Generalized Search Trees for Database Systems

Joseph M. Hellerstein, Jeffrey F. Naughton, Avi Pfeffer
{hellers, naughton}@cs.wisc.edu, avi@cs.berkeley.edu

June, 1995

Abstract

This paper introduces the Generalized Search Tree (GiST), an index structure supporting an extensible set of queries and data types. The GiST allows new data types to be indexed in a manner supporting queries natural to the types; this is in contrast to previous work on tree extensibility which only supported the traditional set of equality and range predicates. In a single data structure, the GiST provides all the basic search tree logic required by a database system, thereby unifying disparate structures such as B+-trees and R-trees in a single piece of code, and opening the application of search trees to general extensibility.

To illustrate the flexibility of the GiST, we provide simple method implementations that allow it to behave like a B+-tree, an R-tree, and an *RD-tree*, a new index for data with set-valued attributes. We also present a preliminary performance analysis of RD-trees, which leads to discussion on the nature of tree indices and how they behave for various datasets.

1 Introduction

An efficient implementation of search trees is crucial for any database system. In traditional relational systems, B+-trees [Com79] were sufficient for the sorts of queries posed on the usual set of alphanumeric data types. Today, database systems are increasingly being deployed to support new applications such as geographic information systems, multimedia systems, CAD tools, document libraries, sequence databases, fingerprint identification systems, biochemical databases, etc. To support the growing set of applications, search trees must be extended for maximum flexibility. This requirement has motivated two major research approaches in extending search tree technology:

1. *Specialized Search Trees:* A large variety of search trees has been developed to solve specific problems. Among the best known of these trees are spatial search trees such as R-trees [Gut84]. While some of this work has had significant impact in particular domains, the approach of developing domain-specific search trees is problematic. The effort required to implement and maintain such data structures is high. As new applications need to be supported, new tree structures have to be developed from scratch, requiring new implementations of the usual tree facilities for search, maintenance, concurrency control and recovery.

2. *Search Trees For Extensible Data Types:* As an alternative to developing new data structures, existing data structures such as B+-trees and R-trees can be made *extensible* in the data types they support [Sto86]. For example, B+-trees can be used to index any data with a linear ordering, supporting equality or linear range queries over that data. While this provides extensibility in the data that can be indexed, it does not extend the set of queries which can be supported by the tree. Regardless of the type of data stored in a B+-tree, the only queries that can benefit from the tree are those containing equality and linear range predicates. Similarly in an R-tree, the only queries that can use the tree are those containing equality, overlap and containment predicates. This inflexibility presents significant

problems for new applications, since traditional queries on linear orderings and spatial location are unlikely to be pertinent for new data types.

In this paper we present a third direction for extending search tree technology. We introduce a new data structure called the Generalized Search Tree (GiST), which is easily extensible both in the data types it can index and in the queries it can support. Extensibility of queries is particularly important, since it allows new data types to be indexed in a manner that supports the queries natural to the types. In addition to providing extensibility for new data types, the GiST unifies previously disparate structures used for currently common data types. For example, both B+-trees and R-trees can be implemented as extensions of the GiST, resulting in a single code base for indexing multiple dissimilar applications.

The GiST is easy to configure: adapting the tree for different uses only requires registering six *methods* with the database system, which encapsulate the structure and behavior of the object class used for keys in the tree. As an illustration of this flexibility, we provide method implementations that allow the GiST to be used as a B+-tree, an R-tree, and an *RD-tree*, a new index for data with set-valued attributes. The GiST can be adapted to work like a variety of other known search tree structures, *e.g.* partial sum trees [WE80], k-D-B-trees [Rob81], Ch-trees [KKD89], Exodus large objects [CDG+90], hB-trees [LS90], V-trees [MCD94], TV-trees [LJF94], etc. Implementing a new set of methods for the GiST is a significantly easier task than implementing a new tree package from scratch: for example, the POSTGRES [Gro94] and SHORE [CDF+94] implementations of R-trees and B+-trees are on the order of 3000 lines of C or C++ code each, while our method implementations for the GiST are on the order of 500 lines of C code each.

In addition to providing an unified, highly extensible data structure, our general treatment of search trees sheds some initial light on a more fundamental question: if any dataset can be indexed with a GiST, does the resulting tree always provide efficient lookup? The answer to this question is "no", and in our discussion we illustrate some issues that can affect the efficiency of a search tree. This leads to the interesting question of how and when one can build an efficient search tree for queries over non-standard domains — a question that can now be further explored by experimenting with the GiST.

1.1 Structure of the Paper

In Section 2, we illustrate and generalize the basic nature of database search trees. Section 3 introduces the Generalized Search Tree object, with its structure, properties, and behavior. In Section 4 we provide GiST implementations of three different sorts of search trees. Section 5 presents some performance results that explore the issues involved in building an effective search tree. Section 6 examines some details that need to be considered when implementing GiSTs in a full-fledged DBMS. Section 7 concludes with a discussion of the significance of the work, and directions for further research.

1.2 Related Work

A good survey of search trees is provided by Knuth [Knu73], though B-trees and their variants are covered in more detail by Comer [Com79]. There are a variety of multidimensional search trees, such as R-trees [Gut84] and their variants: R*-trees [BKSS90] and R+-trees [SRF87]. Other multidimensional search trees include quad-trees [FB74], k-D-B-trees [Rob81], and hB-trees [LS90]. Multidimensional data can also be transformed into unidimensional data using a space-filling curve [Jag90]; after transformation, a B+-tree can be used to index the resulting unidimensional data.

Extensible-key indices were introduced in POSTGRES [Sto86, Aok91], and are included in Illustra [Sto93], both of which have distinct extensible B+-tree and R-tree implementations. These extensible indices allow many types of data to be indexed, but only support a fixed set of query predicates. For example, POSTGRES B+-trees support the usual ordering predicates ($<, \leq, =, \geq, >$), while POSTGRES R-trees support only the predicates Left, Right, OverLeft, Overlap, OverRight, Right, Contains, Contained and Equal [Gro94].

Extensible R-trees actually provide a sizable subset of the GiST's functionality. To our knowledge this paper represents the first demonstration that R-trees can index data that has not been mapped into a spatial domain. However, besides their limited extensibility R-trees lack a number of other features supported by

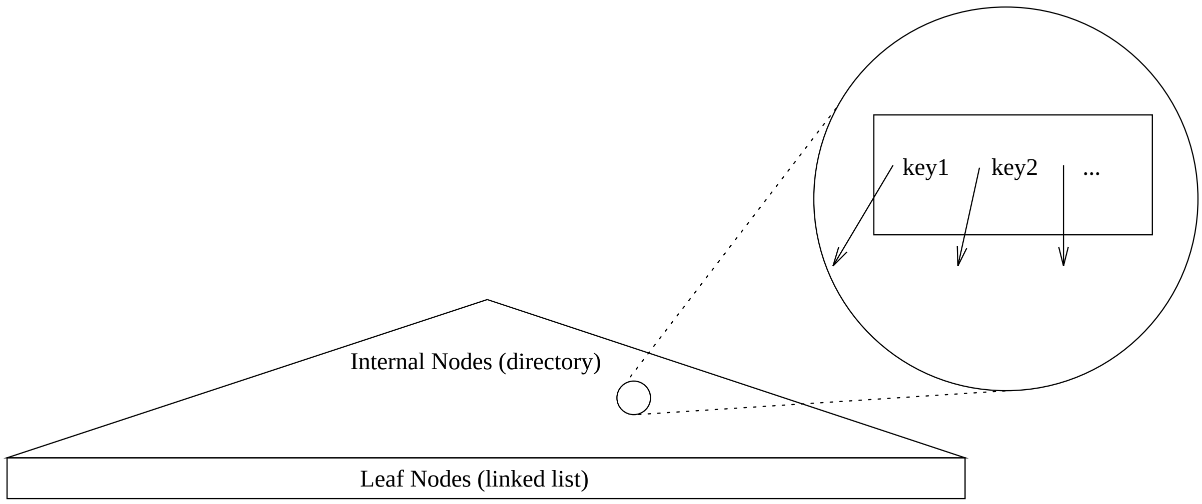

Figure 1: Sketch of a database search tree.

the GiST. R-trees provide only one sort of key predicate (Contains), they do not allow user specification of the PickSplit and Penalty algorithms described below, and they lack optimizations for data from linearly ordered domains. Despite these limitations, extensible R-trees are close enough to GiSTs to allow for the initial method implementations and performance experiments we describe in Section 5.

Classification trees such as ID3 and C4.5 [Qui93] are similar in spirit to the GiST, but have some major differences. Most significantly, classification trees are not intended as search structures, and are not suited for indexing large amounts of data. Additionally, ID3 and C4.5 are built over pre-defined, discrete classes, whereas the GiST is a dynamic structure that develops its own, unspecified data buckets (leaves), with potentially overlapping keys. Finally, classification trees are not generally extensible: they are defined over traditional tuple-like objects with attributes from numeric or finitely enumerated domains. However, extensions of the GiST may be able to leverage some of the lessons learned in the development of classification trees, particularly in implementing methods for picking a way to split nodes (the PickSplit method described below) and for choosing good keys (the Union and Compress methods described below.)

