Generalizing "Search" in Generalized Search Trees

(extended abstract)

Paul M. Aoki[†]
Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, CA 94720-1776

Abstract

The generalized search tree, or GiST, defines a framework of basic interfaces required to construct a hierarchical access method for database systems. As originally specified, GiST only supports record selection. In this paper, we show how a small number of additional interfaces enable GiST to support a much larger class of 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 cost model 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 [INFO97].

The problem is that these interfaces relate to the functions performed by access methods and 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 matters 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 (ADTs) being indexed rather than in terms of pages, records and query processing primitives. Since a GiST implementor need not write any structural maintenance code, *e.g.*, for concurrency control [KORN97], they need not understand the server-specific protocols discussed in the previous paragraph. 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 in [HELL95], GiST does not provide the functionality required by certain advanced applications. For example, database extension modules for multimedia ADTs (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 exact shade of purple." However, a typical image database query is a similarity or nearest-neighbor search, such as, "Find the images *most like* this one." To get this functionality, 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 extended abstract, we show how to extend the original GiST design to support applications requiring

[†] Research supported by the National Science Foundation under grant IRI-9400773 and the Army Research Office under grant FDDAAH04-94-G-0223.

[1] Note that the "toolkit" approach (*e.g.*, [BATO88]) helps only marginally here, since the problem we describe is essentially that of implementing new parts for the toolkit.

specialized index operations. These applications include:

- ranked and nearest neighbor search (spatial and feature vector databases)
- index-assisted sampling
- index-assisted query selectivity estimation
- 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 one of our test applications. In Section 4, we discuss the interfaces and design details of our extensions. Section 5 applies the new extensions to some of our test applications, giving specific examples of its use. (Further discussion of the remaining applications, along with an extensive description of related work for all of the applications, appears in the full paper [AOKI97].) We conclude in Section 6 with a discussion of project status and future directions.

2. A review of GiST

In this section, we review the current state of generalized search tree research. This includes the definition of the GiST structure and the callback architecture by which operations are performed on GiSTs. (In what follows, we use the term ''GiST'' to mean the software framework as well as specific instances; the meaning will be clear from context.) The following sections of the paper will assume reasonable familiarity with these aspects.

2.1. Basic definitions and structure

A GiST is a height-balanced, multiway tree. Each tree node contains a number of entries, $E = \prec p, ptr \succ$, 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]).

Two terms will be used in this paper that require additional explanation. First, for consistency with [HELL95], we call each datum stored in p a ''predicate'' rather than a ''key'' or ''index column.'' Second, we describe the combination of an ADT and any GiST methods associated with that ADT as a *domain*. We use this term in place of ''ADT'' or ''type'' because the same ADT may be used in different ways depending on the operations and semantics supported by the index. An intuitive (albeit non-GiST) example is that an integer ADT may be used to implement both B^+-tree keys or hash table keys (hash values), but the two index structures support different types of search predicates and operations. We therefore say that predicates are associated with a domain rather than a type.

2.2. Callback architecture

The original GiST architecture consists of (1) a set of common *internal* methods provided by GiST and (2) a set of *type-specific* methods provided by the user. The internal methods correspond to the generic functional interfaces specified in other designs: SEARCH, INSERT and DELETE. (An additional internal method, ADJUSTKEYS, serves as a ''helper function'' for INSERT and DELETE.) The basic type-specific methods, which operate on predicates, include CONSISTENT, UNION, PENALTY and PICKSPLIT; the full list appears in Section 4.

The novelty of GiST lies in the manner in which the behavior of the generic internal methods is controlled (customized) by one or more of the type-specific methods. We describe the customization interface for each internal method in turn:

- SEARCH is controlled by the CONSISTENT method, which returns *true* if a node entry predicate $E.p$ matches the query. By default, SEARCH implements a depth-first algorithm in which the decision to follow a given node entry pointer $E.ptr$ is determined by CONSISTENT. CONSISTENT therefore 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 by PENALTY and PICKSPLIT 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 internal method, ADJUSTKEYS, to propagate any needed predicate changes up the tree from the updated leaf. If a node overflows upon insertion, the node entries are divided among the new sibling nodes by PICKSPLIT.
- DELETE is controlled by CONSISTENT. 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 some additional type-specific methods which are used in the construction of new predicates.

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 of an updated node, it invokes UNION over the entries in the updated node to form the new parent predicate. Finally, two optional type-specific methods, COMPRESS and DECOMPRESS, optimize the use of space within a node.

