Generalizing ‘‘Search’’ in Generalized Search Trees

Paul M. Aoki†

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720-1776
aoki@cs.berkeley.edu

Abstract

The *generalized search tree,* or GiST, defines the basic interfaces required to construct a hierarchical access method for database systems. As originally specified, GiSTs only support record selection. In this paper, we show how a small number of additional interfaces enable GiSTs to support a much larger class of search and computation operations. Members of this class, which includes nearest-neighbor and ranked search, user-defined aggregation and index-assisted selectivity estimation, are increasingly common in new database applications. The advantages of implementing these operations in the GiST framework include reduction of user development effort and the ability to use ‘‘industrial strength’’ concurrency and recovery mechanisms provided by expert implementors.

1. Introduction

Access methods are arguably the most difficult user extensions supported by object-relational database management systems. Dozens of database extension modules are available today for commercial database servers. However, none of them ship with access methods that are of the same degree of efficiency, robustness and integration as those provided by the vendors.

The problem is not a lack of access method extension interfaces. The iterator interface (by which the database *invokes* access methods) existed in System R [ASTR76]. Query optimizer interfaces (by which the database *decides* to invoke access methods) were introduced in the early extensible database prototypes (*e.g.*, ADT-INGRES/POSTGRES [STON86] and Starburst [LIND87]). These well-understood interfaces still constitute the commercial state of the art [INFO97b].

The problem is that these functional interfaces do not isolate the primitive operations required to *construct* new access methods. Each access method implementor must write code to pack records into pages, maintain links between pages, read pages into memory and latch them, *etc*. Writing this kind of structural maintenance code for an ‘‘industrial strength’’ access method requires a great deal of familiarity with buffer management, concurrency control and recovery protocols. To make things worse, these protocols are different in every database server.

The generalized search tree, or GiST [HELL95], addresses this problem — in part. Like the previous work in this area, GiST defines a set of interfaces for implementing a search index. However, the GiST interfaces are essentially expressed in terms of the abstract data types being indexed rather than in terms of of pages, records and query processing primitives. Since a GiST implementor need not write any structural maintenance code, they need not understand the server-specific protocols described in the last paragraph.

† Research supported by the National Science Foundation under grant IRI-9400773 and the Army Research Office under grant FD-DAAH04-94-G-0223.

Given that database extension modules tend to be produced by *domain knowledge* experts rather than *database server* experts, we believe that GiST serves the majority of database extenders much better than the previous work.

We say that GiST solves the access method problem "in part" because, as originally specified, GiST does not provide the functionality required by certain advanced applications. For example, database extension modules for multimedia types (images, video, audio, *etc.*) usually include specialized index structures. Unfortunately, these applications need specialized index operations as well. GiSTs only support the relational selection operator, such as, "Find the images containing this shade of purple." However, a typical image database query is, "Find the images *most like* this one." To get this functionality, would-be access method implementors must override one or more of the internal GiST methods. This leaves them with many or all of the pre-GiST implementation issues.

In this paper, we show how to extend the original GiST design to support applications requiring specialized index operations. These applications include:

(1) ranked and nearest neighbor search (spatial and feature vector databases)

(2) index-assisted sampling

(3) index-assisted selectivity estimation

(4) index-assisted statistical computation (*e.g.*, aggregation)

Our goal is not to create low-level interfaces that permit as many optimizations as possible. Instead, we expose simple, high-level interfaces. The idea is to enable (say) a computer vision expert to produce a correct and efficient access method with an interesting search algorithm. In the sections that follow, we describe these interfaces and show how they implement the desired functionality.

The remainder of the paper is organized as follows. Section 2 reviews the original GiST design. Section 3 motivates our changes to that design by giving an extended example of our of our test applications, similarity search. In Section 4, we discuss the interfaces and design details of our extensions. Section 5 applies the new extensions to our test applications, giving specific examples of their use. Section 6 describes related work. We conclude in Section 7 with a discussion of project status and future work.

2. A Review of Generalized Search Trees

In this section, we review the current state of generalized search tree research. This includes the definition of the GiST structure, the callback architecture by which operations are performed on GiSTs, and recent extensions in concurrency control and recovery. The following sections of the paper will assume reasonable familiarity with these design aspects.

2.1. Basic Definitions and Structure

A GiST is a height-balanced, multiway tree. Each tree node contains a number of node entries, $E = \langle p, ptr \rangle$, where p is a predicate that describes the subtree indicated by ptr. The subtrees recursively partition the data records. However, they do not necessarily partition the data space. GiST can therefore model ordered, space-partitioning trees (*e.g.*, B^+-trees [COME79]) as well as unordered, non-space-partitioning trees (*e.g.*, R-trees [GUTT84]).

For consistency with [HELL95], we call each datum stored in p a "predicate."[1] We use the term "key" only when it is part of a standard phrase in database terminology.

In the remainder of this paper, we describe the combination of an abstract data type and any GiST methods associated with that type as a *domain*. Predicates are associated with a particular domain.

2.2. Callback Architecture

The original GiST callback architecture consists of a set of common internal methods (provided by GiST) and a set of type-specific methods (provided by the user). The internal methods correspond to the basic functional interfaces identified in other systems: SEARCH, INSERT and DELETE. The novel aspect of GiST is the way in which the behavior of the internal methods is controlled by the type-specific methods.

- SEARCH is (by default) simply depth-first search. The decision to follow a node entry pointer $E.\,ptr$ is determined by whether or not the node entry predicate $E.\,p$ satisfies the CONSISTENT method with respect to the query. In other words, CONSISTENT takes the place of "key test" routines in conventional database systems. As SEARCH locates CONSISTENT records, they are returned to the user.
- INSERT is controlled in a similar manner. INSERT evaluates the PENALTY method over each entry in the root node and the new entry, E^{new}. It then follows the pointer corresponding to the predicate with the lowest PENALTY relative to E^{new}. Since PENALTY encapsulates the notion of index clustering, this process directs E^{new} to the subtree into which it best "fits." INSERT descends recursively until E^{new} is inserted into a leaf node. INSERT then calls another common method, ADJUSTKEYS, to propagate any needed predicate changes up the tree from the updated leaf.
- DELETE combines different aspects of the other two methods. The records to be deleted are located using CONSISTENT (as in SEARCH), after which any changes to the bounding predicates of the updated nodes must be propagated upward using ADJUSTKEYS (as in INSERT).

GiST defines three additional type-specific methods. Unlike CONSISTENT and PENALTY, these methods do not compare entries or predicates. Instead, they are transformations. The first, UNION, is used to form new predicates out of collections of subpredicates. For example, when ADJUSTKEYS identifies the need to "expand" or "tighten" the predicate corresponding to an updated node, it invokes UNION over the entries in the updated node to form the new parent predicate. The other type-specific methods, COMPRESS and DECOMPRESS, are defined as necessary to optimize the use of space within a node.

2.3. Concurrency Control and Recovery

The GiST concurrency control and recovery protocols [KORN97] do not change the basic GiST framework. The internal concurrency control algorithm is based on rightlinks [LEHM81] and therefore depends on a well-ordered traversal of the tree to detect node modifications. The algorithms in this paper follow the required well-ordering. Therefore, where we use the notation from [HELL95], it should be assumed to be augmented as described in [KORN97].

---

[1] "Index column" might have been clearer, since "predicate" usually implies "Boolean logic predicate" in database systems. However, the most general definition of predicate is "a quality, attribute, or property," which is certainly appropriate in the GiST context.

3. Motivating the GiST Extensions

This section presents a concrete example of one of our test applications, similarity search. A complete example will help us determine the specific features lacking in the original GiST (as described in Section 2). A clear identification of these missing features will make the purpose of the GiST extensions in Section 4 more clear. The first subsection explains these deficiencies in the specific context of similarity search. The second subsection explores the underlying principles.

Recall that we described four test applications in Section 1. These applications will be discussed at length in Section 5. For now, we assert that the important characteristics of these applications can be found in similarity search as well.

3.1. A Similarity Search Tree

Similarity search means retrieval of the record(s) closest to an example (i.e., query) according to some similarity function. Similarity search occurs frequently in feature vector (*e.g.*, multimedia and text) databases as well as spatial databases. When retrieving multiple items, users generally want the results ranked (ordered) by similarity. Similarity search, ranked search and the well-known nearest-neighbor problem are very closely related [GUTI94].

For concreteness, our example will use a specific data structure, the SS-tree [WHIT96a]. The SS-tree is a variant of the clustered file [SALT78] applied to Euclidean space. The tree organizes records into (potentially overlapping) hierarchical clusters, each of which is represented by a centroid point (weighted center of mass) and a bounding sphere. (The SS-tree centers the sphere on the centroid, but this does not give a minimum bounding sphere, so we will not assume that these are dependent predicates here.) Each tree node corresponds to one cluster, and the centroid and bounding radius of each cluster are stored in an entry in the cluster's parent node. The SS-tree insertion algorithm locates the best cluster for a new record by recursively finding the cluster with the closest centroid.