The Predicate Trees of Valduriez and Viemont [VV84] are similar to classification trees, in that they are not used to index data sets, and can be defined only on traditional alphanumeric predicates. Predicate Trees are used to generate hash keys. A even more limited notion is that of the Interval Hierarchy [WE77], which can be used to index a file of simple conjunctive ordering predicates.

Analyses of R-tree performance have appeared in [FK94] and [PSTW93]. This work is dependent on the spatial nature of typical R-tree data, and thus is not generally applicable to the GiST. However, similar ideas may prove relevant to our questions of when and how one can build efficient indices in arbitrary domains.

2 Getting The Gist of Database Search Trees

As an introduction to GiSTs, it is instructive to review search trees in a simplified manner. Most people with database experience have an intuitive notion of how search trees work, so our discussion here is purposely vague: the goal is simply to illustrate that this notion leaves many details unspecified. After highlighting the unspecified details, we can proceed to describe a structure that leaves the details open for user specification.

The canonical rough picture of a database search tree appears in Figure 1. It is a balanced tree, with high fanout. The internal nodes are used as a directory. The leaf nodes contain pointers to the actual data, and are stored as a linked list to allow for partial or complete scanning.

Within each internal node is a series of keys and pointers. To search for tuples which match a query predicate q, one starts at the root node. For each pointer on the node, if the associated key is *consistent* with q, *i.e.* the key does not rule out the possibility that data stored below the pointer may match q, then one traverses the subtree below the pointer, until all the matching data is found. As an illustration, we review the notion of consistency in some familiar tree structures. In B+-trees, queries are in the form of range predicates (*e.g.* "find all i such that $c_1 \leq i \leq c_2$"), and keys logically delineate a range in which the data below a pointer is contained. If the query range and a pointer's key range overlap, then the two are consistent and the pointer is traversed. In R-trees, queries are in the form of region predicates (*e.g.* "find all i such that (x_1, y_1, x_2, y_2) overlaps i"), and keys delineate the bounding box in which the data below a pointer is contained. If the query region and the pointer's key box overlap, the pointer is traversed.

Note that in the above description, the only restriction placed on a key is that it must logically match each datum stored below it, so that the consistency check does not miss any valid data. In B+-trees and R-trees, keys are essentially "containment" predicates: they describe a contiguous region in which all the data below a pointer are contained. Containment predicates are not the only possible key constructs, however. For example, the predicate "$\mathsf{elected\_official}(i) \wedge \mathsf{has\_criminal\_record}(i)$" is an acceptable key if every data item i stored below the associated pointer satisfies the predicate. As in R-trees, keys on a node may "overlap", *i.e.* two keys on the same node may hold simultaneously for some tuple.

This flexibility allows us to generalize the notion of a search key: *a search key may be any arbitrary predicate that holds for each datum below the key.* Given a data structure with such flexible search keys, a user is free to form a tree by organizing data into arbitrary nested sub-categories, labelling each with some characteristic predicate. This in turn lets us capture the essential nature of a database search tree: *it is a hierarchy of categorizations, in which each categorization holds for all data stored under it in the hierarchy.* Searches on arbitrary predicates may be conducted based on the categorization. In order to support searches on a predicate q, the user must provide a Boolean method to tell if q is consistent with a given search key. When this is so, the search proceeds by traversing the pointer associated with the search key. The grouping of data into categories may be controlled by a user-supplied node splitting algorithm, and the characterization of categories can be done with user-supplied search keys. Thus by exposing the key methods and the tree's split method to the user, arbitrary search trees may be constructed, supporting an extensible set of queries. These ideas form the basis of the GiST, which we proceed to describe in detail.

3 The Generalized Search Tree

In this section we present the abstract data type (or "object") *Generalized Search Tree* (GiST). We define its structure, its invariant properties, its extensible methods and its built-in algorithms. As a matter of convention, we refer to each indexed datum as a "tuple"; in an Object-Oriented or Object-Relational DBMS, each indexed datum could be an arbitrary data object.

3.1 Structure

A GiST is a balanced tree of variable fanout between kM and M, $\frac{2}{M} \leq k \leq \frac{1}{2}$, with the exception of the root node, which may have fanout between 2 and M. The constant k is termed the *minimum fill factor* of the tree. Leaf nodes contain $(p, \texttt{ptr})$ pairs, where p is a predicate that is used as a search key, and `ptr` is the identifier of some tuple in the database. Non-leaf nodes contain $(p, \texttt{ptr})$ pairs, where p is a predicate used as a search key and `ptr` is a pointer to another tree node. Predicates can contain any number of free variables, as long as any single tuple referenced by the leaves of the tree can instantiate all the variables. Note that by using "key compression", a given predicate p may take as little as zero bytes of storage. However, for purposes of exposition we will assume that entries in the tree are all of uniform size. Discussion of variable-sized entries is deferred to Section 6.3. We assume in an implementation that given an entry $E = (p, \texttt{ptr})$, one can access the node on which E currently resides. This can prove helpful in implementing the key methods described below.

3.2 Properties

The following properties are invariant in a GiST:

1. Every node contains between kM and M index entries unless it is the root.
2. For each index entry $(p, \texttt{ptr})$ in a leaf node, p is true when instantiated with the values from the indicated tuple (*i.e.* p *holds* for the tuple.)
3. For each index entry $(p, \texttt{ptr})$ in a non-leaf node, p is true when instantiated with the values of any tuple reachable from ptr. Note that, unlike in R-trees, for some entry $(p', \texttt{ptr}')$ reachable from ptr, we do not require that $p' \to p$, merely that p and p' both hold for all tuples reachable from $\texttt{ptr}'$.
4. The root has at least two children unless it is a leaf.
5. All leaves appear on the same level.

Property 3 is of particular interest. An R-tree would require that $p' \to p$, since bounding boxes of an R-tree are arranged in a containment hierarchy. The R-tree approach is unnecessarily restrictive, however: the predicates in keys above a node N must hold for data below N, and therefore one need not have keys on N restate those predicates in a more refined manner. One might choose, instead, to have the keys at N characterize the sets below based on some entirely orthogonal classification. This can be an advantage in both the information content and the size of keys.

3.3 Key Methods

In principle, the keys of a GiST may be arbitrary predicates. In practice, the keys come from a user-implemented object class, which provides a particular set of methods required by the GiST. Examples of key structures include ranges of integers for data from $\mathbb{Z}$ (as in B+-trees), bounding boxes for regions in $\mathbb{R}^n$ (as in R-trees), and bounding sets for set-valued data, *e.g.* data from $\mathcal{P}(\mathbb{Z})$ (as in RD-trees, described in Section 4.3.) The key class is open to redefinition by the user, with the following set of six methods required by the GiST:

- **Consistent**(E,q): given an entry $E = (p, \texttt{ptr})$, and a query predicate q, returns false if $p \wedge q$ can be guaranteed unsatisfiable, and true otherwise. Note that an accurate test for satisfiability is not required here: Consistent may return true incorrectly without affecting the correctness of the tree algorithms. The penalty for such errors is in performance, since they may result in exploration of irrelevant subtrees during search.
- **Union**(P): given a set P of entries $(p_1, \texttt{ptr}_1), \ldots (p_n, \texttt{ptr}_n)$, returns some predicate r that holds for all tuples stored below $\texttt{ptr}_1$ through $\texttt{ptr}_n$. This can be done by finding a predicate r such that $(p_1 \vee \ldots \vee p_n) \to r$.
- **Compress**(E): given an entry $E = (p, \texttt{ptr})$ returns an entry $(\pi, \texttt{ptr})$ where π is a compressed representation of p.
- **Decompress**(E): given a compressed representation $E = (\pi, \texttt{ptr})$, where $\pi = \text{Compress}(p)$, returns an entry $(r, \texttt{ptr})$ such that $p \to r$. Note that this is a potentially "lossy" compression, since we do not require that $p \leftrightarrow r$.
- **Penalty**(E_1, E_2): given two entries $E_1 = (p_1, \texttt{ptr}_1), E_2 = (p_2, \texttt{ptr}_2)$, returns a domain-specific penalty for inserting E_2 into the subtree rooted at E_1. This is used to aid the Split and Insert algorithms (described below.) Typically the penalty metric is some representation of the increase of size from $E_1.p_1$ to Union(E_1, E_2). For example, Penalty for keys from $\mathbb{R}^2$ can be defined as area(Union(E_1, E_2))$-$ area$(E_1.p_1)$ [Gut84].

- **PickSplit**(P): given a set P of $M+1$ entries $(p, \texttt{ptr})$, splits P into two sets of entries P_1, P_2, each of size at least kM. The choice of the minimum fill factor for a tree is controlled here. Typically, it is desirable to split in such a way as to minimize some badness metric akin to a multi-way Penalty, but this is left open for the user.