3. Motivating the GiST extensions

This section presents a concrete example of one of our test applications, similarity search. The similarity search example will enable us to determine a comprehensive list of features lacking in the original GiST. The first subsection explains these deficiencies in the specific context of similarity search. The second subsection explores the underlying issues and principles.

3.1. A similarity search tree

Similarity search means retrieval of the record(s) closest to a 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.

For concreteness, our example will use a specific data structure, the SS-tree [WHIT96]. We choose the SS-tree because it is a feature vector access method that cannot be modelled using the original GiST design.

The SS-tree organizes records into (potentially overlapping) hierarchical clusters, each of which is represented by two predicates: a centroid point (weighted center of mass) and a bounding sphere radius.[2] 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.

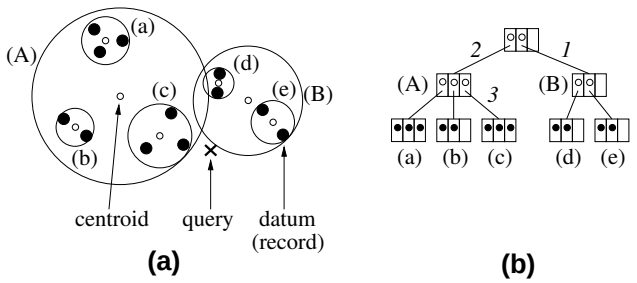

Figure 1. Similarity search using an SS-tree. (a) Spatial coverage diagram. (b) Tree structure diagram.

Similarity search in an SS-tree is quite simple.[3] The algorithm traverses the tree top-down, following the pointer whose corresponding bounding sphere is closest to the query. Note that the spatial distance from the query to a node entry's bounding sphere represents the smallest possible distance to any 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 $\times$ in Figure 1(a). The search begins with the root node, which (as Figure 1(b) shows) contains two bounding spheres, one for node (A) and another for node (B). The bounding sphere of node (B) is closest to the $\times$, 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, its search algorithm is not depth-first. Instead, it "jumps around" in the tree based on the current minimum node distance. Second, unlike GiST's depth-first search, it has search state beyond a simple stack of unvisited nodes. This algorithm-

[2] Even though the SS-tree does center its bounding sphere on the centroid, the bounding sphere need not be the (unique) *minimum* bounding sphere and may be updated independently of the centroid. Also, the centroid is used separately during insertion. (For additional details, see [WHIT96].) Since the SS-tree centroid and radius are often accessed and updated separately, it is more natural to treat them as separate predicates.

[3] The SS-tree search algorithm originally presented in [WHIT96] is based on that of [ROUS95]. We present the algorithm of [HJAL95] here because (1) it is more clear and (2) it has been shown to be I/O-optimal [BERC97].

specific state includes the closest record found and the tightest bounding distance seen. Third, it uses algorithm-specific state 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 radius. The original GiST can handle only single-predicate node entries. Second, the SS-tree contains *non-restrictive* predicates. GiST only uses predicates to restrict, or prune, searches; centroids, on the other hand, can only be used as search hints and are not used to restrict search. Third, SS-trees use batched updates. Specifically, 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. GiST does 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 (as we will show in Section 5). For similarity search, as well as our other sample applications, it turns out that:

- Both the search criteria and stateful computation may be based on what we called ‘‘non-restrictive predicates’’ in Section 3.2 (*i.e.*, metadata, such as cardinality counts and cluster centroids) stored in the index. Metadata cannot currently be stored in GiSTs.
- Search (*i.e.*, tree traversal) may be *directed* by user-specific criteria. GiST provides only depth-first search (the user must rewrite all of SEARCH if an alternative is required).
- The value returned by an index probe may be the result of a *stateful computation* (*i.e.*, one with item-by-item state, such as an aggregate function) over some of the entries stored in the index. GiST can only return leaf index records.
- The stateful computation may be based on approximate values or may be themselves be approximate. There are no mechanisms in GiST to compensate for ‘‘sloppy’’ predicates that can diverge from their expected value, perhaps due to a mechanism like batched updates; however, without this, some access methods become hopelessly inefficient. In order to perform this compensation, we need to be able to *control divergence* between the predicate value and its expected value.