Similarity search in an SS-tree is quite simple.[2] The algorithm traverses the tree top-down, following

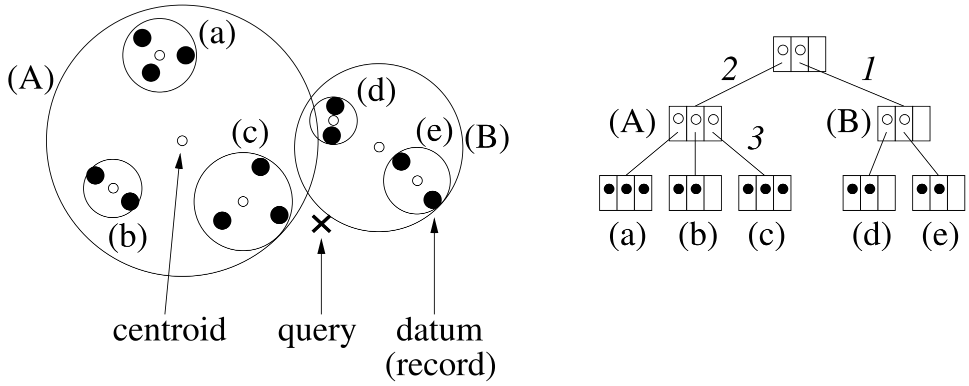

Figure 1. Similarity search using an SS-tree.

[2] The actual SS-tree search algorithm is based on that of [ROUS95]. For clarity, we present the algorithm of [HJAL95] instead.

the pointer whose corresponding bounding sphere is closest to the query. Note that the distance from the query to a node entry's bounding sphere represents the smallest possible distance to a record contained by the subtree represented by that node entry. Therefore, we can stop searching when we find a record that is closer than any unvisited node.

We demonstrate the algorithm using the tree depicted in Figure 1. Our query point is indicated by the **X**. The search begins with the root node, which contains two bounding spheres, one for node (A) and another for node (B). The bounding sphere of node (B) is closest to **X**, so we follow the pointer (tree edge) marked *1*. Examining node (B) gives us the bounding spheres for nodes (d) and (e). Node (A) is closer than either (d) or (e), so we visit node (A) next by following pointer *2*. This, in turn, gives us the bounding spheres for nodes (a), (b) and (c). Node (c) is the closest out of the five unvisited nodes, so we visit node (c) via pointer *3*. Now we have three records. One of the records is closer than any of the four unvisited nodes (as well as its sibling records); the algorithm returns this record.

We can make this algorithm more space-efficient by incrementally pruning branches. As we visit nodes, the bounding spheres of its entries give us *upper* bounds as well as lower bounds on the distance to the nearest neighbor. For example, the bounding sphere of node (d) tells us that we need never visit nodes (a) and (b). This allows us to remember fewer node entries during our search.

3.2. Issues Raised by the SS-tree

The SS-tree search algorithm has three properties that cannot be modelled using GiST. First, it is not depth-first. Instead, it "jumps around" in the tree based on the current minimum node distance. Second, unlike depth-first search, it has search state beyond a simple list of unvisited nodes. This algorithm-specific state includes the closest record found and the tightest bounding distance seen. Third, the algorithm-specific state is used to eliminate nodes from consideration. GiST only prunes subtrees using CONSISTENT.

The SS-tree itself has three structural properties that GiST does not support cleanly. First, the SS-tree has two predicates, a centroid and a bounding sphere. The original GiST can handle only single-predicate node entries. Second, the SS-tree contains non-restrictive predicates. That is, a centroid is a *representative* of the records contained in a subtree rather than a *generalization* of them. Because of this, a centroid can only be used as a search priority hint. GiSTs use predicates as a means of pruning, rather than directing, the search. Third, SS-trees use batched updates. That is, each node accumulates five changes (of arbitrary magnitude) before applying any of them. This is because the insertion of a new record will, in general, change the centroid and radius of every cluster containing it; if predicates are not allowed to diverge from their true values, we must update every node on a leaf-to-root path for every insertion. GiSTs do not support batched updates.

3.3. Generalizing the Issues Raised by the SS-tree

The discussion of the previous subsection reveals several issues that must be addressed to support SS-trees in GiSTs. These issues are shared with other applications. We summarize these issues as follows:

- Search (more specifically, tree traversal) may be *directed* by user-specific criteria. (GiST provides only depth-first search.)
- The returned value may be the result of a *stateful computation* (*i.e.*, one with user-defined state, such as an aggregate function) over some of the entries stored in the index. (GiST can only return leaf index

records.)

- Both the search criteria and stateful computation may be based on *non-restrictive keys* (*i.e.*, metadata, such as cardinality counts and cluster centroids) stored in the index. (Metadata cannot currently be stored in GiSTs.)
- The stateful computation may be *approximate*. (There are no mechanisms in GiST to compensate for ‘‘sloppy’’ predicates, which makes some trees hopelessly inefficient.)

As we will see in the next section, combinations of the following mechanisms allow the user to construct access methods with the characteristics described above.

- multiple-key support
- user-directed traversal control
- user-defined computation state
- user-controlled predicate divergence

4. New GiST Interfaces

In this section, we extend the basic GiST mechanisms. First, we show how the mechanisms extend to support entries that contain multiple predicates. Second, we explain how the user can specify traversals other than depth-first search using a simple priority interface. Third, we illustrate how an aggregation-like iterator interface can support additional traversals as well as index-assisted computations more general than record retrieval. Finally, we (more thoroughly) justify the need for divergence control and demonstrate its uses.

	[HELL95] interface	proposed interface
Basic tree operations	CONSISTENT(E, q) UNION$(\{E\})$ PENALTY(E, E^{new}) PICKSPLIT$(\{E\})$	CONSISTENT(E_i, q_i) UNION$(\{E_i\})$ PENALTY PICKSPLIT$(\{E_i\}, bounds)$
Optional tree operations	COMPRESS(E) DECOMPRESS(E)	COMPRESS(E_i) DECOMPRESS(E_i)
Specialized tree operations	COMPARE(E^1, E^2) FINDMIN(R, q) NEXT(R, q, E)	COMPARE$({E^1}_i, {E^2}_i)$ PRIORITY$(\vec{Q}, \vec{E}, N)$
Stateful computation		STATEINIT$(\vec{Q})$ STATECONSISTENT$(h_i, \{S\})$ STATEITER$(h_i, S.\mathrm{E})$ STATEFINAL$(h_{i)}$
Divergence		ACCURATE$(E_i, {E^{new}}_i, N)$

Table 1. Summary of GiST methods.

For convenience of reference, we summarize our changes in Table 1. The table classifies the old and new operations according to their functionality. In addition, it clearly shows which of the operations of [HELL95] have been deleted, added or replaced.

4.1. Multiple Key Support

In order to support computations over metadata (*e.g.*, subtree cardinality counts and cluster centroids), GiSTs must be able to contain multiple keys. This capability has other, more traditional uses as well.[3] Hence, while our main motivation is the storage of metadata, it makes sense to provide a general multikey mechanism. This subsection describes such a mechanism for GiSTs.

If "multiple keys" is defined as meaning "concatenated keys," most of the GiST methods extend trivially. Each index record contains an entry $E = <\vec{P}, ptr>$ that contains a compound predicate $\vec{P}$ and a tree pointer ptr. $\vec{P}$ contains $|P|$ simple predicates, $<p_1, \ldots, p_{|P|}>$. Similarly, a query $\vec{Q}$ contains simple predicates $<q_1, \ldots, q_{|P|}>$.

- We apply the Union, Compress and Decompress methods of each domain to the individual predicates and concatenate the results. For example, for n compound predicates $\vec{P}^1, \ldots, \vec{P}^n$, $\text{Union}(\{\vec{P}^1, \ldots, \vec{P}^n\}) = <\text{Union}(\{p^1{}_1, \ldots, p^n{}_1\}), \ldots, \text{Union}(\{p^1{}_{|P|}, \ldots, p^n{}_{|P|}\})>$.
- We say $\text{Consistent}(\vec{P}, \vec{Q}) = true$ iff $\text{Consistent}(p_i, q_i) = true$ for all $1 \le i \le |P|$.
- Successive Penalty methods become "tie-breakers." For three predicates $\vec{P^0}$, $\vec{P^1}$ and $\vec{P^2}$, we say $\text{Penalty}(\vec{P^1}, \vec{P^0}) < \text{Penalty}(\vec{P^2}, \vec{P^0})$ iff there exists some $1 \le i \le |P|$ such that $\text{Penalty}(p^1{}_j, p^0{}_j) = \text{Penalty}(p^2{}_j, p^0{}_j)$ for all $1 \le j < i$ and $\text{Penalty}(p^1{}_i, p^0{}_i) < \text{Penalty}(p^2{}_i, p^0{}_i)$.
- PickSplit also uses successive domains to break ties. That is, entries that are duplicates according to the first i domains (a set whose size is strictly non-increasing in i) may be redistributed between the nodes in accordance with PickSplit results for successive domains.