The above are the only methods a GiST user needs to supply. Note that Consistent, Compress, Union and Penalty have to be able to handle any predicate in their input. In full generality this could become very difficult, especially for Consistent. But typically a limited set of predicates is used in any one tree, and this set can be constrained in the method implementation.

There are a number of options for key compression. A simple implementation can let both Compress and Decompress be the identity function. A more complex implementation can have Compress($(p, \texttt{ptr})$) generate a valid but more compact predicate r, $p \rightarrow r$, and let Decompress be the identity function. This is the technique used in SHORE's R-trees, for example, which upon insertion take a polygon and compress it to its bounding box, which is itself a valid polygon. It is also used in prefix B+-trees [Com79], which truncate split keys to an initial substring. More involved implementations might use complex methods for both Compress and Decompress.

3.4 Tree Methods

The key methods in the previous section must be provided by the designer of the key class. The tree methods in this section are provided by the GiST, and may invoke the required key methods. Note that keys are Compressed when placed on a node, and Decompressed when read from a node. We consider this implicit, and will not mention it further in describing the methods.

3.4.1 Search

Search comes in two flavors. The first method, presented in this section, can be used to search any dataset with any query predicate, by traversing as much of the tree as necessary to satisfy the query. It is the most general search technique, analogous to that of R-trees. A more efficient technique for queries over linear orders is described in the next section.

Algorithm Search(R, q)

Input: GiST rooted at R, predicate q

Output: all tuples that satisfy q

Sketch: Recursively descend all paths in tree whose keys are consistent with q.

S1: [Search subtrees] If R is not a leaf, check each entry E on R to determine whether Consistent(E, q). For all entries that are Consistent, invoke Search on the subtree whose root node is referenced by $E.\texttt{ptr}$.

S2: [Search leaf node] If R is a leaf, check each entry E on R to determine whether Consistent(E, q). If E is Consistent, it is a qualifying entry. At this point $E.\texttt{ptr}$ could be fetched to check q accurately, or this check could be left to the calling process.

Note that the query predicate q can be either an exact match (equality) predicate, or a predicate satisfiable by many values. The latter category includes "range" or "window" predicates, as in B+ or R-trees, and also more general predicates that are not based on contiguous areas (*e.g.* set-containment predicates like "all supersets of $\{6, 7, 68\}$".)

3.4.2 Search In Linearly Ordered Domains

If the domain to be indexed has a linear ordering, and queries are typically equality or range-containment predicates, then a more efficient search method is possible using the FindMin and Next methods defined in this section. To make this option available, the user must take some extra steps when creating the tree:

1. The flag **IsOrdered** must be set to true. IsOrdered is a static property of the tree that is set at creation. It defaults to false.

2. An additional method **Compare**(E_1, E_2) must be registered. Given two entries $E_1 = (p_1, \mathtt{ptr}_1)$ and $E_2 = (p_2, \mathtt{ptr}_2)$, Compare returns a negative number if p_1 precedes p_2, returns a positive number if p_1 follows p_2 and returns 0 otherwise. Compare is used to insert entries in order on each node.

3. The PickSplit method must ensure that for any entries E_1 on P_1 and E_2 on P_2, Compare$(E_1, E_2) < 0$.

4. The methods must assure that no two keys on a node overlap, *i.e.* for any pair of entries E_1, E_2 on a node, Consistent$(E_1, E_2.p)$ = false.

If these four steps are carried out, then equality and range-containment queries may be evaluated by calling FindMin and repeatedly calling Next, while other query predicates may still be evaluated with the general Search method. FindMin/Next is more efficient than traversing the tree using Search, since FindMin and Next only visit the non-leaf nodes along one root-to-leaf path. This technique is based on the typical range-lookup in B+-trees.

Algorithm FindMin(R, q)

Input: GiST rooted at R, predicate q

Output: minimum tuple in linear order that satisfies q

Sketch: descend leftmost branch of tree whose keys are Consistent with q. When a leaf node is reached, return the first key that is Consistent with q.

FM1: [Search subtrees] If R is not a leaf, find the first entry E in order such that Consistent(E, q)[1]. If such an E can be found, invoke FindMin on the subtree whose root node is referenced by $E.\mathtt{ptr}$. If no such entry is found, return NULL.

FM2: [Search leaf node] If R is a leaf, find the first entry E on R such that Consistent(E, q), and return E. If no such entry exists, return NULL.

Given one element E that satisfies a predicate q, the Next method returns the next existing element that satisfies q, or NULL if there is none. Next is made sufficiently general to find the next entry on non-leaf levels of the tree, which will prove useful in Section 4. For search purposes, however, Next will only be invoked on leaf entries.

[1] The appropriate entry may be found by doing a binary search of the entries on the node. Further discussion of intra-node search optimizations appears in Section 6.1.

Algorithm Next(R, q, E)

Input: GiST rooted at R, predicate q, current entry E

Output: next entry in linear order that satisfies q

Sketch: return next entry on the same level of the tree if it satisfies q. Else return NULL.

N1: [next on node] If E is not the rightmost entry on its node, and N is the next entry to the right of E in order, and Consistent(N, q), then return N. If $\neg$Consistent(N, q), return NULL.

N2: [next on neighboring node] If E is the righmost entry on its node, let P be the next node to the right of R on the same level of the tree (this can be found via tree traversal, or via sideways pointers in the tree, when available [LY81].) If P is non-existent, return NULL. Otherwise, let N be the leftmost entry on P. If Consistent(N, q), then return N, else return NULL.

3.4.3 Insert

The insertion routines guarantee that the GiST remains balanced. They are very similar to the insertion routines of R-trees, which generalize the simpler insertion routines for B+-trees. Insertion allows specification of the level at which to insert. This allows subsequent methods to use Insert for reinserting entries from internal nodes of the tree. We will assume that level numbers increase as one ascends the tree, with leaf nodes being at level 0. Thus new entries to the tree are inserted at level $l = 0$.

Algorithm Insert(R, E, l)

Input: GiST rooted at R, entry $E = (p, \texttt{ptr})$, and level l, where p is a predicate such that p holds for all tuples reachable from `ptr`.

Output: new GiST resulting from insert of E at level l.

Sketch: find where E should go, and add it there, splitting if necessary to make room.

I1. [invoke ChooseSubtree to find where E should go] Let $L =$ ChooseSubtree(R, E, l)

I2. If there is room for E on L, install E on L (in order according to Compare, if IsOrdered.) Otherwise invoke Split(R, L, E).

I3. [propagate changes upward] AdjustKeys(R, L).

ChooseSubtree can be used to find the best node for insertion at any level of the tree. When the IsOrdered property holds, the Penalty method must be carefully written to assure that ChooseSubtree arrives at the correct leaf node in order. An example of how this can be done is given in Section 4.1.

Algorithm ChooseSubtree(R, E, l)

Input: subtree rooted at R, entry $E = (p, \texttt{ptr})$, level l

Output: node at level l best suited to hold entry with characteristic predicate $E.p$

Sketch: Recursively descend tree minimizing Penalty

CS1. If R is at level l, return R;

CS2. Else among all entries $F = (q, \texttt{ptr}')$ on R find the one such that Penalty(F, E) is minimal. Return ChooseSubtree$(F.\texttt{ptr}', E, l)$.

The Split algorithm makes use of the user-defined PickSplit method to choose how to split up the elements of a node, including the new tuple to be inserted into the tree. Once the elements are split up into two groups, Split generates a new node for one of the groups, inserts it into the tree, and updates keys above the new node.

Algorithm Split(R, N, E)

Input: GiST R with node N, and a new entry $E = (p, \texttt{ptr})$.

Output: the GiST with N split in two and E inserted.

Sketch: split keys of N along with E into two groups according to PickSplit. Put one group onto a new node, and Insert the new node into the parent of N.

SP1: Invoke PickSplit on the union of the elements of N and $\{E\}$, put one of the two partitions on node N, and put the remaining partition on a new node N'.

SP2: [Insert entry for N' in parent] Let $E_{N'} = (q, \texttt{ptr}')$, where q is the Union of all entries on N', and $\texttt{ptr}'$ is a pointer to N'. If there is room for $E_{N'}$ on Parent(N), install $E_{N'}$ on Parent(N) (in order if IsOrdered.) Otherwise invoke Split$(R, \text{Parent}(N), E_{N'})$[2].

SP3: Modify the entry F which points to N, so that $F.p$ is the Union of all entries on N.

Step SP3 of Split modifies the parent node to reflect the changes in N. These changes are propagated upwards through the rest of the tree by step I3 of the Insert algorithm, which also propagates the changes due to the insertion of N'.

The AdjustKeys algorithm ensures that keys above a set of predicates hold for the tuples below, and are appropriately specific.

[2] We intentionally do not specify what technique is used to find the Parent of a node, since this implementation interacts with issues related to concurrency control, which are discussed in Section 6. Depending on techniques used, the Parent may be found via a pointer, a stack, or via re-traversal of the tree.