	[HELL95] interface	proposed interface	discussed in section
Basic tree operations	CONSISTENT UNION PENALTY PICKSPLIT	CONSISTENT UNION PENALTY PICKSPLIT	4.1
Optional tree operations	COMPRESS DECOMPRESS	COMPRESS DECOMPRESS	
Specialized traversal operations	FINDMIN NEXT	PRIORITY	4.2
Stateful computation		STATEINIT STATECONSISTENT STATEITER STATEFINAL	4.3
Divergence control		ACCURATE	4.4

Table 1. Summary of GiST methods.

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 predicate support
- user-directed traversal control
- user-defined computation state
- user-specified predicate divergence control

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 that generalize record retrieval. Finally, we (more thoroughly) justify the need for divergence control and demonstrate its uses.

For convenience of reference, we summarize our interface changes in Table 1. The table classifies the old and new operations according to their functionality. In addition, the table clearly shows which of the operations of [HELL95] have been modified. The basic and optional operations, described in Section 2.2, remain largely unchanged. Additional specialized operations having to do with specific tree traversal algorithms have been generalized. Finally, the stateful computation and divergence control operations are entirely new.

4.1. Multiple predicate support

This subsection describes a mechanism for indexing multiple distinct predicates, which the original GiST cannot support. This capability is useful because index

metadata items are often distinct from the predicates that organize the tree. For example, if we store the number of records contained in each subtree of a B$^+$-tree, this quantity is clearly not "part of" the B$^+$-tree key.

Multiple predicates can be combined in semantically and physically different ways. Semantically, we can assign each predicate a degree of precedence/significance (multikey indexing) or not (multidimensional indexing). Physically, we can either force the user to combine the various predicates into a single ADT or not. Historically, database systems have supported multidimensional indexing using combined ADTs and multikey indexing using separate ADTs. We follow that convention here.[4] Therefore, this subsection will be concerned with the analogue of multikey indexing.

For most of the type-specific methods, multikey-style extensions are easy. 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, $<p_1,...,p_{|P|}>$. Similarly, a query $\vec{Q}$ contains simple predicates $<q_1,...,q_{|P|}>$.

- We apply the UNION, COMPRESS and DECOMPRESS methods of each domain to the individual predicates and concatenate the results. For example, we combine n compound predicates $\vec{P}^1,...,\vec{P}^n$ into a new compound predicate using the rule UNION($\{\vec{P}^1,...,\vec{P}^n\}$) = <UNION($\{p^1{}_1,...,p^n{}_1\}$),...,UNION($\{p^1{}_{|P|},...,p^n{}_{|P|}\}$)>.
- CONSISTENT($\vec{P}, \vec{Q}$) = *true* iff CONSISTENT(p_i, q_i) = *true* for all $1 \le i \le |P|$. Put another way, multikey predicates are conjuncts.
- Recall that PENALTY may be thought of as a way of "scoring" the relative similarity of two predicates. In a multikey context, successive PENALTY methods become "tie-breakers" when previous PENALTY methods return equal results. For three predicates $\vec{P^0}$, $\vec{P^1}$ and $\vec{P^2}$, we say PENALTY($\vec{P^1}, \vec{P^0}$) < PENALTY($\vec{P^2}, \vec{P^0}$) iff there exists some $1 \le i \le |P|$ such that PENALTY($p^1{}_j, p^0{}_j$) = PENALTY($p^2{}_j, p^0{}_j$) for all $1 \le j < i$ and PENALTY($p^1{}_i, p^0{}_i$) < PENALTY($p^2{}_i, p^0{}_i$).
- PICKSPLIT, which divides the entries of a split node among the new nodes, must now maintain the correctness of multiple predicates. Like PENALTY, PICKSPLIT uses successive domains to break ties. That is, the set of node entries that are duplicates according to the first i domains (a set whose size is strictly non-increasing in i), $i < |P|$, may be redistributed between the new sibling nodes in accordance with PICKSPLIT results for the remaining domains. We discuss additional issues and difficulties related to splitting and duplicate values in the full paper.

4.2. Traversal control

Most search algorithms require some degree of control over the order in which nodes are visited. As we have seen in Section 3, standard stack-based traversal algorithms do not provide adequate control for some common applications. In this subsection, we describe a mechanism for specifying user-defined search strategies; we also discuss some of the implementation issues involved in providing a general traversal control interface.

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 an ordered set of PRIORITY methods[5] computed from node entries and the current scan state. When SEARCH visits a node, it adds each entry $E = <\vec{P}, ptr>$ to the priority queue, together with a traversal priority vector, $\vec{T}$. $\vec{T}$ contains one priority value for each of the PRIORITY methods. SEARCH chooses the next node to visit by removing the item with the highest traversal priority from the priority queue (where "highest" priority is determined in a manner analogous to that of finding the lowest PENALTY in Section 4.1).