PickSplit is the most difficult operation to generalize because we must intuit the desired splitting semantics from other methods. Specifically, in our recursive application of PickSplit, we must be careful in our definition of "duplicate." For unordered domains, PickSplit divides the entries according to some arbitrary basis. Any duplicates (as defined by the first i type-specific Equality methods[4]) can be interchanged. For ordered domains, however, PickSplit uses Compare to linearize the entries. In this case, we can only interchange entries within the single sequence of duplicates (again, as defined by the first i type-specific Equality methods) that spans the page "split point."[5]

---

[3] In general, it is unreasonable to expect the user to define new opaque types for each combination of keys that could be stored in an index. A common complaint from POSTGRES [STON91] users was the need to define composite opaque types in order to achieve the functionality of standard multikey B^+-trees. For example, to build a B^+-tree over two columns of type `int` and `text`, the user had to write C functions (!) implementing a new `int_text` type and then create a functional B^+-tree [LYNC88].

[4] Modern extensible databases such as Informix Universal Server require the implicit or explicit definition of Equality for all opaque types. This is obviously more general and useful than the POSTGRES approach of assuming bitwise equality.

[5] Some domain implementations "order" duplicate entries using RIDs or other system-generated identifiers [GRAY93]. Such behavior obviously breaks the GiST domain abstraction. In these implementations, duplicate entries essentially do not exist; therefore, no redistribution can be performed beyond that domain. This is generally undesirable because it prevents proper clustering on subdomains.

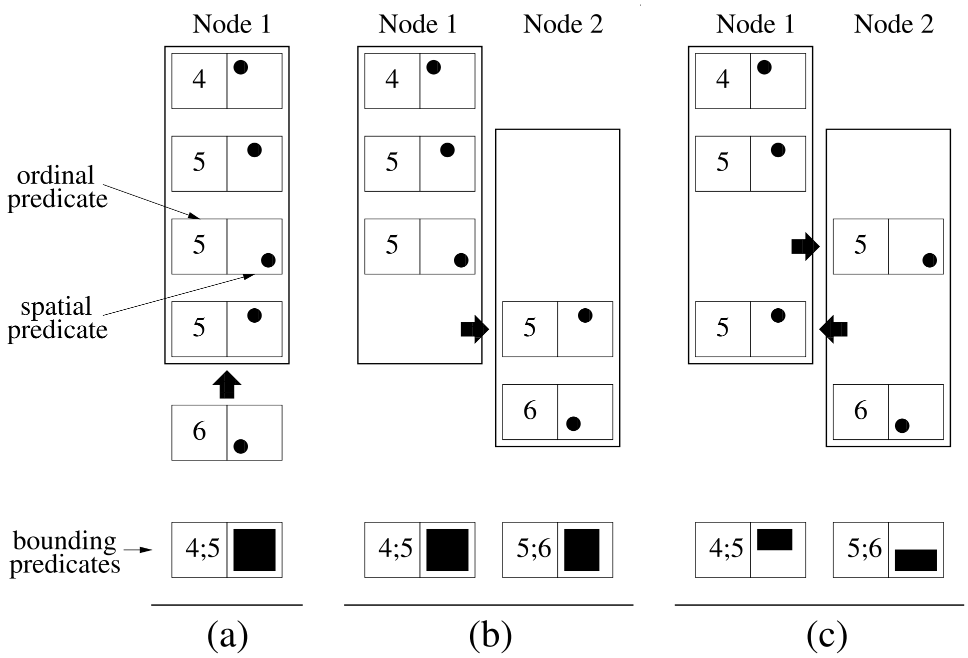

Figure 2. Recursive application of PICKSPLIT.

Figure 2 shows an example of a recursive PICKSPLIT. The entries consist of one ordinal predicate and one spatial predicate. Figure 2(a) shows the initial situation: an insertion is causing a node overflow. Figure 2(b) shows the "50/50" split chosen by the PICKSPLIT method for the first domain. This split is correct and optimal for the first domain but produces large bounding boxes for the second domain. Since the three elements with duplicate ordinal value "5" can be redistributed without affecting the correctness and optimality of the first domain's split, the second domain's PICKSPLIT is allowed to shuffle the "5"s to produce the split shown in Figure 2(c). The first domain's bounding predicate has not improved over Figure 2(b), but the bounding boxes in the second domain have been tightened significantly. Note that duplicate "4"s or "6"s could not be shuffled.

Recursive PICKSPLIT requires a slight generalization of the original definition. In order for PICKSPLIT to shuffle part of a node's entries, we must be able to pass in any occupancy or space bounds that may apply. For example, in Figure 2, we attempt to achieve a 50/50 split by limiting both nodes to no more than three fixed-size entries. Recursive splits must work with progressively smaller bounds.

4.2. Traversal Control

Most search algorithms require some degree of control over the order in which nodes are visited. The standard search tree traversal algorithms are based on stacks. The depth-first search (DFS) induced by a stack does not consider "closeness" and therefore cannot support efficient similarity search. Here, we provide a mechanism for specifying user-defined search strategies. We also discuss some of the implementation issues involved in providing a general traversal control interface.

4.2.1. Generalizing Search Stacks

To "open up" traversal control to the user, we define a new SEARCH method based on a priority queue rather than a stack. The access method implementor provides a set of PRIORITY methods computed from node entries and the current scan state. When SEARCH visits a node, it adds each entry $E = \langle\vec{P}, ptr\rangle$ to the priority queue, together with a traversal priority vector, $\vec{T}$. SEARCH chooses the next node to visit by removing the item with the highest traversal priority from the priority queue.

The priority queue can contain entries corresponding to leaf records as well as internal nodes. There are many cases where we need delivery of records to be delayed until some invariant can be satisfied over all entries visited thus far (similarity search is one such case). It is therefore useful to have a unified mechanism for controlling the delivery of both types of entries.

The priority queue approach subsumes all techniques in which visit order is computed from local information (*i.e.*, which do not use holistic considerations such as "current median object"). For example, many spatial search algorithms visit nodes in some distance-based order. A statistical access method might choose to visit nodes that provide the greatest increase in precision. Finally, we can simulate stacks using a Priority method that returns either a constant value (if we have a priority queue implementation with a stable sort order) or a decreasing counter (if not).

4.2.2. Implementation Issues

The generality of priority queues comes at a price. Three problems are immediately evident. First, stack-based SEARCH implementations need not store full entries or priorities.[6] An unoptimized priority queue implementation therefore consumes more memory than a simple stack. Second, stacks have $O(1)$ insertion/deletion cost for k entries, whereas priority queues have $O(\log k)$ insertion/deletion cost.[7] Third, the number of stack items required to retrieve a record from a leaf node of an n-record search tree is $O(\log n)$, whereas the priority queue requires worst-case $O(n)$ space.[8]

These three problems can be addressed to varying degrees. We can solve the first problem, larger entries, using compression (*e.g.*, supporting NULL predicates and priorities). We can ameliorate the second problem, asymptotic efficiency, with some engineering. Optimizing for the common case, we can implement the priority queue as a "staque" (*i.e.*, by placing a stack on top of the priority queue which contains all objects with the current maximum priority). The third problem results inherently from the fact that the priority queue is more flexible; however, note that for traditional traversals, the behavior will be $O(\log n)$ as usual.

---

[6] The storage of $\vec{P}$ is as yet unmotivated. We will see that storing the predicates in the priority queue will allow us to perform several useful tasks, such as pruning node entries as they are removed from the priority queue.

[7] Some priority queue implementations achieve better — even $O(1)$ — amortized costs. However, these amortized costs are not guaranteed for all workloads and are often not achieved in practice [CORM90].

[8] To be fair, trees with rightlinks also require $O(n)$ stack space in the worst case; this occurs when a scan "chases" a predicate value all the way across a tree level due to concurrent splitting.

4.3. Stateful Computation

In addition to the ability to direct our tree traversal, we also need control over the current state of our traversal (or computation). In this subsection, we describe our interface for maintaining this incremental state. We then give an illustrative example of an application of this interface.

It may be helpful for the user to think of stateful computations as aggregate functions. However, our stateful computations may have side effects as well as having ongoing state that persists between invocations. For example, one stateful computation is the standard aggregate function, COUNT. Other computations actually influence the tree traversal, pruning entries (as in Section 2) or even halting the traversal of new pointers entirely. The latter ability (to stop traversal without halting the delivery of records) makes them unlike iterator-based relational operators (see [GRAE93]).

4.3.1. The Interface

Our primitive methods, modelled on Illustra's user-defined aggregate interface [ILLU95], can be summarized as follows:

- StateInit — Allocates and initializes any internal state. Returns a pointer to this internal state.
- StateIter — Computes the next stage of the iterative computation, updating the internal state as required. StateIter may halt traversal, *i.e.*, that no further node pointers should be followed.
- StateConsistent — Returns a list of node entries to be inserted into the priority queue. For example, it may prune the list of Consistent entries using the current internal state.
- StateFinal — Computes the final (scalar) result of the iterative computation from the internal state.

These methods are invoked, primarily by Search, on an entry-by-entry basis. StateInit is called when the GiST traversal is opened. Search passes all of the Consistent entries in a node through StateConsistent before inserting them into the priority queue; Search also passes each entry removed from the priority queue through StateConsistent before passing them through StateIter. We invoke StateIter on each entry that passes our consistency filters; in addition, any entries remaining in the priority queue when the scan halts will be passed through both StateConsistent and StateIter. Search calls StateFinal when there are no more entries in the priority queue.

