High-Performance Extensible Indexing

Marcel Kornacker*

U. C. Berkeley
marcel@cs.berkeley.edu

Abstract

Today's object-relational DBMSs (ORDBMSs) are designed to support novel application domains by providing an extensible architecture, supplemented by domain-specific database extensions supplied by external vendors. An important aspect of ORDBMSs is support for extensible indexing, which allows the core database server to be extended with external access methods (AMs). This paper describes a new approach to extensible indexing implemented in Informix Dynamic Server with Universal Data Option (IDS/UDO). The approach is is based on the *generalized search tree*, or GiST, which is a template index structure for abstract data types that supports an extensible set of queries. GiST encapsulates core database indexing functionality including search, update, concurrency control and recovery, and thereby relieves the external access method (AM) of the burden of dealing with these issues. The IDS/UDO implementation employs a newly designed GiST API that reduces the number of user defined function calls, which are typically expensive to execute, and at the same time makes GiST a more flexible data structure. Experiments show that GiST-based AM extensibility can offer substantially better performance than built-in AMs when indexing user-defined data types.

1 Introduction

Efficient search tree access methods are crucial for any database system. In traditional relational database management systems, B^+-trees [Com79] suffice for queries posed on the standard SQL data types. Today's extensible object-relational database management systems (ORDBMSs) are being deployed to support new applications such as dynamic web servers, geographic information systems, CAD tools, multimedia and document libraries, sequence databases, fingerprint identification systems, biochemical databases, *etc.* For these applications, new kinds of access methods are required.

Broadly speaking, the research community has responded by developing novel search trees to support each new application. For example, a recent survey article [GG98] describes over 50 alternative index structures for spatial indexing alone. Some of this specialized work has had fundamental impact in particular domains. However, only two or three structures developed since the B^+-tree have enjoyed any significant industrial acceptance.

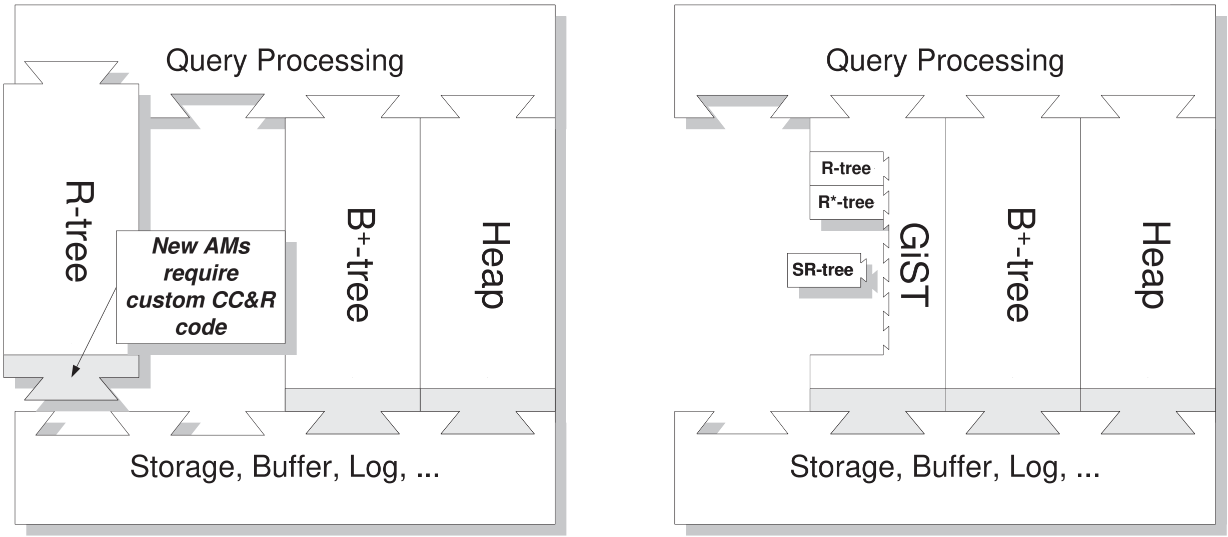
The reason for this is the fundamental complexity and cost involved in developing access methods (AMs) and integrating them into database servers. Designing an AM for use in a commercial ORDBMS requires a very good understanding of concurrency and recovery protocols; integrating an AM into a database server requires a great deal of familiarity with such central components as the lock and log managers. The commercial state of the art in access method extensibility, exemplified by IDS/UDO's Virtual Index Interface [Inf98b] and Oracle's Extensible Indexing Interface [Ora98] and illustrated in principle in Figure 1 (a), does not reduce this complexity. Essentially, these interfaces represent the access method as an iterator data structure; the query executor calls this interface directly to retrieve tuples from the index. An interface like that allows extensibility, but does not reduce the implementation effort of an external AM when compared to a built-in one, if identical levels of concurrency, robustness and integration are desired.[1] As a result, few if any database extension vendors have undertaken the daunting task of implementing a custom-designed, high-quality access method for any of the popular ORDBMSs.

This paper describes the implementation in IDS/UDO of an alternative approach to access method extensibility. This approach is based on the *generalized search tree* (GiST, originally proposed in [HNP95]), a template search tree

*This work was done while the author was a summer intern at Informix Corp.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

[1] The advantage of such an iterator interface is that existing external retrieval engines can easily be interfaced to the database system.



