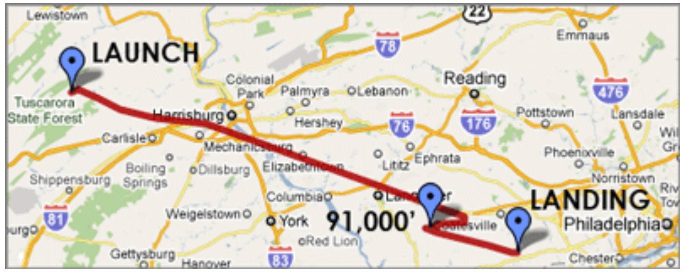
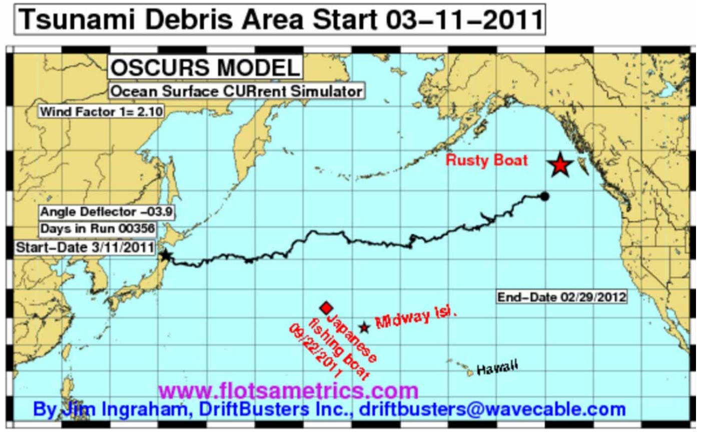
Systems-Oriented Multi-Dimensional Indexes: SP-GiST - The Case for Space-Partitioning Trees

Walid G. Aref - Purdue University and Alexandria University-Egypt

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 Spatial and IBM DB2




Tsunami Debris Area Start 03-11-2011
OSCURS MODEL
Ocean Surface CURrent Simulator
Wind Factor 1= 2.10
Rusty Boat
Angle Deflector -03.9
Days in Run 00356
Start-Date 3/11/2011
End-Date 02/29/2012
Japanese fishing boat 09/22/2011
Midway Isl.
Hawaii
www.flotsametrics.com
By Jim Ingraham, DriftBusters Inc., driftbusters@wavecable.com

Hurdles: (1) Many New Indexes

- One of the major hurdles in implementing non-traditional indexes inside a database engine is:
  - The very wide variety of such indexes
- Example:
  - Spatio-temporal Access Methods (2003)
    ~ 45 indexes
  - Spatio-Temporal Access Methods: Part 2 (2003 - 2010)
    ~ 90 indexes
  - Spatio-Temporal Access Methods: A Survey (2010 - 2017)
    ~ 170 indexes
- Very hard for systems engineers to keep up

Bulletin of the Technical Committee on

Data Engineering

June 2003 Vol. 26 No. 2 IEEE Computer Society

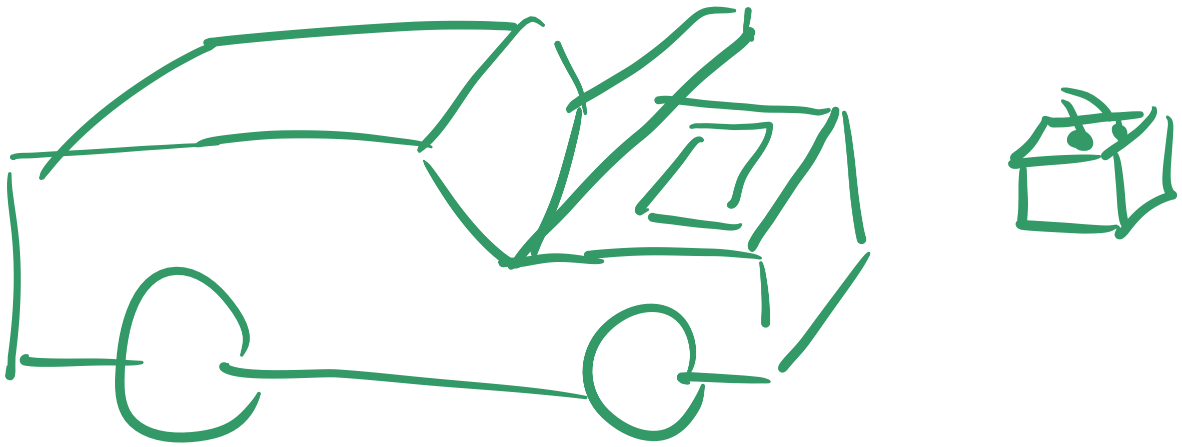
Letters

Special Issue on Infrastructure for Research in Spatio-Temporal Query Processing

Conference and Journal Notices

Hurdles: (2) Overhead in Realizing the Index

- Tremendous overhead associated with realizing and integrating any of these indexes inside the engine
- It can take many programmer-months to realize one index inside the DBMS
- In addition to realizing the logic of the index operations
  1. Concurrency control, locking, lock-free
  2. Recovery in the presence of failures, transaction aborts, and system crashes
  3. Resource management
  4. Disk, memory, and buffer management



Generalized Search Trees (GiST and SP-GiST)

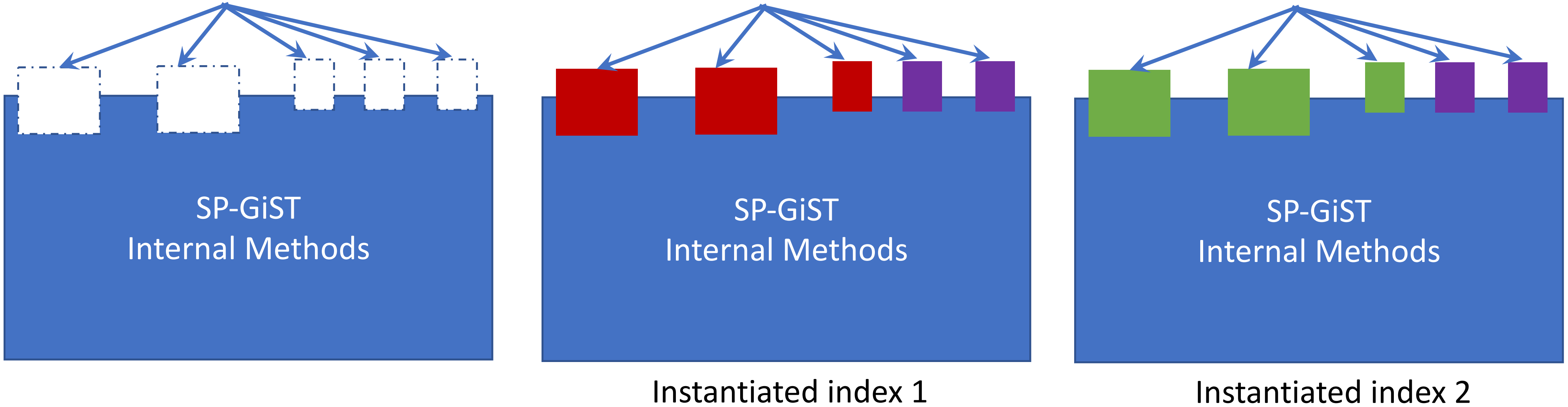
- Software engineering frameworks for rapid prototyping of indexes inside a database engine
- Umbrella indexes that generalize a certain class of indexes
- GiST [VLDB 1995]:
  - Generalizes B+-tree-like trees
    - R-trees, SR-trees, and RD-trees
  - Balanced trees
  - Every node maps to a disk page
- SP-GiST [SSDBM 2001, ICDE 2006]
  - Generalizes quad-tree-like class of indexes
    - Variants of quadtrees and tries
  - Unbalanced skinny trees
  - Multiple tree nodes map to a disk page
- GiST and SP-GiST are realized inside PostgreSQL

Generalized Search Trees (GiST and SP-GiST)

- Have internal methods that furnish general database functionalities
    - e.g., generalized search and insert algorithms
- User-defined external methods and parameters
    - Tailor the generalized index into one instance index from the corresponding index class

Space-partitioning Generalized Search Tree

- Supports disk-based trie variants, disk-based quadtree variants, and disk-based kd-trees
- SP-GiST is realized inside of PostgreSQL (Since PostgreSQL 8.0-8.5, 9.2 till today)
- Has user-specified ***pluggable*** ***modules*** *and* ***parameters*** that when provided:
  - Allows the instantiation of one member space-partitioning index with less effort (in a matter of days)

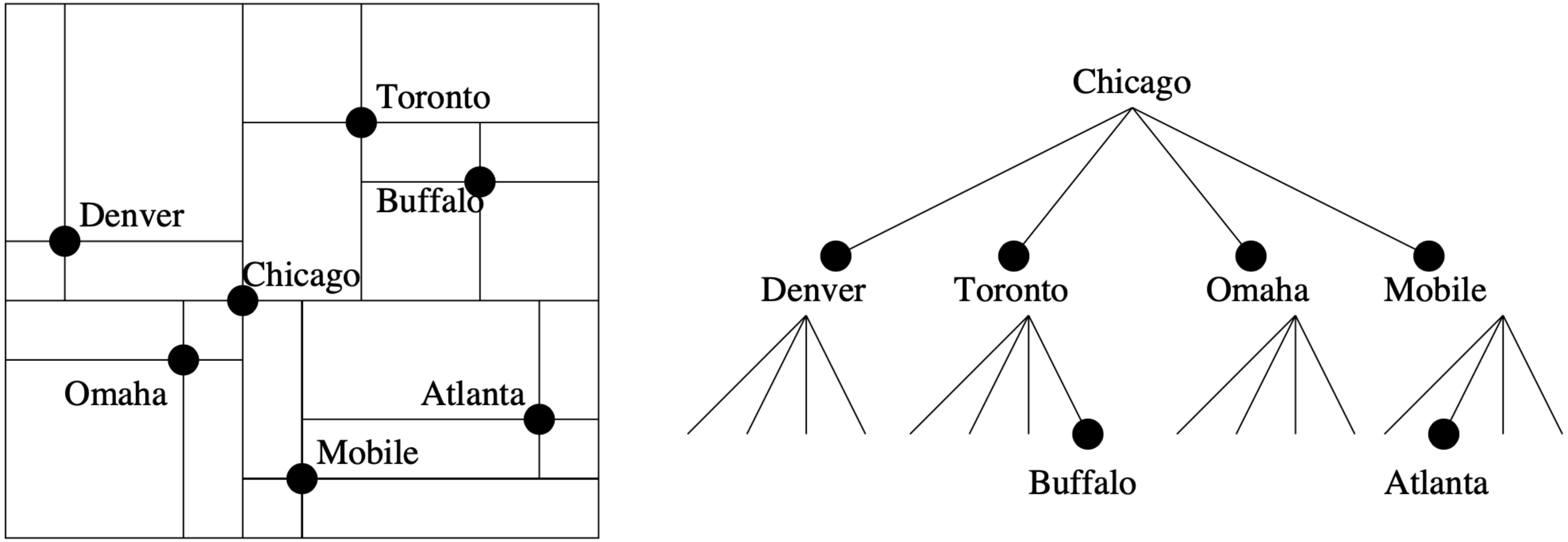


Space-Partitioning Trees

- Partition the multi-dimensional space into disjoint (non-overlapping) regions
- Partitioning can be either
    - Space-driven:
        - Decompose the space into equal-sized partitions regardless of the data distribution
    - Data-driven:
        - Split the data set into equal portions based on some criteria, e.g., based on one of the dimensions

Space-Partitioning Trees

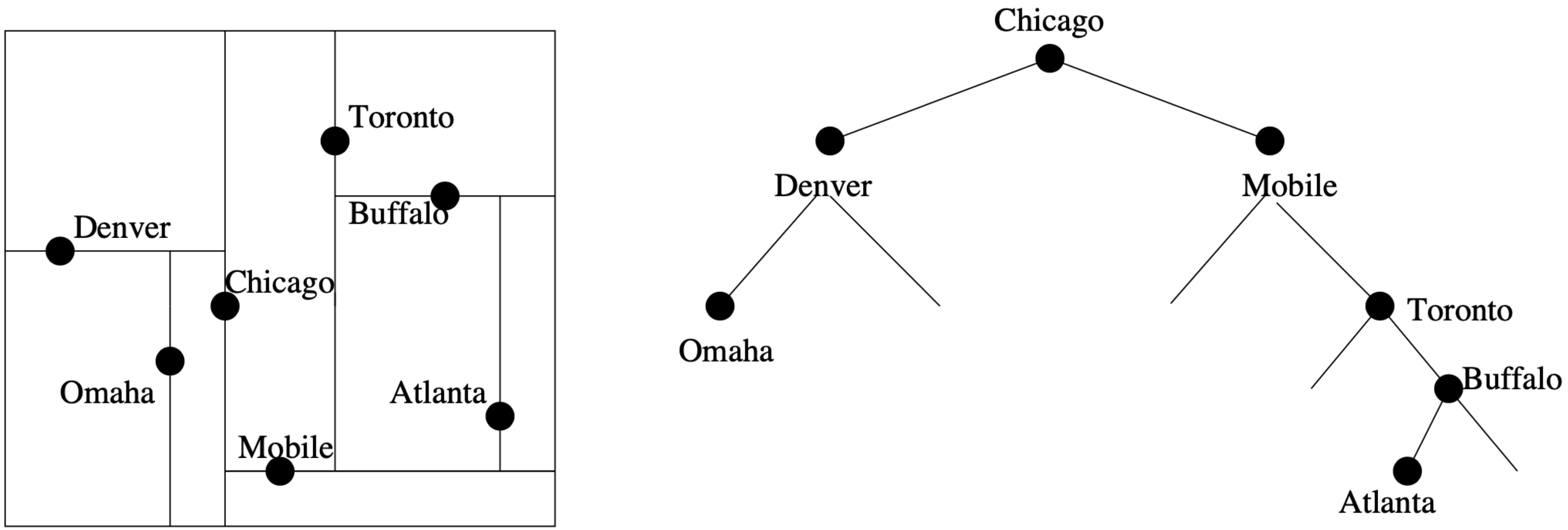
- Partition the multi-dimensional space into disjoint (non-overlapping) regions



(a) Point quadtree

Space-Partitioning Trees (2)