Algorithm AdjustKeys(R, N)

Input: GiST rooted at R, tree node N

Output: the GiST with ancestors of N containing correct and specific keys

Sketch: ascend parents from N in the tree, making the predicates be accurate characterizations of the subtrees. Stop after root, or when a predicate is found that is already accurate.

PR1: If N is the root, or the entry which points to N has an already-accurate representation of the Union of the entries on N, then return.

PR2: Otherwise, modify the entry E which points to N so that $E.p$ is the Union of all entries on N. Then AdjustKeys(R, Parent(N).)

Note that AdjustKeys typically performs no work when IsOrdered = true, since for such domains predicates on each node typically partition the entire domain into ranges, and thus need no modification on simple insertion or deletion. The AdjustKeys routine detects this in step PR1, which avoids calling AdjustKeys on higher nodes of the tree. For such domains, AdjustKeys may be circumvented entirely if desired.

3.4.4 Delete

The deletion algorithms must maintain the balance of the tree, and attempt to keep keys as specific as possible. The CondenseTree algorithm uses the B+-tree "borrow or coalesce" technique for underflow if there is a linear order. Otherwise it uses the R-tree reinsertion technique.

Algorithm Delete(R, E)

Input: GiST R, leaf entry $E = (p, \texttt{ptr})$

Output: balanced GiST with E removed

Sketch: Remove E from its leaf node. If this causes underflow, adjust tree accordingly. Update predicates in ancestors to keep them as specific as possible.

D1: [Find node containing entry] Invoke Search$(R, E.p)$ and find leaf node L containing E. Stop if E not found.

D2: [Delete entry.] Remove E from L.

D3: [Propagate changes.] Invoke CondenseTree(R, L);

D4: [Shorten tree.] If the root node has only one child after the tree has been adjusted, make the child the new root.

Algorithm CondenseTree(R, L)

Input: GiST R containing leaf node L

Output: GiST with invariant properties maintained

Sketch: If L has fewer than kM elements, either eliminate L and relocate its entries, or borrow entries from elsewhere to put on L. Propagate node elimination upwards as necessary. Adjust all predicates on the path to the root, making them more specific as appropriate.

CT1: [Initialize.] Set $N = L$. Set Q, the set of eliminated nodes, to be empty.

CT2: If N is the root, go to CT6. Otherwise let $P = \text{Parent}(N)$, and let E_N be the entry on P that points to N.

CT3: [Handle under-full node] If N has fewer than kM entries:

CT3.1 [If not IsOrdered, delete N] If not IsOrdered, add the elements of N to set Q, delete E_N from P, and invoke AdjustKeys(R, P).

CT3.2 [If IsOrdered, try to borrow entries, else coalesce with a neighbor] Otherwise let N' be the neighboring node in order. If the number of keys on N and N' combined is $2kM$ or greater, then evenly split the entries on N and N' between the two nodes. Otherwise, place the entries from N onto N', and delete E_N from P. Invoke AdjustKeys(R, N'), and AdjustKeys(R, P).

CT4: [Adjust covering predicate] If E_N was not deleted from P, then AdjustKeys(R, N);

CT5: [Propagate deletes] If E_N was deleted from P, let $N = P$, and goto CT2;

CT6: [Re-insert orphaned entries] If Q is not empty, invoke Insert$(R, E, \text{level}(E))$ for all elements E of Q.

In some implementations, it is considered preferable to leave a node under-full after a delete, in the expectation that it will fill up soon thereafter [JS93]. To support such behavior, step D3 of algorithm Delete could invoke AdjustKeys(R, L), and avoid CondenseTree.

4 The GiST for Three Applications

In this section we briefly describe implementations of key classes used to make the GiST behave like a B+-tree, an R-tree, and an RD-tree, a new R-tree-like index over set-valued data.

4.1 GiSTs Over $\mathbb{Z}$ (B+-trees)

In this example we index integer data. Before compression, each key in this tree is a pair of integers, representing the interval contained below the key. Particularly, a key $<a, b>$ represents the predicate Contains$([a, b), v)$ with variable v. The query predicates we support in this key class are Contains(interval, v), and Equal(number, v). The interval in the Contains query may be closed or open at either end. The boundary of any interval of integers can be trivially converted to be closed or open. So without loss of generality, we assume below that all intervals are closed on the left and open on the right.

The implementations of the Contains and Equal query predicates are as follows:

- **Contains**$([x, y), v)$ If $x \leq v < y$, return true. Otherwise return false.

- **Equal**(x, v) If $x = v$ return true. Otherwise return false.

Now, the implementations of the GiST methods:

- **Consistent**(E, q) Given entry $E = (p, \texttt{ptr})$ and query predicate q, we know that $p = \text{Contains}([x_p, y_p), v)$, and either $q = \text{Contains}([x_q, y_q), v)$ or $q = \text{Equal}(x_q, v)$. In the first case, return true if $(x_p < y_q) \wedge (y_p > x_q)$ and false otherwise. In the second case, return true if $x_p \leq x_q < y_p$, and false otherwise.

- **Union**$(E_1 = ([x_1, y_1), \texttt{ptr}_1), \ldots, E_n = ([x_n, y_n), \texttt{ptr}_n))$ Return $[\text{MIN}(x_1, \ldots, x_n), \text{MAX}(y_1, \ldots, y_n))$.

- **Compress**$(E = ([x, y), \texttt{ptr}))$ If E is the leftmost key on a non-leaf node, return a 0-byte object. Otherwise return x.

- **Decompress**$(E = (\pi, \texttt{ptr}))$ We must construct an interval $[x, y)$. If E is the leftmost key on a non-leaf node, let $x = -\infty$. Otherwise let $x = \pi$. If E is the rightmost key on a non-leaf node, let $y = \infty$. If E is any other key on a non-leaf node, let y be the value stored in the next key (as found by the Next method.) If E is on a leaf node, let $y = x + 1$. Return $([x, y), \texttt{ptr})$.

- **Penalty**$(E = ([x_1, y_1), \texttt{ptr}_1), F = ([x_2, y_2), \texttt{ptr}_2))$ If E is the leftmost pointer on its node, return $\text{MAX}(y_2 - y_1, 0)$. If E is the rightmost pointer on its node, return $\text{MAX}(x_1 - x_2, 0)$. Otherwise return $\text{MAX}(y_2 - y_1, 0) + \text{MAX}(x_1 - x_2, 0)$.

- **PickSplit**(P) Let the first $\lfloor \frac{|P|}{2} \rfloor$ entries in order go in the left group, and the last $\lceil \frac{|P|}{2} \rceil$ entries go in the right. Note that this guarantees a minimum fill factor of $\frac{M}{2}$.

Finally, the additions for ordered keys:

- **IsOrdered** = true

- **Compare**$(E_1 = (p_1, \texttt{ptr}_1), E_2 = (p_2, \texttt{ptr}_2))$ Given $p_1 = [x_1, y_1)$ and $p_2 = [x_2, y_2)$, return $x_1 - x_2$.

There are a number of interesting features to note in this set of methods. First, the Compress and Decompress methods produce the typical "split keys" found in B+-trees, *i.e.* $n-1$ stored keys for n pointers, with the leftmost and rightmost boundaries on a node left unspecified (*i.e.* $-\infty$ and ∞). Even though GiSTs use key/pointer pairs rather than split keys, this GiST uses no more space for keys than a traditional B+-tree, since it compresses the first pointer on each node to zero bytes. Second, the Penalty method allows the GiST to choose the correct insertion point. Inserting (*i.e.* Unioning) a new key value k into a interval $[x, y)$ will cause the Penalty to be positive only if k is not already contained in the interval. Thus in step CS2, the ChooseSubtree method will place new data in the appropriate spot: any set of keys on a node partitions the entire domain, so in order to minimize the Penalty, ChooseSubtree will choose the one partition in which k is already contained. Finally, observe that one could fairly easily support more complex predicates, including disjunctions of intervals in query predicates, or ranked intervals in key predicates for supporting efficient sampling [WE80].

4.2 GiSTs Over Polygons in $\mathbb{R}^2$ (R-trees)

In this example, our data are 2-dimensional polygons on the Cartesian plane. Before compression, the keys in this tree are 4-tuples of reals, representing the upper-left and lower-right corners of rectilinear bounding rectangles for 2d-polygons. A key $(x_{ul}, y_{ul}, x_{lr}, y_{lr})$ represents the predicate $\text{Contains}((x_{ul}, y_{ul}, x_{lr}, y_{lr}), v)$, where (x_{ul}, y_{ul}) is the upper left corner of the bounding box, (x_{lr}, y_{lr}) is the lower right corner, and v is the free variable. The query predicates we support in this key class are Contains(box, v), Overlap(box, v), and Equal(box, v), where box is a 4-tuple as above.

