Generalized Search Trees for Database Systems

Matthias Springer
matthias.springer@student.hpi.uni-potsdam.de

Seminar Paper
Beauty is our business, summer term 2012

September 12, 2012
supervised by Prof. Dr. Felix Naumann

1 INDEX DATA STRUCTURES

Every database system can basically store arbitrary data. We can always store tuples as a list of binary large objects (blobs) and, if we want to evaluate a query, check for every tuple if it satisfies the query[1]. The crux of the matter is that this usually takes too much time. An index data structure is an efficient lookup data structure, which locates relevant tuples without having to analyze all tuples. The key idea is to save some data redundantly, which costs hard disk and main memory space, but accelerates operations on the database.

Common index data structures are B^+ trees, hash indices and R trees. Every index data structure is suitable for a specific type of data. For instance, most database systems use B^+ trees for linearly ordered data like numbers, because B^+ trees allow efficient range queries. PostgreSQL 9.1 has built-in support for B^+ trees, hash indices and GIN indices[2].

The development of new index data structures is a cumbersome job, because we have to think about complex problems like concurrency control and recovery. Besides, we have to reimplement algorithms for searching and inserting into the index data structure all over again, for every index data structure. Hellerstein et al. estimated that the implementation of actual data-type-specific algorithms and data structures made up only 20 percent of the code for implementing an index data structure for PostgreSQL[1]. The PostgreSQL documentation states that programmers can "define their own index [data structures], but that is fairly complicated" *[PostgreSQL 9.1.5 Documentation]*[2].

This paper describes the development of new index data structures for all kinds of data. It is based on the paper *Generalized Search Trees for Database Systems*[1] by *Joseph M. Hellerstein*, *Jeffrey F. Naughton* and *Avi Pfeffer*.

2 GENERALIZED SERACH TREES FOR DATABASE SYSTEMS

The Generalized Search Tree for Database Systems (GiST) simplifies the development of new index data structures. Instead of implementing a complete index data structure, we only have to implement data-type-specific and query-specific functionality. Thus, we distinguish between two types of procedures.

- *GiST procedures* are only implemented once, for instance by the database system producer. Among GiST procedures are algorithms for searching and inserting into the GiST. These procedures also take care of concurrency control and recovery. GiST

[1] We also say, the tuple is consistent with the query.
[2] A GIN index is a modified version of a B^+ tree.

- *Data type procedures* are implemented for every index data structure. DATA Among data type procedures are implementations of every supported query operation, as well as four other procedures. All data type procedures are entirely data-type-specific, such that they could be implemented without understanding the concept of the GiST.

2.1 Notation

A GiST uses first-order predicate formulas as keys.

Definition $\mathbb{P}$ is the set of all predicate formulas. $\mathbb{T}$ is the set of all tuples.

Definition *Single predicates* are predicate formulas without logical operators. For convenience, we call predicate formulas simply *predicates*.

- $\top$ (true) and $\bot$ (false) are predicates.
- Binary functions $f : \mathbb{T} \to \{\top, \bot\}$ are predicates.
- If $a(t)$ is a predicate, then $\neg a(t)$ is a predicate.
- If $a(t)$ and $b(t)$ are predicates, then $a(t) \circ b(t)$ are predicates, with $\circ \in \{\wedge, \vee, \rightarrow, \leftarrow, \leftrightarrow\}$

Predicates always take exactly one parameter, which has to be a tuple[3].

Definition An inner node $N = \{\langle \mathtt{desc}_M, \mathtt{ptr}_M \rangle \mid M \text{ is a node}\}$ is a set of key-pointer pairs. $\mathtt{desc}_M$ is the *description* of node M and $\mathtt{ptr}_M$ is a pointer to node M. For leaf nodes, M is a tuple.

Definition By *subtree* A we denote the subtree which is rooted at the node A. The *description* $\mathtt{desc}_A$ of a subtree A is the key predicate associated to the pointer, which points to node A. The description of the whole tree is $\mathtt{desc}_R = \top$, where R is the root.

2.2 Structure

A GiST is a balanced search tree, which is very similar to a B^+ tree. In contrast to B^+ trees, however, every node has the same number of keys and pointers. Every node has a the same number of slots, which can be occupied by predicate-pointer pairs. Leaf nodes point to tuples and inner nodes point to other nodes.