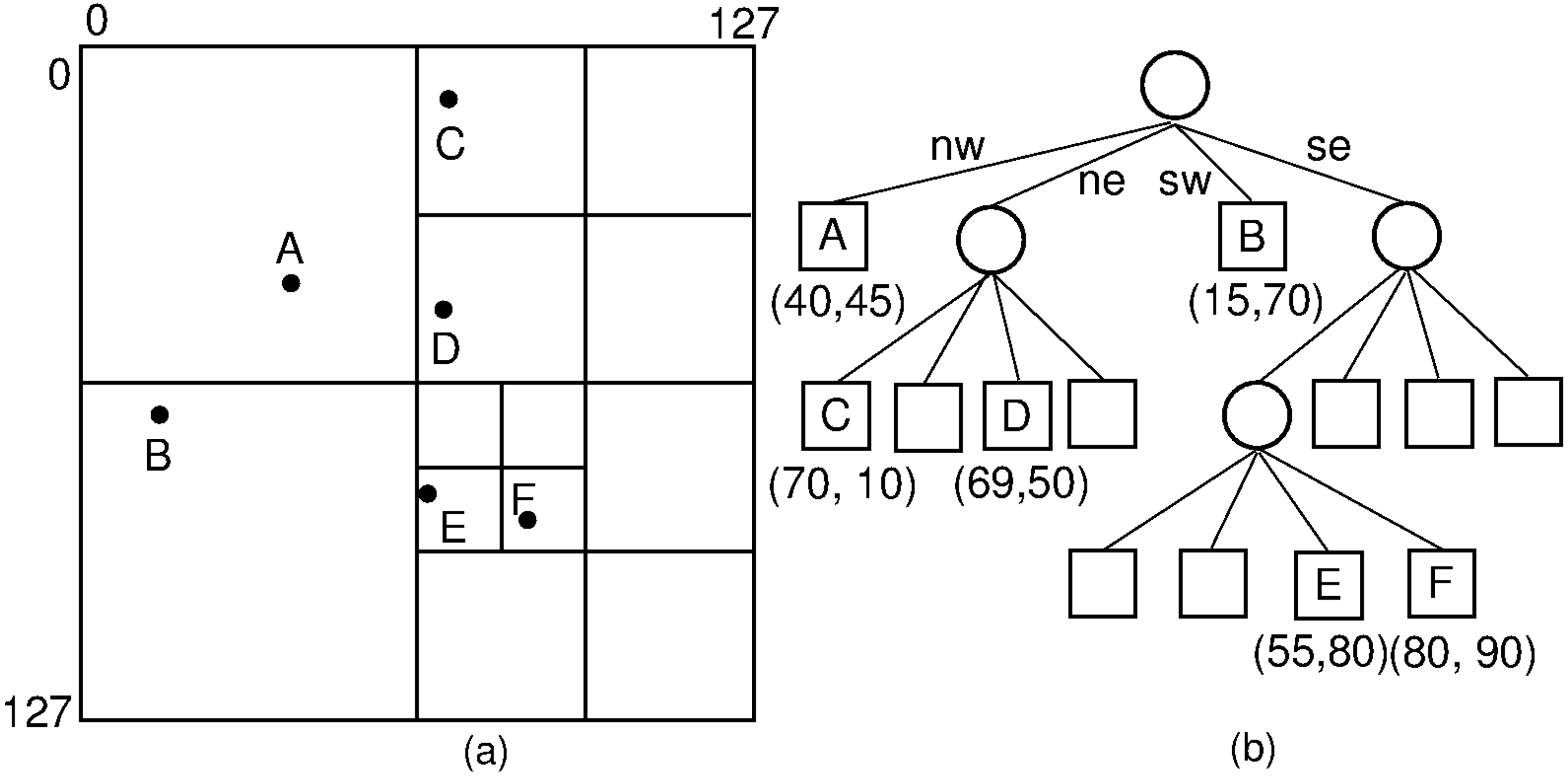
- Partition the multi-dimensional space into disjoint (non-overlapping) regions



(b) kd-tree

Space-Partitioning Trees (3)

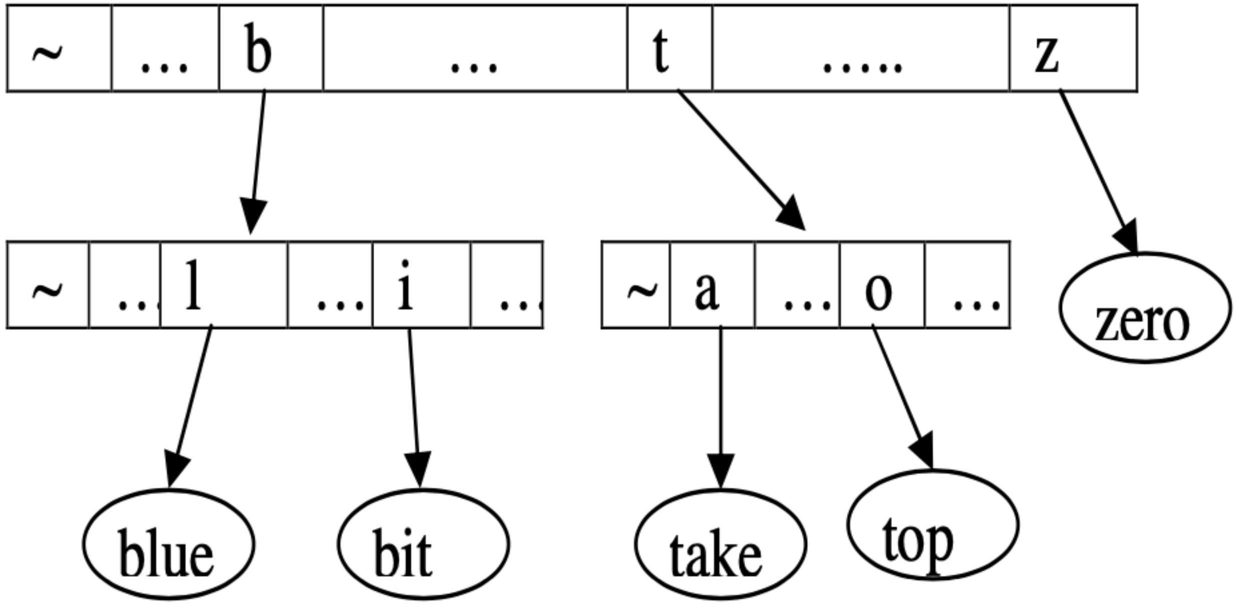
- Partition the multi-dimensional space into disjoint (non-overlapping) regions



(c) PR Quadtree

Space-Partitioning Trees (4)

- Partition the multi-dimensional space into disjoint (non-overlapping) regions



(d) Trie

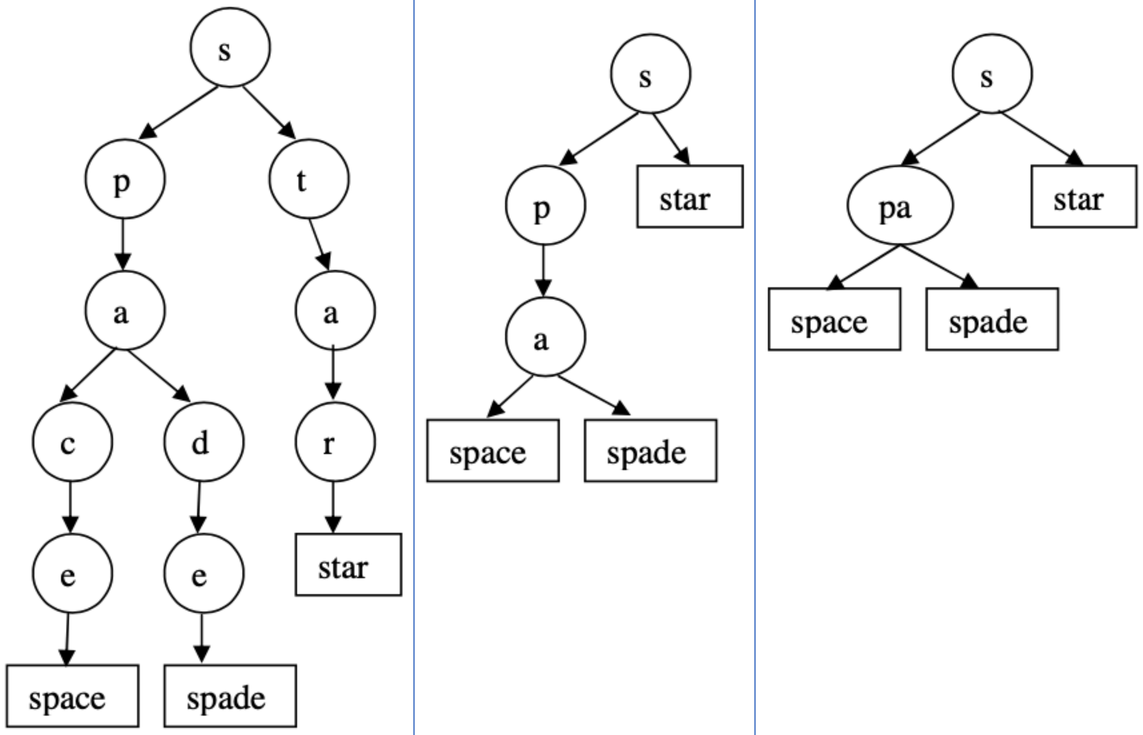
SP-GiST Parameters and Pluggable Modules

- Introduce SP-GiST Parameters and Pluggable Modules in the context of the trie data structure
- By setting these parameters differently we can realize various types of trie data structures

SP-GiST
Internal Methods

SP-GiST Parameters and Pluggable Modules (2)

- ***Path Shrink***
  - Vertical shrinking
  - Avoid lengthy and skinny paths from a root to a leaf
  - Paths of one child can be collapsed into one node
  - ***Leaf Shrink***: Shrinking single child nodes at the leaf level nodes → Patricia trie
  - ***Path Shrink***: allow for shrinking single child nodes at the non-leaf level nodes
  - ***No Tree Shrink***: No shrinking at all → The regular trie



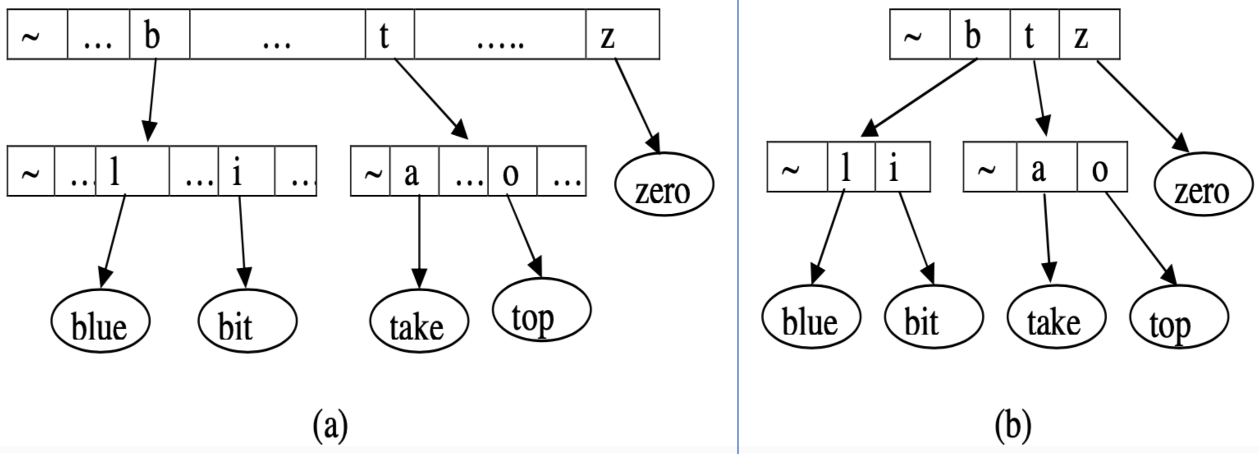
(a)
No Tree Shrink

(b)
Leaf Shrink:
Patricia Trie

(c)
Path Shrink

SP-GiST Parameters and Pluggable Modules (3)

- ***Node Shrink***
  - Horizontal shrinking
  - Avoid empty partitions – problem with space-driven partitioning
  - The question is: Do we allow that empty partitions be omitted?
  - ***Node Shrink***: Eliminate empty partitions → The forest trie
  - ***No Node Shrink***: The standard trie

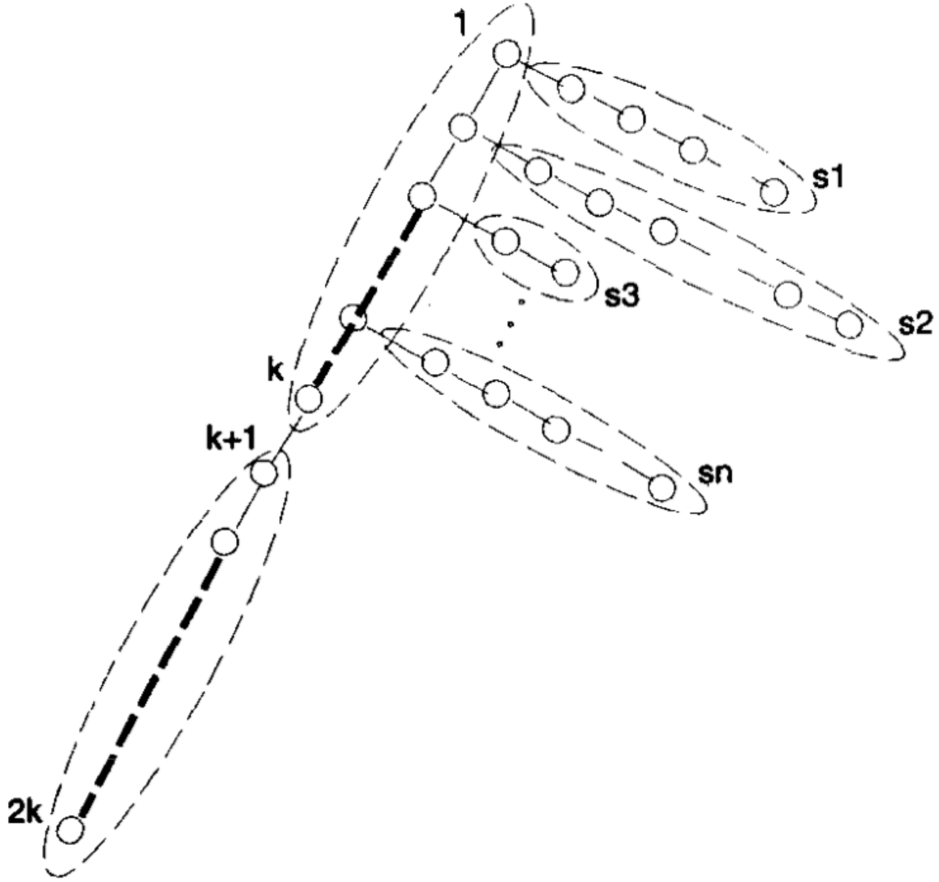


No Node Shrink

Node Shrink

SP-GiST Parameters and Pluggable Modules (4)

- ***Clustering***:
  - Address most serious issue facing disk-based space-partitioning trees
  - Problem: Tree nodes do not map directly to disk pages
    - Much smaller than disk pages.
  - Question: How do we pack tree nodes into disk pages?
    - Objective: Reduce disk I/Os for tree search and update
  - An optimal node-packing algorithm already exists that solves this issue [VLDB 96].