(a) Standard ORDBMS (b) ORDBMS with GiST

Figure 1: Access method interfaces – the database extender's perspective.

structure that is easily extensible in both the data types it can index and the query types it can support. GiST encapsulates core indexing functionality such as search and update operations, concurrency and recovery. The GiST interface, like the existing extensibility interfaces, defines a set of functions for implementing an external AM. However, the GiST interface raises the level of abstraction, only requiring the AM developer to implement the semantics of the data type that is being indexed and those operational properties that distinguish a particular AM from other tree-structured AMs. An AM extension based on this interface typically needs only a small percentage of the (tens of) thousands of lines of code required for a full access method implementation. The level of abstraction offered by the interface relieves the AM developer of the burden of understanding concurrency and recovery protocols and the corresponding components of the database servers. Instead, it is the ORDBMS vendor who implements the concurrency and recovery protocols within GiST, using the existing, low-level extensibility interface to add GiST to the database server (illustrated in Figure 1 (b)). Given that database extension vendors tend to be *domain knowledge* experts rather than *database server* experts, this approach to access method extensibility should result in much higher-quality access methods at substantially reduced development cost for the extension vendor. For the ORDBMS vendor, implementing GiST is no more complex than implementing any other fully integrated AM.

A key ingredient of ORDBMSs is the ability to call user-defined functions (UDFs) that are external to the database server. Since the reliability of the server must not be compromised, it must take precautionary steps to insulate itself from malfunctioning UDFs. In IDS/UDO, a UDF is executed in the same address space as the server, but calling a UDF still involves some overhead: installation of a signal handler to catch segmentation violations and bus errors,[2] allocation of additional stack space, if necessary, and checking of parameters for NULL values. This makes a UDF call considerably more expensive than a regular function call. In Oracle and DB2, UDFs can be executed in a separate address space, which even adds to the cost. When dividing the full functionality of an AM between the database server and an external extension module, as GiST does, UDF calls become inevitable, which can become a performance problem. To address this issue, the original GiST interface was redesigned to reduce as much as possible the number of UDF calls. The new interface is also more flexible, giving external AMs the option of customizing how data is stored on index pages.

The remainder of this paper is structured as follows: Section 2 gives an overview of the GiST data structures; Section 3 describes how the GiST concept was implemented in IDS/UDO and gives examples that highlight some of the features; Section 4 describes some of the concurrency and recovery implementation issues that would arise in a typically ORDBMS and Section 5 compares the performance of GiST-based R-trees with their built-in counterparts in IDS/UDO.

2 Generalized Search Tree Overview

A GiST is a balanced tree which provides "template" algorithms for navigating the tree structure and modifying the tree structure through page splits and deletes. Like all other (secondary) index trees, the GiST stores *(key, RID)* pairs in the leaves; the RIDs (record identifiers) point to the corresponding records on the data pages. Internal pages contain *(predicate, child page pointer)* pairs; the predicate evaluates

[2] These mechanisms are specific to Unix. On Windows NT, similar mechanisms are used.

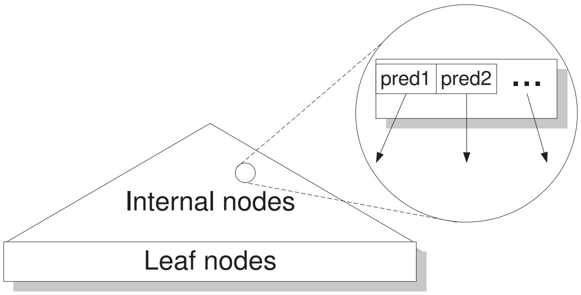


Figure 2: Sketch of a database search tree.

to true for any of the keys contained in or reachable from the associated child page. Figure 2 illustrates this organization, which captures the essence of a tree-based index structure: a hierarchy of predicates, in which each predicate holds true for all keys stored under it in the hierarchy. A B^+-tree [Com79] is a well known example with those properties: the entries in internal pages represent ranges which bound values of keys in the leaves of the respective subtrees. Another example is the R-tree [Gut84], which contains bounding rectangles as predicates in the internal pages. The predicates in the internal pages of a search tree will subsequently be referred to as subtree predicates (SPs).

Apart from these structural requirements, a GiST does not impose any restrictions on the key data stored within the tree or their organization within and across pages. In particular, the key space need not be ordered, thereby allowing multidimensional data. Moreover, the pages of a single level need not partition or even cover the entire key space, meaning that (a) overlapping SPs of entries at the same tree level are allowed and (b) the union of all SPs can have "holes" when compared to the entire key space. The leaves, however, partition the set of stored RIDs, so that exactly one leaf entry points to a given data record.

A GiST supports the standard index operations: SEARCH, which takes a predicate and returns all leaf entries satisfying that predicate; INSERT, which adds a *(key, RID)* pair to the tree; and DELETE, which removes such a pair from the tree. It implements these operations with the help of a set of external functions supplied by the access method developer. This set of external functions, which forms the GiST interface, encapsulates the semantics of the data domain to be indexed and the organization of predicates within the tree; a specific implementation of this interface is an *AM extension*. The combination of generic Gist algorithms for search, insert and delete operations and the AM extension constitutes a fully functional AM.