We store each iterator's state in a master state descriptor. This descriptor contains several other pieces of state. These include the traversal priority queue and several flags (*e.g.*, whether traversal has been halted, whether entries will be inserted or have been removed from the priority queue, *etc.*).

Since our stateful computations subsume relational selection, they subsume the standard logic by which Consistent is applied in Search. That is, one can implement the standard Consistent logic as a StateConsistent method that returns only the Consistent entries of a node.

4.3.2. An Example: Implementing Ordered Traversal

The pruning technique of Section 2 is one example of a stateful computation. Here, we provide another. We show how the combination of priority queues and stateful computation eliminates the need for the special ordered traversal methods (FindMin and Next) described in [HELL95].

The ordered traversal methods were added to allow GiST to emulate the "left edge"/"leaf scan" range traversal of standard B$^+$-trees. However, in the new framework, FindMin is simply a combination of depth-first traversal (*i.e.*, a counter-based Priority) and a StateConsistent method that returns only the first (leftmost) Consistent entry from the entries passed into it. When the search reaches the leaf level, StateConsistent returns each leaf's rightlink[9]; traversal stops (*i.e.*, the rightlink is not added to the priority queue) when the rightmost entry in the node is not Consistent.

Note that such "leaf scans" only work for multiple ordered predicates, *i.e.*, for B$^+$-trees. The existence of non-partitioning domains makes it very difficult to prevent the repeated delivery of records. Figure 3 shows a scenario in which three different tree descents (labelled (1), (2) and (3)) deliver seven records to the user when there are actually only four matching records. The problem is that the records marked (a), (b), (c) and (d) have accidentally formed a valid increasing sequence because the spatial domain is not partitioned. Since the leading predicate is not unique, subtree predicates cannot stop a leaf scan from leaving the subtree in which it began. Even the mechanism which detects split nodes for concurrency control purposes [KORN97] (which solves an analogous problem of deciding when to stop traversing rightlinks) cannot prevent the delivery of duplicates in this case.

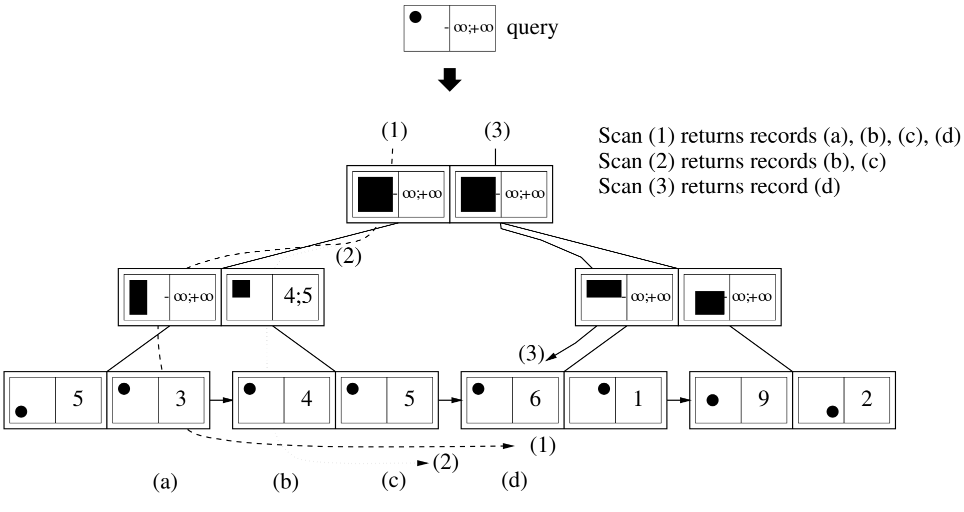

Figure 3. Mixing ordered and unordered domains causes repeated record delivery.

---

[9] Note that the rightlinks of the GiST concurrency control scheme exist only to provide *temporary* paths between nodes that have split. Node deletion may result in an tree level that (by itself) does not form a connected graph. This is different from the B^{link}-tree [LEHM81] and Π-tree [LOME92], which require tree levels to be connected graphs. If leaf scan traversals are required, the GiST implementation must provide extra Delete logic to ensure the continuity of the current level's rightlink chain.

4.4. Divergence Control

It is not immediately clear why divergence between parent and child node predicates should be allowed. After all, looser bounds can only increase the number of ''false positive'' predicates, increasing the number of nodes visited during SEARCH. Certainly, structures such as the B^+-tree and the R-tree do not allow such divergence. Here, we justify the need for divergence control in new applications. We then provide a simple interface for controlling divergence.

In general, incremental predicate updates are expensive. (Recall that we alluded to this in Section 2.) To see this, consider the fact that GiST supports predicates which are the UNION of child predicates. Such predicates always represent a kind of aggregate function (albeit not necessarily a SQL aggregate). Examples include traditional bounding predicates (MIN/MAX), cardinality/ranking (COUNT), frequency moments (SUM), and cluster centroids (AVG). The key difference between the latter three and bounding predicates is that the latter three are guaranteed to be perturbed by any insertion or deletion in a subtree — these predicates are representatives of the underlying data rather than generalization. Hence, all access methods proposed for them are always subject to high update cost; ADJUSTKEYS will modify each node in a leaf-to-root path. This is clearly unacceptable in an online environment, and this cost is the problem addressed by divergence control.[10]

We control divergence using a new GiST method, ACCURATE. ACCURATE assesses two entries according to some criterion specified by the user. ADJUSTKEYS calls ACCURATE when an entry update occurs to a node. If ACCURATE = *false* for the new UNION of the updated node and the parent predicate that formerly described it, ADJUSTKEYS installs the new UNION in place of the old predicate.

Acceptable divergence between the parent predicate and child node UNION predicate may be based on arbitrary criteria. These criteria may be value-based or structural. Some relevant value-based criteria include simple difference for cardinality counts, Euclidean distance for vector-space centroids and partial area difference for spatial bounding boxes. Relevant structural criteria include the node's height in the tree.

We do not expect this facility to be easy to use in general. Fortunately, it adds no work in the common case, since the default ACCURATE is simply the EQUALITY method defined for each opaque type. That is, a parent entry predicate must be equal to the UNION of the predicates in the child node. In addition, it is clearly more general than the alternative. In spite of contrary claims [SRIV88, WHIT96a], batched update is of questionable value in almost any domain because it provides no bounds on the imprecision of the answers provided to the user.

4.5. Summary

Table 1 summarizes our changes to the GiST interface. The type-specific methods have not changed substantially, having been generalized to multiple keys. The new aggregate-like stateful computation interface and the diverge control interface have been added. Finally, the special ordered domain traversal methods are gone, since their functionality has been subsumed by PRIORITY and the stateful computation

[10] It should be noted that the cost of incremental bounding predicate updates can be very high as well. The tree structures in the literature have acceptable update cost only because updates do not ''usually'' change their parent predicates. For example, one can construct an R-tree workload that causes leaf-to-root updates for every new entry, but such is not the common case.

interface.

As previously mentioned, the extensions listed in this section require modifications to SEARCH. Pseudocode for our new SEARCH algorithm is contained in Appendix A.

5. Applications

In this section, we show how to support many common traversal and computation operations in the framework just described. We describe the implementation of each of our test applications in turn. By giving concrete examples, the needs of each application and their solution in our framework should be clear. For each application, we also describe how divergence control can be applied to improve update costs.

5.1. Similarity Search

We have already given an example of similarity search in Section 2. Here, we give a more complete description of the different types of similarity search proposed in the literature and suggest how they can be modelled using our GiST extensions.

Many algorithms have been proposed for similarity search, all based on traversal priority queues. The traversal priority of a node is determined in whole or in part by a lower-bound distance from the query to the node's bounding predicate. The algorithms differ in their formulation of priority, and each priority formulation results in different space complexity and I/O behavior. However, since all of the algorithms have the same $O(n)$ time complexity, each one represents a viable tradeoff. The basic algorithms include:

- *Priority DFS* searches all subtrees of a node in a local proximity order. It has been applied to a wide variety of space-partitioning [FRIE77, SANT89] and non-space-partitioning [BERC96, KATA97, ROUS95, WHIT96a] trees. It is not I/O-optimal and must apply pruning heuristics to reduce the number of unnecessary node visits. However, since it is based on DFS, it has the advantage of requiring only $O(\log n)$ space.
- *Priority bucket* visits buckets (leaf nodes) in proximity order, probing each unvisited subtree in a global proximity order. It is I/O-optimal for space-partitioning trees [ARYA93, HENR94], but the subtree priority queue requires $O(n)$ space. (This is distinct from the $O(\log n)$ space required for each probe.)
- *Priority node* is the search algorithm presented in Section 3. It is a generalization of priority-bucket and visits individual nodes (as opposed to subtrees) in a global proximity order. This algorithm is optimal in terms of both I/O [BERC97, HJAL95, YANG95] and distance calculations [WHIT96b]. Both types of optimality hold for partitioning and non-partitioning trees. The algorithm has the disadvantage of requiring $O(n)$ space for the node priority queue. Furthermore, the algorithm requires a (strictly) greater number of priority queue operations than priority-bucket.