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

SP-GiST Parameters and Pluggable Modules (5)

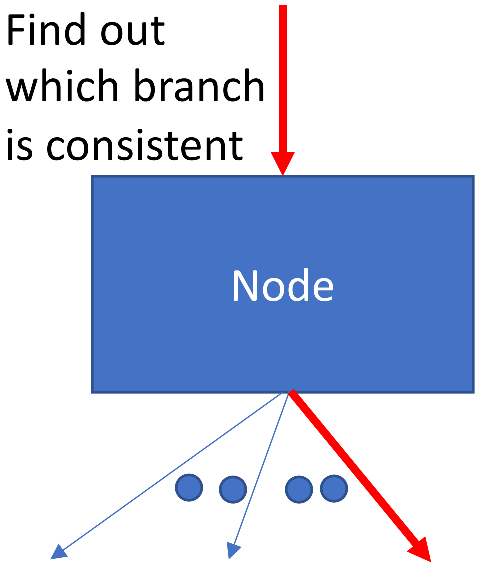
- ***NodePredicate***:
  - *S*pecifies the predicate to be used in the index nodes of the tree
    - A letter in a trie (e.g., "= a")
    - A quadrant in a quadtree
      - ($\geq$ x1 $\wedge$ $\leq$ x2 $\wedge$ $\geq$ y1 $\wedge$ $\leq$ y2)
- ***KeyType***:
  - Gives the type of the data in the leaf level of the tree
  - E.g.,
    - "Point" is the key type in an MX quadtree
    - "Word" is the key type in a trie
    - These data types have to be pre-defined by the user.

SP-GiST Parameters and Pluggable Modules (6)

- ***NumberofSpacePartitions***:
  - *S*pecifies the number of disjoint partitions produced at each decomposition
    - E.g.,
      - 4 for a quadtree
      - 2 for kd-tree
      - 8 for an octree
      - 27 for an English Dictionary trie
- ***Resolution***:
  - Limits the number of space decompositions and is set depending on the required granularity
- ***BucketSize***:
  - *S*pecifies the maximum number of data items a data node can hold

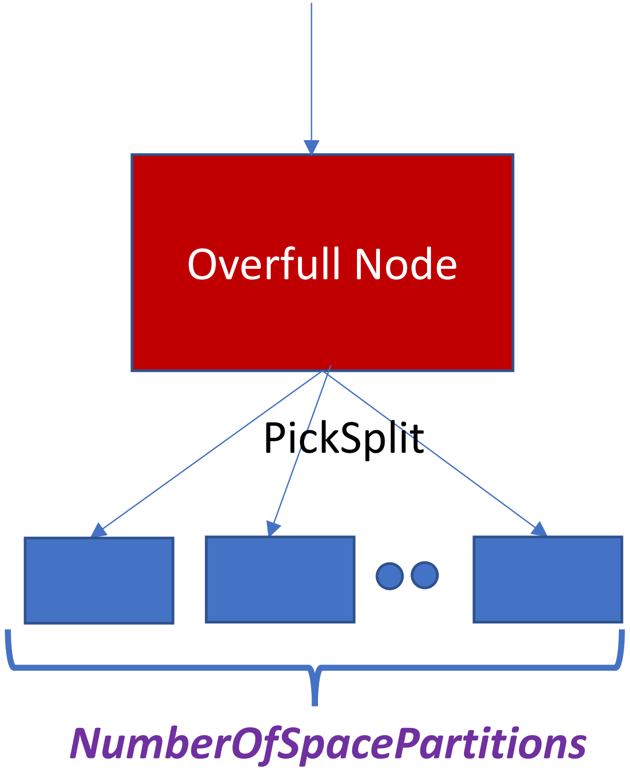
SP-GiST Parameters and Pluggable Modules (7)

- ***Consistent(Node Entry, Query Predicate, level)***:
- Find out which child branch in the node is consistent with the query predicate (e.g., range query or point query predicates)
  - Boolean function
    - Return: True, when ***Node Entry*** satisfies the ***Query Predicate***
False, otherwise
  - Used by the SP-GiST tree search method as a navigation guide through the tree
  - Test may differ based on what level in the tree we are at
    - E.g., in kd-tree may check x dimension vs. y dimension at a different level

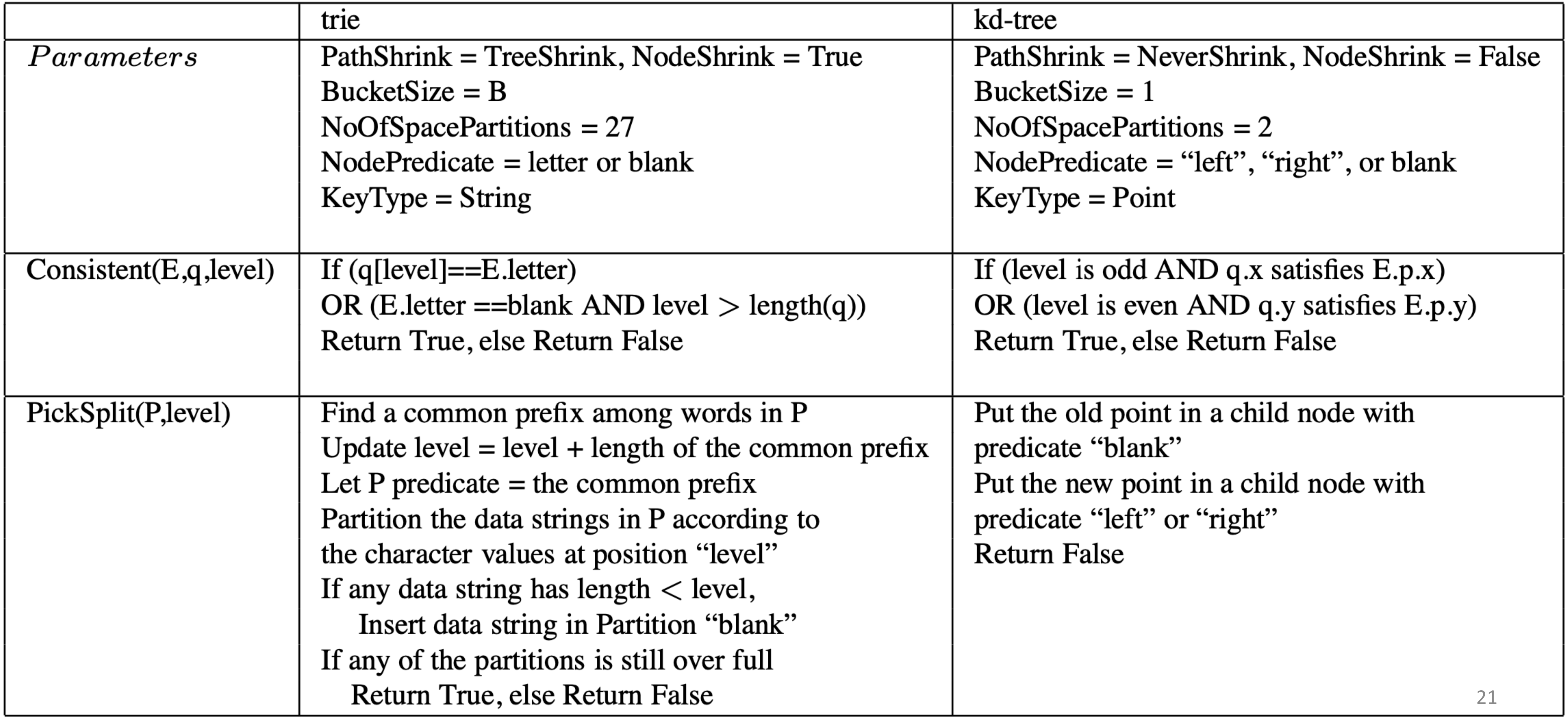


SP-GiST Parameters and Pluggable Modules (8)

- ***PickSplit(EntriesInNode, level, splitNodes, splitPredicates)***:
  - Specifies how the space is decomposed and how the data items are distributed over the new partitions
  - Invoked by internal method *Insert()* when a node-split is needed
  - ***EntriesInNode:***
    - *S*et of entries that cannot fit in a node ***PickSplit***
    - *BucketSize*+1 entries
  - Define tree's strategy on how to split the entries into a number of partitions (equal to ***NumberOfSpacePartitions****)*
  - Returns a Boolean that indicates if further partitioning should take place
  - ***Level:*** Used in the splitting criterion because splitting will depend on the current decomposition level of the tree
    - E.g., in a trie, at Level i, splitting will be according to the *i*th character of the word
  - ***splitNodes***: Result of pickSplit
    - Array of buckets; each contains elements to be inserted in each child node
  - ***splitPredicates***: Array of predicates; one for each child



Realization of a Trie and a kd-tree using SP-GiST



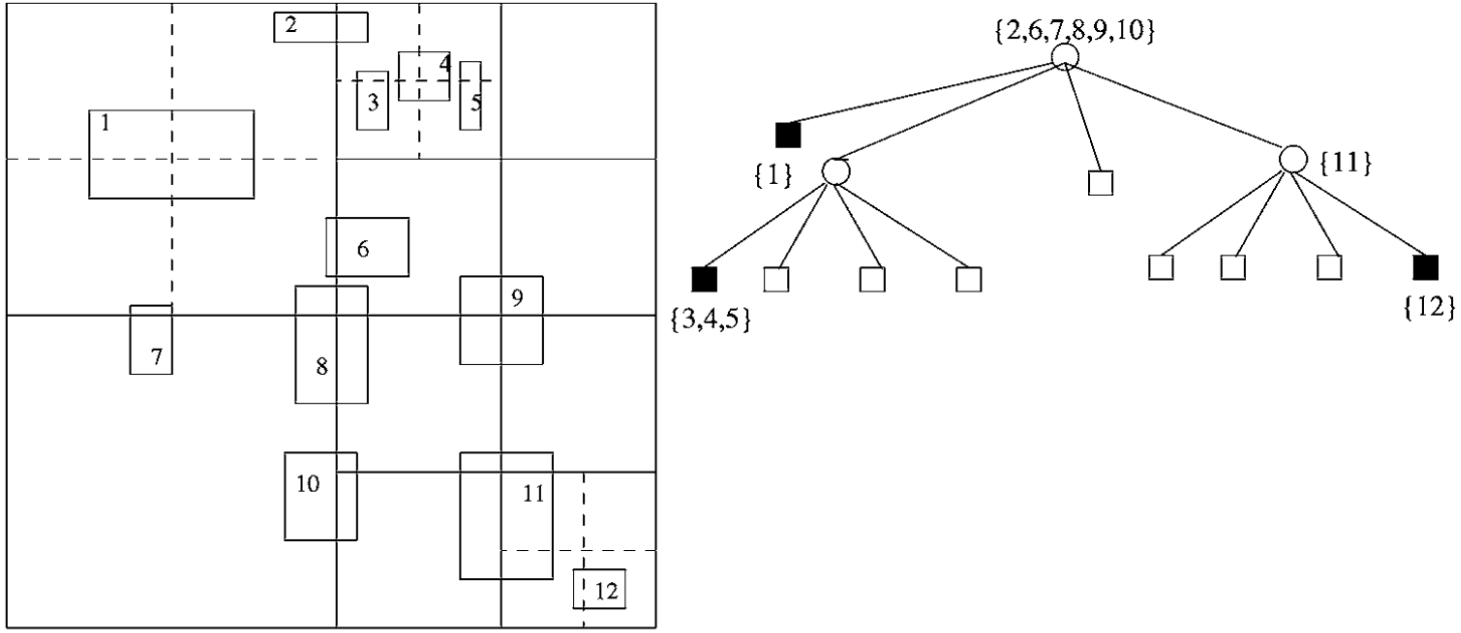
	trie	kd-tree

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

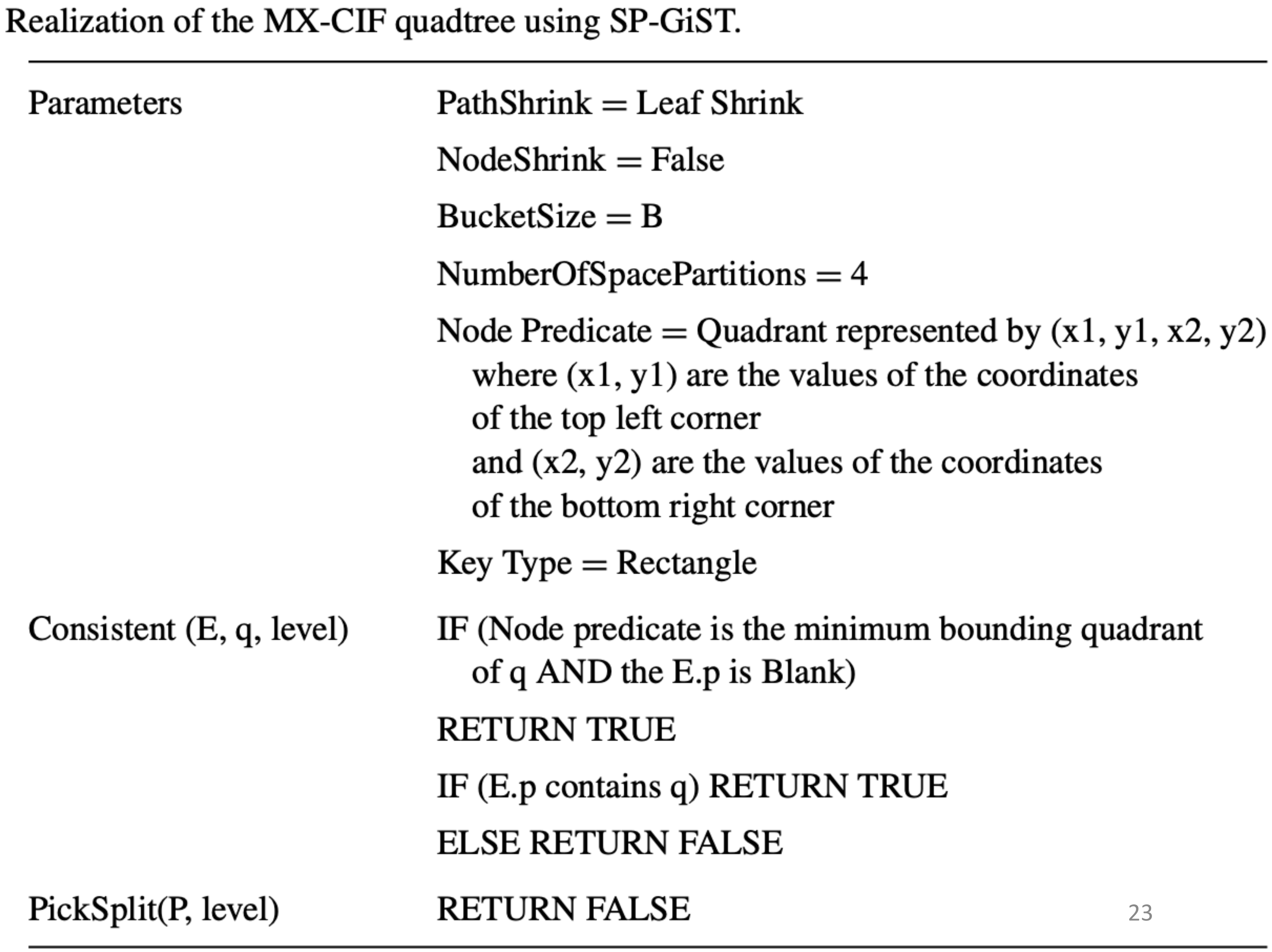
Realization of the MX-CIF Quadtree using SP-GiST