The following overview outlines the generic algorithms and the role of the AM extension within those algorithms. To show the extent to which the GiST interface was redesigned, the GiST interface functions mentioned here are those of the original GiST design. The redesigned interface and a descriptions of the its functions in the context of the generic algorithms are the topic of the next section.

SEARCH

In order to find all leaf entries satisfying the search qualification, we recursively descend *all* subtrees for which the parent entry's predicate is consistent with the search qualification. Interpretation of the search qualification and its evaluation against data stored in the tree is handled by the interface function *consistent()*, which takes a query predicate and a page entry as arguments and returns true if the entry matches the predicate.

INSERT

Given a new *(key, RID)* pair, we must find a single leaf to insert it on. Note that, unlike B-trees, GiSTs allow overlapping SPs, so there may be more than one leaf where the key could be inserted. We traverse a single path from root to leaf, using as the guiding principle for selecting the next child pointer to follow an *insertion penalty*. This is supplied by the *penalty()* interface function, which takes the new key and a page entry as arguments and returns the corresponding penalty (a numerical value). Conceptually, the AM extension is presented with the new key and an SP and computes a penalty value that typically reflects how much the SP needs to be expanded to accomodate the new key. At each traversed index page, the entry with the smallest insertion penalty is chosen for further traversal. The insertion penalty expresses the AM extension's insertion strategy, i.e., which path is taken when locating the target leaf.

If the target leaf overflows as a result of the insertion, it is split; if the parent also overflows, the splitting is carried out recursively. The *pick_split()* interface function determines the split strategy by specifying which of the entries on a page move to the new right sibling page during a split.

If the leaf's ancestors' predicates do not include the new key, they must be expanded, so that the path from the root to the leaf reflects the new key. The *union()* interface function computes the expanded predicate as the union of the old SP and the new key. Like page splitting, expansion of predicates in parent entries is carried out recursively until we find an ancestor page whose predicate does not require expansion.

DELETE

In order to find the leaf containing the key we want to delete, we again traverse multiple subtrees as in SEARCH. Once the leaf is located and the key is found on it, we remove the *(key, RID)* pair and, if possible, shrink the ancestors' SPs. The *union()* interface function computes the contracted SP as the union of all the entries on the corresponding page.

Although the GiST abstraction prescribes algorithm for search and update operations, the AM designer still has full control over clustering, page utilization and the subtree predicates, which are the performance-relevant structural characteristics of an index. The insertion and split strategies of an AM extension, expressed by the interface functions *penalty()* and *pick_split()*, determine where predicates are placed and how they are moved around; they control page utilization and clustering. The subtree predicates, which greatly influence the performance of search operations, are not interpreted by the GiST algorithms directly; to GiST, they are only sequences of bytes. The AM extension determines the semantics of those predicates and communicates this through the GiST interface functions *consistent()* and *union()*.

3 GiST-Based Index Extension Architecture

When implementing a GiST-based AM extension architecture in a commercial-strength ORDBMS, several issues need to be addressed:

- The existing form of datatype extensibility needs to be retained: In some ORDBMSs, the built-in AMs are extensible in the type of data they can index. For example, a B-tree can be made to work with character strings and user-defined data. It is only required that the data type to be indexed has some particular characteristics (such as a defined total order in the case of a B-tree). In order for an AM to index this new data type, the data type implementor need only provide a set of functions that express the particular characteristics required by the AM (in the B-tree example, this would be a comparison function). This kind of datatype-extensible indexing is already a standard feature in currently at least two ORDBMSs (Informix and Oracle), and it is desirable that a GiST-based extension architecture retain this feature.

- UDF calls are expensive and need to be used judiciously: A high number of calls to AM extension UDFs can have a negative impact on the performance of index operations and make the case for extensible indexing less compelling. The original GiST interface as specified in [HNP95] interacts with the AM extension on a per-page entry basis, which results in a large number of UDF calls. A commercial-strength GiST implementation should reduce this overhead.

- Customizable intra-page storage format: The original GiST design assumed that index pages are organized like an unordered collection of data items (page entries are independent of one another and can be inserted and removed without maintaining any particular order on the page). While this is very general, it precludes optimization of the intra-page data layout, which can be used to compress the data or simplify its access. The B-tree is the most well-known AM that takes advantage of customized intra-page data layout: the page entries are ordered within a page to avoid full scans for lookups. Additionally, internal pages compress interval predicates by storing only the right interval boundary (and using the left neighbor's predicate as the left interval boundary). A GiST-based approach to AM extensibility should not preclude customized page layouts.