5.1.1. Modelling Similarity Search

For brevity, we will only describe how to model priority-node. The other algorithms are depth-based variations as described in Section 4.2.1. We show how to implementation of the basic traversal algorithm, pruning optimizations, and tree divergence.

The basic traversal algorithm requires only a PRIORITY method that computes a lower bound on the distance to a index entry. Records are also inserted into the traversal priority queue. By prioritizing record entries ahead of internal node entries, we can deliver records to the user when they appear at the top of the priority queue.

As mentioned in Section 3, pruning optimizations are possible for kth nearest neighbor search (where k is known *a priori*). These optimizations are based on the principle that we need never visit a node, or even insert it into the priority queue, if it is more distant than the *upper* bound distance to the kth closest entry that we have seen. (It can be shown that upper bound distances give no advantage in terms of bounding the actual search [BERC97].) These techniques are easily implemented using a STATECONSISTENT method that maintains a separate priority queue, specifically for pruning, of size k.

5.1.2. Application of Divergence Control

Some types of similarity search use cluster centroids rather than bounding predicates. That is, the prioritized search is driven by the distance from the query to the centroid (rather than the minimal distance to the bounding predicate). This kind of heuristic traversal is common in non-Euclidean similarity search. If updates are performed online, we may choose to allow the centroids to diverge from their true values using ACCURATE.

5.2. Sampling

Sampling scans are beginning to appear in commercial database systems. Sampling has many financial and scientific applications [OLKE95]. It can also be used to improve response time for decision support queries that do not need completely accurate answers [INFO97a].

Sampling access methods support some type of randomized probing operation. Augmented trees can be used, as can some variation of acceptance/rejection (A/R) sampling, or some combination. This subsection shows how to emulate both types of sampling.

Sampling is easy using trees augmented with ranks [KNUT73] or other weighting [WONG80] information. To sample from an n-record index, we choose a random number $k \in [1, n]$ and return the kth record by following the pointers whose corresponding ranges contain k (see Figure 4(a)). This is discussed in undergraduate textbooks [CORM90].

A/R sampling is more complex. Conceptually, we choose a random path from root to leaf. As we descend this path, we discover some piece of auxiliary information associated with each node. Based on this auxiliary information, we choose to accept or reject the node (and therefore the path). Our A/R decision criterion is designed to ensure that we return records whose expected distribution is consistent with that of the data set; simply following random pointers in each node does not sample records with equal probability, and sometimes we want records according to some predicate-based distribution. However, rejections may cause us to probe the index several times before returning a record. For example, we can simulate ranked sampling in unranked B^+-trees by assuming a conceptual tree in which each node has the same (*i.e.*, maximum) fanout. If the path to the kth record in our conceptual tree turns out to be impossible because some node does not contain enough entries, we start over with a new path [OLKE89]. Similar algorithms are possible for spatial access methods [OLKE93].

Figure 4(b) is a conceptual example of A/R sampling. Since we have no actual ranks in the tree, we pretend each subtree is full (*i.e.*, each node has maximal fanout). We therefore estimate that the tree contains 27 records. Our first attempt, scan (1), selects record 7. The bogus ranks lead us to a subtree that does not exist, so we reject this path. Our second attempt, scan (2), selects record 16 (which is record 11 in

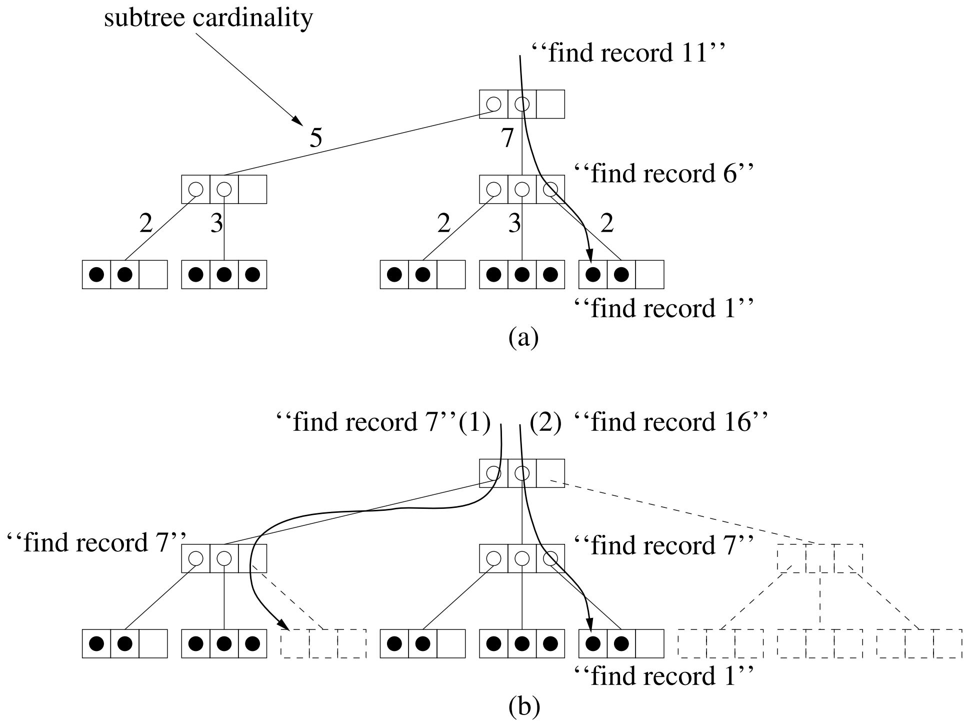

Figure 4. Index-based sampling algorithms.

(a) Sampling from a ranked tree.

(b) Sampling using acceptance/rejection techniques (conceptual diagram).

the physical tree). This time, our path does not follow any nonexistent pointers, so we are able to use the *ersatz* ranking to locate this record.

5.2.1. Modelling Sampling Trees

Ranked trees can be supported trivially in our framework. Cardinality counts (weights) can be maintained as metadata predicates and can be updated automatically by the existing UNION/ADJUSTKEYS mechanisms (*i.e.*, the UNION of multiple counts is just the SUM aggregate). STATEINIT chooses an initial record k; SEARCH simply aggregates the counts/weights in the nodes it visits using STATEITER, following the pointers corresponding to k. Each descent returns one sampled record.

A/R sampling is harder to model. A sampling “scan” is essentially the standard depth-first GiST search, but it does not follow pointers based on CONSISTENT. Instead, as the scan proceeds, STATECONSISTENT uses the metadata in the current node's entries to select exactly zero or exactly one of the entries. If zero entries are selected, the node has been rejected and STATECONSISTENT returns the root (this act restarts the scan). Otherwise, the scan follows the chosen pointer.

5.2.2. Application of Divergence Control

Ranked trees are a good application for divergence control. In fact, Oracle has implemented a specific type of divergent B$^+$-tree in Rdb 7.0 [SMIT96]. In essence, their AdjustKeys algorithm makes the parent predicate slightly larger than the Union of the predicates in the child node. This use of "sloppy" predicates significantly reduces the rate of forced updates. When used for sampling, such "pseudo-ranked" trees [ANTO92] become a combination of ranked sampling and A/R sampling; paths must be rejected when they correspond to subtrees whose existence has been erroneously predicted by the "sloppy" parent predicates.

5.3. Selectivity Estimation

In spite of recent advances in selectivity estimation, "asking the database" by probing an index can still be the only cost-effective way to compute the result size of a restriction query. There are two main reasons for this. The first has to do with database extensions, and the second has to do with deficiencies in existing query optimizer technology.

First, analytic selectivity estimation techniques do not exist for many new, user-defined types and functions. Nearly all of the advances in selectivity estimation have resulted from applications of classic statistics to the traditional ordinal domains. It will take time to develop the same theoretical understanding for user-defined types, a situation that has been described as a "nightmare" for query optimizer implementors [CHAU97]. Extensible database systems typically provide selectivity function interfaces (*e.g.*, the `am_scancost` interface in Informix Universal Server [INFO97b]) but vendors obviously cannot provide domain-specific guidance for implementing these functions.

Second, the non-parametric statistics used by commercial query optimizers have well-known vulnerabilities. For example, the result size estimate for a range query may have a very high degree of relative imprecision, even if its absolute imprecision is bounded using frequent value statistics and small histogram bins. The difference between returning 5 records and 50 records from a table can matter a great deal when it is the driving table for a large join query. Practitioners have recognized this problem and have shown that index-assisted approaches are a viable solution; Digital Rdb/VMS [ANTO93] (now Oracle Rdb) and IBM SQL/400 [ANDE88] (now IBM DB2/400) have long supplemented the standard selectivity estimation techniques by probing unranked search trees.