- Quadtree variation for storing rectangles
- Associate each rectangle *R* with the quadtree node corresponding to the smallest quadtree block that contains R
- Rectangles can be associated with both leaf and non-leaf nodes
- Subdivision stops when a node’s block contains no rectangles
- More than one rectangle can be associated with a given node



Realization of the MX-CIF Quadtree using SP-GiST

- PickSplit is not applicable
  - In the MX-CIF insertion algorithm, there is not much choice as to where a rectangle gets inserted
  - No choices to be made for PickSplit



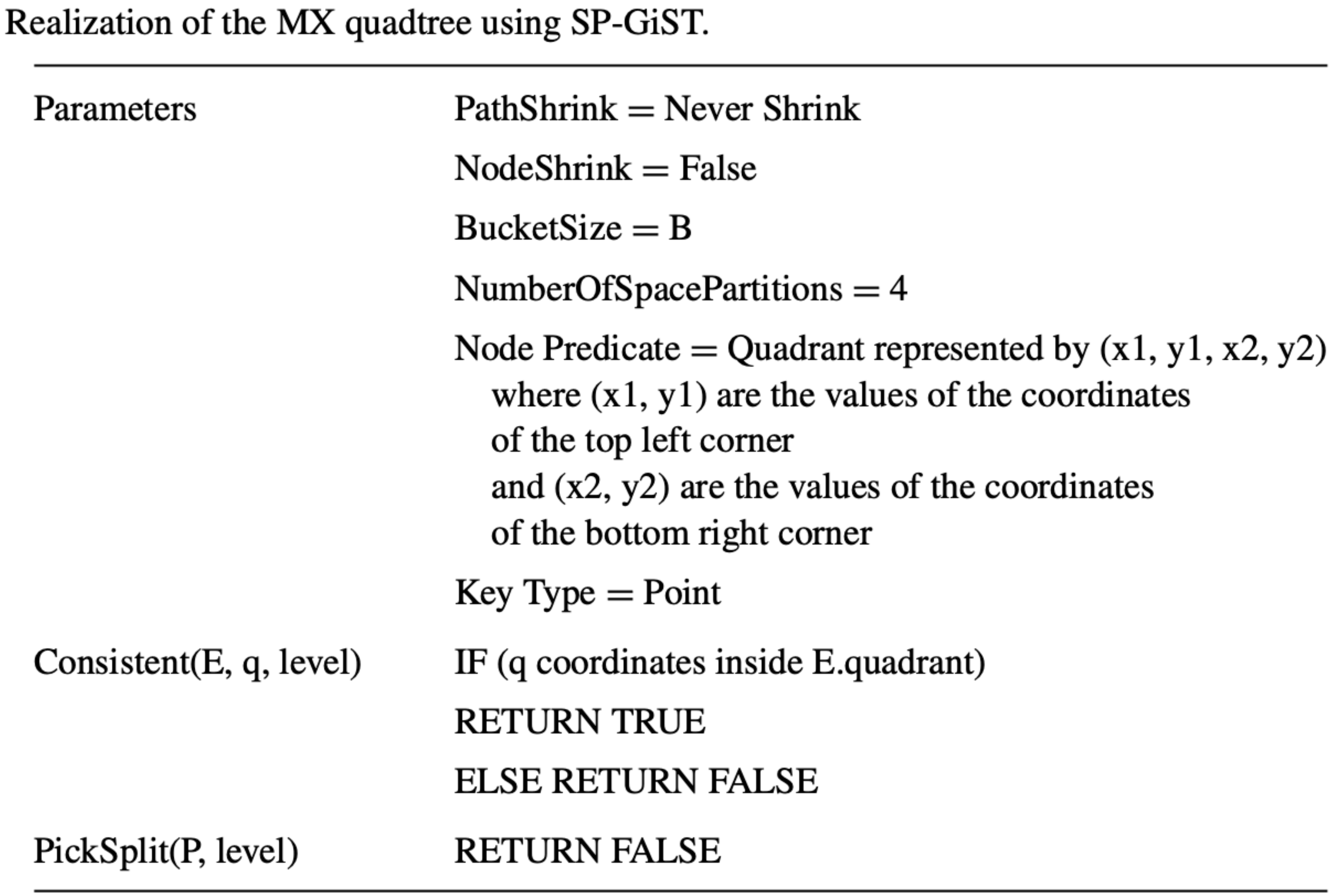
Realization of the MX-CIF quadtree using SP-GiST.

Parameters	PathShrink = Leaf Shrink
	NodeShrink = False
	BucketSize = B
	NumberOfSpacePartitions = 4
	Node Predicate = Quadrant represented by (x1, y1, x2, y2) where (x1, y1) are the values of the coordinates of the top left corner and (x2, y2) are the values of the coordinates of the bottom right corner
	Key Type = Rectangle

Consistent (E, q, level)	IF (Node predicate is the minimum bounding quadrant of q AND the E.p is Blank)
	RETURN TRUE
	IF (E.p contains q) RETURN TRUE
	ELSE RETURN FALSE

PickSplit(P, level)	RETURN FALSE

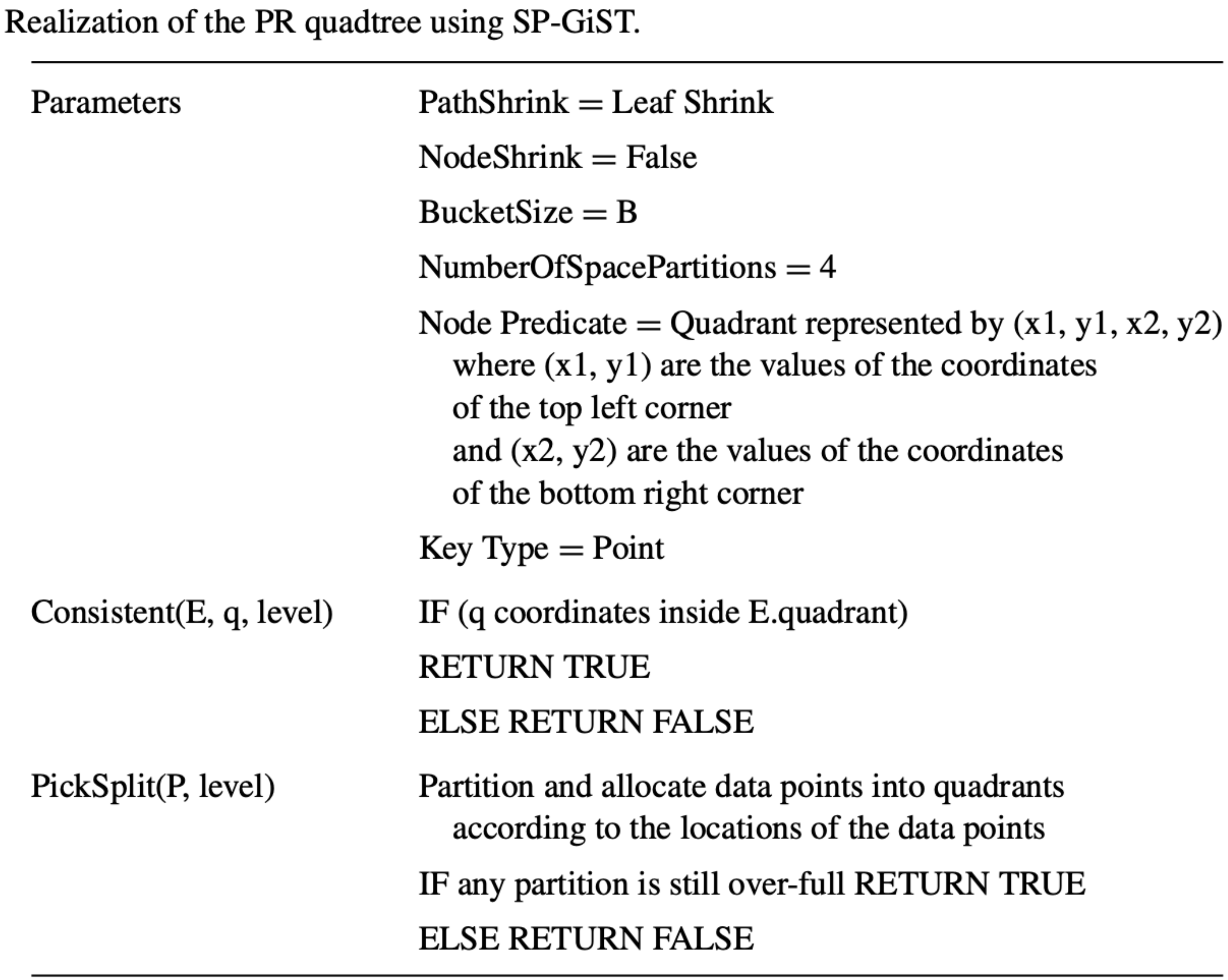
Realization of the MX Quadtree using SP-GiST



Realization of the MX quadtree using SP-GiST.

Parameters	PathShrink = Never Shrink
	NodeShrink = False
	BucketSize = B
	NumberOfSpacePartitions = 4
	Node Predicate = Quadrant represented by (x1, y1, x2, y2) where (x1, y1) are the values of the coordinates of the top left corner and (x2, y2) are the values of the coordinates of the bottom right corner
	Key Type = Point
Consistent(E, q, level)	IF (q coordinates inside E.quadrant)
	RETURN TRUE
	ELSE RETURN FALSE
PickSplit(P, level)	RETURN FALSE

Realization of the PR Quadtree using SP-GiST



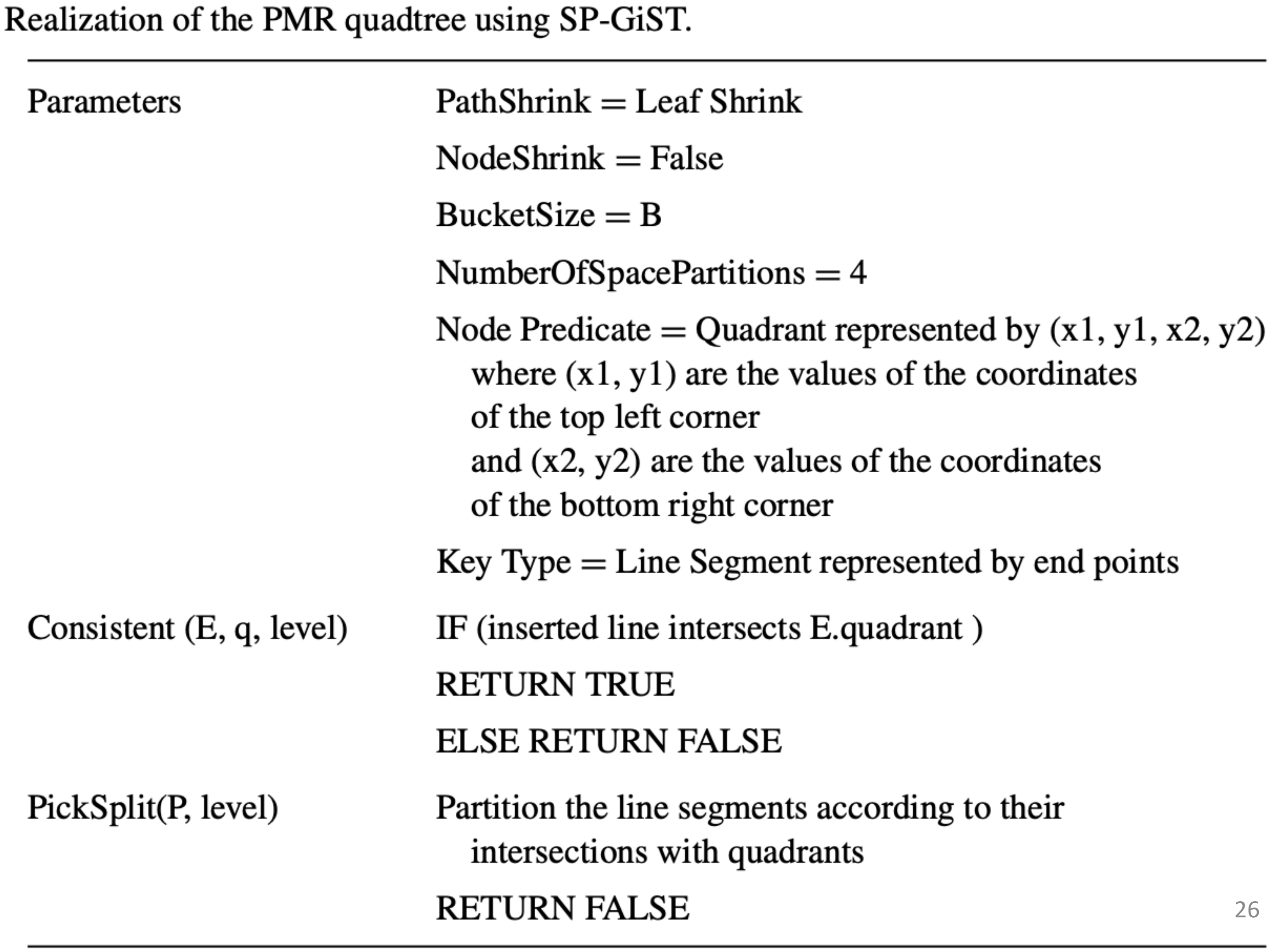
Realization of the PR quadtree using SP-GiST.

Parameters	PathShrink = Leaf Shrink
	NodeShrink = False
	BucketSize = B
	NumberOfSpacePartitions = 4
	Node Predicate = Quadrant represented by (x1, y1, x2, y2) where (x1, y1) are the values of the coordinates of the top left corner and (x2, y2) are the values of the coordinates of the bottom right corner
	Key Type = Point

Consistent(E, q, level)	IF (q coordinates inside E.quadrant)
	RETURN TRUE
	ELSE RETURN FALSE

PickSplit(P, level)	Partition and allocate data points into quadrants according to the locations of the data points IF any partition is still over-full RETURN TRUE ELSE RETURN FALSE

Realization of the PMR Quadtree using SP-GiST



Realization of the PMR quadtree using SP-GiST.