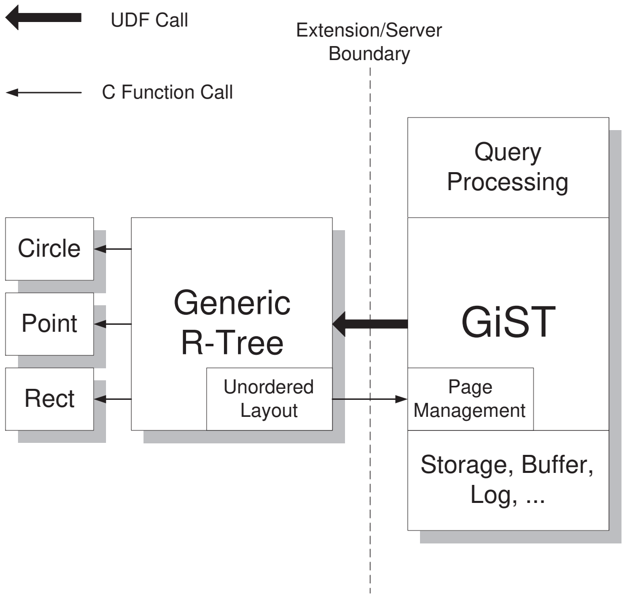


Figure 3: Example of an R-tree in the GiST AM extension architecture

3.1 Architecture Overview

In the IDS/UDO GiST-based AM extension architecture, the full functionality of an AM is divided up into three components: the GiST core inside the database server and the AM extension and a data type adapter in the external database extension module. Figure 3 shows the example of an R-tree extension in this architecture.

GiST Core

The generic GiST algorithms, including the concurrency and recovery protocols, are implemented in the GiST core. It is part of the database server and interacts with the AM extension via the redesigned *GiST interface*, which consists of 11 functions that each AM extension needs to implement. Compared to the original GiST interface, which encapsulates data semantics and the split and insertion strategies, this interface also encapsulates the layout of index pages: the generic GiST algorithms update pages and extract information from them solely through GiST interface functions calls. All of these function calls into the AM extension are executed as UDFs, so that the database server is insulated against failures in the AM extension. In order to reduce the number of UDF calls, the *consistent()* and *penalty()* functions of the original interface have been converted into functions that operate on one entire page instead of individual page entries.

To allow AM extensions to implement customized page layouts, the GiST core exports a GiST-specific page management interface as part of the standard server API (SAPI, see [Inf98a]). This interface is a very thin layer on top of the server-internal page management interface. The latter implements a standard slotted page organization and includes functions to add, update, remove and read page entries, along with various locking and logging options, as well as functions to create and free pages. In contrast, the exported, GiST-specific interface is greatly simplified, being stripped off all logging and locking-related functions and parameters. Furthermore, no page creation and deletion are possible and the target page of each function is implicit (it is the currently "active" page in the tree, i.e., the page that is being traversed, inserted into, etc.). These restrictions do not limit the AM developer's page layout design, but they reduce the potential for doing unwanted damage (since logging, locking and page creation and deletion are handled by the core GiST algorithms, not the AM extension, exposing this functionality to the AM would have no benefit). Also, calls to SAPI functions from the AM extension execute as regular C function calls within the server address space, so there is no need to "ship" the currently active page to the AM extension; copy overhead is therefore avoided. ORDBMSs that execute UDFs outside the server address space could employ careful mapping of address space regions to obtain the same effect.

AM Extension

The AM extension implements the GiST interface and resides in an extension module outside the database server. The AM extension itself specifies an interface, the *extension interface*, that encapsulates the behavior of the data it can index. This interface contains all the functions needed for the supported query operators and to implement the split and insertion strategies. For example, the B-tree extension's interface specifies a comparison function, which is needed to support range queries and perform insertions. The R-tree extension's interface specifies a minimum of seven functions: four of those (namely *overlap()*, *contains()*, *equal()* and *within()*) implement the corresponding search operators, while the other three (*(union()*, *size()* and *intersect()*) are used in the implementations of the split and insertion strategies.

An extension's interface is implemented for every datatype to be indexed by a *datatype adapter* module. For performance reasons, calls by the AM extension to the adapter module are executed as regular C function calls. Since the AM extension functions themselves are called as UDFs, the database server is still insulated from failures in any of the external functions.

The AM extension implements its desired page layout using the GiST-specific page management interface exported by the server. Due to the modular nature of the architecture, user-defined page layouts can be implemented as libraries and reused within other AM extensions (indicated in Figure 3 for the R-tree extension). A standard page layout, which implements the original GiST unordered page layout, is available for AM extensions that do not require customization.

In the current implementation, the B-tree extension occupies about 500 lines of code, excluding comments. The R-tree extension occupies around 800 lines of C code, 150 of which are calls to the unordered page layout and could have been generated automatically. The unordered page layout library is fairly small itself, taking up only about 600 lines of code.

Datatype-specific AM adapter

This user-defined component implements an AM extension's interface for a particular datatype. Typically, datatype adapters are fairly small: our B-tree/integer adapter consists of a 10-line comparison function. An R-tree adapter for simple geospatial objects occupies less than 300 lines of code.

3.2 GiST Interface

The functions of the GiST interface are summarized in Table 1. To provide context, I will go through each index operation chronologically, explaining each interface function as it is called by the generic algorithm.

Each of the GiST interface functions requires as a parameter a pointer to the datatype adapter module, through which the AM extension calls the datatype-specific functions. The GiST core obtains this pointer by calling a UDF that is registered with the database for the specific AM extension/datatype combination. The adapter itself is an array of pointers to functions that implement the AM extension's interface.

