Space-partitioning Trees in PostgreSQL: Realization and Performance *

Mohamed Y. Eltabakh Ramy Eltarras Walid G. Aref
Computer Science Department, Purdue University
{meltabak, rhassan, aref}@cs.purdue.edu

Abstract

Many evolving database applications warrant the use of non-traditional indexing mechanisms beyond B+-trees and hash tables. SP-GiST is an extensible indexing framework that broadens the class of supported indexes to include disk-based versions of a wide variety of space-partitioning trees, e.g., disk-based trie variants, quadtree variants, and kd-trees. This paper presents a serious attempt at implementing and realizing SP-GiST-based indexes inside PostgreSQL. Several index types are realized inside PostgreSQL facilitated by rapid SP-GiST instantiations. Challenges, experiences, and performance issues are addressed in the paper. Performance comparisons are conducted from within PostgreSQL to compare update and search performances of SP-GiST-based indexes against the B+-tree and the R-tree for string, point, and line segment data sets. Interesting results that highlight the potential performance gains of SP-GiST-based indexes are presented in the paper.

1 Introduction

Many emerging database applications warrant the use of non-traditional indexing mechanisms beyond B+-trees and hash tables. Database vendors have realized this need and have initiated efforts to support several non-traditional indexes, e.g., (Oracle [37], and IBM DB2 [1]).

One of the major hurdles in implementing non-traditional indexes inside a database engine is the very wide variety of such indexes. Moreover, there is tremendous overhead associated with realizing and integrating any of these indexes inside the engine. Generalized search trees (e.g., GiST [21] and SP-GiST [3, 4]) are designed to address this problem.

Generalized search trees (GiST [21]) and Space-partitioning Generalized search trees (SP-GiST [3, 4]) are software engineering frameworks for rapid prototyping of indexes inside a database engine. GiST supports the class of balanced trees (B+-tree-like trees), e.g., R-trees [7, 20, 34], SR-trees [25], and RD-trees [22], while SP-GiST supports the class of space-partitioning trees, e.g., tries [10, 16], quadtrees [15, 18, 26, 30], and kd-trees [8]. Both frameworks have internal methods that furnish general database functionalities, e.g., generalized search and insert algorithms, as well as user-defined external methods and parameters that tailor the generalized index into one instance index from the corresponding index class. GiST has been tested in prototype systems, e.g., in Predator [36] and in PostgreSQL [39], and is not the focus of this study.

The purpose of this study is to demonstrate feasibility and performance issues of SP-GiST-based indexes. Using SP-GiST instantiations, several index types are realized rapidly inside PostgreSQL that index string, point, and line segment data types. In addition, several advanced search operations are developed inside the SP-GiST framework. In particular, in addition to the standard index maintenance and search mechanisms, we realized the nearest-neighbor (NN) search algorithm proposed in [23] to support NN search over space partitioning trees. Performance comparisons are conducted from within PostgreSQL to compare update and search performances of (1) a disk-based trie variant against the B+-tree for a variety of string dataset collections, (2) a disk-based kd-tree variant against the R-tree for two-dimensional point dataset collections, and (3) a disk-based quadtree variant (the PMR-quadtree [30]) against the R-tree for line segment datasets. In addition to the performance gains and the advanced search functionalities provided by SP-GiST indexes, it is the ability to rapidly prototype these indexes inside a DBMS that is most attractive.

The contributions of this paper are as follows:

1. We realized SP-GiST inside PostgreSQL to extend the available access methods to include the class of space-partitioning trees, e.g., quadtrees, tries, kd-trees and suffix trees. Our implementation methodology makes SP-GiST portable, i.e., SP-GiST is realized inside PostgreSQL without recompiling PostgreSQL.

2. We extended the index operations in SP-GiST to in-

*This work was supported in part by the National Science Foundation under Grants IIS-0093116, IIS-0209120, and 0010044-CCR.

clude *prefix* and *regular expression match*, and a generic incremental NN search for SP-GiST-based indexes.

3. We conducted extensive experiments from within PostgreSQL to compare the performance of SP-GiST indexes against the B+-tree and R-tree. Our results show that a disk-based SP-GiST trie performs more than 2 orders of magnitude better than the B+-tree for a *regular expression match* search, and that a disk-based SP-GiST kd-tree performs more than 300% better than an R-tree for a *point match* search.

4. We realized a disk-bsed suffix tree index using SP-GiST to support *substring match* searching. Our experiments demonstrate that the suffix tree performs more than 3 orders of magnitude better than existing techniques.

5. We made the PostgreSQL version of SP-GiST available for public access and download at: www.cs.purdue.edu/spgist.

The rest of this paper proceeds as follows. In Section 2, we highlight related work. In Section 3, we overview space-partitioning trees, the challenges they have from database indexing point of view, and how these challenges are addressed in SP-GiST. Section 4 describes the implementation of SP-GiST inside PostgreSQL. Section 5 presents a new nearest-neighbor search functionality for SP-GiST. In Section 6, we present the performance results of a disk-based SP-GiST trie vs. the B+-tree for string data sets, and a disk-based SP-GiST kd-tree and PMR quadtree vs. the R-tree for two-dimensional point and line segment data sets, respectively. Section 7 contains concluding remarks.

2 Related Work

Multidimensional searching is a fundamental operation for many database applications. Several index structures beyond B-trees [6, 11] and hash tables [14, 31] have been proposed for multidimensional data, e.g., [17, 29, 33, 35]. These index structures include the R-tree and its variants, e.g., [7, 20, 34], the quadtree and its variants, e.g., [15, 18, 26, 41], the kd-tree [8] and its disk-based variants, e.g., [9, 32], and the trie and its variants [2, 10, 16]. Extensions to the B-tree have been proposed to index multidimensional data, e.g., [5, 13]. Extensible indexing frameworks have been proposed to instantiate a variety of index structures in an efficient way and without modifying the database engine. Extensible indexing frameworks are first proposed in [38]. GiST (*Generalized Search Trees*) is an extensible framework for B-tree-like indexes [21]. SP-GiST (*Space Partitioning Generalized Search Trees*) is an extensible framework for the family of space-partitioning trees [3, 4, 19]. Extensible indexing structures are important in the context of object-relational database management systems to support new data types. The implementation of GiST in *Informix Dynamic Server with Universal Data Option* (IDS/UDO) is presented in [27]. Commercial databases have supported extensible indexing frameworks, e.g., IBM DB2 [1], and Oracle [37]. The performance of various index structures have been studied extensively. For example, a model for the R-tree performance is proposed in [40]. R-tree and quadtree variants are compared in [24] and from within Oracle Spatial in [28].