Parameters	PathShrink = Leaf Shrink
	NodeShrink = False
	BucketSize = B
	NumberOfSpacePartitions = 4
	Node Predicate = Quadrant represented by (x1, y1, x2, y2) where (x1, y1) are the values of the coordinates of the top left corner and (x2, y2) are the values of the coordinates of the bottom right corner
	Key Type = Line Segment represented by end points

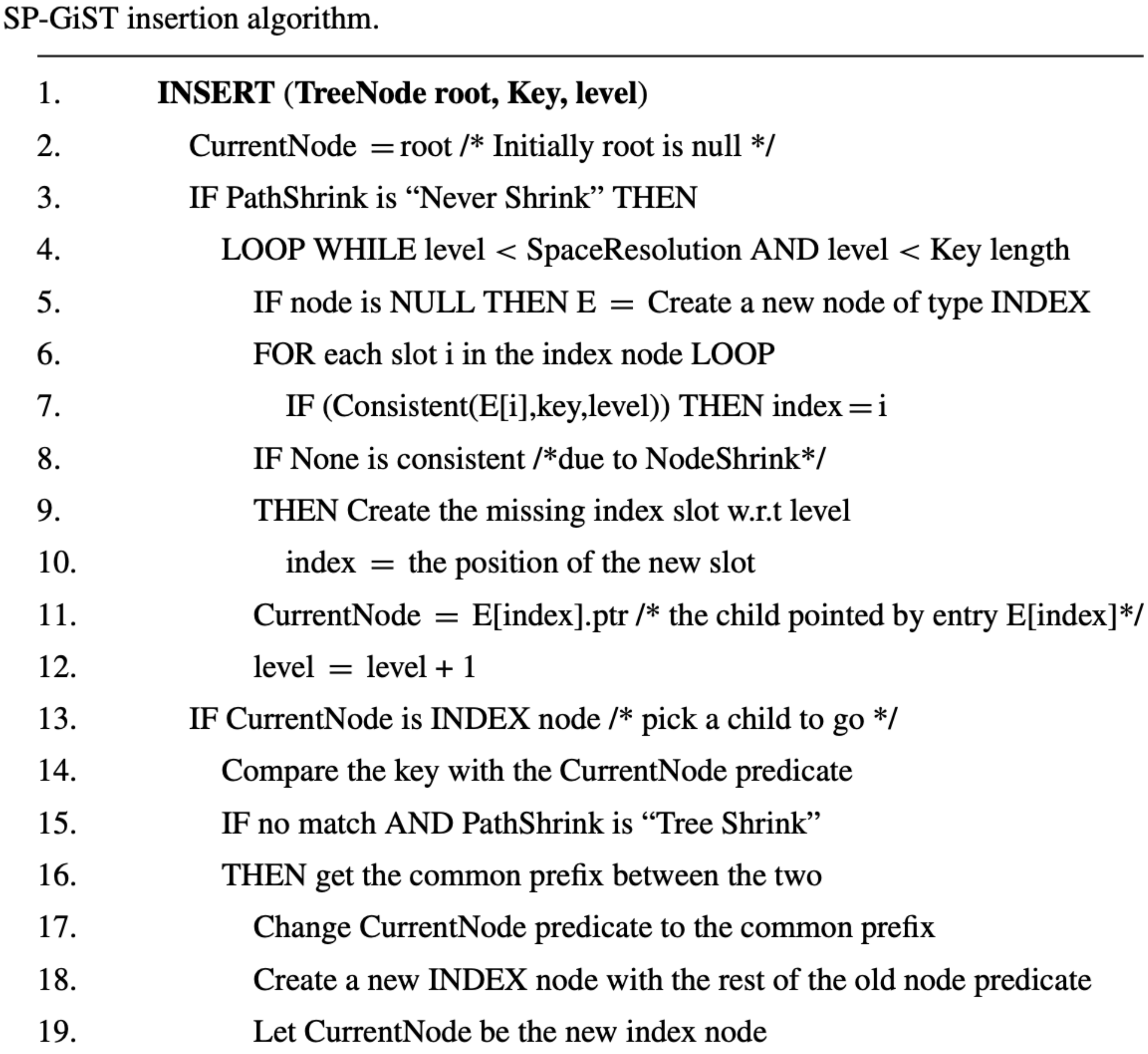
Consistent (E, q, level)	IF (inserted line intersects E.quadrant)
	RETURN TRUE
	ELSE RETURN FALSE

PickSplit(P, level)	Partition the line segments according to their intersections with quadrants RETURN FALSE

SP-GiST Internal Methods

- SP-GiST provides a set of *internal* methods that are common for all space-partitioning trees
  - The *Insert()*, *Search()*, and *Delete()* methods
  - The core of SP-GiST and are the same for all SP-GiST-based indexes

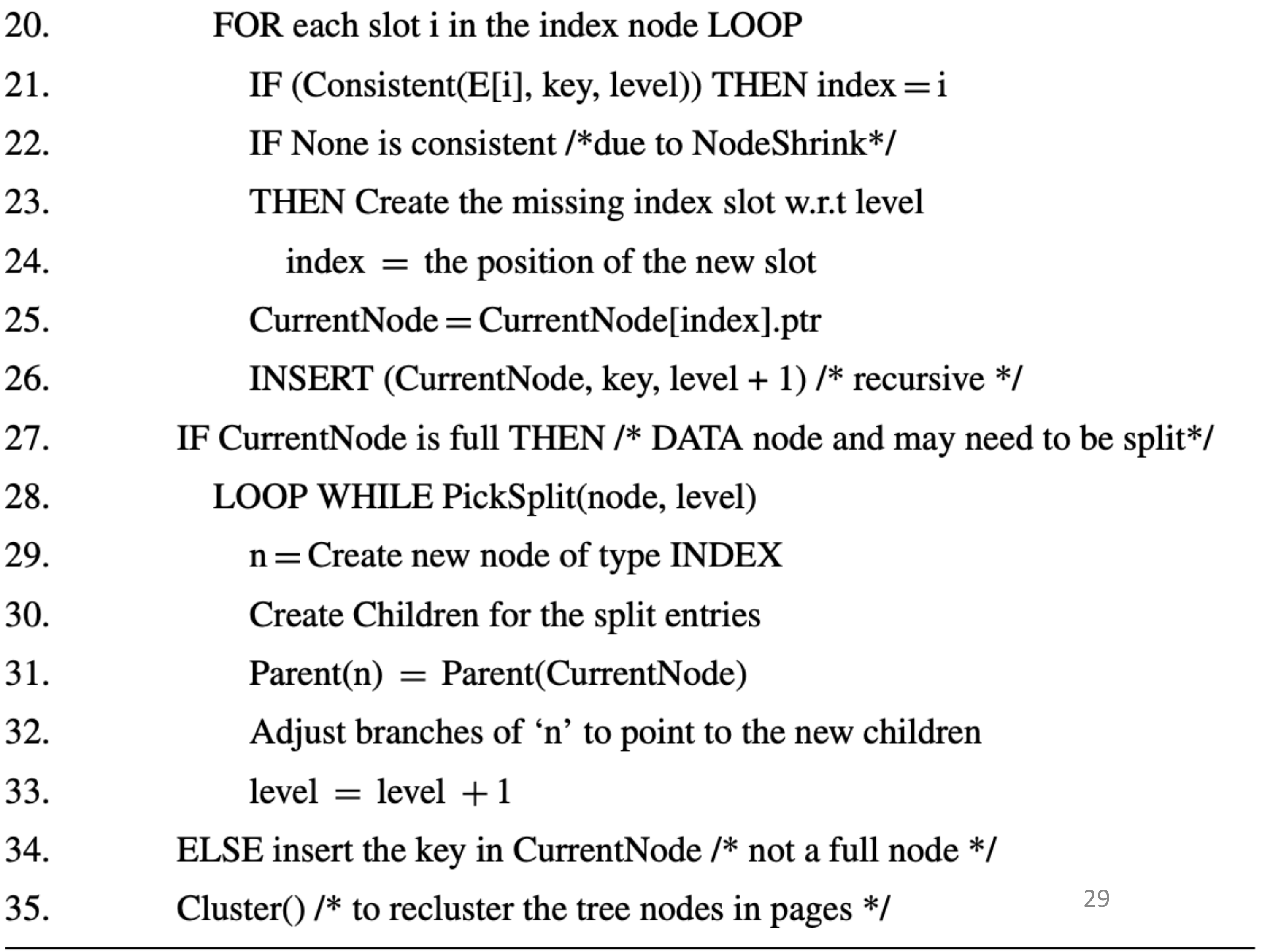
SP-GiST Insert



SP-GiST insertion algorithm.

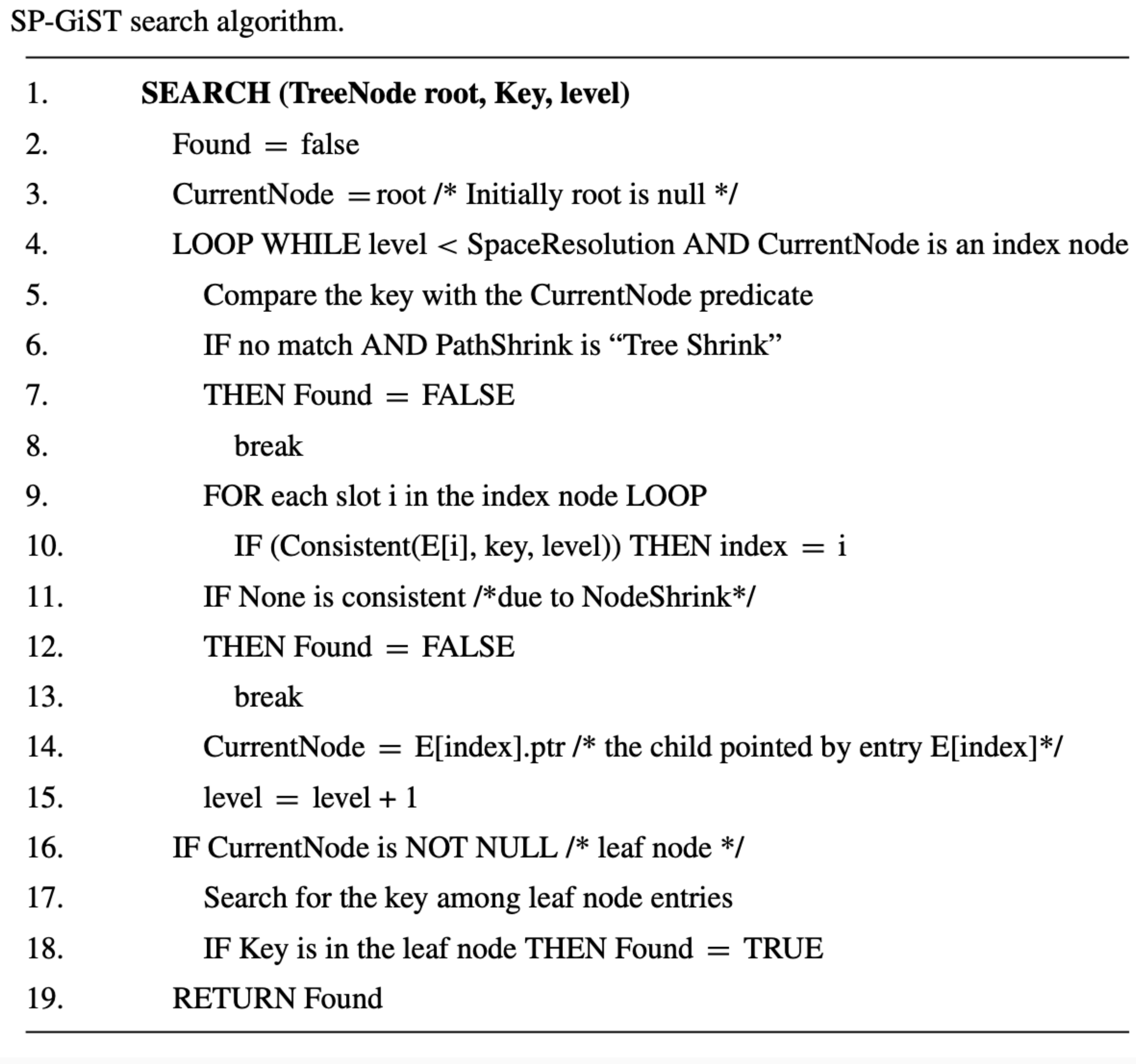
```
INSERT (TreeNode root, Key, level)
  CurrentNode = root /* Initially root is null */
  IF PathShrink is “Never Shrink” THEN
    LOOP WHILE level < SpaceResolution AND level < Key length
      IF node is NULL THEN E = Create a new node of type INDEX
      FOR each slot i in the index node LOOP
        IF (Consistent(E[i],key,level)) THEN index = i
      IF None is consistent /*due to NodeShrink*/
      THEN Create the missing index slot w.r.t level
        index = the position of the new slot
      CurrentNode = E[index].ptr /* the child pointed by entry E[index]*/
      level = level + 1
  IF CurrentNode is INDEX node /* pick a child to go */
    Compare the key with the CurrentNode predicate
    IF no match AND PathShrink is “Tree Shrink”
    THEN get the common prefix between the two
      Change CurrentNode predicate to the common prefix
      Create a new INDEX node with the rest of the old node predicate
      Let CurrentNode be the new index node
```

SP-GiST Insert (Cont’d)



```
    FOR each slot i in the index node LOOP
        IF (Consistent(E[i], key, level)) THEN index = i
        IF None is consistent /*due to NodeShrink*/
        THEN Create the missing index slot w.r.t level
            index = the position of the new slot
        CurrentNode = CurrentNode[index].ptr
        INSERT (CurrentNode, key, level + 1) /* recursive */
IF CurrentNode is full THEN /* DATA node and may need to be split*/
    LOOP WHILE PickSplit(node, level)
        n = Create new node of type INDEX
        Create Children for the split entries
        Parent(n) = Parent(CurrentNode)
        Adjust branches of ‘n’ to point to the new children
        level = level + 1
ELSE insert the key in CurrentNode /* not a full node */
Cluster() /* to recluster the tree nodes in pages */
```

SP-GiST Search

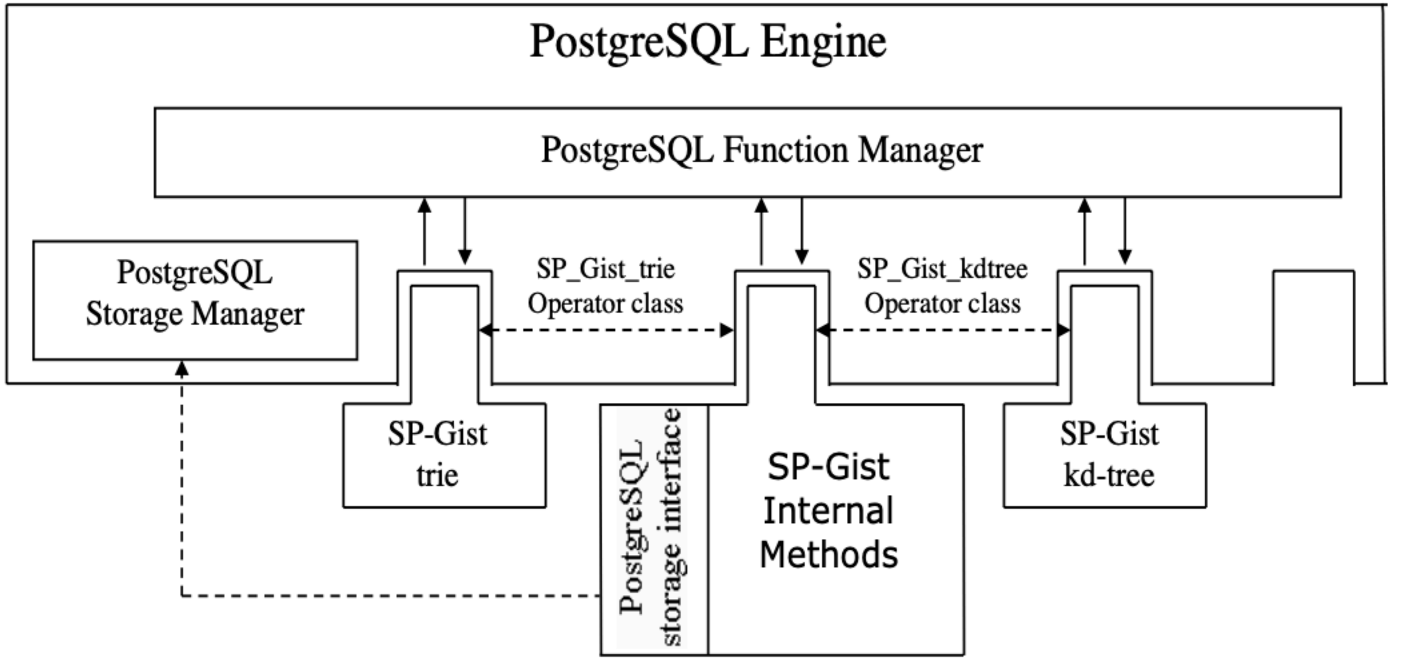


SP-GiST search algorithm.