SEARCH

To guide tree traversal, the generic search algorithm calls the *search()* function, which, given the currently traversed page, returns the slot indices of those entries that match the query descriptor. For leaf pages, the matching items' heap pointers and predicates—extracted with the *get_key()* function—are returned to the query executor. For internal pages, the child pointers are extracted from the matching items and stored on a stack for future traversal. The query descriptor is assembled by the parser and passed as a parameter into the *search()* function, which then uses SAPI functions to extract the operator and the qualification constants. These server interface calls can involve catalog lookup overhead, which the AM extension may want to avoid incurring for each traversed page. The *begin_scan()* function, called before traversal begins, gives the AM extension an opportunity to extract and store the necessary information from the query descriptor, which is then passed into the *search()* function (as the *state_ptr* parameter). When the search operation is finished, *end_scan()* is called to free up the data allocated in *begin_scan()*.

INSERT

The insertion operation begins by traversing the tree from the root to the insertion target leaf, at each page on the path picking as the next subtree to traverse the child pointer of

Function	Input Parameters	Output Parameters	Purpose
insert()	predicate, heap_ptr, adapter		insert (*predicate*, *heap_ptr*) entry on page
remove()	slots[], num_slots, adapter		remove items corresponding to *slots[]* from page
update_pred()	slot_num, key, adapter		update predicate part of entry on internal page
begin_scan()	query_descr, adapter	state_ptr	transform query descriptor into AM-specific format
search()	query_descr, state_ptr, adapter	matches[], num_matches	return slot indices of matching items on page
end_scan()	state_ptr, adapter		deallocate data allocated in *begin_scan()*
get_key()	slot_num, adapter	key	extract single entry's predicate from page
pick_split()	adapter, orig_SP	right_entries[], num_right, left_SP, right_SP	determine which entries of a page are to be moved to the new right sibling page and compute the SPs for the resulting left and right page
find_min_pen()	new_key, adapter	slot_num	find page entry with smallest insertion penalty on internal page
union()	SP, is_valid_SP, new_key, adapter	SP, SP_changed	compute a page's SP
eq_op()	adapter		returns AM-specific equality operator number

Table 1: GiST interface summary

the minimum-penalty entry returned by the *find_min_pen()* interface function. At the leaf, the *insert()* interface function physically adds the new item to the leaf page, or signals an overflow, at which point a split is performed. To perform the split, the *pick_split()* function returns the slot numbers of the entries to move to the new right sibling, along with the new SPs for the left and right page produced by the split. The split is then installed in the parent: the old SP for the left page is updated via *update_pred()* and a new entry for the new right page is inserted into the parent with the *insert()* function. Recursive splitting due to parent page overflows are handled in the same way. The actual splitting of the original target page is performed by creating the new right sibling as an exact copy of the page and then removing the unnecessary entries from both pages with the *remove()* interface function. After the split has been completed, the insertion of the new data item can be re-attempted.

If the target page does not overflow, the insertion proceeds without a page split, but must check after calling *insert()* whether the target leaf's SP needs to be updated. This is achieved with a call to the *union()* interface function, which computes the new SP, given the old one and the new item, and also indicates whether the SP has changed. If it has changed, it is installed in the corresponding entry in the parent page with the *update_pred()* interface function. If this causes the parent's SP to change, the SP updates are performed recursively.

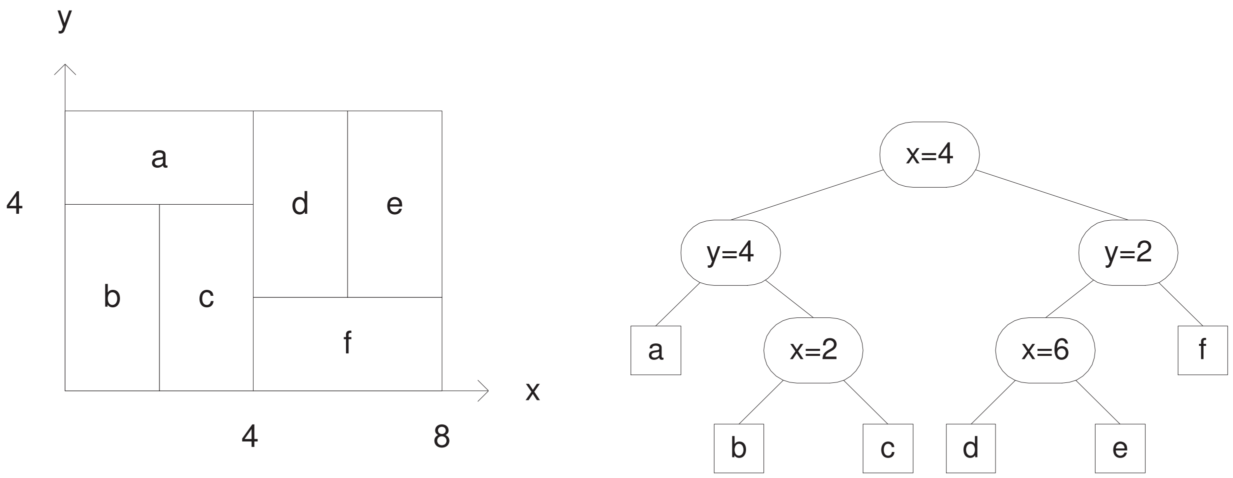
DELETE

There are two scenarios for a delete operation. If it is preceded by a search operation in the same index, the leaf that holds the item to be deleted has already been located, and the deletion of the item can be performed immediately via the *remove()* interface function. If an initial lookup of the target item is necessary, it is performed like a search operation for an equality operator. The query descriptor is assembled using the operator number returned by the *eq_op()* interface function.