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 kinds of node entries.

The priority queue approach subsumes all techniques in which visit order is computed from local information (*i.e.*, information that can be determined solely by looking at one node entry and the current state). For example, many spatial search algorithms visit nodes in some distance-based order. A statistical access method might

[4] Of course, this convention is largely due to the fact that multikey indexing is a useful practice when using B$^+$-trees with traditional SQL types, which all systems support, whereas multidimensional indexing is much less useful. In general, it is unreasonable to expect the user to define new ADTs for each combination of keys that could be stored in an index. A common complaint from POSTGRES [STON91] users was the need to define combined ADTs in order to achieve the functionality of standard multikey B$^+$-trees. For example, to build a multikey B$^+$-tree over columns of type `int` and `text`, the user had to write C functions implementing a new `int_text` ADT and then create a functional B$^+$-tree [LYNC88].

[5] For example, one can associate separate PRIORITY methods for each predicate if desired. This is cleaner than forcing the user to define a completely new PRIORITY method for every combination of predicates that have been indexed.

choose to visit nodes that provide the greatest increase in precision. Finally, we can easily simulate stacks.

The generality of priority queues comes at a price. Two problems are immediately evident. First, stack-based SEARCH implementations need not store full entries or priorities.[6] Therefore, stacks consume less memory than an unoptimized priority queue implementation. Second, stacks have $O(1)$ insertion/deletion cost for k entries, whereas priority queues have $O(\log k)$ insertion/deletion cost.[7]

These 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).

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 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 (we briefly described in Section 3.1 how node entries can be pruned from the search queue).

Our new methods are modelled on Illustra's user-defined aggregate interface [ILLU95]. Each *iterator* consists of four methods. STATEINIT and STATEFINAL perform initialization and finalization, respectively, whereas STATECONSISTENT and STATEITER implement the computation over the node entries. They can be summarized as follows:

- STATEINIT allocates and initializes any internal state. It is called by SEARCH when the GiST traversal is opened and returns a pointer to the internal state.
- STATECONSISTENT returns a list of node entries to be inserted into the priority queue. For example, it may be used to prune the current node's list of CONSISTENT entries using the internal state. SEARCH passes all of the CONSISTENT entries in a node[8] through STATECONSISTENT before inserting them into the priority queue; in addition, SEARCH passes each entry taken from the priority queue through STATECONSISTENT before passing it to STATEITER.
- STATEITER computes the next stage of the iterative computation, updating the internal state as required. STATEITER may halt traversal, *i.e.*, indicate that no further node pointers should be followed. As previously mentioned, SEARCH invokes STATEITER on each entry removed from the priority queue that is STATECONSISTENT.
- STATEFINAL computes the final (scalar) result of the iterative computation from the internal state. 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 generic 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.*).

In the full paper, 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]. We also show that such ordered "leaf scans" can only work correctly for multiple predicates over ordinal domains, *i.e.*, for multikey B^+-trees.

4.4. Divergence control

In our new framework, a parent node entry predicate and the UNION of its child subtree predicates may diverge. That is, we permit the predicate contained by a parent node entry to be an inaccurate description of the subtree to which it points, whereas the original GiST assumes that they are exact replicas. Divergence control is the mechanism by which we enforce the following constraint: the replicated data items may only differ in ways that permit us to reason about one item given the other. (To avoid creating new jargon, we have modelled our terminology

[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 amortized costs. However, these amortized costs are not guaranteed for all workloads and are often not achieved in practice [CORM90].

[8] The reason for doing all entries at once (rather than individually) has to do with B^+-tree ordered traversal. For example, when descending the left edge of a subtree defined by a range predicate, only one entry pointer from a given node (that corresponding to the lower bound of the predicate range) will be traversed rather than all of them. Checking the consistency of all of a node's entries at once allows us to do this.

after that of the *epsilon serializability* literature [PU91].)

It is not immediately clear why divergence between parent and child node predicates should be allowed. For example, too-large bounding predicates increase the number of "false positive" predicates, thereby increasing the number of nodes visited during SEARCH. 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, one which often (but not always) resembles an 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 far more likely (or even certain, in the case of COUNT) to be perturbed by any insertion or deletion in a subtree — these predicates are representatives of the underlying data rather than generalizations. 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 our divergence control mechanism.[9]

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. In short, ACCURATE specifies what is "accurate enough" for the application.