You can use single predicates to define certain properties, which tuples can have. It is your job to implement single predicates, such that GiST algorithms can evaluate, whether a tuple satisfies a predicate formula.

Example The predicate $\text{has_cancer}(t) \wedge \text{is_female}(t)$ is a valid key for a medical database. This predicate holds true for every female person t with cancer.

[3] $\top$ and $\bot$ are constants and ignore the parameter.

The description of every node must hold true for every tuple reachable from that node.

Definition Let N be a node, $\text{tuples}(N)$ be the set of all tuples reachable from N and $\text{childs}(N)$ be the set of all child nodes of N.

1. Let R be the root of the tree. $\texttt{desc}_R := \top$
2. $\forall s \in \text{tuples}(N)\colon \texttt{desc}_N(s)$

In contrast to a B^+ tree, a GiST does not require descriptions on a higher level to be logical consequences of descriptions on a lower level.

- *B^+ tree:* $\forall C \in \text{childs}(N)\colon \forall t \in \mathbb{T}\colon \texttt{desc}_C(t) \rightarrow \texttt{desc}_N(t)$
- *GiST:* $\forall C \in \text{childs}(N)\colon \forall t \in \text{tuples}(C)\colon \texttt{desc}_C(t) \wedge \texttt{desc}_N(t)$ (deduced directly from *(1)*)

In contrast to B^+ trees, keys on the same level do not necessarily have to be disjunct within a GiST.

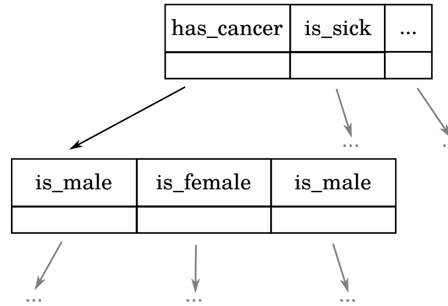


Figure 1: A GiST for a medical database. has_cancer is not a logical consequence of is_male, i.e. male people without cancer exist. The predicates has_cancer and is_sick are not disjunct, because people suffering from cancer are considered sick. Thus, if we wanted to search for people with cancer, you would have to search in both subtrees.

We did not specify an explicit formula for $\texttt{desc}_N$. There are many predicates which satisfy the requirements in the previous definition. We will discuss how to calculate $\texttt{desc}_N$ in section 3.2.

3 A GIST FOR INTEGER SET-VALUED DATA

We will develop a GiST, which will index sets of integers. The GiST will support these single query predicates.

- contains(i, t) holds true for sets t, which contain i.
- contains_any(l, r, t) holds true for sets t, which contain at least one number i with $l \leqslant i < r$.
- contains_all(l, r, t) holds true for sets t, which contain all numbers i with $l \leqslant i < r$.
- range(l, r, t) holds true for sets t, which contain *only* numbers i with $l \leqslant i < r$.

We use range(l, r, t) as the only key predicate inside the nodes.

3.1 Search

CONSISTENT(p, q) decides if $p \wedge q$ is satisfiable. DATA

Example Let $p = \text{range}(1, 10, t)$ and $q = \text{contains}(2, t) \wedge \text{contains}(3, t)$. $p \wedge q$ is satisfiable, for instance with $t = \{2, 3, 6, 8\}$.

Example Let $p = \text{range}(1, 10, t)$ and $q = \text{contains_all}(5, 15, t)$. $p \wedge q$ is not satisfiable, because t cannot include all integers i with $5 \leqslant i < 15$ and only include integers i with $1 \leqslant i < 10$ at the same time.

For a single predicate q, we calcuate $\text{consistent}(\text{range}(l, r, t), q)$ as follows[4].

- $\text{consistent}(\text{range}(l, r, t), \top)$
- $\neg\text{consistent}(\text{range}(l, r, t), \bot)$
- $\text{consistent}(\text{range}(l, r, t), \text{contains}(i, t)) \leftrightarrow l \leqslant i < r$
- $\text{consistent}(\text{range}(l_1, r_1, t), \text{contains_any}(l_2, r_2, t)) \leftrightarrow l_2 < r_1 \wedge l_1 < r_2$[5]
- $\text{consistent}(\text{range}(l_1, r_1, t), \text{contains_all}(l_2, r_2, t)) \leftrightarrow l_1 \leqslant l_2 \wedge r_1 \geqslant r_2$[6]
- $\text{consistent}(\text{range}(l_1, r_1, t), \text{range}(l_2, r_2, t))$[7], e.g. $t = \emptyset$

Many algorithms exist for deciding satisfiability for predicate formulas[8], e.g. Beth's tableaux[3] method. We will not discuss them in this paper.