The next two sections sketch the implementations of two particular AMs to illustrate the flexibility of the GiST interface.

3.3 Example: GiST-Based B-Trees

The B-tree extension implements a sorted page layout, which it maintains during *insert()* calls with the help of the datatype-specific comparison function. The *remove()* function compacts the slots after deleting the requested entry from a page. The *search()* and *find_min_pen()* functions perform a binary search, again using the datatype-specific



(a) Six 2-dimensional rectangles (b) K-d-tree representation

Figure 4: K-d-tree example

comparison function, to locate the range of entries that match the query descriptor or to find the entry for the subtree that is appropriate for the new key.

B-trees partition the data space at each level of an index and therefore an insertion never causes an SP to expand. As a result, the *union()* function only indicates that the SP has not changed. In a simple B-tree extension, the *get_key()* function would only return a pointer to the predicate stored on the page. For B-trees that support prefix compression for string keys, the *get_key()* function would need to assemble in a private buffer the full string predicate of the entry from the entries on the page and return a pointer to that buffer. The *update_pred()* function simply overwrites an entry's predicate with the new data; in the case of prefix compression, the new predicate is compared to neighboring page entries to determine the compressed predicate.

Predicates in internal pages store only the right boundary of the interval they represent. The rightmost entry of a page carries a 0-length predicate to signal ∞, which requires the extension's binary search routine to filter out such predicates before calling the datatype's comparison function.

3.4 Example: K-d-tree Page Layout

The k-d-tree [Ben75] is a multidimensional binary search tree that is very efficient for storing iso-oriented rectangles that partition a given space. They are used in hB-trees [LS90], a multidimensional point access method that partitions the data space, as the page layout on non-leaf pages. Figure 4 shows an example of six rectangles in 2-dimensional space and their k-d-tree representation. By organizing rectangles into a tree structure, sides that are common to multiple rectangles need only be stored once, resulting in space savings. On the other hand, each rectangle in a k-d-tree needs to refer to the nodes on its path to reconstruct its coordinates. A simple, unstructured page layout cannot map this hierarchical structure efficiently into a sequence of page entries (it could extract every rectangle from the tree and store each one as a separate page entry with its full set of coordinates, but the advantages of the k-d-tree data structure in terms of compression and searching would be lost).

A k-d-tree page layout can be implemented by mapping each node of the k-d-tree onto a page entry. Internal node entries have four components: the coordinate value, two pointers to child nodes (with pointers being stored as slot indices) and one pointer back to the parent node. The root node entry is assigned slot 0 on every page, and is stored similarly to internal nodes, but without the parent pointer. Leaf node entries represent data rectangles, which are stored as a parent pointer and a heap pointer—the predicate data can be recovered from the ancestor nodes. Figure 5 illustrates this for the left branch (representing data rectangles a, b and c) of the k-d-tree shown in Figure 4.

The *search()* function traverses the k-d-tree and returns the slot indices of the matching k-d-tree leaf node page entries. The *get_key()* function, given a slot index of a k-d-tree leaf entry, can reconstruct the corresponding rectangle by traversing the tree from the leaf to the root. The *insert()* function adds a new rectangle to the tree by creating a new k-d-tree leaf entry and an entry for the required new internal k-d-tree node. The *remove()* function reverses this process, removing both the k-d-tree leaf and internal node page entries. Both update functions must be careful not to alter the existing slot assignment, otherwise they will invalidate the k-d-tree child node pointers stored in the other page entries.

Since a k-d-tree partitions the data space, SPs do not expand and the *union()* function signals that to the caller. For the same reason, a new key can only go into one specific subtree, which the *find_min_pen()* function finds by traversing the k-d-tree. If the split strategy is to bisect the k-d-tree at the root, the *pick_split()* function traverses the

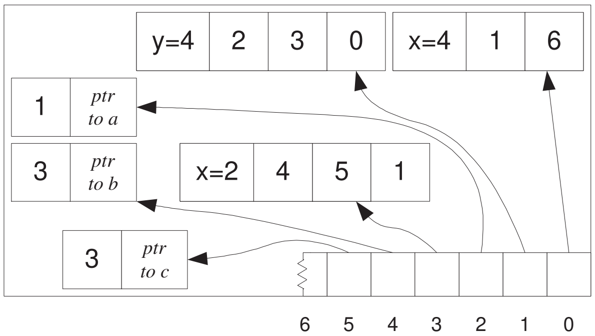


Figure 5: Example of k-d-tree page layout (root: slot 0, internal nodes: slots 1 and 3, leaf nodes: slots 2, 4 and 5)

right subtree of the root and returns the slot indices of its *leaf* nodes, together with SPs for the left and right page of the split, which can be constructed from the root.