Acceptable divergence between the parent predicate and child node UNION predicate may be based on arbitrary criteria. These criteria may be either 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.

[9] It should be noted that the cost of bounding predicate updates can be very high for conventional data structures as well. Many access methods 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).

We do not expect this facility, though simple, 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 ADT.[10] That is, a parent entry predicate must be equal to the UNION of the predicates in the child node. The alternative, batched updates [SRIV88, WHIT96], is of questionable value in almost any domain because it provides no bounds on the imprecision of the answers provided to the end-user.

5. Applications

In this section, we show how to support several common traversal and computation operations in the framework just presented. We describe the implementation of two test applications, similarity search and selectivity estimation, in turn. These concrete examples illustrate the needs of each application and their solution in our framework. For each application, we also describe how divergence control can be applied to improve update costs. The other two applications mentioned previously, sampling and statistics access methods, are discussed in the full paper.

5.1. Similarity search

We have already given an example of similarity search in Section 2. In the full paper, we give a more complete description of the different kinds of similarity search proposed in the literature and suggest how they can be modelled using our GiST extensions. Here, we show how to implement the basic traversal algorithm, pruning optimizations, and tree divergence.

Modelling similarity search: The basic traversal algorithm requires only a PRIORITY method that computes a lower bound on the distance to an index entry. Leaf index 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 Sections 3.1 and 4.3, 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 its entry 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 reduce the memory required but give no advantage in terms of bounding the actual search

[10] Extensible systems such as Informix Universal Server typically require the definition of EQUALITY for all ADTs (defaulting to bit-equality).

[BERC97].) These techniques are easily implemented using a StateConsistent method that maintains a separate, additional priority queue, specifically for pruning, of size k.

Application of divergence control: Some kinds 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. 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. First, analytic selectivity estimation techniques may simply not exist for novel user-defined types and functions, and the techniques used for ordinal domains may not extend straightforwardly. Extensible database systems do provide selectivity function interfaces (*e.g.*, the `am_scancost` interface in Informix Universal Server [INFO97]) but can only provide general guidance for their use. Second, the non-parametric statistics used by commercial query optimizers necessarily have vulnerabilities related to their fixed resolution. Practitioners have recognized the latter problem and have shown that index-assisted approaches can be a cost-effective 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 performing index probes.

In this section, we explore the notion of partial tree traversal as a means of estimating how much of the tree matches the query. (Index-assisted sampling is another possibility; we review this technique in the full paper.) The basic idea is that, given both the predicate and the cardinality (number of leaf records) of a subtree, we can estimate the number of records matching the query by computing the overlap between the predicate and the query. This process can be applied recursively, so it gives us statistics of arbitrary resolution.

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

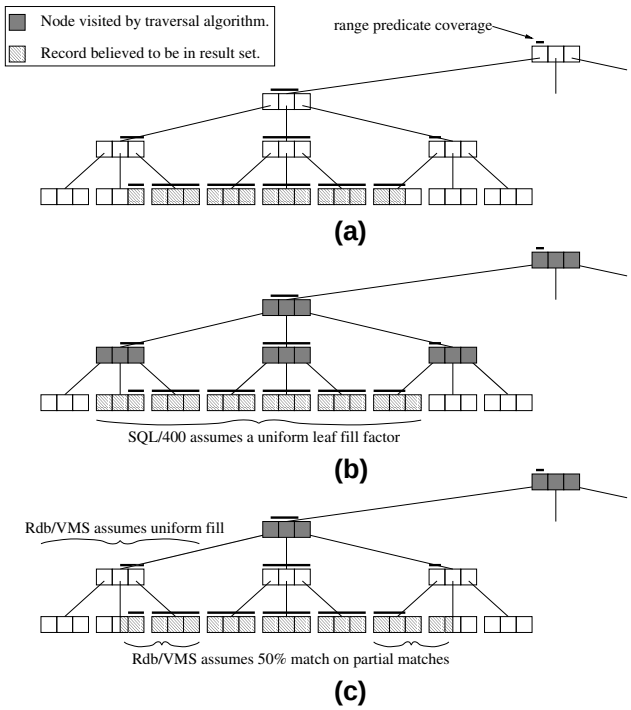

Figure 2. B$^+$-tree descent strategies: (a) The actual query predicate coverage. (b) Descent to level above leaves in SQL/400. (c) Descent to "split level" in Rdb/VMS.

subtrees with higher imprecision (*e.g.*, those with partial predicate matches) before those with lower imprecision (complete predicate matches). Finally, adding auxiliary information to each node may produce better estimates. For example, guessing the number of records in a subtree (*e.g.*, using fanout estimates) will be much less accurate than using the actual subtree cardinality (if we can afford to maintain it).