3 Space-partitioning Trees: Overview, Challenges, and SP-GiST

The main characteristic of space-partitioning trees is that they partition the multi-dimensional space into disjoint (non-overlapping) regions. Refer to Figures 1, 2, and 3, for a few examples of space-partitioning trees. Partitioning can be either (1) space-driven (e.g., Figure 2), where we decompose the space into equal-sized partitions regardless of the data distribution, or (2) data-driven (e.g., Figure 3), where we split the data set into equal portions based on some criteria, e.g., based on one of the dimensions.

There are many types of trees in the class of space-partitioning trees that differ from each other in various ways. Without loss of generality, and for the simplicity of this discussion, we highlight below some of the important variations in the context of the trie data structure.

- **Path Shrinking** (refer to Figure 1) - The problem is that we may want to avoid lengthy and skinny paths from a root to a leaf. Paths of one child can be collapsed into one node. For example, the Patricia trie allows for leaf-shrinking (Shrinking single child nodes at the leaf level nodes, e.g., Figure 1(b)), while it is also possible to allow for path-shrinking (Shrinking single child nodes at the non-leaf level nodes, e.g., Figure 1(c)), or even no shrinking at all (Figure 1(a)).

- **Node Shrinking** (refer to Figure 2) - The problem is that with space-driven partitions, some partitions may end up being empty. So, the question is: Do we allow that empty partitions be omitted? For example, the difference between the standard trie (Figure 2(a)) and the forest trie (Figure 2(b)) is that the latter allows for empty partitions to be eliminated.

- **Clustering** - This is one of the most serious issues when addressing disk-based space-partitioning trees. The problem is that tree nodes do not map directly to disk pages. In fact, tree nodes are usually much smaller than disk pages. So, the question is: How do we pack

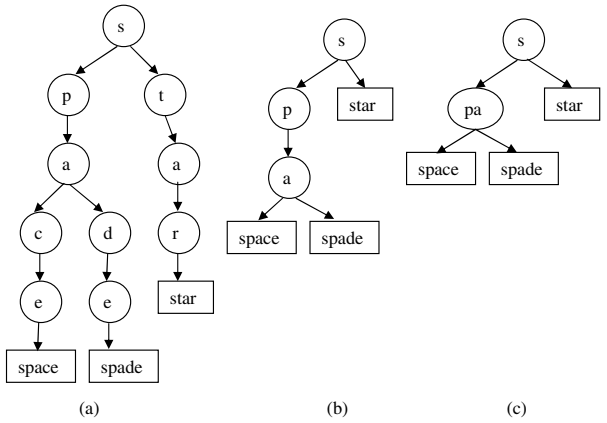


Figure 1. Trie variants. (a) No tree shrink, (b) Leaf shrink, (c) Path shrink.

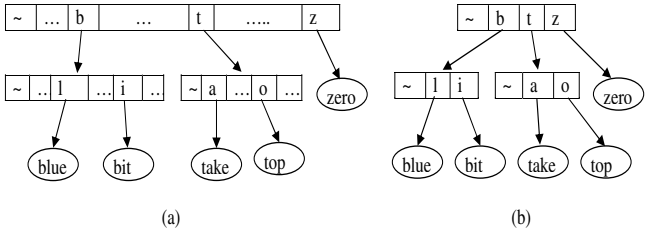


Figure 2. Trie variants. (a) No node shrink, (b) Node shrink.

tree nodes into disk pages with the objective of reducing disk I/Os for tree search and update? An optimal node-packing algorithm already exists that solves this issue [12].

Other characteristics of importance to space-partitioning trees include the bucket size of leaf nodes, the resolution of the underlying space, the support for various data types, the splitting of nodes (when to trigger a split and how node splitting is performed), and how concurrency control of space-partitioning trees is performed. For more discussion on these issues as they relate to space-partitioning trees, the reader is referred to [3, 4, 19].

3.1 SP-GiST

SP-GiST is an extensible indexing framework that broadens the class of supported indexes to include disk-based versions of a wide variety of space-partitioning trees, e.g., disk-based trie variants, quadtree variants, and kd-trees.

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 and are the same for all SP-GiST-based indexes. 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:

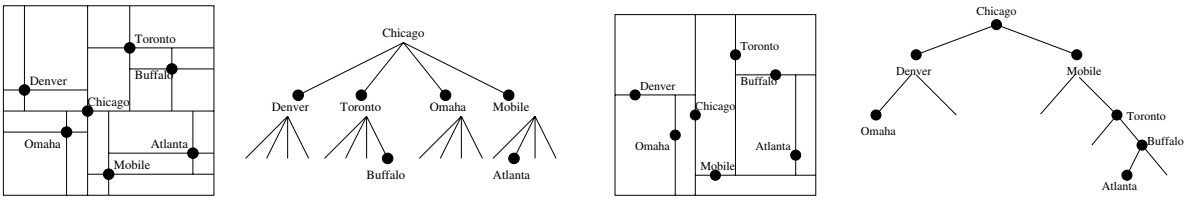
- *NodePredicate:* This parameter specifies the predicate type at the index nodes.
- *KeyType:* This parameter specifies the data type stored at the leaf nodes.
- *NumberofSpacePartitions:* This parameter specifies the number of disjoint partitions produced at each decomposition.
- *Resolution:* This parameter limits the number of space decompositions and is set depending on the required granularity.
- *PathShrink:* This parameter specifies how the index tree can shrink. *PathShrink* takes one of three possible values: *NeverShrik*, *LeafShrink*, and *TreeShrink*.
- *NodeShrink:* A *Boolean* parameter that specifies whether the empty partitions should be kept in the index tree or not.
- *BucketSize:* This parameter specifies the maximum number of data items a data node can hold.