To be cost-effective, index-assisted selectivity estimation should perform significantly less work than actually answering the query. If our goal is fast convergence using the fewest I/Os, three heuristics immediately suggest themselves. First, we should generally visit nodes at higher levels before nodes at lower levels because of their larger subtree cardinalities. Second, we should descend subtrees with higher imprecision (partial predicate matches, loose divergence bounds) before those with lower imprecision (complete predicate matches, tight divergence bounds). Finally, adding auxiliary information to each node may produce better estimates. For example, guessing the number of records in a subtree using fanout estimates will be much less accurate than obtaining the actual cardinality from ranking information.

5.3.1. Modelling Selectivity Estimation Trees

The option implemented by Rdb/VMS and SQL/400, selectivity estimation using unranked trees, requires very little beyond the ability to traverse the index using Consistent. Both systems descend the tree

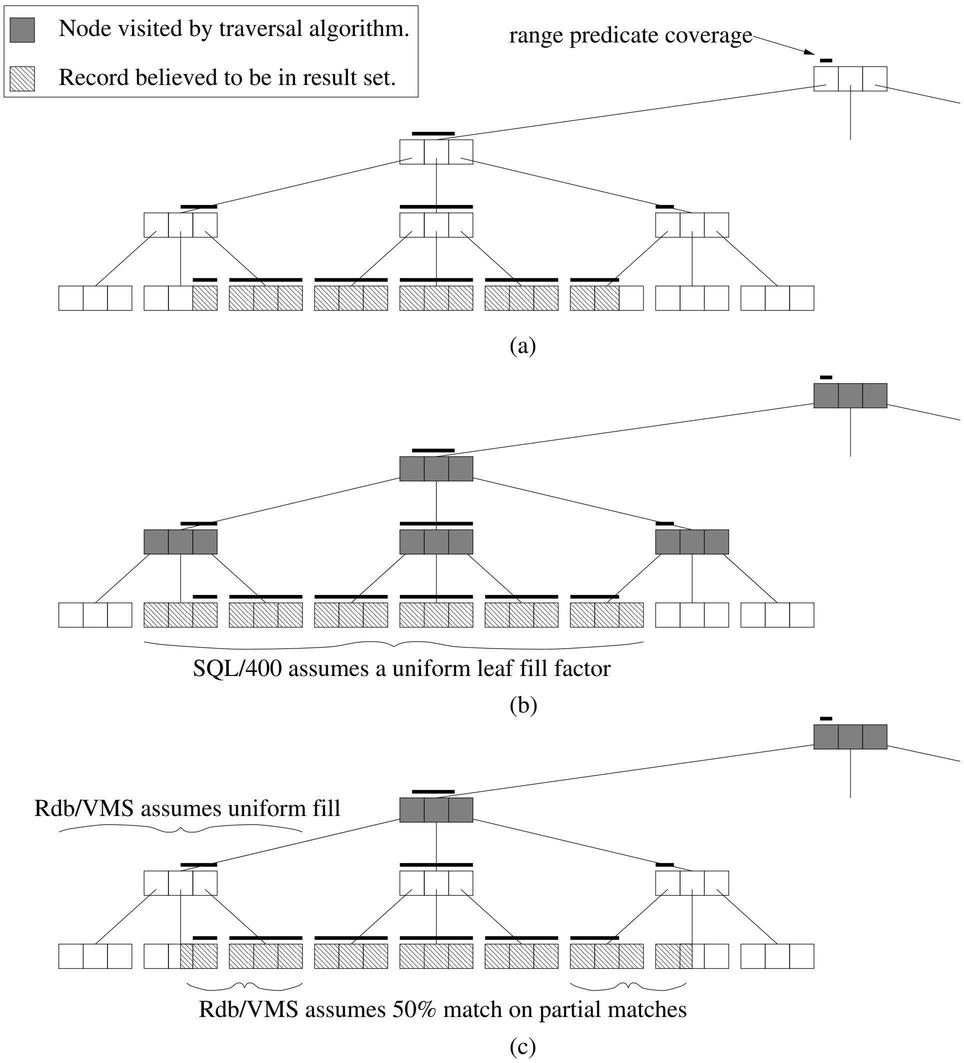

Figure 5. B^+-tree descent strategies for selectivity estimation:

(a) The actual query predicate coverage.

(b) Descent to level above leaves in SQL/400 [ANDE88].

(c) Descent to ‘‘split level’’ in Rdb/VMS [ANTO93].

part-way, using simple uniformity models and fanout estimates to ‘‘guess’’ the tree structure below that point. Figure 5 gives an example of how these approaches work in a B^+-tree.[11]

---

[11] SQL/400 actually uses AS/400 radix trees, which are not height-balanced. The discussion here therefore takes some liberties with the ideas of [ANDE88]. For example, we ignore SQL/400's pilot probes, which only serve to estimate the radix tree height.

Emulation of the SQL/400 approach turns out to be very simple. STATECONSISTENT stops returning entries for a given subtree when the scan is one level above the leaves (Figure 5(b)). When there are no more non-leaf entries to be visited, the scan halts and STATEITER multiplies the number of leaf node entries accumulated in the priority queue by the mean leaf fanout (occupancy).

Emulation of Rdb/VMS uses a similar approach. STATECONSISTENT stops descent when more than one entry in the current node is CONSISTENT (Figure 5(c)). STATEITER combines the current tree level, the mean fanout and the number of CONSISTENT entries in the terminal node (with partially matching entries counting as ½) to calculate $estimate = fanout^{level} \times entries$.

Obviously, more sophisticated estimation and traversal algorithms are possible. For example, one can use Selinger-style uniformity models [SELI79] in any domain in which we can sensibly measure degree of overlap. For example, this is straightforward in multidimensional domains [WHAN94]. Again, this simply requires replacing STATEITER.

5.3.2. Application of Divergence Control

Ranked selectivity estimation trees can again be "pseudo-ranked" (as with sampling). Figure 5 happens to show an example where the unranked tree schemes work relatively well (*i.e.*, the actual number of records is 15, and both schemes give an estimate of 18). However, Variable-length keys cause large fanout variance. This variance greatly reduces the estimation accuracy of unranked trees. Using ranked trees to estimate subtree cardinality reduces the severity of this problem. As previously discussed for sampling trees, the standard GiST UNION/ADJUSTKEYS logic can maintain subtree cardinality counts automatically. Each count is actually a range of possible values rather than a single value (as is normally the case with ranked trees), and the ACCURATE method and the STATEITER (estimation) method must account for this. Divergence leads to some estimation inaccuracy, but at least the inaccuracy has tight bounds.

For completeness, we discuss emulation of the pseudo-ranked estimation in Oracle Rdb. Oracle Rdb uses the same traversal and partial-match logic as Rdb/VMS, so only STATEITER changes (to add the counts for each CONSISTENT entry instead of multiplying). The traversal picture looks the same as Figure 5(c).

5.4. Aggregation Using Statistical Access Methods

Statistical access methods such as SIAM [GHOS86] and TBSAM [SRIV88] are a generalization of ranked trees. Ranked trees store simple order statistics (*i.e.*, subtree counts), whereas statistical access methods can store more complex statistics (*e.g.*, qth frequency moments).

A statistical query is the computation of a scalar aggregate function over all records CONSISTENT with the restriction predicate(s). A statistical access method improves the efficiency of such queries by storing preaggregated metadata in the root of each subtree. (This is different from other index-based approaches, which are typically designed to accelerate restriction and grouping [O'NE97].) Consider the following query, which computes the variance of household incomes within a circular region:

```
SELECT VARIANCE(income)
FROM   households
WHERE  Contains(Circle('...'), location)
```

One approach is to use a primary R-tree to locate the appropriate households and then compute VARIANCE over the matching records. Alternatively, we can maintain Σ *income*, Σ ($income^2$) and COUNT as "keys" in each subtree root; variance is easily computed using the individual sums of these quantities over all subtrees that fully match the restriction predicate. The performance of the two alternatives can differ wildly. For example, if the query happens to cover the entire domain space, the first approach scans the entire index and the alternative approach visits one index node.

5.4.1. Modelling Statistical Access Methods

Statistical access methods can be emulated using the following callbacks. The traversal algorithm aggregates all entries for which CONSISTENT($\vec{P}, \vec{Q}$) = *true* and UNION($\vec{Q}, \vec{P}$) = $\vec{P}$ (*i.e.*, full matches) and follows all pointers for which CONSISTENT($\vec{P}, \vec{Q}$) = *true* but UNION($\vec{Q}, \vec{P}$) ≠ $\vec{P}$ (*i.e.*, partial matches). The latter test occurs in STATECONSISTENT and the aggregation occurs in STATEITER and STATEFINAL.

Figure 6 shows how this traversal algorithm works for an index over an ordinal domain (*e.g.*, TBSAM). For such domains, only the ''edges'' of the range query contain partial predicate matches, so at most two edge descents must be made. A more general structure (*e.g.*, a statistical R-tree) can have many ''edge'' predicates because of bounding predicate overlap, resulting in many more descents. Hence, significant work remains in cost tradeoffs for more general trees.

5.4.2. Application of Divergence Control

Statistical applications are not as well-suited for divergence control as selectivity estimation. The inaccuracy of arbitrary frequency moments of arbitrary attributes (*e.g.*, salary) is potentially much higher than that of order statistics (cardinalities). Furthermore, the fact that the answer is being delivered to the user suggests that the inaccuracy of the answer must be much more explicit than in selectivity estimation. In fact, an ideal interface would allow the user to specify the desired accuracy; general accuracy-specification interfaces require more study.