SEARCH(N, q) finds all tuples satisfying the query predicate q. GiST

The search is initiated by calling $\text{search}(R, q)$, where R is the root of the GiST.

Algorithm 1: Pseudocode for $\text{search}(N, q)$

```
Data: N (node or tuple), q (query predicate)
Result: Output all tuples satisfying q

if N is tuple ∧ q(N) then
    output(N)
else
    for C ∈ childs(N) do
        if consistent(desc_C, q) then
            search(C, q)
```

The search algorithm might traverse more than one path in the tree, since the key predicates do not have to be disjunct. Although a GiST is a balanced search

[4] We say $\text{consistent}(p, q)$ if $\text{consistent}(p, q)$ returns $\top$ and $\neg\text{consistent}(p, q)$ otherwise.

[5] Both intervals intersect.

[6] $[l_1; r_1)$ contains $[l_2; r_2)$ entirely.

[7] It does not make sense to use range as a query predicate. Thus we need no implementation for range. We must implement all other single predicates for search.

[8] All predicates are monadic, therefore satisfiability is decidable.

tree, the number of visited nodes is not bound by $\mathcal{O}(\log n)$ but bound by $\mathcal{O}(n)$[9], where n is the number of tuples.

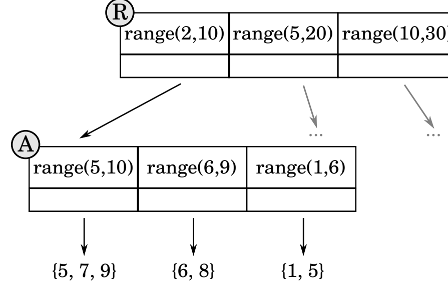


Figure 2: Example for search(R, contains(7, t)). range(2, 10, t) and range(5, 20, t) are consistent with contains(7, t), so search is called on these two childs. Let us take a look at search(A, contains(7, t)). The algorithm again follows the two consistent keys range(5, 10, t) and range(6, 9, t). Now search is called on two tuples. The algorithm evaluates contains(7, {5, 7, 9}) and contains(7, {6, 8})[a], but only the latter tuple is output.

[a] We must implement every single (query) predicate as a data type procedure. The algorithm for evaluating a whole predicate formula is a GiST procedure.

3.2 Insert

DESCRIBE(P) calculates a description for a tuple or a set of predicates. DATA

Definition Let t be a tuple. describe(t) calculates a (key) predicate, such that $\forall q \in \mathbb{P} : q(t) \rightarrow$ consistent(describe(t), q). Let P be a set of predicates. describe(P) calculates a predicate, such that $\forall t \in \mathbb{T}$: $\bigvee_{p \in P} p(t) \rightarrow$ describe(P)(t).

Example For a tuple $t = \{0, 1, 5\}$, describe(t) = range(0, 6, t)[10] is a solution.

describe(P) calculates a predicate r, such that r is a logical consequence of any predicate (description) within P. For integer set-valued tuple, we can calculate describe as follows.

$$\text{describe}(\{\text{range}(l_1, r_1, t), \text{range}(l_2, r_2, t), \ldots\}) = \text{range}(\min_{i \in \mathbb{N}} l_i, \max_{i \in \mathbb{N}} r_i, t)$$

Example For a set of tuples $P = \{\text{range}(2, 10, t), \text{range}(1, 5, t), \text{range}(15, 20, t)\}$, describe(P) = range(1, 20, t) is a valid solution.

In the previous example describe(P) = range(0, 1000, t) and describe(P) = $\top$ are also valid solutions[11]. For efficiency reasons you should calculate a predicate, which describes P as accurate as possible.

[9] Worst case scenario: all key predicates are $\top$, so we visit all nodes.
[10] We only allowed range as a key predicate. contains_any(0, 6, t) is valid, but not allowed.
[11] $\top$ is not allowed, because range is the only key predicate.