For example, to instantiate the trie variants presented in Figure 1(a), (b), and (c), we set *PathShrink* to *NeverShrink*, *LeafShrink*, and *TreeShrink*, respectively. To instantiate the trie variants presented in Figures 2(a) and 2(b) we set *NodeShrink* to FALSE and TRUE, respectively. In the case of the quadtree and the kd-tree presented in Figures 3, *NoOfSpacePartitions* is set to 4 and 2, respectively.

The SP-GiST external methods include the method *PickSplit()* to specify 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-split is needed. Another external method is the *Consistent()* method 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 dictionary trie and the 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.

SP-GiST provides a default clustering technique that maps index nodes into disk pages [3, 4]. The clustering technique is based on [12] and is proven to generate minimum page-height trees.



(a) Point quadtree

(b) kd-tree

Figure 3. Example point quadtree and kd-tree.

	trie	kd-tree
Parameters	PathShrink = TreeShrink, NodeShrink = True BucketSize = B NoOfSpacePartitions = 27 NodePredicate = letter or blank KeyType = String	PathShrink = NeverShrink, NodeShrink = False BucketSize = 1 NoOfSpacePartitions = 2 NodePredicate = "left", "right", or blank KeyType = Point
Consistent(E,q,level)	If (q[level]==E.letter) OR (E.letter ==blank AND level $>$ length(q)) Return True, else Return False	If (level is odd AND q.x satisfies E.p.x) OR (level is even AND q.y satisfies E.p.y) Return True, else Return False
PickSplit(P,level)	Find a common prefix among words in P Update level = level + length of the common prefix Let P predicate = the common prefix Partition the data strings in P according to the character values at position "level" If any data string has length $<$ level, Insert data string in Partition "blank" If any of the partitions is still over full Return True, else 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 trie and kd-tree using SP-GiST.

4 Implementation Issues

In this section we discuss implementation issues in realizing SP-GiST inside PostgreSQL. First, we give an overview of the main extensible features of PostgreSQL. Then, we discuss the implementation of SP-GiST.

4.1 PostgreSQL Extensibility

PostgreSQL is an open-source object-relational database management system. PostgreSQL is extensible as most of its functionalities are table-driven. Information about the available data types, access methods, operators, etc., is stored in the system catalog tables. PostgreSQL incorporates user-defined functions into the engine through dynamically loadable modules, e.g., shared libraries. These loadable modules can be used to implement the functionality of new operators or access methods. The implementation of SP-GiST inside PostgreSQL makes use of the following features:

- **Defining New Interface Routines:** Each access method in PostgreSQL has a set of associated functions that perform the functionality of that access method. These functions are called, *interface routines*. The interface routines can be implemented as loadable modules.

- **Defining New Operators:** In the operator definition, we specify the data types on which the operator works. We also specify a set of properties that the query optimizer can use in evaluating the access methods.

- **Defining New Operator Classes:** Operator classes specify the data type and the operators on which a certain access method can work. In addition to linking an access method with data types and operators, operator classes allow users to define a set of functions called *support functions*, that are used by the access method to perform internal functions.

4.2 Realizing SP-GiST Inside PostgreSQL

The access methods currently supported by PostgreSQL (version 8.0.1) are: **Heap access:** Sequential scan over the relation, **B+-tree:** The default index access method, **R-tree:**

SP-GiST insert statement	INSERT INTO pg_am VALUES ('SP_GiST', 0, 20, 20, 0, 'f', 'f', 'f', 't', 'spgistgettuple', 'spgistinsert', 'spgistbeginscan', 'spgistrescan', 'spgistendscan', 'spgistmarkpos', 'spgistrestrpos', 'spgistbuild', 'spgistbulkdelete', '-' , 'spgistcostestimate');

Column name	Column description	SP-GiST function/value
amname	Name of the access method	SP_GiST
amowner	User ID of the owner	0
amstrategies	Max number of operator strategies for this access method	20
amsupport	Max number of support functions for this access method	20
amorderstrategy	The strategy number for entries ordering	0
amcanunique	Support unique index flag	FALSE
amcanmulticol	Support multicolumn flag	FALSE
amindexnulls	Support null entries flag	FALSE
amconcurrent	Support concurrent update flag	TRUE
amgettuple	"Next valid tuple" function	'spgistgettuple'
aminsert	"Insert this tuple" function	'spgistinsert'
ambeginscan	"Start new scan" function	'spgistbeginscan'
amrescan	"Restart this scan" function	'spgistrescan'
amendscan	"End this scan" function	'spgistendscan'
ammarkpos	"Mark current scan position" function	'spgistmarkpos'
amrestrpos	"Restore marked scan position" function	'spgistrestrpos'
ambuild	"Build new index" function	'spgistbuild'
ambulkdelete	Bulk-delete function	'spgistbulkdelete'
amvacuumcleanup	Post-VACUUM cleanup function	—
amcostestimate	Function to estimate cost of an index scan	'spgistcostestimate'

Table 2. pg_am catalog table entry for SP-GiST.

To support queries on spatial data, **Hash:** To support simple equality queries, **GiST:** Generalized index framework for the B-tree-like structures. By realizing SP-GiST inside PostgreSQL, we extend the access methods to include the family of space-partitioning trees, e.g., the kd-tree, the trie, the quadtree, and their variants. In the following, we discuss how we implement SP-GiST inside PostgreSQL.

- **Realization of SP-GiST Internal Methods**

  SP-GiST internal methods are the core part of the SP-GiST framework, and they are shared among all the space partitioning tree structures. To realize the internal methods, we use PostgreSQL access methods' interface routines (See Section 4.1). A new row is inserted into the *pg_am* table to introduce SP-GiST to PostgreSQL as a new access method (See Table 2). *pg_am* is a system catalog table that stores the information about the available access methods. The internal methods are defined as the interface routines of that access method.

  In Table 2 we illustrate the *pg_am* table entry for SP-GiST. The name of the new access method is set to 'SP_GiST'. We set the maximum number of the possible strategies (i.e., operators linked to an access method), and the maximum number of possible support functions to 20. Since SP-GiST index entries do not follow a certain order, we set the value of the *amorderstrategy* to 0. This value means that there is no strategy for ordering the index entries. The SP-GiST internal methods (e.g., *spgistgettuple(), spgistinsert()*, etc.) are assigned to the corresponding interface routine columns (e.g., *amgettuple, aminsert*, etc.).