3.5 Summary

The redesigned GiST AM extension architecture addresses all three issues mentioned at the beginning of this section:

- Datatype extensibility: By separating the external portion of an AM into an AM extension and a datatype adapter, full datatype extensibility of GiST-based, user-defined AMs can be achieved.
- UDF call overhead: By changing the level of abstraction of the GiST interface from a call-per-entry interface as in the original GiST specification to a call-per-page interface, the number of UDF calls required by index operations is reduced significantly.
- Page layout customization: A further advantage of the call-per-page interface is that it hides the details of the page layout from the GiST core, allowing the AM extension to customize page layout via the SAPI page management interface. This additional flexibility does not necessarily come at the price of an increased implementation effort for the AM developer, because page layout functionality can be separated into libraries and re-used across AM extensions.

Furthermore, it has a number of additional advantages over the current state-of-the-art iterator-style AM extension interfaces:

- AM development is greatly simplified. The above-mentioned B-tree and R-tree extensions were written and debugged in a matter of hours rather than weeks or months. Also, page layout code can be reused across AMs very easily, because it is separated from concurrency and logging considerations. With custom locking and logging protocols intermixed with page layout functionality, such reuse is normally not possible.
- AM stability is improved, because an AM extension is implemented in terms of a (relatively) stable GiST interface and need not rely on interfaces to server-internal services, such as the lock and log manager.
- Intricate concurrency and recovery protocols need only be implemented and tested once, which greatly improves reliability (and oftentimes performance, because externally-developed AMs tend to have somewhat less efficient protocols).
- Despite part of an AM being outside the server, the indices generated by that AM are still fully integrated into the DBMS, with all the advantages: integrated storage management, backup and recovery, support for SQL isolation semantics, built-in concurrency.

4 Implementation Issues

The implementation of the GiST AM extension architecture in IDS/UDO includes concurrency control and recovery. Despite the changes to the GiST interface in the IDS/UDO implementation, the implemented protocols follow mostly what is described in [KMH97]; only in some cases did they need to be adapted. This section describes the implementation details of the protocols.

4.1 Concurrency Control

The locking protocol that allows search and update operations to execute concurrently in the tree is an adaption of the B-link tree technique. All the pages at each level are chained together via links to their right siblings; the addition of this rightlink allows traversing operations to compensate for missed splits by following rightlinks. For this strategy to work, a traversing operation must be able to (1) detect a page split and (2) determine when to stop following rightlinks (a page can split multiple times, in which case the traversing operation must follow as many rightlinks as there are missed page splits). To this end, every page is extended with a sequence number (PSN) in addition to the rightlink. During a page split, a global counter variable is incremented and its new value assigned to the original page's PSN. The new right sibling page receives the original page's old rightlink and PSN. A traversing operation can now detect a split by memorizing the global counter value when reading the parent entry and comparing it with the PSN of the current page. If the latter is higher, the page must have been split and the operation follows the current page's rightlink until it sees a page with a PSN less than or equal to the one originally memorized.

A key component is the global counter variable used to generate page sequence numbers. The counter needs to be incremented atomically and needs to be recoverable in order for split detection to work after a crash. The original paper advocates using the log sequence number of the most recently written log record as a system-global counter variable. This design choice is not possible in IDS/UDO, so instead each index is equipped with an anchor page that holds an index-global counter, among other things. Update operations only consist of simple increment operations, which results in a short critical section around the update and therefore keeps contention low. Making the counter variable recoverable involves writing log records; to amortize the cost of a log write, we only log every 100th increment. This logging is done in advance of the actual increments, so that the logged value nevers falls behind any actual PSN in the index. Putting the counter variable "inside" the index simplifies the implementation considerably, because no changes to server-internal data structures and their recovery are necessary. So far, the degree of concurrency that this allows seems to be adequate.

4.2 Recovery

The GiST logging protocol supports high-concurrency transactional update operations on the index trees by separating them into their content-changing (item insertion and deletion at the leaf level) and structure-modifying parts (page splits, parent entry updates, page deletions). The content change is logged as part of the initiating transaction, whereas the structure modification is logged as an individually committed atomic unit of work (also referred to as atomic actions [LS92] or nested top actions [MHL+92] in the literature). This protocol is not affected by the API change in the IDS/UDO implementation, except for a small detail. When redoing or undoing a leaf insertion or deletion operation, it is not sufficient to perform a simple insertion or deletion of a single page entry, because the AM extension might implement these operations as a sequence of calls to the page interface. Instead, the recovery process must call the AM extension's *insert()* and *remove()* functions.

5 Performance Measurements

A comparison of GiST-based R-trees with the built-in R-trees available in IDS/UDO 9.2 shows that GiST-based AMs not only enjoy software engineering benefits, but can also offer *higher* performance than built-in AMs with datatype extensibility. As mentioned in Section 3.1, the built-in R-tree can be used to index any datatype by supplying datatype-specific functions that implement the query operators and some additional functions needed for splitting and insertion (namely *size()* and *union()*). These datatype-specific functions are provided by the extension module that implements the user-defined type and execute as UDFs. The performance comparison involves individual search and insert operations on a three-level R-tree, which were executed on a Sun machine with a 167MHz UltraSparc CPU. The timings were obtained with the quantify profiling tool, and show the number of cycles needed for full SQL SELECT and INSERT statements.