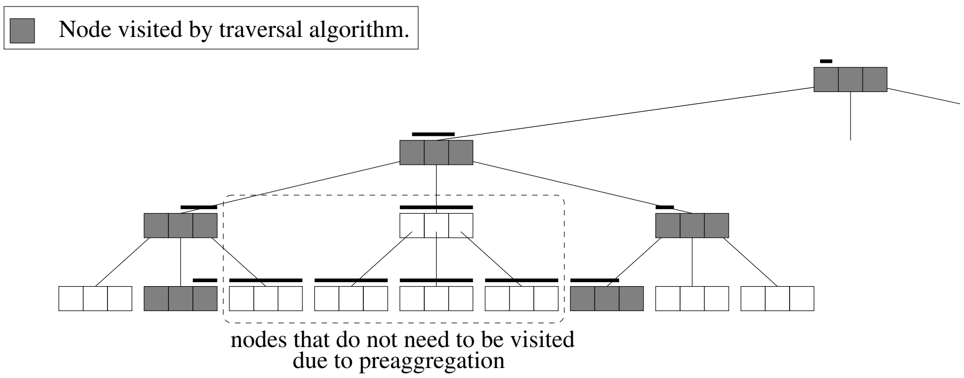

Figure 6. Traversal of a B^{+}-tree-like statistics access method.

6. Related Work

With few exceptions, the work on specialized traversal/computation algorithms, generalized storage structures and semantic divergence control have been orthogonal. Since we have covered the first topic at such length in the preceding sections, we discuss only the latter two here.

Although there have been many extensible database projects, there have been few attempts to capture the primitive structural operations required to implement predicate-based search structures. The GENESIS file management interface [BATO85, BATO88] remains the most comprehensive framework to date. However, GENESIS focuses on ease-of-use for implementors who select from a set of reusable software components. By contrast, GiST [HELL95] focuses on identifying the interfaces that simplify the implementation of the components which are most often *non*-reusable to the greatest possible degree. GiST alone has an integrated concurrency control and recovery scheme capable of addressing all possible search structures built upon it [KORN97]. Neither system appears to have actually implemented search or computation operations more general than record selection.

There have been relatively few discussions of imprecise index structures. Several authors have proposed batched updates (*e.g.*, [SRIV88, WHIT96a]). Antoshenkov's work on pseudo-ranked B^+-trees [ANTO92] is more rigorous but applies only to ordinal domains. To avoid creating new jargon, we have consciously modelled some of our conceptual framework and terminology after that of the *epsilon serializability* (ESR) literature [PU91]. It is possible that additional concepts can be borrowed from that literature as well. However, there are enough considerations unique to divergent GiSTs (*e.g.*, the hierarchical nature of both the data and traversal) that we consider them to be separate problems.

7. Conclusions, Status and Future Work

In this paper, we have shown how two mechanisms, traversal priority callbacks and aggregation-like iterators, enable users them to emulate many of the special-purpose index traversal algorithms proposed in the literature. These traversal mechanisms, combined with multikey support, significantly enhance the ability of GiSTs to support new database applications. A final mechanism, divergence control, enables us to implement these specialized structures as efficient, dynamic indices.

We have given specific details of how these (largely orthogonal) mechanisms support four important applications. We have a limited implementation of our framework in POSTGRESQL 6.1 (`URL:http://postgresql.org/`) and are presently implementing our test applications in this framework.

Future directions include:

Selectivity estimation: We are actively investigating improved techniques for selectivity estimation using GiSTs. Salient issues include:

- Better estimators of the number of records in an index node (*i.e.*, subtree) matching a predicate (*i.e.*, STATEITERS optimized for particular domains). Examples include the use of fractal estimators for the spatial domain [BELU95].
- Balancing I/O cost and estimation accuracy in traversal strategies. The previously proposed descent strategies have many obvious vulnerabilities. Two more attractive options are: descent to a predetermined degree of relative imprecision and descent to a dynamically determined "tailoff" in the reduction

of imprecision. Tailoff detection is particularly natural if we perform a priority queue traversal in which the priority is computed from the imprecision.

- Effects of improved accuracy on optimization (*i.e.*, when is the cost of tree descent warranted?). For example, a query optimizer might only invoke index estimation when the uncertainty of its histogram-based estimate is high. The estimated impact on the rest of the query plan might also be considered.
- Many researchers have pointed out that multidimensional selectivity estimation can benefit from specialized main memory data structures that resemble condensed search trees (*e.g.*, [MURA88]). Such structures could easily be constructed using GiST; the costs and benefits of this approach relative to that of augmenting secondary memory structures (as discussed here) are not well-understood.

In addition, it will be interesting to compare the sensitivity of the more complex estimators to divergence as well as the practicality of creating and maintaining the statistics required to support them (notably, those based on fractals).

Tuning issues: Controlled divergence is a powerful tool for balancing the update rate and inaccuracy of access methods, such as TBSAM and centroid trees, that would otherwise be impractical. However, requiring the user to turn performance ‘‘knobs’’ for each index is not reasonable. We need general parameter guidelines or, better yet, automatic tuning methods.

Further traversal-order generalizations: Marcel Kornacker posed the following interesting question: ‘‘What traversal strategies do not fit in a priority queue model?’’ We have not yet been able to find a reasonable traversal algorithm that cannot be emulated using priorities. When such an algorithm is found, it may not prove to be important enough to warrant a change to the callback interface — the priority queue methods (actually, SEARCH) can always be overridden for that specific GiST.

Acknowledgements

Joe Hellerstein, Marcel Kornacker and Allison Woodruff have provided many comments that have improved the presentation and generality of the concepts in this paper. In particular, Marcel’s skepticism about non-priority-based traversals influenced the design of the traversal control interface.

References

[ANDE88] M. J. Anderson, R. L. Cole, W. S. Davidson, W. D. Lee, P. B. Passe, G. R. Ricard and L. W. Youngren, “Index Key Range Estimator,” U. S. Patent 4,774,657, IBM Corp., Armonk, NY, Sep. 1988. Filed June 6, 1986.

[ANTO92] G. Antoshenkov, “Random Sampling from Pseudo-Ranked B^+ Trees,” *Proc. 18th Int. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, Aug. 1992, 375-382.

[ANTO93] G. Antoshenkov, “Dynamic Query Optimization in Rdb/VMS,” *Proc. 9th IEEE Int. Conf. on Data Eng.*, Vienna, Austria, Apr. 1993, 538-547.

[ARYA93] S. Arya and D. M. Mount, “Algorithms for Fast Vector Quantization,” *Proc. 3rd Data Compression Conf.*, Snowbird, UT, 1993, 381-390.

[ASTR76] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade and V. Watson, “System R: Relational Approach to Database Management,” *Trans. Database Systems 1*, 2 (June 1976), 97-137.

[BATO85] D. S. Batory, "Modeling the Storage Architectures of Commercial Database Systems," *Trans. Database Systems 10*, 4 (Dec. 1985), 463-528.

[BATO88] D. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell and T. E. Wise, "GENESIS: An Extensible Database Management System," *IEEE Trans. on Software Eng. 14*, 11 (1988), 1711-1730.

[BELU95] A. Belussi and C. Faloutsos, "Estimating the Selectivity of Spatial Queries Using the 'Correlation' Fractal Dimension," *Proc. 21st Int. Conf. on Very Large Data Bases*, Zurich, Switzerland, Sep. 1995, 299-310.

[BERC96] S. Berchtold, D. A. Keim and H.-P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," *Proc. 22nd Int. Conf. on Very Large Data Bases*, Mumbai (Bombay), India, Sep. 1996, 28-39.

[BERC97] S. Berchtold, C. Böhm, D. A. Keim and H.-P. Kriegel, "A Cost Model for Nearest Neighbor Search," *Proc. 16th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, Tucson, AZ, May 1997, 78-86.

[CHAU97] S. Chaudhuri, "Query Optimization at the Crossroads (panel session)," *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, Tucson, AZ, May 1997, 509.

[COME79] D. Comer, "The Ubiquitous B-tree," *Computing Surveys 11*, 2 (1979), 122-137.

[CORM90] T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, NY, 1990.

[FRIE77] J. H. Friedman, J. L. Bentley and R. A. Finkel, "An Algorithm for Finding the Best Matches in Logarithmic Expected Time," *Trans. Math. Software 3*, 3 (Sep. 1977), 209-226.

[GHOS86] S. Ghosh, "SIAM: Statistics Information Access Method," in *Proc. 3rd Int. Wksp. on Statistical and Scientific Database Management* (Luxembourg, July 1986), R. Cubitt, B. Cooper and G. Özsoyoglu (ed.), EUROSTAT, Luxembourg, 1986, 286-293.

[GRAE93] G. Graefe, "Query Evaluation Techniques for Large Databases," *Computing Surveys 25*, 2 (June 1993), 73-170.

[GRAY93] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, CA, 1993.