  Estimating the cost of the SP-GiST index scan is performed by function *spgistcostestimate()*, which is assigned to column *amcostestimate*. *spgistcostestimate()* uses the generic cost estimate functions provided by PostgreSQL. Four cost parameters are estimated:

  1. **Index selectivity:** The index selectivity is the estimated fraction of the underlying table rows that will be retrieved during the index scan. The selectivity depends on the operator being used in the query. We associate with each operator that we define, a procedure that estimates the selectivity of that operator.
  2. **Index correlation:** The index correlation is set to 0 because there is no correlation between the index order and the underlying table order.

Query type	Query Semantic
Equality query	Return the keys that exactly match the query predicate.
Prefix query	Return the keys that have a prefix that matches the query predicate.
Regular expression query	Return the keys that match the query regular expression predicate.
Substring query	Return the keys that have a substring that matches the query predicate.
Range query	Return the keys that are within the query predicate range.
NN query	Return the keys sorted based on their distances from the query predicate.

Table 3. The semantic of the query types.

trie		kd-tree	
Equality operator '='	*Prefix match operator* '?='	*Equality operator* '@'	*inside operator* '∧'
CREATE OPERATOR = (leftarg = VARCHAR, rightarg = VARCHAR, procedure = trieword_equal, commutator = =, restrict = eqsel,);	CREATE OPERATOR ?= (leftarg = VARCHAR, rightarg = VARCHAR, procedure = trieword_prefix, restrict = likesel,);	CREATE OPERATOR @ (leftarg = POINT, rightarg = POINT, procedure = kdpoint_equal, commutator = @, restrict = eqsel,);	CREATE OPERATOR ∧ (leftarg = POINT, rightarg = BOX, procedure = kdpoint_inside, restrict = contsel,);

Table 4. The trie and kd-tree operator definitions.

3. **Index startup cost:** The startup cost is the CPU cost of evaluating any complex expressions that are arguments to the index. These expressions are evaluated once at the beginning of the index scan.

4. **Index total cost:** The total cost is the sum of the startup cost plus the disk I/O cost. The estimated disk I/O cost depends on the index selectivity and the index size.

SP-GiST internal methods are implemented as a dynamically loadable module that is loaded by the PostgreSQL dynamic loader when the index is first used. Therefore, the implementation of the internal methods is completely portable, and does not even require recompiling PostgreSQL's code.

- **Definition of SP-GiST Operators**

The various SP-GiST index structures have different sets of operators (external methods) to work on. For the trie index structure, we define the three operators; '=', '#=', and '?=', to support the equality queries, the prefix queries, and the regular expression queries, respectively. For the regular expression queries, the SP-GiST trie supports currently, the wildcard character; '?', that matches any single character. In the case of the kd-tree, we define two operators; '@' and '∧', to support the equality and range queries, respectively. We define one operator for the suffix tree, i.e., '@=', to support the substring match queries. The nearest-neighbor search, NN_search, (see Section 5) is defined as the operator '@@' that can be called from the SQL like all other operators. The NN distance function for each index structure is defined in the *NN_Consistent()* external method (see Section 5). For example, the kd-tree and quadtree may use the Euclidean distance function, while the trie may use the Hamming distance function. The semantics of the query types are given in Table 3.

An example of the operators' definitions is given in Table 4. Each operator is linked to a procedure that performs the operator's functionality, e.g., *trieword_equal(), kdpoint_equal()*. Other properties can be defined for each operator. For example, the *commutator* clause specifies the operator that the query optimizer should use, if it decides to switch the original operator's arguments.

Estimating the selectivity of each operator is performed by the procedures defined in the *restrict* clause. We use procedures provided by PostgreSQL, e.g., *eqsel(), contsel(), likesel()*. *eqsel()* estimates the selectivity of the equality operators. *contsel()* estimates the selectivity of the containment operators (i.e., range search), whereas, *likesel()* estimates the selectivity of the similarity operators, e.g., *LIKE* operator. The query optimizer uses these procedures to estimate the index selectivity and the index scan cost.

- **Realization of SP-GiST External Methods**

The SP-GiST external methods and interface parameters capture the differences among the various types of SP-GiST index structures. To realize the external methods inside PostgreSQL, we use the access methods' support functions. The support functions are provided within the definition of the operator classes (See Section 4.1). The definitions

trie	kd-tree	suffix tree
CREATE OPERATOR CLASS SP_GiST_trie FOR TYPE VARCHAR USING SP_GiST AS OPERATOR 1 =, AS OPERATOR 2 #=, AS OPERATOR 3 ?=, AS OPERATOR 20 @@, FUNCTION 1 trie_consistent, FUNCTION 2 trie_picksplit, FUNCTION 3 trie_NN_consistent, FUNCTION 4 trie_getparameters;	CREATE OPERATOR CLASS SP_GiST_kdtree FOR TYPE POINT USING SP_GiST AS OPERATOR 1 @, OPERATOR 2 ∧, OPERATOR 20 @@, FUNCTION 1 kdtree_consistent, FUNCTION 2 kdtree_picksplit, FUNCTION 3 kdtree_NN_consistent, FUNCTION 4 kdtree_getparameters;	CREATE OPERATOR CLASS SP_GiST_suffix FOR TYPE VARCHAR USING SP_GiST AS OPERATOR 1 @=, AS OPERATOR 20 @@, FUNCTION 1 suffix_consistent, FUNCTION 2 suffix_picksplit, FUNCTION 3 suffix_NN_consistent; FUNCTION 4 suffix_getparameters;

Table 5. The trie, kd-tree, and suffix tree operator class definitions.

	trie		kd-tree	
Index creation	CREATE TABLE word_data (name VARCHAR(50), id INT); CREATE INDEX sp_trie_index ON word_data USING SP_GiST (name SP_GiST_trie);		CREATE TABLE point_data (p POINT , id INT); CREATE INDEX sp_kdtree_index ON point_data USING SP_GiST (p SP_GiST_kdtree);	
	equality query	regular expression query	equality query	range query
Queries	SELECT * FROM word_data WHERE name = 'random';	SELECT * FROM word_data WHERE name ?= 'r?nd?m';	SELECT * FROM point_data WHERE p @ '(0,1)';	SELECT * FROM point_data WHERE p ∧ '(0,0,5,5)';

Table 6. The trie and kd-tree index creation and querying.

of the trie operator class (*SP-GiST_trie*), the kd-tree operator class (*SP-GiST_kdtree*), and the suffix tree operator class (*SP-GiST_suffix*) are given in Table 5. *SP-GiST_trie*, and *SP-GiST_suffix* use the data type VARCHAR, whereas, *SP-GiST_kdtree* uses the data type POINT.