For historical reasons, trees that maintain subtree cardinality counts are generally known as *ranked* trees. Trees with approximate counts are described as *pseudo-ranked*, and conventional trees that do not maintain extra information are called *unranked*. In the latter case, the cardinality of a subtree is usually estimated by max_fanout^{height} or $mean_fanout^{height}$.

Modelling selectivity estimation trees: We now describe how to emulate the techniques used by Rdb/VMS and SQL/400. Both systems implement selectivity estimation using unranked trees, which requires very little beyond the ability to traverse the index using Consistent. They descend the tree part-way, using simple uniformity models and fanout estimates to "guess" the tree structure below that point. Figure 2 gives an example

of how these approaches work in a B$^+$-tree.[11]

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 2(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 (leaf occupancy).

Emulation of Rdb/VMS uses a similar approach. StateConsistent stops descent the first time more than one entry in the current node is Consistent (Figure 2(c)). The level of this terminal node is called the "split level." 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 the estimate.

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

Application of divergence control: Ranked trees are a good application for divergence control. Figure 2 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, large fanout variance (resulting, *e.g.*, from variable-length keys) is not unusual. This variance greatly reduces the estimation accuracy of unranked trees. Using ranked trees to estimate subtree cardinality reduces the severity of this problem, and the standard GiST Union/AdjustKeys logic can maintain subtree cardinality counts automatically. However, cardinality counts have the leaf-to-root update problem described in Section 4.4, so allowing sloppy counts makes sense. Each count stored in the tree becomes an interval 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 does lead to some estimation inaccuracy, but at least the inaccuracy has tight bounds.

Oracle has implemented pseudo-ranked B$^+$-trees 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 counts significantly reduces the rate of non-leaf-node updates [ANTO92]. 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 using fanout statistics). The traversal picture looks the same as Figure 2(c).

[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.

6. Conclusions, status and future work

In this paper, we have shown how two mechanisms, traversal priority callbacks and aggregation-like iterators, enable users to emulate many of the special-purpose index traversal algorithms proposed in the literature. These traversal mechanisms, combined with multiple predicate 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 details of how these (largely orthogonal) mechanisms support several important applications. We have a limited implementation of our framework in POSTGRESQL 6.1 and are presently implementing our test applications in this framework.

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

- 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 concepts; the costs and benefits of this approach relative to that of augmenting secondary memory structures (as discussed here) are not well-understood.
- 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.

There are many other possible directions, which we discuss in the full paper [AOKI97].

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. The feedback of Dr. Sunita Sarawagi and the anonymous referees is gratefully acknowledged.

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'l 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'l Conf. on Data Eng.*, Vienna, Austria, Apr. 1993, 538-547.

[AOKI97] P.M. Aoki, "Generalizing ''Search'' in Generalized Search Trees," Tech. Rep. UCB//CSD-97-950, Univ. of California, Berkeley, CA, June 1997.

[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," *ACM Trans. Database Sys. 1*, 2 (June 1976), 97-137.

[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. Software Eng. 14*, 11 (Nov. 1988), 1711-1730.

[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.

[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, 1990.

[GUTT84] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM SIGMOD Int'l Conf. on Mgmt. 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'l Conf. on Very Large Data Bases*, Zürich, Switzerland, Sep. 1995, 562-573.

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

[INFO97] "Guide to the Virtual-Table Interface, Version 9.01," Part Number 000-3692, Informix Corp., Menlo Park, CA, Jan. 1997.

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

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

[LYNC88] C.A. Lynch and M. Stonebraker, "Extended User-Defined Indexing with Application to Textual Databases," *Proc. 14th Int'l 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'l Conf. on Mgmt. of Data*, Chicago, IL, June 1988, 28-36.

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

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

[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'l Conf. on Mgmt. 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.

[SRIV88] J. Srivastava and V.Y. Lum, "A Tree Based Access Method (TBSAM) for Fast Processing of Aggregate Queries," *Proc. 4th IEEE Int'l 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'l 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. 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.

[WHIT96] D.A. White and R. Jain, "Similarity Indexing with the SS-tree," *Proc. 12th IEEE Int'l Conf. on Data Eng.*, New Orleans, LA, Feb. 1996, 516-523.