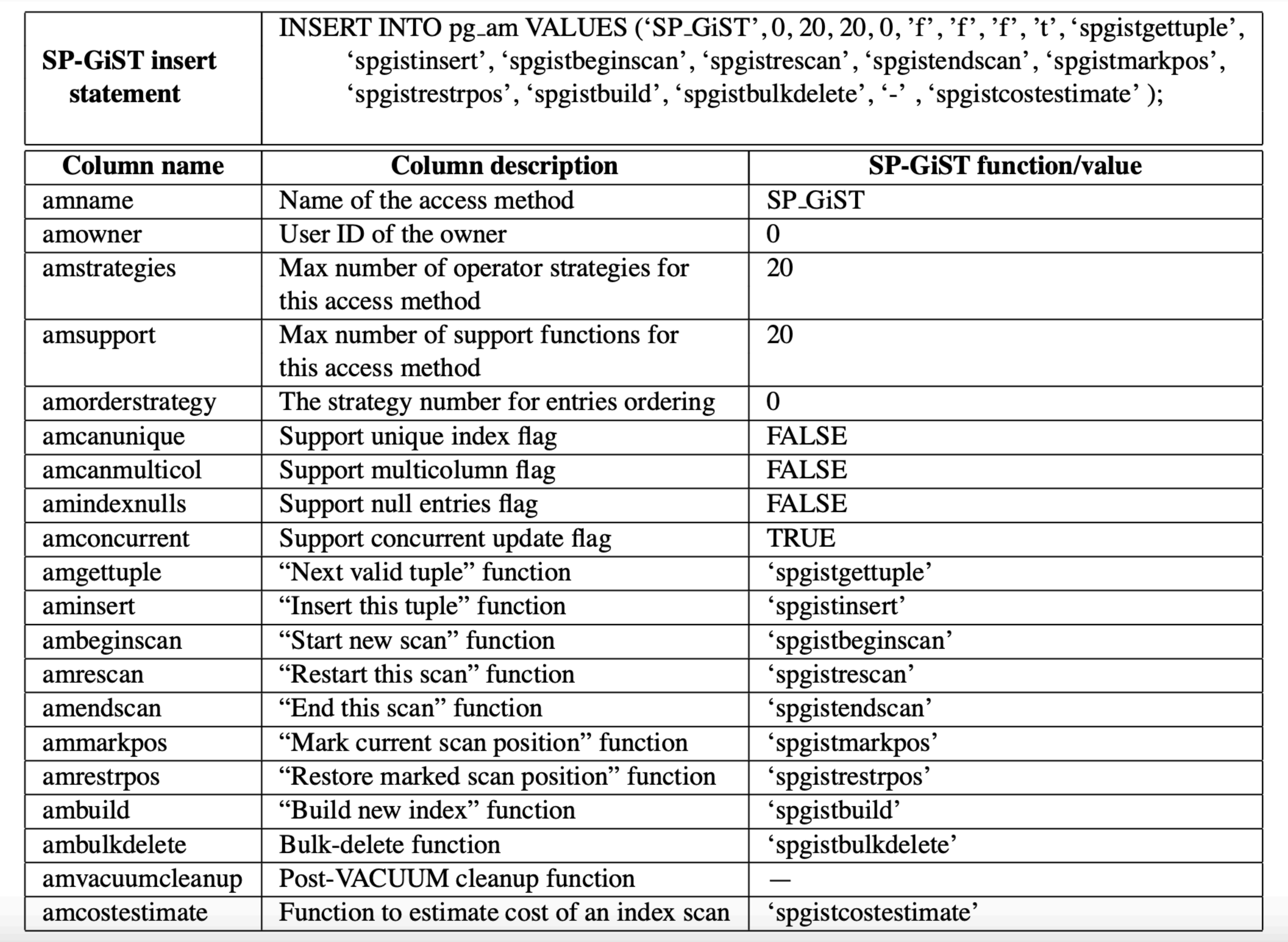
```
SEARCH (TreeNode root, Key, level)
  Found = false
  CurrentNode = root /* Initially root is null */
  LOOP WHILE level < SpaceResolution AND CurrentNode is an index node
    Compare the key with the CurrentNode predicate
    IF no match AND PathShrink is "Tree Shrink"
    THEN Found = FALSE
      break
    FOR each slot i in the index node LOOP
      IF (Consistent(E[i], key, level)) THEN index = i
    IF None is consistent /*due to NodeShrink*/
    THEN Found = FALSE
      break
    CurrentNode = E[index].ptr /* the child pointed by entry E[index]*/
    level = level + 1
  IF CurrentNode is NOT NULL /* leaf node */
    Search for the key among leaf node entries
    IF Key is in the leaf node THEN Found = TRUE
  RETURN Found
```

Realizing SP-GiST Inside PostgreSQL

- PostgreSQL:
  - An ***open-source*** object-relational database management system
  - 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.


PostgreSQL Engine
PostgreSQL Function Manager
PostgreSQL Storage Manager
SP_Gist_trie Operator class
SP_Gist_kdtree Operator class
SP-Gist trie
PostgreSQL storage interface
SP-Gist Internal Methods
SP-Gist kd-tree

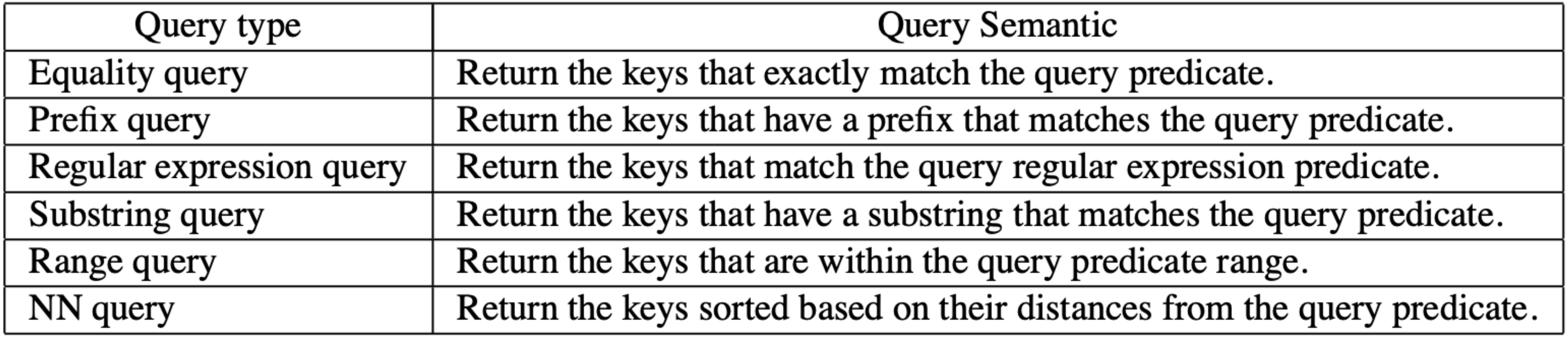
Operator classes specify the data type and the operators on which a certain access method can work



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’

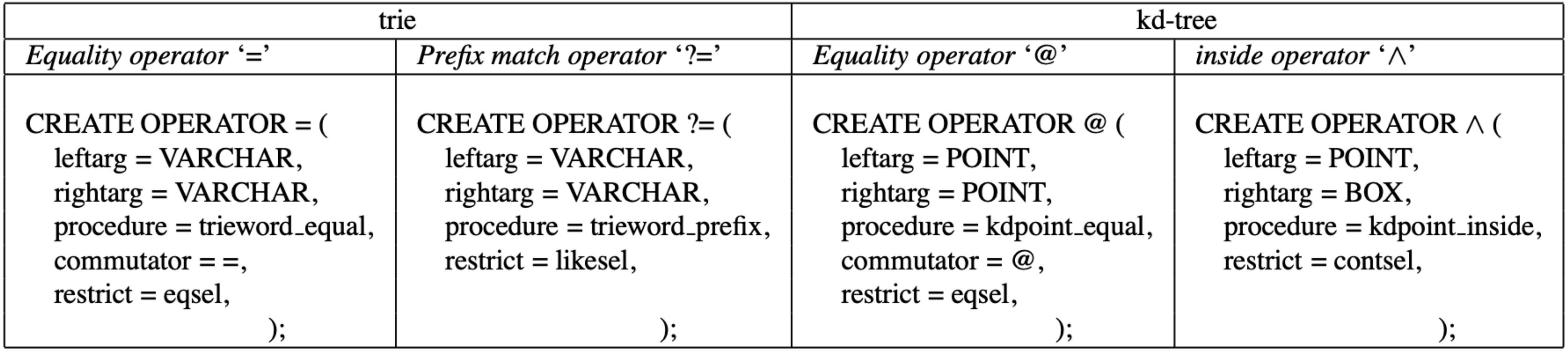
Supported Query Types in SP-GiST



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.

- We only allow for the wildcard, ‘?’, that matches any single character

Operator Definitions for the Trie and the kd-tree in SP-GiST



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,
			);

Operator Class for Trie and kd-tree using SP-GiST

- Trie Operators '=', '#=', and '?=' to support the equality queries, the prefix queries, and the regular expression queries

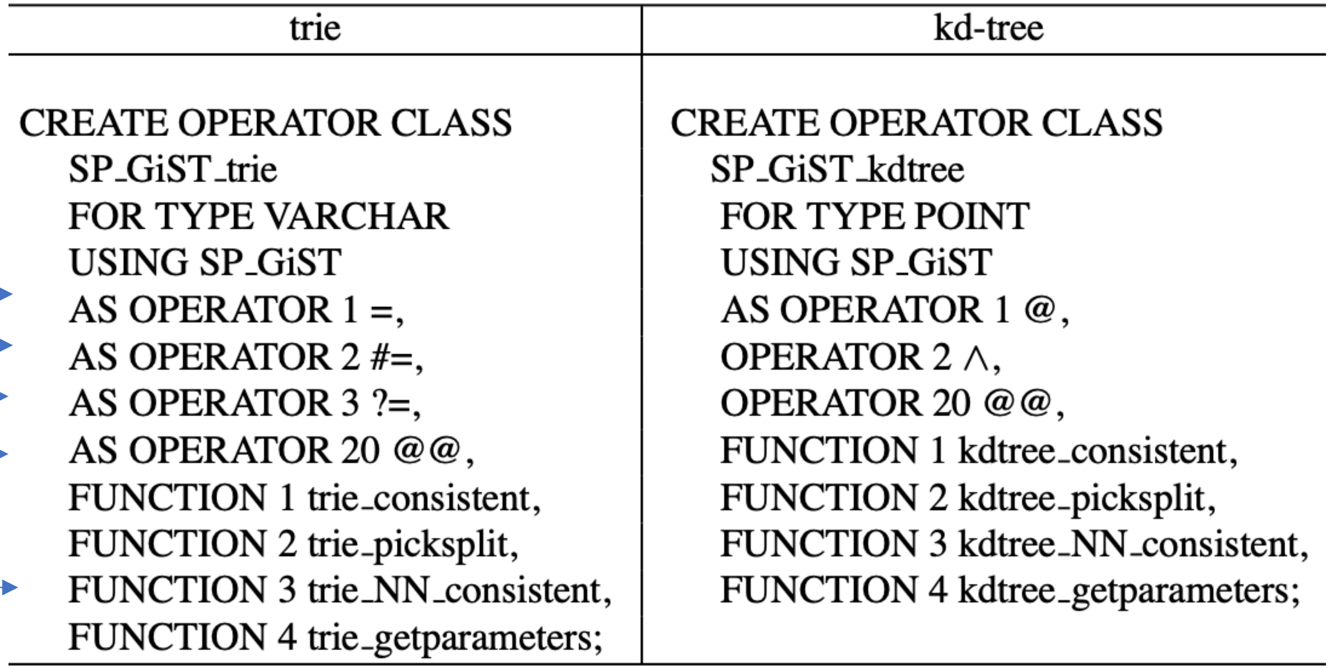
Equality operator

Prefix operator

RE operator

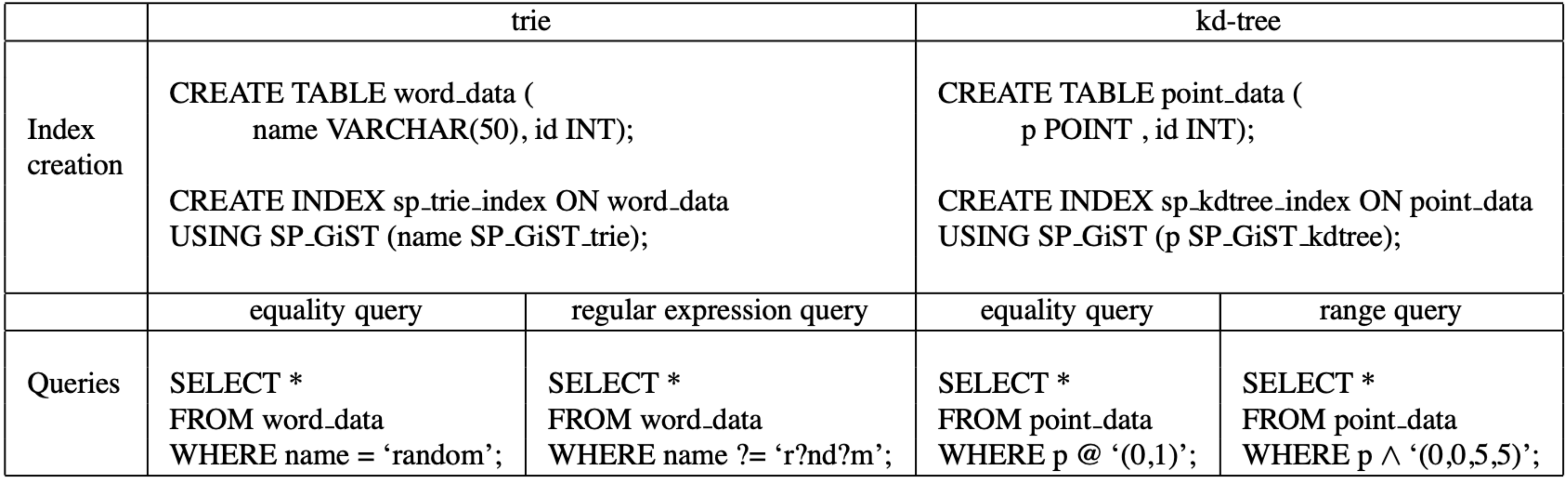
NN search operator

NN Distance function



trie	kd-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;