SPLIT(P) seperates a set of predicates into two disjunct sets. DATA

Definition Let P be a set of predicates. split creates two sets P_1 and P_2 with $P_1 \cup P_2 = P$, $P_1 \cap P_2 = \emptyset$ and $|\#P_1 - \#P_2| \leqslant 1$.

For efficiency reasons, you should choose P_1 and P_2 in such a way, that the descriptions of P_1 and P_2, which are generated by describe, are as different as possible. In other words, the descriptions should overlap as little as possible. The overlap O is defined as follows.

$$O = \{t \in \mathbb{T} \mid \text{describe}(P_1)(t)\} \cap \{t \in \mathbb{T} \mid \text{describe}(P_2)(t)\}$$

Example For tuples $P = \{\text{range}(2,5,t), \text{range}(1,8,t), \text{range}(6,20,t)\}$, split($P$) should generate $P_1 = \{\text{range}(2,5,t), \text{range}(1,8,t)\}$ and $P_2 = \{\text{range}(6,20,t)\}$ with describe(P_1) $= \text{range}(1,8,t)$ and describe(P_2) $= \text{range}(6,20,t)$. The overlap is $O = \{6,7\}$, because these numbers are contained in both ranges.

$P_1 = \{\text{range}(1,8,t), \text{range}(6,20,t)\}$ and $P_2 = \{\text{range}(2,5,t)\}$ result in a bigger overlap $O = \{2,3,4\}$, which is a worse solution.

To calculate split(P) for integer set-valued tuples, you can sort the elements of P by the left range value and split the list in the middle[12].

PENALTY(p, t) calculates how well two predicates fit together. DATA

Definition Let p and t be key predicates. penalty(p, t) calculates, to which extent p needs to be generalized in order to be a logical consequence of t.

$$\text{penalty}(p,t) = \#\{u \in \mathbb{T} \mid \text{describe}(\{p,t\})(u)\} - \#\{u \in \mathbb{T} \mid p(u)\}$$

Example Let $p_1 = \text{range}(1,20,t)$, $p_2 = \text{range}(10,25,t)$ and $t = \text{range}(5,15,t)$. To decide, whether t fits best to p_1 or to p_2, we calculate the penalty for both predicates.

$$\text{describe}(p_1, t) = \text{range}(1,20,t)$$

$$\text{penalty}(p_1, t) = \#\{u \mid \text{range}(1,20,u)\} - \#\{u \mid \text{range}(1,20,u)\} = 0$$

$$\text{describe}(p_2, t) = \text{range}(5,25,t)$$

$$\text{penalty}(p_2, t) = \#\{u \mid \text{range}(5,25,u)\} - \#\{u \mid \text{range}(1,20,u)\} = 5$$

p_1 has a lower penalty than p_2, so t fits better to p_2. This is what we expected, too, since the range t is completely contained in the range p_1, whereas t just intersects p_2 partly.

[12]This algorithm does not always generate the optimal solution, but it is simple and fast.

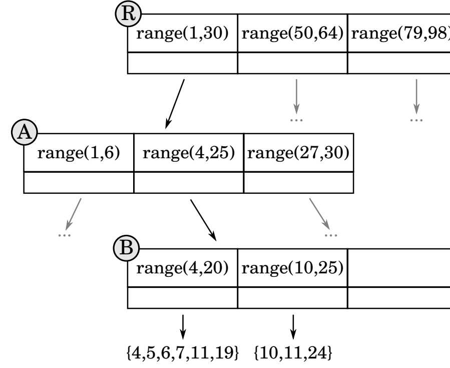
For integer set-valued tuples, we can calculate penalty as follows[13].

$$\text{penalty}(\text{range}(l_p, r_p, u), \text{range}(l_t, r_t, u)) = \max\{l_p - l_t, 0\} + \max\{r_t - r_p, 0\}$$

$\max\{l_p - l_t, 0\}$ is the expansion of p to the left and $\max\{r_t - r_p, 0\}$ is the expansion of p to the right.

CHOOSESUBTREE(N, p) determines into which leaf node we insert a tuple with description p. GiST

Figure 3: Example for chooseSubtree. We insert a tuple t with $p = \text{describe}(t) = \text{range}(15, 31, t)$ into the GiST with root R. chooseSubtree(R, t) calculates penalty$(\text{range}(1, 30, t), p) = 1$, penalty$(\text{range}(50, 64, t), p) = 35$ and an even worse penalty for the third node. Therefore, we continue with chooseSubtree(A, t). We choose the node B with penalty$(\texttt{desc}_B, p) = 6$. B is a leaf, chooseSubtree$(R, t) = B$.



