistency_Reference requires around 50% of the time taken by the stand-alone DISTINCT operator approach. The reason is that Consistency_Reference saves the manipulation time of the hash table, e.g., insertion and searching times.

To measure the performance in the case of the non-coordinate-maintained indexes, we store the objects' coordinates in a separate table outside the index tree. Any coordinate retrieval operation is counted as one I/O request. Figures 5, 6, and 7 depict the performance of the duplicate elimination techniques of the non-coordinate-maintained indexes. Figure 5 illustrates that the proposed hashing technique requires more storage than the stand-alone DISTINCT operator approach. This is because Embedded_Hashing stores all encountered objects including objects that do not satisfy the query. In Embedded_Hashing, the *intersection* and *containment* queries have the same storage overhead because the required storage depends on the encountered objects not on the objects that satisfy the query. In Figure 6, we present the number of I/O requests performed by each technique. The stand-alone DISTINCT operator approach performs much more I/O requests because it retrieves the coordinates of the encountered objects each time an object appears, whereas Embedded_Hashing retrieves the coordinates of each encountered object only once. The *intersection* and *containment* queries require the same number of I/O requests because this number depends on the encountered objects not on the objects that satisfy the query. In Figure 7, we present the execution time taken by each technique. The figure illustrates that Embedded_Hashing requires less execution time since it performs less I/O operations.

8 Conclusion

We presented generic duplicate elimination techniques for the class of space partitioning trees in the context of SP-GiST. The proposed techniques are implemented inside the SP-GiST INDEX-SCAN operator. Hence, we avoid using an expensive stand-alone DISTINCT operator in the query plan. We considered two cases for the index structures based on whether or not the objects' coordinates are stored inside the index tree. We proposed the Consistency_Reference technique to eliminate duplicates in indexes that store the objects' coordinates and the Embedded_Hashing technique to eliminate duplicates in indexes that do not store the objects' coordinates. The theoretical and experimental analysis illustrate the efficiency of the proposed techniques with respect to storage requirements, I/O operations, and processing time compared to adding a separate duplicate elimination operator in the query plan.

Acknowledgments

The work of Mourad Ouzzani was supported in part by a Lilly Endowment grant and a US DHS (PURVAC) grant.

References

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on linear key for a rectangle retrieval problem. In *IJCV*, pages 1–13, 1983.

[2] W. G. Aref, D. Barbará, and P. Vallabhaneni. The handwritten trie: Indexing electronic ink. In *SIGMOD*, pages 151–162, 1995.

[3] W. G. Aref and I. F. Ilyas. An extensible index for spatial databases. In *SSDBM*, pages 49–58, 2001.

[4] W. G. Aref and I. F. Ilyas. Sp-gist: An extensible database index for supporting space partitioning trees. *J. Intell. Inf. Syst.*, 17(2-3):215–240, 2001.

[5] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *CIKM*, pages 347–354, 1994.

[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[7] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE TSE-5:333–340*, 1979.

[8] W. A. Burkhard. Hashing and trie algorithms for partial match retrieval. *ACM Transactions Database Systems*, 1(2):175–187, 1976.

[9] J. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *ICDE*, page 535, 2000.

[10] R. Elmasri and S. B. Navathe. Fundamentals of database systems. In *Benjamin/Cummings, Redwood City*, 1989.

[11] M. Y. Eltabakh, R. H. Eltarras, and W. G. Aref. Space-partitioning trees in postgresql: Realization and performance. In *ICDE*, pages 100–111, 2006.

[12] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[13] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[14] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[15] T. M. Ghanem, R. Shah, M. F. Mokbel, W. G. Aref, and J. S. Vitter. Bulk operations for space-partitioning trees. In *ICDE*, pages 29–40, 2004.

[16] G. Kedem. The quad-cif tree: A data structure for hierarchical on-line algorithms. In *conference on Design automation*, pages 352–357, 1982.

[17] T. Matsuyama, L. V. Hao, and M. Nagao. A fi le organization for geographic information systems. In *IJCV*, pages 303–318, 1984.

[18] H. G. Molina, J. D. Ullman, and J. Widom. Database systems: The complete book. In *Prentice Hall*, 2001.

[19] R. C. Nelson and H. Samet. A population analysis for hierarchical data structures. In *SIGMOD*, pages 270–277, 1987.

[20] J. T. Robinson. The k-d-b-tree: a search structure for large multidimensional dynamic indexes. In *SIGMOD*, pages 10–18, 1981.

[21] H. Samet. The design and analysis of spatial data structures. In *Addison-Wesley, Reading MA*, 1990.

[22] C. A. Shaffer. Application of alternative quadtree representations. In *Ph.D. dissertation, TR-1672, Computer Science Departement, Univ. of Maryland, Collage Park, MD*, 1986.

[23] F. Wang. Relational-linear quadtree approach for two-dimensional spatial representation and manipulation. *TKDE*, 3(1):118–122, 1991.