Trie and kd-tree Index Creation and Querying



	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)’;

Generic Nearest-Neighbor Algorithm for SP-GiST

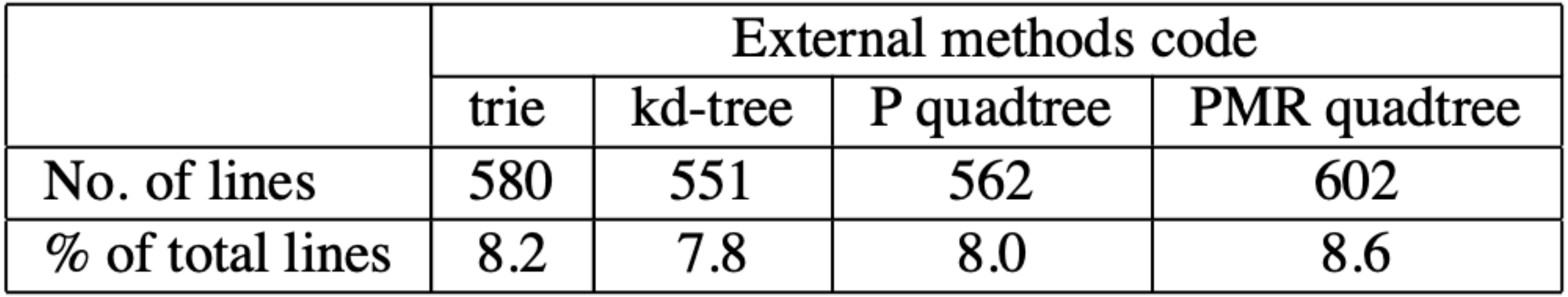
```
Insert the root node into the priority queue with minimum distance
While (priority queue is not empty)
{
        Retrieve the top of the queue into P
        If (P is an object) Then
                Report P as the next NN to the query object
        Else
                Compute the minimum distances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

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

Experiments

- Demonstrate the extensibility of SP-GiST to rapidly prototype new indexes and
- Highlight strengths and weaknesses of SP-GiST indexes over B+-tree and R-tree indexes
- Realized in PostgreSQL using SP-GiST the following disk-based indexes:
  - Disk-based Patricia trie
  - Disk-based kd-tree
  - Disk-based point quadtree
  - Disk-based PMR quadtree
  - Disk-based suffix tree

Number and Percentage of External Methods' Code Lines

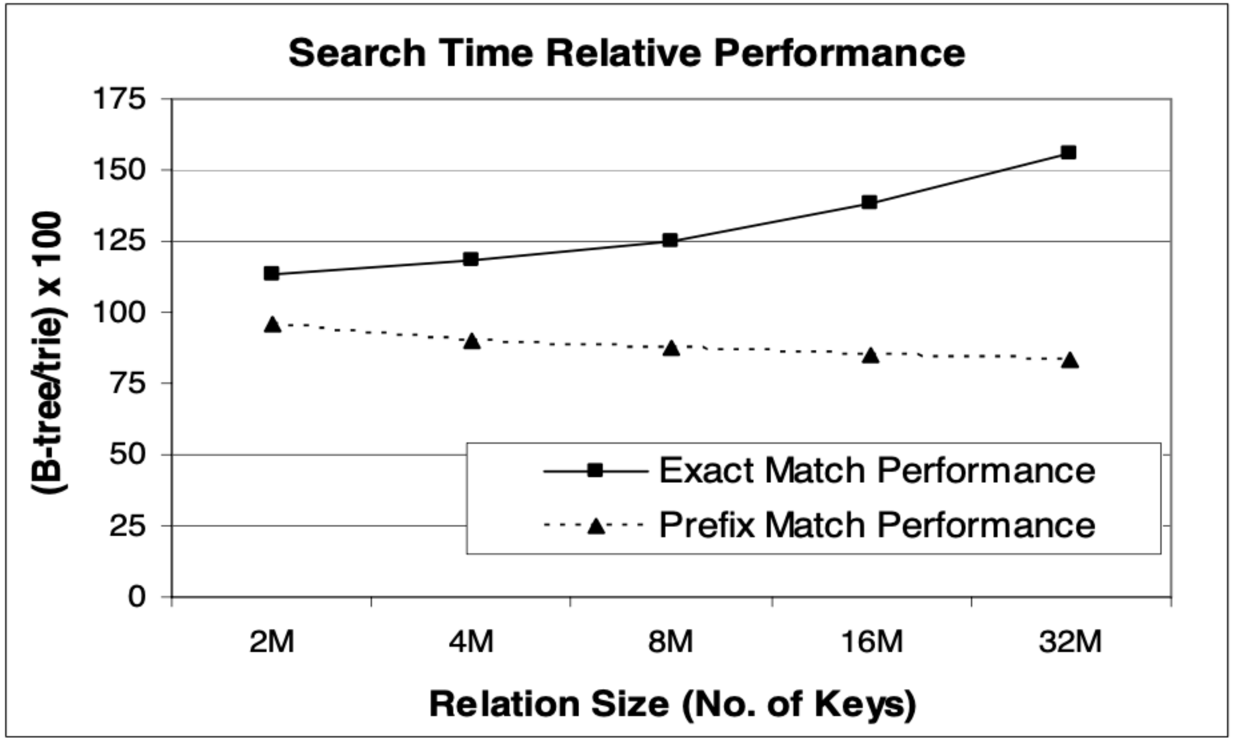


	External methods code			
	trie	kd-tree	P quadtree	PMR quadtree
No. of lines	580	551	562	602
% of total lines	8.2	7.8	8.0	8.6

- Developer provides < 10% of total index coding
- The other 90% of the code is provided as the SP-GiST core

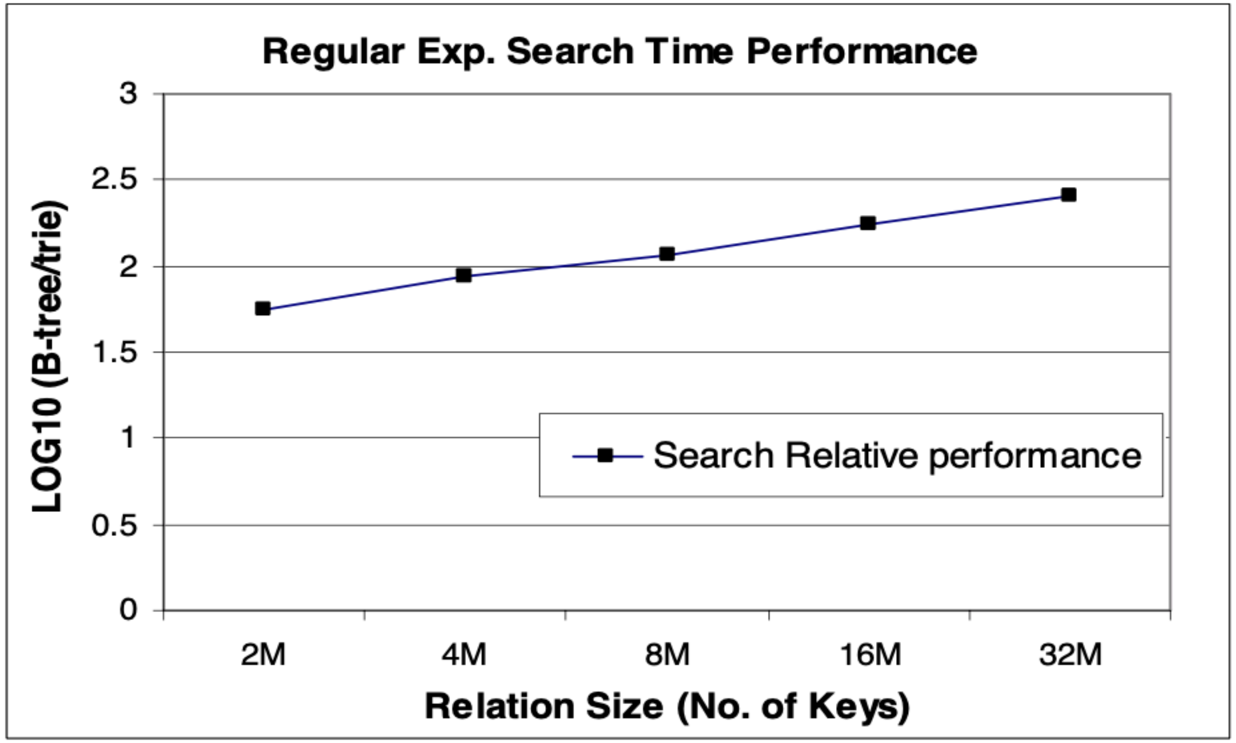
SP-GiST Patricia Trie vs. B+-tree for Text String Data

- ***Exact match search***:
  - Patricia trie has > 150% search time improvement over the B+-tree.
  - Patricia trie scales better with the increase in data size
- ***Prefix match search***:
  - B+-tree has better performance
  - Keys sorted in the B+-tree leaf nodes
  - Better packing into disk pages in contrast to the trie



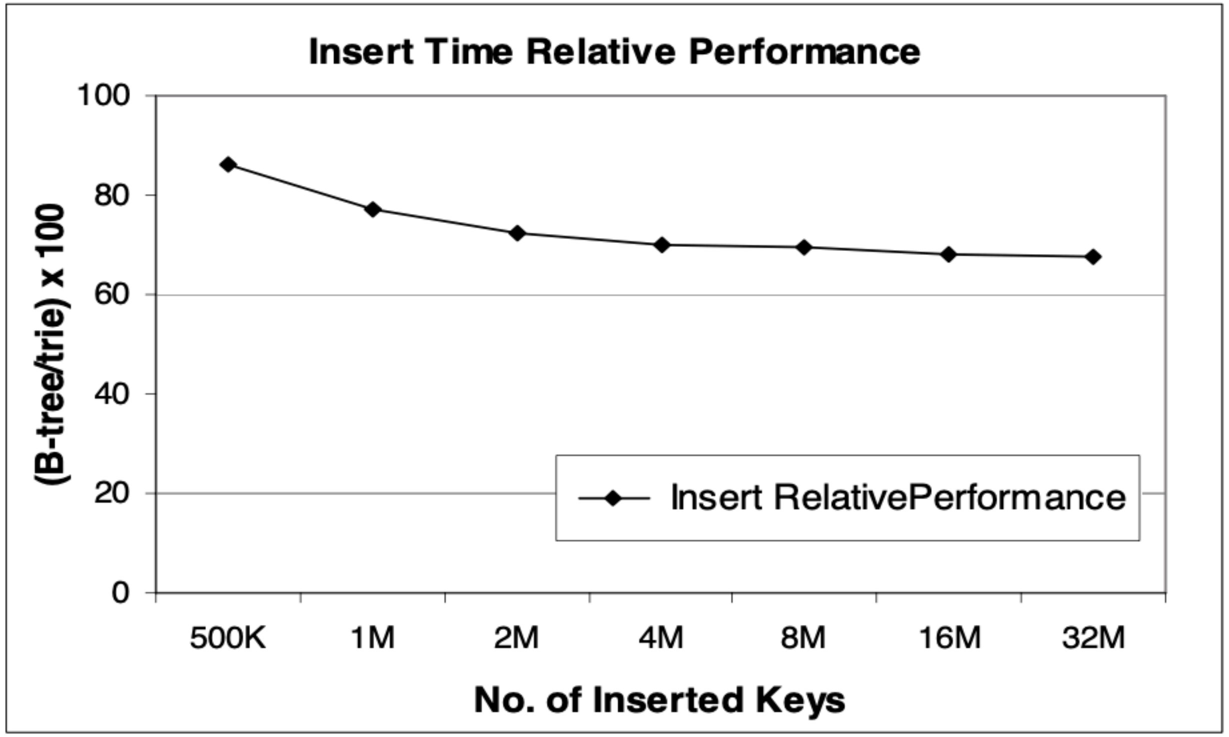
SP-GiST Patricia Trie vs. B+-tree for Text String Data

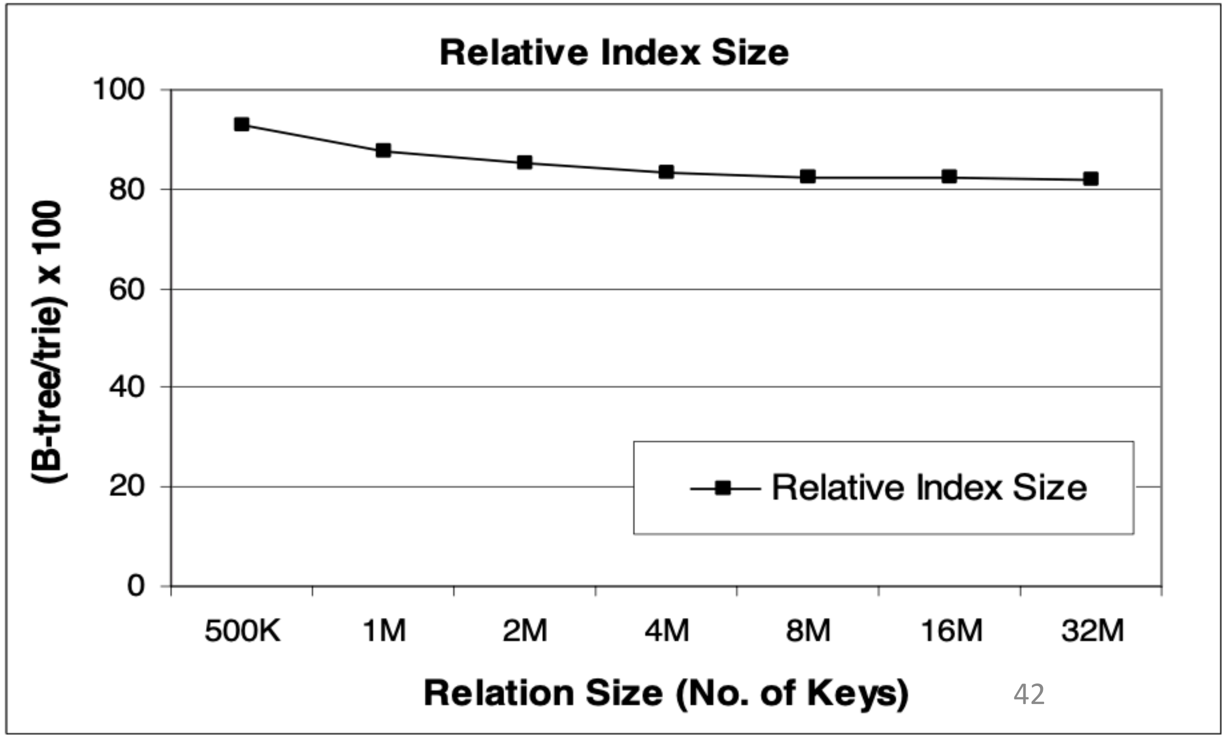
- ***Regular expression search***: We only allow for the wildcard, ‘?’, that matches any single character
- B+-tree performance is very sensitive to the positions of the wildcard; ‘?’ in the search string
  - E.g., ?at?y
- Trie > 2 orders of magnitude search time improvement over the B+-tree



SP-GiST Patricia Trie vs. B+-tree Insertion Time and Index Size

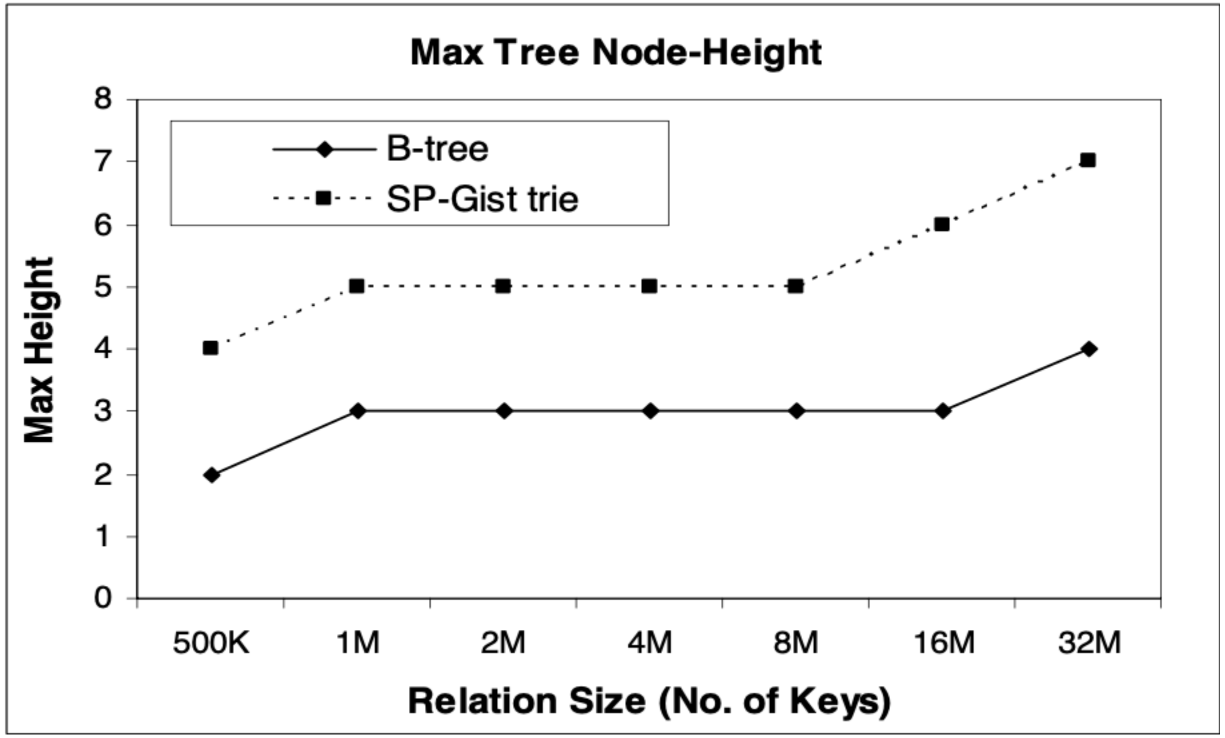
- B+-tree scales better
- Trie involves a higher number of nodes and a higher number of node splits than the B+-tree because the trie node size is much smaller than the B+-tree node size

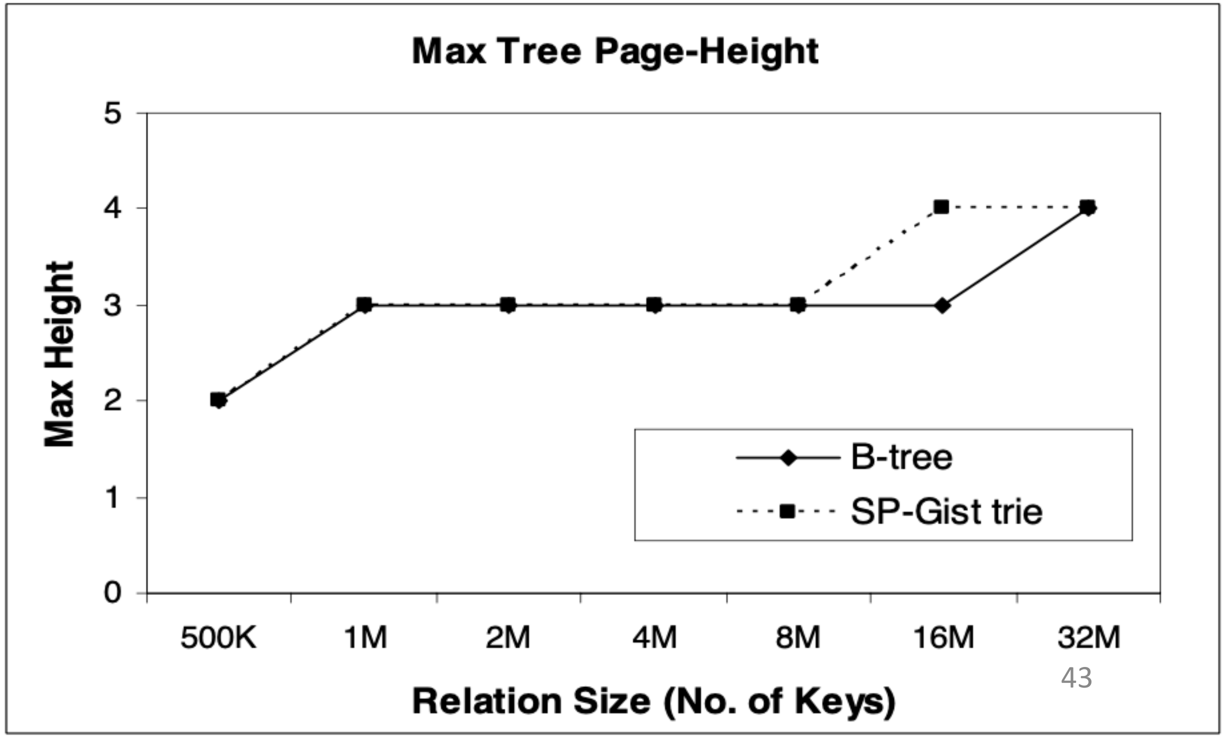




SP-GiST Patricia Trie vs. B+-tree Node and Page Heights

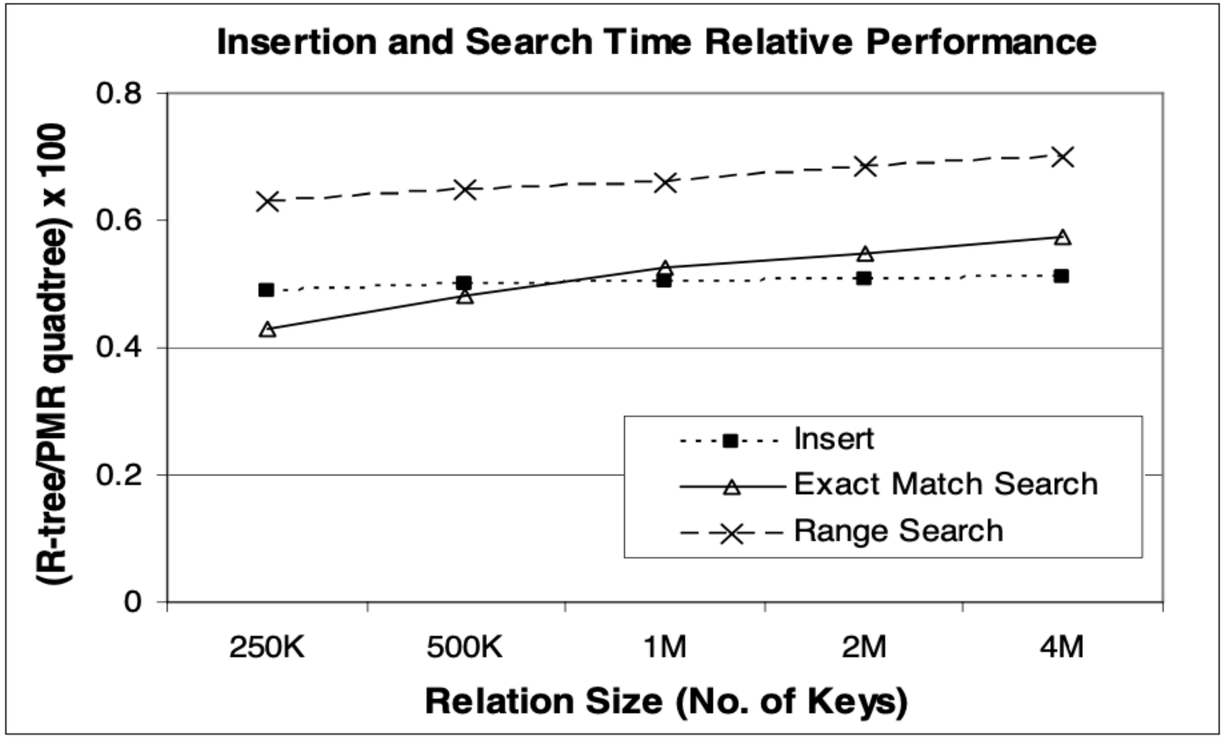
- Maximum tree height in nodes and pages