Two examples for creating and querying the trie and kd-tree indexes are given in Table 6. The *USING* clause in the *CREATE INDEX* statement specifies the name of the access method to be used, that is 'SP_GiST' in our case. We then specify the column name to be indexed, and the corresponding operator class.

SP-GiST external methods are implemented as a dynamically loadable module that is loaded when the index is first used.

In Figure 4, we illustrate the architecture of SP-GiST inside PostgreSQL. The implementation of the SP-GiST core (i.e., internal methods) is fully isolated from the implementation of the SP-GiST extensions (i.e., external methods). The link between the core and the extensions is achieved through PostgreSQL operator classes. The communication among the methods is through the PostgreSQL function manager. The portability is achieved because both the SP-GiST core and

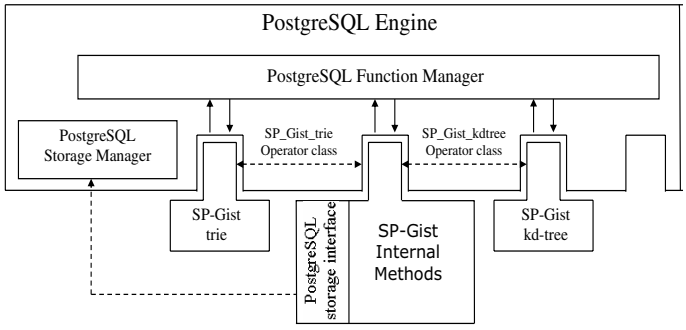


Figure 4. SP-GiST architecture inside PostgreSQL.

extensions are loadable modules. That is, SP-GiST can be realized inside PostgreSQL without recompiling PostgreSQL. We extended the internal methods to include functions, i.e., *PostgreSQL storage interface*, to communicate with the PostgreSQL storage manager for the allocation and retrieval of disk pages.

5 New Nearest-Neighbor Search in SP-GiST

We extended SP-GiST core internal methods to support incremental nearest-neighbor searching. Our extension is an adaptation of the algorithm in [23]. The outline of the

```
Insert the root node into the priority queue with minimum distance 0
While (priority queue is not empty)
    {
        - Retrieve the top of the queue into P
        - If (P is an object) Then
              - Report P as the next NN to the query object
        - Else
              - Compute the minimum distances between
                the query object and P's children
              - Insert P's children into their proper positions
                in the queue based on their distances
    }
```

Figure 5. Generic NN algorithm for SP-GiST

algorithm is given in Figure 5. The algorithm prioritizes and visits the space partitions based on their minimum distances from the query object. The partitions are maintained sorted in increasing order of their distances in a priority queue. Initially, the queue contains the root node with a minimum distance of 0. The algorithm recursively replaces the node at the top of the queue by the node's children (inserted in their proper positions based on their minimum distances) until a database object reaches the top of the queue. This object is reported as the next NN to the query object. The algorithm is incremental and can be used in a query pipeline such that every call to the algorithm (get-next) returns the next NN object.

To make the algorithm generic for all space-partitioning trees (not only for quadtrees and kd-trees), we modified the algorithm. For example, in the case of a trie, the NN algorithm has to remember the minimum distance of the parent node in order to compute the minimum distance of the children. The NN algorithm stores the minimum distance of a parent in the priority queue and uses it to compute the minimum distances of the parents children and stores them in the priority queue entries of each child.

To realize the NN search algorithm inside SP-GiST, we added the new internal method *NN_Search()* and the new external method *NN_Consistent()*. *NN_Search()* maintains a priority queue by retrieving the top of the queue P, to either report P as the next NN to the query object if P is a database object or replace P with its child nodes if P is an index node. *NN_Search()* is aware of neither the index data type nor how the distance function is computed. *NN_Consistent()* guides the *NN_Search()* method during the search. *NN_Consistent()* computes and returns the minimum distances between the query object and the index nodes or database objects sent to it from *NN_Search()*. *NN_Search()* then sorts these nodes and objects based on their distances and insert them into their proper positions in the priority queue.

	External methods code			
	trie	kd-tree	P quadtree	PMR quadtree
No. of lines	580	551	562	602
% of total lines	8.2	7.8	8.0	8.6

Table 7. Number and percentage of external methods' code lines

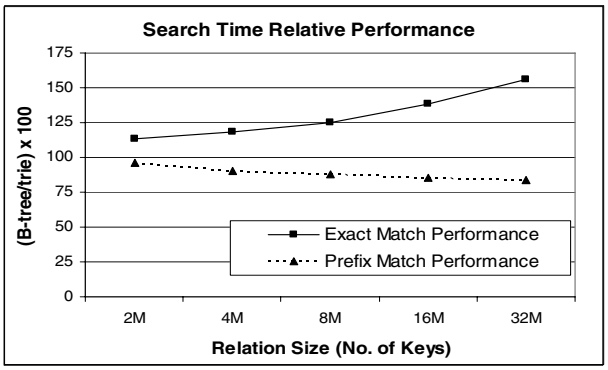


Figure 6. The search performance of the B+-tree vs. the patricia trie.

6 Experiments

Our main objective of this paper is not to show the superiority of one index structure over the other as we believe that the index performance depends heavily on the nature of data and the type of applications. Our objectives are to demonstrate the extensibility of SP-GiST to rapidly prototype new indexes and to highlight several strengths and weaknesses of SP-GiST indexes over B+-tree and R-tree indexes.

We realized the following index structures in PostgreSQL using SP-GiST: a disk-based patricia trie, kd-tree, point quadtree, PMR quadtree, and suffix tree. In Table 7, we provide the number and percentage of code lines that we added to realize these index structures. The table illustrates that the external methods that a developer needs to provide represent less than 10% of the total index coding. The other 90% of the code is provided as the SP-GiST core.