[GUTI94] R. H. Güting, "An Introduction to Spatial Database Systems," *VLDB J. 3*, 4 (Oct. 1994), 357-399.

[GUTT84] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM SIGMOD Int. Conf. on Management of Data*, Boston, MA, June 1984, 47-57.

[HELL95] J. M. Hellerstein, J. F. Naughton and A. Pfeffer, "Generalized Search Trees for Database Systems," *Proc. 21st Int. Conf. on Very Large Data Bases*, Zurich, Switzerland, Sep. 1995, 562-573.

[HENR94] A. Henrich, "A Distance-Scan Algorithm for Spatial Access Structures," *Proc. 2nd ACM Wksp. on Advances in Geographic Information Systems*, Gaithersburg, MD, Dec. 1994, 136-143.

[HJAL95] G. R. Hjaltason and H. Samet, "Ranking in Spatial Databases," in *Advances in Spatial Databases* (Proc. 4th Int. Symp. on Spatial Databases, Portland, ME, Aug. 1995), M. J. Egenhofer and J. R. Herring (ed.), Springer Verlag, LNCS Vol. 951, Berlin, Germany, 1995, 83-95.

[ILLU95] "Illustra User's Guide, Server Release 3.2," Part Number DBMS-00-42-UG, Illustra Information Technologies, Inc., Oakland, CA, Oct. 1995.

[INFO97a] "Informix-OnLine Extended Parallel Server Version 8.1 for the UNIX Operating System," Part Number 000-21384-77, Informix Corp., Menlo Park, CA, Jan. 1997.

[INFO97b] "Guide to the Virtual-Table Interface, Version 9.01," Part Number 000-3692, Informix Corp., Menlo Park, CA, Jan. 1997.

[KATA97] N. Katayama and S. Satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, Tucson, AZ, May 1997, 369-380.

[KNUT73] D. E. Knuth, *The Art of Computer Programming, Volume III: Sorting and Searching*, Addison Wesley, Reading, MA, 1973.

[KORN97] M. Kornacker, C. Mohan and J. M. Hellerstein, "Concurrency and Recovery in Generalized Search Trees," *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, Tucson, AZ, May 1997, 62-72.

[LEHM81] P. L. Lehman and S. B. Yao, "Efficient Locking for Concurrent Operations on B-trees," *Trans. Database Systems 6*, 4 (Dec. 1981), 650-670.

[LIND87] B. Lindsay, J. McPherson and H. Pirahesh, "A Data Management Extension Architecture," *Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data*, San Francisco, CA, May 1987, 220-226.

[LOME92] D. Lomet and B. Salzberg, "Access Method Concurrency with Recovery," *Proc. 1992 ACM SIGMOD Int. Conf. on Management of Data*, San Diego, CA, June 1992, 351-360.

[LYNC88] C. A. Lynch and M. Stonebraker, "Extended User-Defined Indexing with Application to Textual Databases," *Proc. 14th Int. Conf. on Very Large Data Bases*, Los Angeles, CA, Aug. 1988, 306-317.

[MURA88] M. Muralikrishna and D. J. DeWitt, "Equi-depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries," *Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data*, Chicago, IL, June 1988, 28-36.

[O'NE97] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes," *Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data*, Tucson, AZ, May 1997, 38-49.

[OLKE89] F. Olken and D. Rotem, "Random Sampling from B^+ Trees," *Proc. 15th Int. Conf. on Very Large Data Bases*, Amsterdam, Netherlands, Aug. 1989, 269-277.

[OLKE93] F. Olken and D. Rotem, "Sampling from Spatial Databases," *Proc. 9th IEEE Int. Conf. on Data Eng.*, Vienna, Austria, Apr. 1993, 199-208.

[OLKE95] F. Olken and D. Rotem, "Random Sampling from Databases: A Survey," *Statistics and Computing 5*, 1 (Mar. 1995), 25-42.

[PU91] C. Pu and A. Leff, "Replica Control in Distributed Systems: An Asynchronous Approach," *Proc. 1991 ACM SIGMOD Int. Conf. on Management of Data*, Denver, CO, May 1991, 377-386.

[ROUS95] N. Roussopoulos, S. Kelley and F. Vincent, "Nearest Neighbor Queries," *Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data*, San Jose, CA, May 1995, 71-79.

[SALT78] G. Salton and A. Wong, "Generation and Search of Clustered Files," *Trans. Database Systems 3*, 4 (Dec. 1978), 321-346.

[SANT89] O. Santana, G. Rodriguez, M. Diaz and A. Plácido, "The Infinite Distance in the Determination of the Nearest Euclidean M-Neighbours in the K-D-B Tree," *Proc. IEEE Int. Wksp. on Tools for AI*, Fairfax, VA, Oct. 1989, 146-152.

[SELI79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data*, Boston, MA, June 1979, 23-34.

[SMIT96] I. Smith, "Oracle Rdb: What's New," in *DECUS Spring '96* (St. Louis, MO), DECUS, Littleton, MA, June 1996, IM-016. Presentation only.

[SRIV88] J. Srivastava and V. Y. Lum, "A Tree Based Access Method (TBSAM) for Fast Processing of Aggregate Queries," *Proc. 4th IEEE Int. Conf. on Data Eng.*, Los Angeles, CA, Feb. 1988, 504-510.

[STON86] M. R. Stonebraker, “Inclusion of New Types in Relational Data Base Systems,” *Proc. 2nd IEEE Int. Conf. on Data Eng.*, Los Angeles, CA, Feb. 1986, 262-269.

[STON91] M. Stonebraker and G. Kemnitz, “The POSTGRES Next-Generation Database Management System,” *Comm. of the ACM 34*, 10 (Oct. 1991), 78-92.

[WHAN94] K.-Y. Whang, S.-W. Kim and G. Wiederhold, “Dynamic Maintenance of Data Distribution for Selectivity Estimation,” *VLDB J. 3*, 1 (Jan. 1994), 29-51.

[WHIT96a] D. A. White and R. Jain, “Similarity Indexing with the SS-tree,” *Proc. 12th IEEE Int. Conf. on Data Eng.*, New Orleans, LA, Feb. 1996, 516-523.

[WHIT96b] D. A. White and R. Jain, “Algorithms and Strategies for Similarity Retrieval,” Tech. Rep. VCL-96-101, Visual Computing Laboratory, Univ. of California, San Diego, La Jolla, CA, July 1996.

[WONG80] C. K. Wong and M. C. Easton, “An Efficient Method for Weighted Sampling Without Replacement,” *SIAM J. Computing 9*, 1 (Feb. 1980), 111-113.

[YANG95] Q. Yang, A. Vellaikal and S. Dao, “MB$^+$-tree: A New Index Structure for Multimedia Databases,” *Proc. Int. Wksp. on Multi-Media Database Management Systems*, Blue Mountain Lake, NY, Aug. 1995, 151-158.

Appendix A: Priority Search Algorithm

```
global NIters;
global Iters[NIters];    // state iterator metadata

Algorithm InitSearch(Q⃗, RootNode)
  H⃗ = new Handle;

  // initialize the priority queue with the root node
  new.T⃗ = 0;
  new.e.P⃗ = NULL;
  new.e.ptr = RootNode;
  PQInsert(H.PQ, new);

  // initialize the iteration state
  for each i ∈ [1,NIters]
    H.iter[i] = StateInit(Iters[i], Q⃗);
  endfor
  H.status = iterate;

  return H⃗;
end
```

```
Algorithm Search(H⃗, Q⃗, RootNode)
  if (H⃗ = NULL)
    H⃗ = InitSearch(Q⃗, RootNode);
  endif

  if (H.status = done)
    return∅
  endif

  while (H.PQ ≠ ∅)
    // return any records already marked as ready
    cur = PQRemove(H.PQ));
    if (cur.return = true)
      return cur.e;
    endif

    // prune entry after extraction from the queue
    {S} = ∅;
    ListInsert({S}, cur);
    for each i ∈ [1,NIters]
      {S} = StateConsistent(H.iter[i], {S});
    endfor
    if ({S} = ∅)
      continue while;
    endif
    cur = ListRemove({S});

    // process this entry; StateIter may set H.status
    for each i ∈ [1,NIters]
      new.e.P[i] = StateIter(H.iter[i], cur.e);
    endfor
    if (new.e.P⃗ ≠ NULL)
      new.return = true;
      PQInsert(H.PQ, new);
    endif

    // fetch and process the next node
    if (H.status = iterate)
      n = NodeRead(cur.e.ptr);
      {S} = ∅;
      // extract all entries (including rightlinks, if any)
      for each e ∈ n
        if (Consistent(e, Q) = true)
          new.e = e;
          for each i ∈ [1,H.nKeys]
            new.T[i] = Priority(Q⃗, e, n);
          endfor
          ListInsert({S}, new);
        endif
      endfor
      // prune entries before insertion into the queue
      for each i ∈ [1,NIters]
        {S} = StateConsistent(H.iter[i], {S});
      endfor
      for each new ∈ {S}
        PQInsert(H.PQ, new);
      endfor
    endif
  endwhile

  H.status = done;
  return StateFinal(H⃗);
end
```