The implementations of the query predicates are as follows:

- **Contains**$((x^1_{ul}, y^1_{ul}, x^1_{lr}, y^1_{lr}), (x^2_{ul}, y^2_{ul}, x^2_{lr}, y^2_{lr}))$ Return true if
$$(x^1_{lr} \geq x^2_{lr}) \wedge (x^1_{ul} \leq x^2_{ul}) \wedge (y^1_{lr} \leq y^2_{lr}) \wedge (y^1_{ul} \geq y^2_{ul}).$$
Otherwise return false.
- **Overlap**$((x^1_{ul}, y^1_{ul}, x^1_{lr}, y^1_{lr}), (x^2_{ul}, y^2_{ul}, x^2_{lr}, y^2_{lr}))$ Return true if
$$(x^1_{ul} \leq x^2_{lr}) \wedge (x^2_{ul} \leq x^1_{lr}) \wedge (y^1_{lr} \leq y^2_{ul}) \wedge (y^2_{lr} \leq y^1_{ul}).$$
Otherwise return false.
- **Equal**$((x^1_{ul}, y^1_{ul}, x^1_{lr}, y^1_{lr}), (x^2_{ul}, y^2_{ul}, x^2_{lr}, y^2_{lr}))$ Return true if
$$(x^1_{ul} = x^2_{ul}) \wedge (y^1_{ul} = y^2_{ul}) \wedge (x^1_{lr} = x^2_{lr}) \wedge (y^1_{lr} = y^2_{lr}).$$
Otherwise return false.

Now, the GiST method implementations:

- **Consistent**$((E, q)$ Given entry $E = (p, \texttt{ptr})$, we know that $p = \text{Contains}((x^1_{ul}, y^1_{ul}, x^1_{lr}, y^1_{lr}), v)$, and q is either Contains, Overlap or Equal on the argument $(x^2_{ul}, y^2_{ul}, x^2_{lr}, y^2_{lr})$. For any of these queries, return true if $\text{Overlap}((x^1_{ul}, y^1_{ul}, x^1_{lr}, y^1_{lr}), (x^2_{ul}, y^2_{ul}, x^2_{lr}, y^2_{lr}))$, and return false otherwise.
- **Union**$(E_q = ((x^1_{ul}, y^1_{ul}, x^1_{lr}, y^1_{lr}), \texttt{ptr}_1), \ldots, E_n = (x^n_{ul}, y^n_{ul}, x^n_{lr}, y^n_{lr}))$
Return $(\text{MIN}(x^1_{ul}, \ldots, x^n_{ul}),\ \text{MAX}(y^1_{ul}, \ldots, y^n_{ul}),\ \text{MAX}(x^1_{lr}, \ldots, x^n_{lr}),\ \text{MIN}(y^1_{lr}, \ldots, y^n_{lr}))$.
- **Compress**$(E = (p, \texttt{ptr}))$ Form the bounding box of polygon p, *i.e.*, given a polygon stored as a set of line segments $l_i = (x^i_1, y^i_1, x^i_2, y^i_2)$, form $\pi = (\forall_i \text{MIN}(x^i_{ul}),\ \forall_i \text{MAX}(y^i_{ul}),\ \forall_i \text{MAX}(x^i_{lr}),\ \forall_i \text{MIN}(y^i_{lr}))$. Return $(\pi, \texttt{ptr})$.
- **Decompress**$(E = ((x_{ul}, y_{ul}, x_{lr}, y_{lr}), \texttt{ptr}))$ The identity function, *i.e.*, return E.
- **Penalty**(E_1, E_2) Given $E_1 = (p_1, \texttt{ptr}_1)$ and $E_2 = (p_2, \texttt{ptr}_2)$, compute $q = \text{Union}(E_1, E_2)$, and return $\text{area}(q) - \text{area}(E_1.p)$. This metric of "change in area" is the one proposed by Guttman [Gut84].
- **PickSplit**(P) A variety of algorithms have been proposed for R-tree splitting. We thus omit this method implementation from our discussion here, and refer the interested reader to [Gut84] and [BKSS90].

The above implementations, along with the GiST algorithms described in the previous chapters, give behavior identical to that of Guttman's R-tree. A series of variations on R-trees have been proposed, notably the R*-tree [BKSS90] and the R+-tree [SRF87]. The R*-tree differs from the basic R-tree in three ways: in its PickSplit algorithm, which has a variety of small changes, in its ChooseSubtree algorithm, which varies only slightly, and in its policy of reinserting a number of keys during node split. It would not be difficult to implement the R*-tree in the GiST: the R*-tree PickSplit algorithm can be implemented as the PickSplit method of the GiST, the modifications to ChooseSubtree could be introduced with a careful implementation of the Penalty method, and the reinsertion policy of the R*-tree could easily be added into the built-in GiST tree methods (see Section 7.) R+-trees, on the other hand, cannot be mimicked by the GiST. This is because the R+-tree places duplicate copies of data entries in multiple leaf nodes, thus violating the GiST principle of a search tree being a hierarchy of *partitions* of the data.

Again, observe that one could fairly easily support more complex predicates, including n-dimensional analogs of the disjunctive queries and ranked keys mentioned for B+-trees. Other examples include arbitrary variations of the usual overlap or ordering queries, *e.g.* "find all polygons that overlap more than 30% of this box", or "find all polygons that overlap 12 to 1 o'clock", which for a given point p returns all polygons that are in the region bounded by two rays that exit p at angles 90° and 60° in polar coordinates. Note that this infinite region cannot be defined as a polygon made up of line segments, and hence this query cannot be expressed using typical R-tree predicates.

4.3 GiSTs Over $\mathcal{P}(\mathbb{Z})$ (RD-trees)

In the previous two sections we demonstrated that the GiST can provide the functionality of two known data structures: B+-trees and R-trees. In this section, we demonstrate that the GiST can provide support for a new search tree that indexes set-valued data.

The problem of handling set-valued data is attracting increasing attention in the Object-Oriented database community [KG94], and is fairly natural even for traditional relational database applications. For example, one might have a university database with a table of students, and for each student an attribute courses_passed of type setof(integer). One would like to efficiently support containment queries such as "find all students who have passed all the courses in the prerequisite set {101, 121, 150}."

We handle this in the GiST by using sets as containment keys, much as an R-tree uses bounding boxes as containment keys. We call the resulting structure an RD-tree (or "Russian Doll" tree.) The keys in an RD-tree are sets of integers, and the RD-tree derives its name from the fact that as one traverses a branch of the tree, each key contains the key below it in the branch. We proceed to give GiST method implementations for RD-trees.

Before compression, the keys in our RD-trees are sets of integers. A key S represents the predicate Contains(S, v) for set-valued variable v. The query predicates allowed on the RD-tree are Contains(set, v), Overlap(set, v), and Equal(set, v).

The implementation of the query predicates is straightforward:

- **Contains**(S, T) Return true if $S \supseteq T$, and false otherwise.
- **Overlap**(S, T) Return true if $S \cap T \neq \emptyset$, false otherwise.
- **Equal**(S, T) Return true if $S = T$, false otherwise.

Now, the GiST method implementations:

- **Consistent**$(E = (p, \texttt{ptr}), q)$ Given our keys and predicates, we know that $p =$ Contains(S, v), and either $q =$ Contains(T, v), $q =$ Overlap(T, v) or $q =$ Equal(T, v). For all of these, return true if Overlap(S, T), and false otherwise.
- **Union**$(E_1 = (S_1, \texttt{ptr}_1), \ldots, E_n = (S_n, \texttt{ptr}_n))$ Return $S_1 \cup \ldots \cup S_n$.
- **Compress**$(E = (S, \texttt{ptr}))$ A variety of compression techniques for sets are given in [HP94]. We briefly describe one of them here. The elements of S are sorted, and then converted into a set of n disjoint ranges $\{[l_1, h_1], [l_2, h_2], \ldots, [l_n, h_n]\}$ where $l_i \leq h_i$, and $h_i < l_{i+1}$. The conversion is done using the following algorithm:

```
Initialize: consider each element a_m ∈ S to be a range [a_m, a_m].
while (more than n ranges remain) {
      find the pair of adjacent ranges with the least interval between them;
      form a single range of the pair;
}
```

  The resulting structure is called a *rangeset*. It can be shown that this algorithm produces a rangeset of n items with minimal addition of elements not in S [HP94].

- **Decompress**$(E = (\text{rangeset}, \texttt{ptr}))$ Rangesets are easily converted back to sets by enumerating the elements in the ranges.
- **Penalty**$(E_1 = (S_1, \texttt{ptr}_1), E_2 = (S_2, \texttt{ptr}_2)$ Return $|E_1.S_1 \cup E_2.S_2| - |E_1.S_1|$. Alternatively, return the change in a weighted cardinality, where each element of $\mathbb{Z}$ has a weight, and $|S|$ is the sum of the weights of the elements in S.