For the experimental results, we conduct our experiments from within PostgreSQL. We compare the performance of the SP-GiST trie against the performance of the B+-tree in the context of text string data. We also compare the performance of the SP-GiST kd-tree and PMR quadtree against the performance of the R-tree in the context of point and line segment data, respectively. We compare the performance of the suffix tree against sequential scanning because the other access methods do not support the substring match operations.

For the patricia trie versus B+-tree experiments, we generate datasets with size ranges from 500K words to 32M

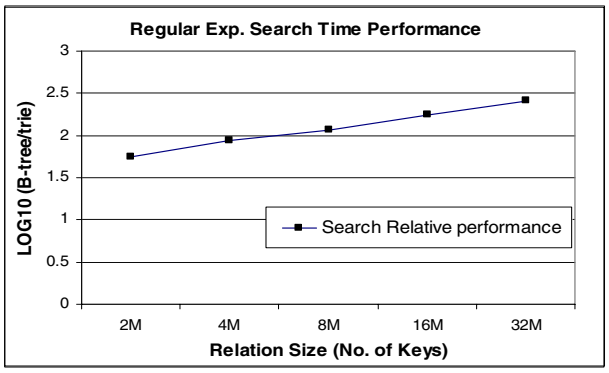


Figure 7. The regular exp. search of the B+-tree vs. the patricia trie.

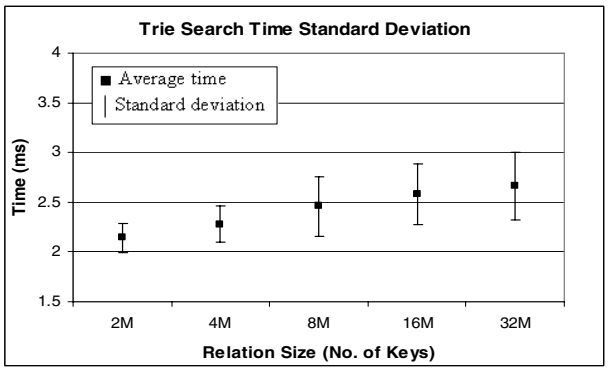


Figure 8. The trie search time standard deviation.

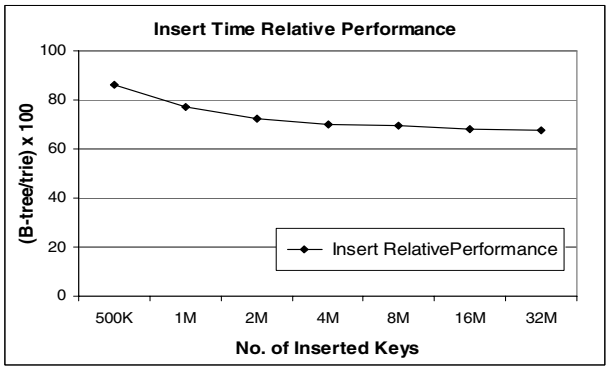


Figure 9. The insert performance of the B+-tree vs. the trie.

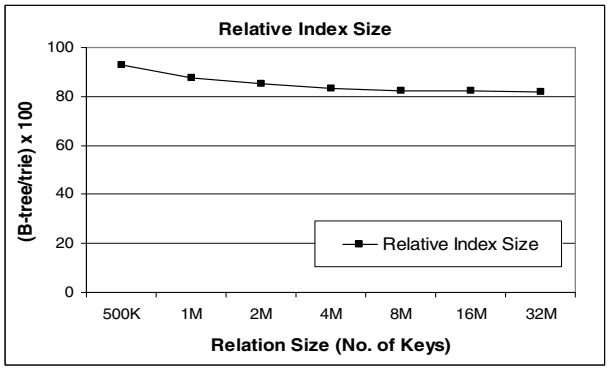


Figure 10. The index size of the B+-tree vs. the trie.

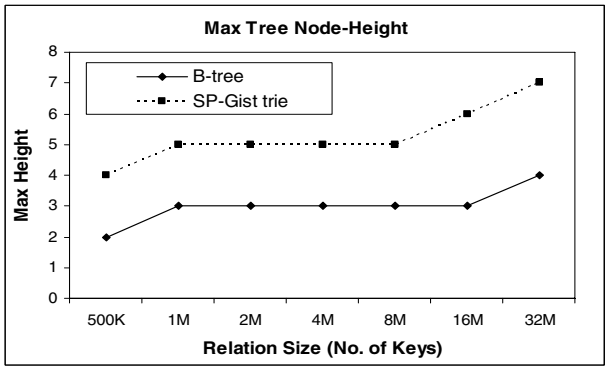


Figure 11. The maximum tree height in nodes.

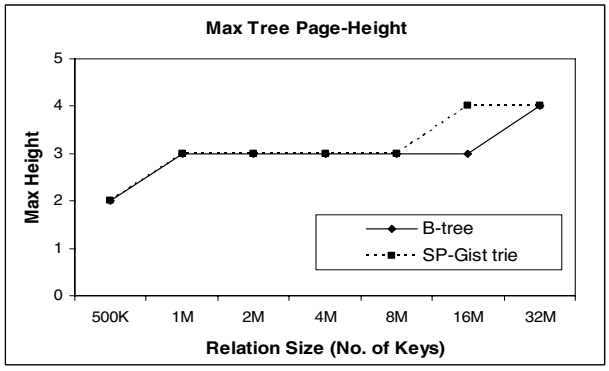


Figure 12. The maximum tree height in pages.

words. The word size (key size) is uniformly distributed over the range [1, 15], and the alphabet letters are from 'a' to 'z'. Our experiments illustrate that the trie has a better search performance than that of the B+-tree. In Figures 6 and 7, we demonstrate the performance of three search operations; *exact match*, *prefix match*, and *regular expression match*. Figure 6 illustrates that in the case of the *exact match* search, the trie has more than 150% search time improvement over the B+-tree, and that, the trie scales better especially with the increase in the data size.