Algorithm 2: Pseudocode for chooseSubtree(N, t)

Data: N (node), t (tuple to insert)
Result: Return leaf node to insert tuple into

if N *is leaf* **then**
  return(N)
else
  return $\left(\text{chooseSubtree}\left(\arg\min_{c \in \text{childs}(N)} \text{penalty}(\texttt{desc}_c, \text{describe}(t)), t\right)\right)$

chooseSubtree is a greedy algorithm, which inspects one path from the root to a leaf. It always chooses the node with the lowest penalty. It does not necessarily find the optimal node to insert the tuple into.

SPLITINSERT(N, t) inserts a node or tuple t into a node N. If N has no free slot, split creates two nodes from the N and t. splitInsert recursively inserts predicate-pointer pairs for these two nodes in the parent node. GiST

[13] For the insert procedure, we merely need to calculate penalty for the only key predicate range.

In the worst case, splitInsert terminates after the root is split. Thus splitInsert performs $\mathcal{O}(\log n)$ insertions, where n is the number of tuples in the GiST.

Algorithm 3: Pseudocode for splitInsert(N, t)

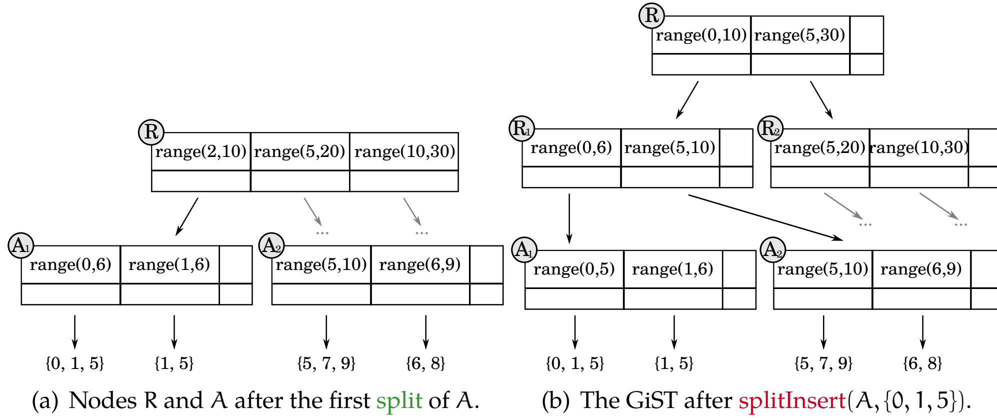
Data: N (node), t (node or tuple)
Result: t part of N, return last insertion node

if N *is full* **then**
  $P_1, P_2 = \text{split}^a(\{\langle \texttt{desc}_c, \texttt{ptr}_c \rangle \mid c \in \text{childs}(N)\} \cup \{\langle \text{describe}(t), \texttt{ptr}_t \rangle\})$
  $\text{parent}(N)^b.\text{remove}^c(\langle \texttt{desc}_N, \texttt{ptr}_N \rangle)$
  splitInsert$(\text{parent}(N), P_1)$
  return(splitInsert$(\text{parent}(N), P_2)$)
else
  $N.\text{add}(\langle \text{describe}(t), \texttt{ptr}_t \rangle)$
  return(N)

[a] For convenience, we use split to split predicate-pointer pairs instead of just pointers.
[b] If R is the root, parent(R) generates a new root node and returns the new node.
[c] Removing predicate-pointer pairs from an empty root node has no effect.

We use $\texttt{desc}_N$ for nodes N which are already part of the GiST. In contrast, we use describe(N) for new nodes or tuples, for which we do not yet have a description.

Example We want to insert the tuple $\{0, 1, 5\}$ into node A in figure 2.



(a) Nodes R and A after the first split of A. (b) The GiST after splitInsert$(A, \{0, 1, 5\})$.

Figure 4: (a) Node A is full, so it is split into the nodes A_1 and A_2. (b) $\langle \text{range}(2, 10, t), \texttt{ptr}_A \rangle$ is removed from node A. $\langle \text{range}(0, 6, t), \texttt{ptr}_{A_1} \rangle$ is inserted into R, but to insert $\langle \text{range}(5, 10, t), \texttt{ptr}_{A_2} \rangle$, node R must be split, too. parent(R) creates a new root and the predicate-pointer pairs for the split nodes R_1 and R_2 are inserted.