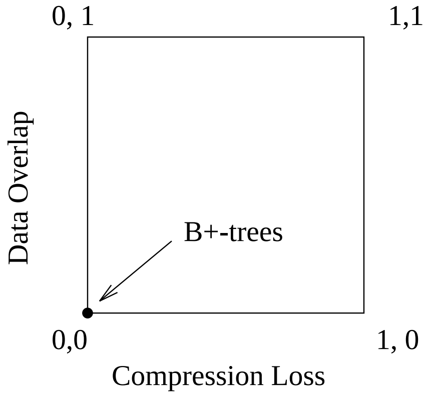

Figure 2: Space of Factors Affecting GiST Performance

- **PickSplit**(P) Guttman's quadratic algorithm for R-tree split works naturally here. The reader is referred to [Gut84] for details.

This GiST supports the usual R-tree query predicates, has containment keys, and uses a traditional R-tree algorithm for PickSplit. As a result, we were able to implement these methods in Illustra's extensible R-trees, and get behavior identical to what the GiST behavior would be. This exercise gave us a sense of the complexity of a GiST class implementation (c. 500 lines of C code), and allowed us to do the performance studies described in the next section. Using R-trees did limit our choices for predicates and for the split and penalty algorithms, which will merit further exploration when we build RD-trees using GiSTs.

5 GiST Performance Issues

In balanced trees such as B+-trees which have non-overlapping keys, the maximum number of nodes to be examined (and hence I/O's) is easy to bound: for a point query over duplicate-free data it is the height of the tree, *i.e.* $O(\log n)$ for a database of n tuples. This upper bound cannot be guaranteed, however, if keys on a node may overlap, as in an R-tree or GiST, since overlapping keys can cause searches in multiple paths in the tree. The performance of a GiST varies directly with the amount that keys on nodes tend to overlap.

There are two major causes of key overlap: data overlap, and information loss due to key compression. The first issue is straightforward: if many data objects overlap significantly, then keys within the tree are likely to overlap as well. For example, any dataset made up entirely of identical items will produce an inefficient index for queries that match the items. Such workloads are simply not amenable to indexing techniques, and should be processed with sequential scans instead.

Loss due to key compression causes problems in a slightly more subtle way: even though two sets of data may not overlap, the keys for these sets may overlap if the Compress/Decompress methods do not produce exact keys. Consider R-trees, for example, where the Compress method produces bounding boxes. If objects are not box-like, then the keys that represent them will be inaccurate, and may indicate overlaps when none are present. In R-trees, the problem of compression loss has been largely ignored, since most spatial data objects (geographic entities, regions of the brain, etc.) tend to be relatively box-shaped.[3] But this need not be the case. For example, consider a 3-d R-tree index over the dataset corresponding to a plate of spaghetti: although no single spaghetto intersects any other in three dimensions, their bounding boxes will likely all intersect!

[3]Better approximations than bounding boxes have been considered for doing spatial joins [BKSS94]. However, this work proposes using bounding boxes in an R*-tree, and only using the more accurate approximations in main memory during post-processing steps.

The two performance issues described above are displayed as a graph in Figure 2. At the origin of this graph are trees with no data overlap and lossless key compression, which have the optimal logarithmic performance described above. Note that B+-trees over duplicate-free data are at the origin of the graph. As one moves towards 1 along either axis, performance can be expected to degrade. In the worst case on the x axis, keys are consistent with any query, and the whole tree must be traversed for any query. In the worst case on the y axis, all the data are identical, and the whole tree must be traversed for any query consistent with the data.

In this section, we present some initial experiments we have done with RD-trees to explore the space of Figure 2. We chose RD-trees for two reasons:

1. We were able to implement the methods in Illustra R-trees.
2. Set data can be "cooked" to have almost arbitrary overlap, as opposed to polygon data which is contiguous within its boundaries, and hence harder to manipulate. For example, it is trivial to construct n distant "hot spots" shared by all sets in an RD-tree, but is geometrically difficult to do the same for polygons in an R-tree. We thus believe that set-valued data is particularly useful for experimenting with overlap.

To validate our intuition about the performance space, we generated 30 datasets, each corresponding to a point in the space of Figure 2. Each dataset contained 10000 set-valued objects. Each object was a regularly spaced set of ranges, much like a comb laid on the number line (*e.g.* $\{[1,10],[100001,100010],$ $[200001,200010],\ldots\}$). The "teeth" of each comb were 10 integers wide, while the spaces between teeth were 99990 integers wide, large enough to accommodate one tooth from every other object in the dataset. The 30 datasets were formed by changing two variables: *numranges*, the number of ranges per set, and *overlap*, the amount that each comb overlapped its predecessor. Varying numranges adjusted the compression loss: our Compress method only allowed for 20 ranges per rangeset, so a comb of $t > 20$ teeth had $t - 20$ of its inter-tooth spaces erroneously included into its compressed representation. The amount of overlap was controlled by the left edge of each comb: for overlap 0, the first comb was started at 1, the second at 11, the third at 21, etc., so that no two combs overlapped. For overlap 2, the first comb was started at 1, the second at 9, the third at 17, etc. The 30 datasets were generated by forming all combinations of numranges in {20, 25, 30, 35, 40}, and overlap in {0, 2, 4, 6, 8, 10}.

For each of the 30 datasets, five queries were performed. Each query searched for objects overlapping a different tooth of the first comb. The query performance was measured in number of I/Os, and the five numbers averaged per dataset. A chart of the performance is shown in Appendix A. More illustrative is the 3-d plot shown in Figure 3, where the x and y axes are the same as in Figure 2, and the z axis represents the average number of I/Os. The landscape is much as we had expected: it slopes upwards as we move away from 0 on either axis.

While our general insights on data overlap and compression loss are verified by this experiment, a number of performance variables remain unexplored. Two issues of concern are *hot spots* and the *correlation factor* across hot spots. Hot spots in RD-trees are integers that appear in many sets. In general, hot spots can be thought of as very specific predicates satisfiable by many tuples in a dataset. The correlation factor for two integers j and k in an RD-tree is the likelihood that if one of j or k appears in a set, then both appear. In general, the correlation factor for two hot spots p, q is the likelihood that if $p \vee q$ holds for a tuple, $p \wedge q$ holds as well. An interesting question is how the GiST behaves as one denormalizes data sets to produce hot spots, and correlations between them. This question, along with similar issues, should prove to be a rich area of future research.

6 Implementation Issues

In previous sections we described the GiST, demonstrated its flexibility, and discussed its performance as an index for secondary storage. A full-fledged database system is more than just a secondary storage manager,

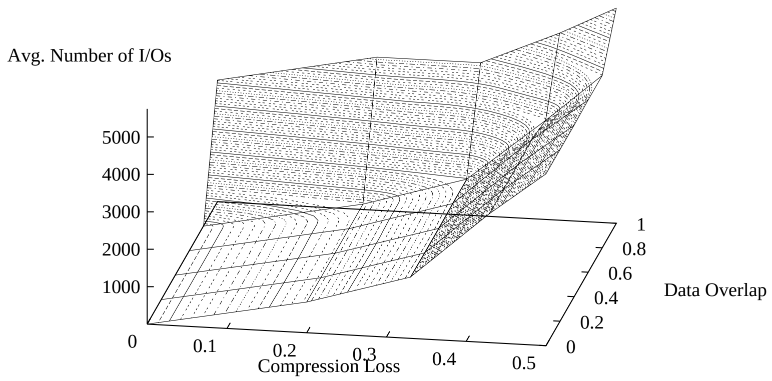

Figure 3: Performance in the Parameter Space

This surface was generated from the data in Appendix A. Compression loss was calculated as (numranges − 20)/numranges, while data overlap was calculated as overlap/10.

however. In this section we briefly address how the GiST can be implemented, taking into account important database system issues.

6.1 In-Memory Efficiency: Streamlining Intra-node Operations

The GiST is an index for secondary storage. As a result, the algorithms described above focus on minimizing node accesses, which are the only operations that involve I/O. However, even secondary storage structures should be implemented with an eye on algorithmic efficiency, and in some situations it is beneficial to streamline in-memory operations on a single node. We facilitate such optimizations in the GiST for situations where they can be exploited, although the default behavior is general-purpose and can be relied upon for correctness in all scenarios.

For the sake of simplicity and generality, the algorithms described above compare input predicates with each entry on a node. More efficient schemes may be used for particular domains. For example, in ordered domains the appropriate entries on a node may be found via binary search. Another example is the hB-tree, in which the entries on a node are themselves indexed via a k-d tree [Ben79], which is used for finding the appropriate entries on the node. Many alternative optimizations may exist, depending on the domain of the key predicates.

To facilitate these techniques, additional extensibility in the GiST may optionally be leveraged for high performance. The Node object in the GiST may be subclassed, and methods for searching, inserting and deleting entries on a node may be specialized. For the sake of brevity, we do not describe the Node interface in detail here, but instead give an illustrative example. The default behavior of the Node object is to store entries in a typical slotted-page format, and do intra-node search in a linear fashion. The behavior of an hB-tree may be achieved instead, by specifying a subclass of the Node object which has a k-d tree member, and overloaded member functions for intra-node insertion, deletion, and search which maintain and exploit the k-d tree. Improved Node methods such as these can streamline the intra-node search operations mentioned in steps S1, S2, FM1, FM2, and CS2. Small overheads may be incurred in steps which insert, delete or modify entries on nodes.