- Although trie has higher maximum node-height, the max. page-height is almost the same as the B+-tree page-height
- Thanks to SP-GiST's clustering technique that minimizes the tree maximum page height





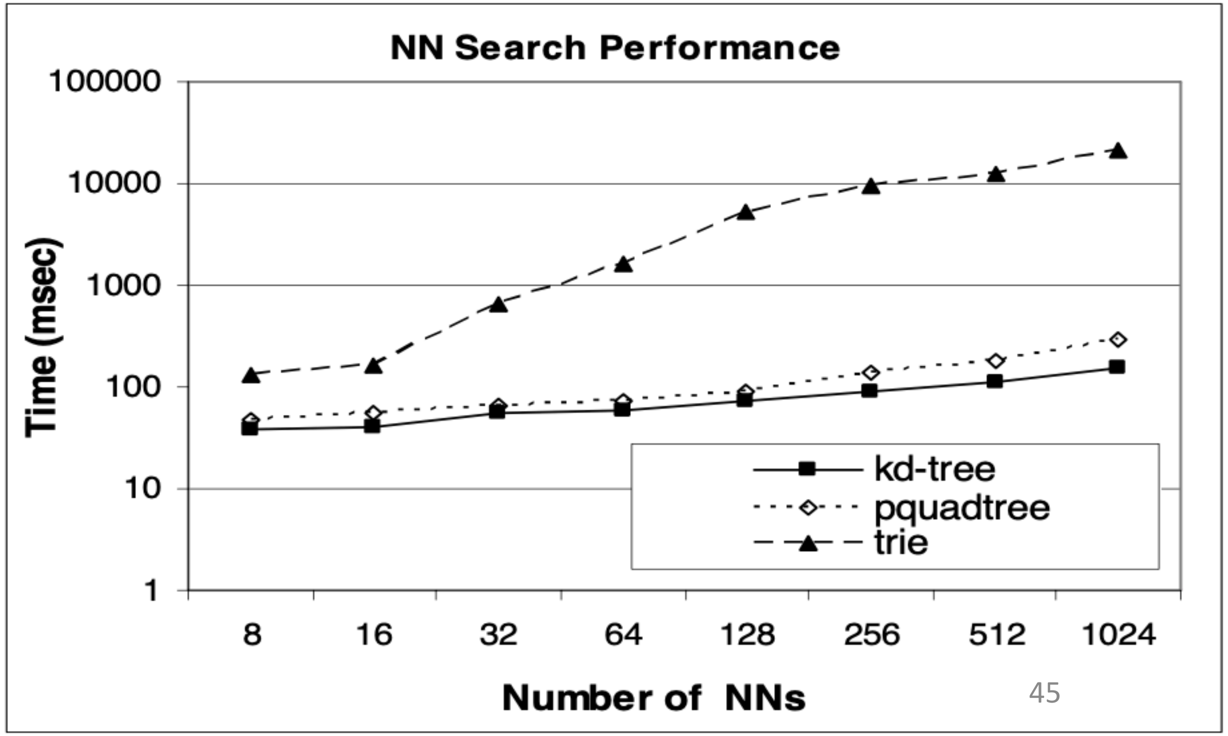
Performance of the R-tree vs. the PMR Quadtree

- PMR quadtree against the R-tree for indexing line segment datasets
- Measured:
  - Insertion time
  - *Exact match* search time
  - *Range (window)* search time
- R-tree has a better performance thar 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
- Search performance gap decreases with the increase of the data size



SP-GiST Nearest-Neighbor Search Performance

- Various SP- GiST instantiations of index structures
    - kd-tree
    - point quadtree
    - patricia trie
- Euclidean distance used as the distance function for the kd-tree and point quadtree
- Hamming distance used as the distance function for the trie
- Varied k from 8 to 1024
- Trie is much slower than kd-tree and point quadtree
- Comparison in trie is performed character by character
    - Makes convergence to the next NN relatively slow
    - Comparison in the kd-tree and quadtree is Partition-based



Conclusions

- SP-GiST as a generalized search tree inside a database engine
- Realized various versions of tries, quadtrees, kd-trees, and suffix trees
- Experiments demonstrate potential gain of SP-GiST indexes
    - Trie has > 150% search performance improvement over B+-tree in the case of the *exact match* search,
    - Has > 2 orders of magnitude search performance gain over the B+-tree for *regular expression match* search.
    - kd-tree also has more than 300% search performance improvement over the R-tree for *point match* search
    - realized NN-search inside SP-GiST and substring match operations
    - In addition to performance gains and the advanced search functionalities of SP-GiST indexes, the ability to rapidly prototype these indexes inside a DBMS is most attractive

Further Reading

- Mohamed Y Eltabakh, Mourad Ouzzani, Walid G. Aref, Duplicate Elimination in Space-partitioning Tree Indexes, 19th International Conference on Scientific and Statistical Database Management (SSDBM), 2007.
  - Consistency Reference is based on the idea of reporting an object at a certain point that is computed at the query run-time. 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.
- Thanaa M. Ghanem, Rahul Shah, Mohamed F. Mokbel, Walid G. Aref, Jeffrey Scott Vitter, Bulk operations for space-partitioning trees, 20th International Conference on Data Engineering (ICDE), Pages 29-40, 2004.
  - Bulk loading and bulk insertion

Thank you!

The authors acknowledge the support of the National Science Foundation under Grant Numbers III-1815796 andIIS-1910216.