For the *regular expression match* search (Figure 7), our experiments illustrate that the trie achieves more than 2 orders of magnitude search time improvement. Recall that, we only allow for the wildcard, '?', that matches any single character. We notice that the B+-tree performance is very sensitive to the positions of the wildcard; '?' in the search string. For example, if '?' appears in the 2nd or the 3rd positions, then the B+-tree performance will degrade significantly. Moreover, if '?' appears as the first character in the search string, then the B+-tree index will not be used at all, and a sequential scan is performed. The reason for this sensitivity is that the B+-tree makes use only of the search string's prefix that proceeds any wildcards. In contrast, the trie makes use of any non-wildcard characters in the search string to navigate in the index tree. Therefore, the trie is much more tolerant for the *regular expression match* queries. For example, to search for expression '?at?r', the trie matches all the entries of the tree root node with '?', then the 2nd and the 3rd tree levels are filtered based on letters 'a' and 't', respectively. At the 4th level of the tree, the entries of the reached nodes are matched with '?', and then the 5th level is filtered based on letter 'r'.

For the *prefix match* search (Figure 6), our experiments illustrate that the B+-tree has a better performance over the trie. The reason is that having the keys sorted in the B+-tree leaf nodes, allows the B+-tree to answer the *prefix match* queries efficiently. In contrast, the trie has to fork the navigation in the index tree in order to reach all the keys that match the search string.

In Figure 8, we present the search time standard deviation of the trie in the case of the *exact match* search to study the effect of the variation of the tree depth on the search performance. The insertion time and the index size of the B+-tree and the trie are presented in Figures 9 and 10, respectively. The figures demonstrate that the B+-tree scales better with respect to both factors. The reason is that the trie involves a higher number of nodes and a higher number of node splits than the B+-tree because the trie node size is much smaller than the B+-tree node size. In Figures 11 and 12, we present the B+-tree and the trie maximum tree height in nodes and pages, respectively. Although the trie has higher maximum node-height, as it is an unbalanced tree, the maximum page-height is almost the same as the

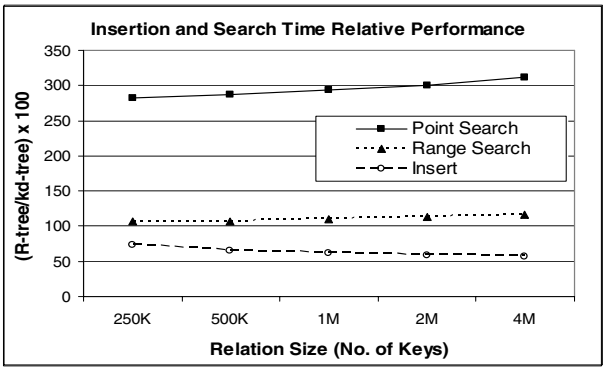


Figure 13. The performance of the R-tree vs. the kd-tree.

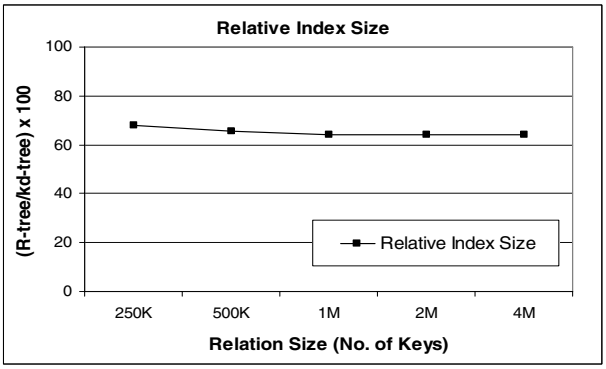


Figure 14. The index size of the R-tree vs. the kd-tree.

B+-tree page-height. Recall that SP-GiST uses a clustering technique that tries to minimize the tree maximum page height, which is effective.

For the comparison of the kd-tree against the R-tree, we conduct our experiments over two-dimensional point datasets. The x-axis and the y-axis range from 0 to 100. We generate datasets of sizes that range from 250K to 4M two-dimensional points. We illustrate in Figure 13 the search performance under two search operations; the *point match* search and the *range* search. The figure illustrates that the SP-GiST kd-tree has more than 300% search time improvement over the R-tree in the case of the *point match* search, and it has around 125% performance gain in the case of the *range* search. However, the experiments demonstrate that the R-tree has a better insertion time (Figure 13) and a better index size (Figure 14) than the kd-tree. The reason is that the kd-tree is a binary search tree, where the node size (*BucketSize*) is 1, and almost every insert results in a node split. Therefore, the number of the kd-tree nodes is very large, and in order for the storage clustering technique to reduce the tree page-height, it has to degrade the index page utilization, which results in an increase in the index size.

In Figure 15, we compare the performance of the

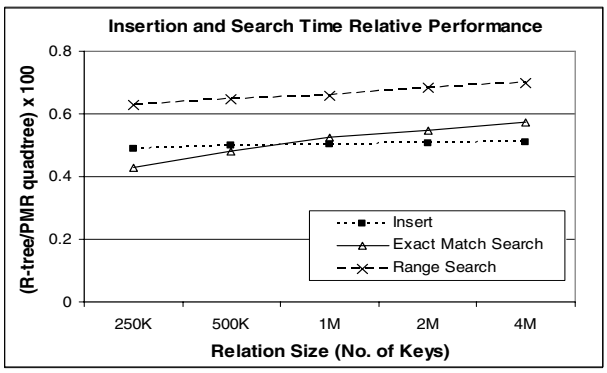


Figure 15. The performance of the R-tree vs. the PMR quadtree.

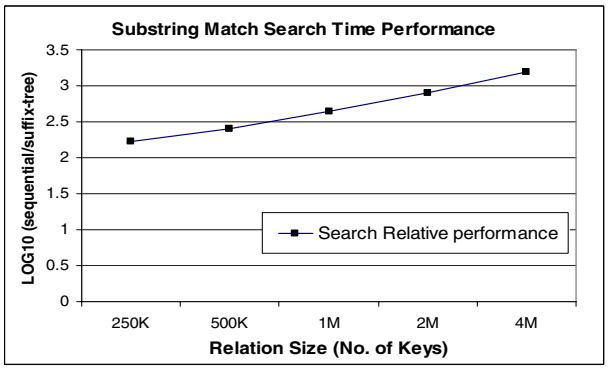


Figure 16. Suffix tree search performance.

PMR quadtree against the R-tree for indexing line segment datasets. We measured the insertion time and the *exact match* and *range (window)* search times. The figure illustrates that the R-tree has a better insertion and search performance than that of the PMR quadtree. The relative insertion performance between the R-tree and the PMR quadtree is almost constant with the increase in the data size. Whereas, the search performance gap decreases with the increase of the data size. Similar results are presented in [28]. The experiments in [28] show that under certain query types, e.g., *overlap queries*, the quadtree may have a better search performance than the R-tree.