Another in-memory optimization is to avoid the overhead of calling user methods for each operation. This can be done by writing methods as *inline* code, and recompiling the system. This is discussed in more detail in Section 6.6.

6.2 Concurrency Control, Recovery and Consistency

High concurrency, recoverability, and degree-3 consistency are critical factors in a full-fledged database system. Concurrency control for B+-trees is a well-understood problem, with Lehman and Yao's *B-link* variant [LY81] being a typical way of implementing high-concurrency B+-trees. Recovery for B-link trees has been explored by Lomet and Salzberg [LS92], who show that a Π-tree, which generalizes the B-link tree, can be reconstructed gradually from any interim state. Until recently, no analogous results existed for R-trees. The main stumbling block was the fact that Lehman and Yao's techniques were based on linear ordering in the data — sideways pointers were introduced at each level of the tree to connect the nodes of each level into a linked list, ordered by key value. R-tree data has no such natural ordering, and imposing an artificial ordering upsets the balancing techniques for the tree.

Fortunately, recent work extends Lehman and Yao's sideways pointer techniques, along with recovery techniques, to R-trees. Ng and Kameda [NK94] do so by generating a *pending update* list at each node of the tree, and applying Lomet and Salzberg's results on Π-trees to this context. Banks, Kornacker and Stonebraker [BKS94, KB95] have a simpler solution that marks keys and nodes with sequence numbers, and uses the sequence numbers to determine order among the nodes. Both approaches provide solutions for degree-3 consistency — Ng and Kameda via aborting transactions that read phantoms, and Banks, *et al.* via predicate locking.

These techniques can be used in GiSTs in the same way they are used in R-trees. The sequence number approach seems more attractive for the GiST, since it is a simple generalization of the original B-link approach. As a result, the original approach could be used in the case of linearly ordered data, and the more complex sequence number approach for unordered data. A full implementation of concurrency, recoverability, and consistency in the GiST will certainly have to revisit some issues, but the bulk of the problems have already been addressed.

6.3 Variable-Length Keys

It is often convenient to allow keys to vary in length, particularly because of the Compress method available in GiSTs. Unfortunately, variable length keys can cause problems for the GiST (and for R-trees), due to the following possible scenario:

1. A new entry is to be inserted in leaf node L
2. L is full, so a new node L' is generated, and the data on L is split among L and L'
3. During split, the new entry is assigned to L. This results in the key above L being readjusted.
4. The key above L grows in size, and no longer fits on the parent node, so...
5. the entry for the key above L must be removed from the parent node

At this point, we are in the unpleasant situation of having *both* L and L' detached from our tree. The traditional split propagation algorithm only works for one detached node, not for two. Some new technique is required to handle this.

For linearly ordered domains, the situation outlined above cannot occur, since key propagation is not necessary (recall the discussion at the end of Section 3.4.3.) For unordered domains, the problem can indeed occur. We suggest two possible solutions:

- **Reinsertion:** The scenario described can be handled by calling the Insert method to reinsert the two orphaned nodes at their original level. Some care must be taken to ensure that the problem described above does not re-occur on reinsert.

- **New Entry on New Node:** An alternative is to force the new entry to be placed on L' after the split. Then we know that some sufficiently small key can be found that is appropriate for L: in particular, the old key for L is still valid after the split. The old key may be unacceptably general though, and it is advisable to search for a more specific key that is small enough to fit.

The first solution has the advantage of being flexible, and potentially improving the performance of the tree through refined data placement [BKSS90]. It may require some modifications to the concurrency control techniques described in the previous section. The second solution, while simpler, presents the problem of finding a good key that requires sufficiently little storage. This is always possible if one can ensure that as the generality of a key decreases, the storage required for its compressed key does not increase. Although this cannot be guaranteed in general, it does seem natural for some domains, *e.g.* sets.

6.4 Bulk Loading

In unordered domains, it is not clear how to efficiently build an index over a large, pre-existing dataset. To do so, some ordering must be used to sort the data, and then the sorted data must be partitioned into a linked list of nodes, so that a tree may be constructed over the list. A good sort order and partitioning results in a relatively full tree with low overlap at the keys. Various sort orders have been developed for R-trees (*e.g.* [KF93], [Jag90], etc.), but these solutions are specific to the spatial domain and thus not generally applicable for an extensible structure like the GiST. Extending this work to new domains should prove interesting. An extensible BulkLoad method may be added to the GiST to accommodate bulk loading for various domains.

6.5 Optimizer Integration

To integrate GiSTs with a query optimizer, one must let the optimizer know which query predicates match each GiST. This can be done by registering the predicates supported by the Consistent method with the optimizer. For trees with IsOrdered = TRUE, one can additionally specify to the optimizer those predicates that can be evaluated using the FindMin/Next technique.

When planning a query, if the optimizer sees a Boolean Factor [SAC+79] based on one of the registered predicates, then it knows it should consider probing the GiST as one of its potential access paths. The question of estimating the cost of probing a GiST is more difficult, and we defer it until Section 7.

6.6 Coding Details

We propose implementing the GiST in two ways: the Extensible GiST will be designed for easy extensibility, while the Template GiST will be designed for maximal efficiency. With a little care, these two implementations can be built off of the same code base, without replication of logic.

The Extensible GiST package is an object library providing a GiST class, and a GiSTkey class. The latter is a skeleton from which users may inherit the method names described in Section 3.3, to provide their own domain-specific implementations. In C++ terminology, these methods are *virtual member functions*, meaning that each time a method, *e.g.* Compress, is invoked on a key object, the appropriate implementation of Compress is looked up and invoked. Invoking virtual member functions can be inefficient, but this architecture allows for great flexibility: new key classes may be implemented on demand, dynamically linked into a running DBMS, and used to index new data types without halting the DBMS. This is analogous to the extensible indexing scheme used in POSTGRES, and is suitable both for prototyping, and for systems where high availability is important.

The Template GiST package provides a *template* GiST class, which is a C++ source code library. Users develop a key class of their liking, say myGiSTkey, with the appropriate methods. Then they can declare a class GiST<myGiSTkey> in their code. After compiling the code for GiST<myGiSTkey>, they have an index that behaves as desired without the inefficiencies of invoking virtual member functions. Note that myGiSTkey may implement some key methods as *inline* code, particularly for trivial operations such as identity function Compress and Decompress methods. This eliminates the pathlength involved in making a function call, thus masking the complexity of unexploited extensibility features. The Template GiST is in the spirit of "extensible database toolkits" such as EXODUS, and can be used to build custom-designed, efficient systems.

7 Summary and Future Work

The incorporation of new data types into today's database systems requires indexes that support an extensible set of queries. To facilitate this, we isolated the essential nature of search trees, providing a clean characterization of how they are all alike. Using this insight, we developed the Generalized Search Tree, which unifies previously distinct search tree structures. The GiST is extremely extensible, allowing arbitrary data sets to be indexed and efficiently queried in new ways. This flexibility opens the question of when and how one can generate effective search trees.

Since the GiST unifies B+-trees and R-trees into a single structure, it is immediately useful for systems which require the functionality of both. In addition, the extensibility of the GiST also opens up a number of interesting research problems which we intend to pursue:

- *Indexability:* The primary theoretical question raised by the GiST is whether one can find a general characterization of workloads that are amenable to indexing. The GiST provides a means to index arbitrary domains for arbitrary queries, but as yet we lack an "indexability theory" to describe whether or not trying to index a given data set is practical for a given set of queries.

- *Indexing Non-Standard Domains:* As a practical matter, we are interested in building indices for unusual domains, such as sets, terms, images, sequences, graphs, video and sound clips, fingerprints, molecular structures, etc. Pursuit of such applied results should provide an interesting feedback loop with the theoretical explorations described above. Our investigation into RD-trees for set data has already begun: we have implemented RD-trees in SHORE and Illustra, using R-trees rather than the GiST. Once we shift from R-trees to the GiST, we will also be able to experiment with new PickSplit methods and new predicates for sets.

- *Query Optimization and Cost Estimation:* Cost estimates for query optimization need to take into account the costs of searching a GiST. Currently such estimates are reasonably accurate for B+-trees, and less so for R-trees. Recently, some work on R-tree cost estimation has been done [FK94], but more work is required to bring this to bear on GiSTs in general. As an additional problem, the user-defined GiST methods may be time-consuming operations, and their CPU cost should be registered with the optimizer [HS93]. The optimizer must then correctly incorporate the CPU cost of the methods into its estimate of the cost for probing a particular GiST.

- *Lossy Key Compression Techniques:* As new data domains are indexed, it will likely be necessary to find new lossy compression techniques that preserve the properties of a GiST.