ADJUSTKEYS(R,N) updates the descriptions above the insertion. GiST

Algorithm 4: Pseudocode for adjustKeys(R, N)

Data: R (root node), N leaf node of insertion
Result: All descriptions within R are accurate

if $N \neq R \wedge \mathit{describe}(N) \neq \mathtt{desc}_N$ **then**
    $\mathtt{desc}_N :=$ describe(N)
    adjustKeys$(R, \text{parent}(N))$

After inserting a predicate-pointer pair into a node, all descriptions of nodes on the way to the root must be updated. Otherwise the search algorithm might not realize that a subtree is relevant for a query. splitInsert already updates all descriptions up to the highest node of insertion. Higher nodes must be updated, until the root is reached or, at any point, the description does not change anymore. In that case, the description is already accurate.

INSERT(R, t) inserts a tuple t into a GiST with root R. GiST

Algorithm 5: Pseudocode for insert(R, t)

Data: R (root node), t (tuple to insert)
Result: t inserted into R

$T :=$ chooseSubtree(R, t)
adjustKeys$(R,$ splitInsert$(T, t))$

4 EFFICIENCY AND IMPLEMENTATION ISSUES

CONSISTENT(p, q) Satisfiability for monadic predicates is decidable, but it is $\mathcal{NP}$-complete. The search algorithm operates correctly, even if consistent produces false positives, because every tuple is checked before it is output. In other words, you can use a heuristic for satisfiability, which may always return $\top$. However, the quality of consistent has a severe effect on the performance of search. False positives make search traverse parts of the tree, which are irrelevant for a query. It is your job to develop a consistent procedure which does not produce too many false positive and can be calculated fast enough.

DESCRIBE(P) Technically describe may always return $\top$, since $\top$ is a logical consequence of every predicate. However, it is critical for the performance of search that describe generates an accurate description. Again, poor descriptions will not make search fail, but they result in unnecessary tree traversals.

In section 2.2, we defined $\mathtt{desc}_N$ to hold true for all reachable tuples. We explicitly did not require $\mathtt{desc}_N$ to be a logical consequence of all reachable

(lower) descriptions. However, describe(P) only generates predicates which are a logical consequence of P. Therefore, a GiST, as defined in this paper[14], cannot benefit from this feature. The definition of describe(P) could be changed to consider all reachable tuples instead of only the descriptions, but this would take considerably more time. A GiST could, however, benefit from our definition of $\mathtt{desc}_N$ at bulk loading.

SPLIT(P) You should implement split in such a way, that it generates two sets of tuples with a minimal overlap. A big overlap increases the probability for search to traverse a high number of subtrees, because multiple child nodes might seem relevant.

5 CONCLUSION

The GiST is a framework, which simplifies the development of new index data structures. The GiST procedures adjustKeys, insert, search and splitInsert contain algorithms and database-specific functionality like locking and recovery. For the development of a new index data structure, we only have to implement the data type procedures consistent, describe, penalty and split[15].

Non-GiST implementations of index data structures are usually more efficient than GiST implementations, as they allow greater optimizations. The basic idea of the GiST was never to provide maximum performance but to facilitate development. An implementation of the GiST exists for PostgreSQL[16].

REFERENCES

[1] HELLERSTEIN, J. M., NAUGHTON, J. F., AND PFEFFER, A. Generalized search trees for database systems. In *Proceedings of the 21th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1995), VLDB '95, Morgan Kaufmann Publishers Inc., pp. 562–573.

[2] POSTGRESQL. Postgresql 9.1.5 documentation. create index. http://www.postgresql.org/docs/9.1/static/sql-createindex.html, 2012. [Online; accessed 09/10/2012].

[3] WIKIPEDIA. Method of analytic tableaux — wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Method_of_analytic_tableaux&oldid=494342495, 2012. [Online; accessed 09/10/2012].

[14] It is defined in the same matter in the paper of Hellerstein et al.

[15] Hellerstein et al. use two additional procedures (compress, decompress) for space efficiency.

[16] For more information, see http://www.sai.msu.su/~megera/postgres/gist/ and the PostgreSQL documentation[2].