With respect to the suffix tree performance, we illustrate in Figure 16, the significant performance gain of using the suffix tree index to support the *substring match* search. The performance gain is more than 3 orders of magnitude over the sequential scan search. The other index types do not support the *substring match* search.

We measured the NN search performance for various SP-GiST instantiations of index structures, mainly, the kd-tree, the point quadtree, and the patricia trie. The Euclidean distance is used as the distance function for the kd-tree and point quadtree, while the Hamming distance is used as the distance function for the trie. In Figure 17, we illustrate the

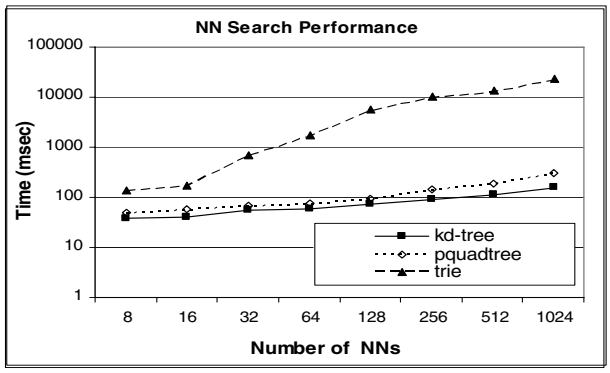


Figure 17. NN search performance

execution time taken to answer the NN query. We inserted 2M tuples in each relation and varied the required number of NNs from 8 to 1024 (we assume that the number of required NNs is controlled by the application using cursors). The figure illustrates that NN search over the trie is much slower than that over the kd-tree and point quadtree. The reason is that the comparison in the case of the trie is performed character by character which makes the convergence to the next NN relatively slow. Whereas, the comparison in the case of the kd-tree and quadtree is Partition-based. Moreover, the Hamming distance has a slow progress compared to the Euclidean distance as the Hamming distance updates the distance value with either 0 or 1 only at each step.

7 Conclusion and Future Research

We presented a serious attempt at implementing and realizing SP-GiST-based indexes inside PostgreSQL. We realized several index structures, i.e., the trie, kd-tree, point quadtree, PMR quadtree, and suffix tree. Several implementation challenges, experiences, and performance issues are addressed in the paper. Our experiments demonstrate the potential gain of the SP-GiST indexes. For example, the trie has more than 150% search performance improvement over the B+-tree in the case of the *exact match* search, and it has more than 2 orders of magnitude search performance gain over the B+-tree in the case of the *regular expression match* search. The kd-tree also has more than 300% search performance improvement over the R-tree in the case of the *point match* search. Several advanced search operations are realized inside SP-GiST such as NN search and substring match operations. In addition to the performance gains and the advanced search functionalities provided by SP-GiST indexes, it is the ability to rapidly prototype these indexes inside a DBMS that is most attractive. Our experiments demonstrate also several weaknesses of SP-GiST indexes that need to be addressed in future research. For example, the insertion time and the index size of the SP-GiST indexes involve higher overhead than those of the B+-tree and the

R-tree indexes.

References

[1] Ibm corp.: Ibm db2 universal database application development guide, vs. 6. 1999.

[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] R. Bayer. The universal b-tree for multidimensional indexing: general concepts. In *WWCA*, pages 198–209, 1997.

[6] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

[7] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r* -tree: An efficient robust access method for points and rectangles. In *SIGMOD Record, 19(2).*, 1990.

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

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

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

[11] D. Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[12] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan. Clustering techniques for minimizing external path length. In *VLDB*, pages 342–353, 1996.

[13] G. Evangelidis, D. B. Lomet, and B. Salzberg. The hb-pi-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *VLDB Journal*, 6(1):1–25, 1997.

[14] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashinga fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

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

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

[17] V. Gaede and O. Gőnther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

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

[19] 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.

[20] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[21] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *VLDB*, pages 562–573, 1995.

[22] J. M. Hellerstein and A. Pfeffer. The rd-tree: An index structure for sets. In *Univ. of Wisconsin CS Technical Report 1252*, 1994.

[23] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *SDD*, pages 83–95, 1995.

[24] E. G. Hoel and H. Samet. A qualitative comparison study of data structures for large line segment databases. In *SIGMOD*, pages 205–214, 1992.

[25] N. Katayama and S. Satoh. The sr-tree: an index structure for high-dimensional nearest neighbor queries. In *SIGMOD*, pages 369–380, 1997.

[26] G. Kedem. The quad-cif tree: A data structure for hierarchical on-line algorithms. In *19th conference on Design automation*, pages 352–357, 1982.

[27] M. Kornacker. High-performance extensible indexing. In *VLDB*, pages 699–708, 1999.

[28] R. Kothuri, S. Ravada, and D. Abugov. Quadtree and r-tree indexes in oracle spatial: a comparison using gis data. In *SIGMOD*, pages 546–557, 2002.

[29] R. K. Kothuri and S. Ravada. Efficient processing of large spatial queries using interior approximations. In *SSTD*, pages 404–424, 2001.

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

[31] R. L. Rivest. Partial-match retrieval algorithms. In *SIAM J. Comput., 5(1)*, pages 19–50, 1976.

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

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

[34] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *VLDB*, pages 507–518, 1987.

[35] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. Multidimensional access methods: Trees have grown everywhere. In *VLDB*, pages 13–14, 1997.

[36] P. Seshadri. Predator: A resource for database research. In *SIGMOD Record, 27(1),*, pages 16–20, 1998.

[37] J. Srinivasan, R. Murthy, S. Sundara, N. Agarwal, and S. DeFazio. Extensible indexing: a framework for integrating domain-specific indexing schemes into oracle8i. In *ICDE*, pages 91–100, 2000.

[38] M. Stonebraker. Inclusion of new types in relational data base systems. In *ICDE*, pages 262–269, 1986.

[39] M. Stonebraker and G. Kemnitz. The postgres next generation database management system. *Commun. ACM*, 34(10):78–92, 1991.

[40] Y. Theodoridis and T. Sellis. A model for the prediction of r-tree performance. In *PODS*, pages 161–171, 1996.

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