- *Algorithmic Improvements:* The GiST algorithms for insertion are based on those of R-trees. As noted in Section 4.2, R*-trees use somewhat modified algorithms, which seem to provide some performance gain for spatial data. In particular, the R*-tree policy of "forced reinsert" during split may be generally beneficial. An investigation of the R*-tree modifications needs to be carried out for non-spatial domains. If the techniques prove beneficial, they will be incorporated into the GiST, either as an option or as default behavior. Additional work will be required to unify the R*-tree modifications with the techniques for concurrency control and recovery.

Finally, we believe that future domain-specific search tree enhancements should take into account the generality issues raised by GiSTs. There is no good reason to develop new, distinct search tree structures if comparable performance can be obtained in a unified framework. The GiST provides such a framework, and we plan to implement it as a C++ library package, so that it can be exploited by a variety of systems.

Acknowledgements

Thanks to Praveen Seshadri, Marcel Kornacker, Mike Olson, Kurt Brown, Jim Gray, and the anonymous reviewers for their helpful input on this paper. Many debts of gratitude are due to the staff of Illustra Information Systems — thanks to Mike Stonebraker and Paula Hawthorn for providing a flexible industrial research environment, and to Mike Olson, Jeff Meredith, Kevin Brown, Michael Ubell, and Wei Hong for their help with technical matters. Thanks also to Shel Finkelstein for his insights on RD-trees. Simon Hellerstein is responsible for the acronym GiST. Ira Singer provided a hardware loan which made this paper possible. Finally, thanks to Adene Sacks, who was a crucial resource throughout the course of this work.

References

[Aok91] P. M. Aoki. Implementation of Extended Indexes in POSTGRES. *SIGIR Forum*, 25(1):2–9, 1991.

[Ben79] J. L. Bentley. Multidimensional Binary Search Trees in Database Applications. *IEEE Transactions on Software Engineering*, SE-5(4):339–353, July 1979.

[BKS94] Douglas Banks, Marcel Kornacker, and Michael Stonebraker. High Concurrency Locking in R-Trees. Technical Report Sequoia 2000 94/56, University of California, Berkeley, June 1994.

[BKSS90] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R*-tree: An Efficient and Robust Access Method For Points and Rectangles. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Atlantic City, May 1990, pages 322–331.

[BKSS94] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. Multi-Step Processing of Spatial Joins. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, May 1994, pages 197–208.

[CDF+94] Michael J. Carey, David J. DeWitt, Michael J. Franklin, Nancy E. Hall, Mark L. McAuliffe, Jeffrey F. Naughton, Daniel T. Schuh, Marvin H. Solomon, C. K. Tan, Odysseas G. Tsatalos, Seth J. White, and Michael J. Zwilling. Shoring Up Persistent Applications. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Minneapolis, May 1994, pages 383–394.

[CDG+90] M.J. Carey, D.J. DeWitt, G. Graefe, D.M. Haight, J.E. Richardson, D.H. Schuh, E.J. Shekita, and S.L. Vandenberg. The EXODUS Extensible DBMS Project: An Overview. In Stan Zdonik and David Maier, editors, *Readings In Object-Oriented Database Systems*. Morgan-Kaufmann Publishers, Inc., 1990.

[Com79] Douglas Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, June 1979.

[FB74] R. A. Finkel and J. L. Bentley. Quad-Trees: A Data Structure For Retrieval On Composite Keys. *ACTA Informatica*, 4(1):1–9, 1974.

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. In *Proc. 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 4–13, Minneapolis, May 1994.

[Gro94] The POSTGRES Group. POSTGRES Reference Manual, Version 4.2. Technical Report M92/85, Electronics Research Laboratory, University of California, Berkeley, April 1994.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure For Spatial Searching. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, June 1984, pages 47–57.

[HP94] Joseph M. Hellerstein and Avi Pfeffer. The RD-Tree: An Index Structure for Sets. Technical Report #1252, University of Wisconsin at Madison, October 1994.

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries With Expensive Predicates. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993, pages 267–276.

[Jag90] H. V. Jagadish. Linear Clustering of Objects With Multiple Attributes. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Atlantic City, May 1990, pages 332–342.

[JS93] T. Johnson and D. Shasha. Inserts and Deletes on B-trees: Why Free-At-Empty is Better Than Merge-At-Half. *Journal of Computer Sciences and Systems*, 47(1):45–76, August 1993.

[KB95] Marcel Kornacker and Douglas Banks. High-Concurrency Locking in R-Trees. Submitted for publication, 1995.

[KF93] Ibrahim Kamel and Christos Faloutsos. On Packing R-Trees. *Second International Conference on Information and Knowledge Management (CIKM)*, November 1993.

[KG94] Won Kim and Jorge Garza. Requirements For a Performance Benchmark For Object-Oriented Systems. In Won Kim, editor, *Modern Database Systems: The Object Model, Interoperability and Beyond.* ACM Press, June 1994.

[KKD89] Won Kim, Kyung-Chang Kim, and Alfred Dale. Indexing Techniques for Object-Oriented Databases. In Won Kim and Fred Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394. ACM Press and Addison-Wesley Publishing Co., 1989.

[Knu73] Donald Ervin Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Co., 1973.

[LJF94] King-Ip Lin, H. V. Jagadish, and Christos Faloutsos. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB Journal*, 3:517–542, October 1994.

[LS90] David B. Lomet and Betty Salzberg. The hB-Tree: A Multiattribute Indexing Method. *ACM Transactions on Database Systems*, 15(4), December 1990.

[LS92] David Lomet and Betty Salzberg. Access Method Concurrency with Recovery. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 351–360, San Diego, June 1992.

[LY81] P. L. Lehman and S. B. Yao. Efficient Locking For Concurrent Operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[MCD94] Maurício R. Mediano, Marco A. Casanova, and Marcelo Dreux. V-Trees — A Storage Method For Long Vector Data. In *Proc. 20th International Conference on Very Large Data Bases*, pages 321–330, Santiago, September 1994.

[NK94] Vincent Ng and Tiko Kameda. The R-Link Tree: A Recoverable Index Structure for Spatial Data. In *Proc. Fifth International Conference on Database and Expert Systems Applications (DEXA '94)*, pages 163–172, Athens, 1994.

[PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an Analysis of Range Query Performance in Spatial Data Structures. In *Proc. 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 214–221, Washington, D. C., May 1993.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan-Kaufmann Publishers, Inc., 1993.

[Rob81] J. T. Robinson. The k-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes. In *Proc. ACM-SIGMOD International Conference on Management of Data*, pages 10–18, Ann Arbor, April/May 1981.

[SAC+79] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, June 1979.

[SRF87] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The R+-Tree: A Dynamic Index For Multi-Dimensional Objects. In *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518, Brighton, September 1987.

[Sto86] Michael Stonebraker. Inclusion of New Types in Relational Database Systems. In *Proceedings of the IEEE Fourth International Conference on Data Engineering*, pages 262–269, Washington, D.C., February 1986.

[Sto93] Michael Stonebraker. The Miró DBMS. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993, page 439.

[VV84] Patrick Valduriez and Yann Viemont. A Multikey Hashing Scheme Using Predicate Trees. In *Proc. ACM-SIGMOD International Conference on Management of Data*, Boston, June 1984, pages 107–114.

[WE77] Kai C. Wong and Murray Edelberg. Interval hierarchies and their application to predicate files. *ACM Transactions on Database Systems*, 2(3):223–232, September 1977.

[WE80] C. K. Wong and M. C. Easton. An Efficient Method for Weighted Sampling Without Replacement. *SIAM Journal on Computing*, 9(1):111–113, February 1980.

A Query Performance Over Comb Data

numranges	overlap	Query 1	Query 2	Query 3	Query 4	Query 5	Average
20	0	7	2	2	2	2	3
20	2	7	2	2	2	2	3
20	4	7	2	2	2	2	3
20	6	7	2	2	2	2	3
20	8	8	2	2	2	2	3.2
20	10	3254	3252	3252	3252	3252	3252.4
25	0	7	4087	6	2	2	820.8
25	2	7	4067	6	2	2	816.8
25	4	7	4063	6	2	2	816
25	6	7	4086	6	2	2	820.6
25	8	8	4084	7	2	2	820.6
25	10	4096	4095	4095	4095	4095	4095.2
30	0	7	4087	4086	6	2	1637.6
30	2	7	4067	4066	6	2	1629.6
30	4	7	4063	4062	6	2	1628
30	6	8	4086	4086	6	2	1637.6
30	8	8	4084	4081	7	2	1636.4
30	10	4096	4095	4095	4095	4095	4095.2
35	0	7	5734	5734	5734	10	3443.8
35	2	7	5730	5729	5729	6	3440.2
35	4	7	5727	5726	5726	6	3438.4
35	6	8	5753	5753	5753	7	3454.8
35	8	10	5533	5534	5534	7	3323
35	10	5002	5002	5002	5002	5002	5002
40	0	8	5734	5734	5734	5734	4588.8
40	2	8	5730	5729	5729	5729	4585
40	4	8	5727	5726	5726	5726	4582.6
40	6	9	5753	5753	5753	5753	4604.2
40	8	10	5756	5754	5754	5754	4605
40	10	5751	5749	5749	5749	5749	5749.4