Extensibility functionality—both AM and datatype extensibility—involves additional cost in comparison to purely hardwired AMs. This cost consists of:

- function descriptor setup: Before calling a UDF, a handle to it must be obtained, which can involve a catalog lookup and permissions checking.
- UDF call overhead: The cost of a single UDF call in IDS/UDO, which is executed in the same address space as the database server, is around 1350 cycles for the test scenario described in this section. In other ORDBMS, a UDF call might involve a context switch and interprocess communication, which would make it far more expensive.

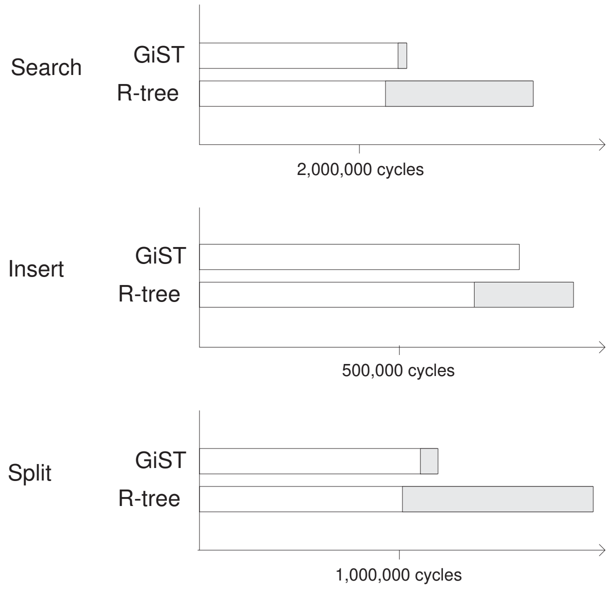


Figure 6: Comparison of GiST-based and built-in R-trees (shaded portion indicates UDF call overhead)

The total execution times, excluding time spent in operating system calls, for three different operations is shown in Figure 6. The operations are: a select with a rectangle containment qualification that retrieves only a single rectangle, but traverses 78 pages in the tree; an insert operation; an insert operation that causes the leaf page to split.

When the built-in R-tree executes a search, it calls the *rectangle_contains()* UDF for every entry on the traversed leaf pages (the *rectangle_overlaps()* UDF for entries on traversed internal pages), resulting in a total of 1359 UDF calls. In contrast, the GiST-based R-tree only calls the *search()* UDF once for every *page* it traverses, requiring only 80 UDF calls (78 plus 2 for *begin_scan()* and *end_scan()*). During the insertion of a new item, the built-in R-tree makes 182 UDF calls, most of these while checking the traversed internal pages for the best subtree to insert in. The GiST-based R-tree subsumes those calls into a single call to the *find_min_pen()* UDF per page, and thereby reduces the total number of calls to only four (two to *find_min_pen()*, one to *insert()* and one to *union()*). When the insertion causes the leaf page to split, the performance gap widens even more: the built-in B-tree makes 704 UDF calls, most of those to find out how to split the page, whereas the GiST-based R-tree only needs 66 UDF calls, 56 of these during the split to extract and insert keys on the new right page.

In all three scenarios, the high number of UDF calls in the built-in R-tree causes it to perform substantially worse than the GiST-based R-tree, resulting in performance losses between 14 and 40 percent. The built-in R-tree has a slight advantage when it comes to function descriptor setup (it uses fewer UDFs, which explains why performance of both AMs is not identical when UDF call overhead is subtracted), but this cannot make up for the large number of UDF calls.

6 Summary

This paper presents a GiST-based approach to extensible indexing implemented in IDS/UDO. The GiST abstraction allows a clean separation of the functionality of an AM into generic tree search and update algorithms as well as generic concurrency and recovery protocols; the AM-specific code, which consists of data domain-specific code and split and insertion strategies, resides in an extension module outside the database server. This has two advantages, that have previously not been realized together: (1) the AM developer need not be concerned with the internals of the database server in general and with the intricacies of concurrency and recovery protocols in particular, (2) the AM is tightly integrated into the database from an operational point of view, offering the same high degree of concurrency and reliability as built-in AMs. The savings in implementation time of new AMs using this architecture are substantial: the B-tree and R-tree extensions are both below 1000 lines of C code, and were written and debugged in a matter of hours rather than weeks or months.

The IDS/UDO implementation of this approach features a GiST interface that allows the external AM developer to take full control of the internal layout of index pages. Additionally, this interface also reduces UDF calling overhead, which can degrade the performance of datatype-extensible AMs. On average, an insert or search operation will only make one UDF call per visitited page. With built-in datatype-extensible AMs, this number can be much higher. A comparison of GiST-based and built-in R-trees in IDS/UDO demonstrates this effect: although the built-in R-tree has lower initial setup cost, the large number of UDF calls reduces performance between approximately 14 and 40 percent for single insertion and search operations in comparison to the GiST-based R-tree.

Acknowledgement

Paul Aoki made valuable contributions to the introduction of this paper and was generally willing to discuss ideas related to this work. Out of the many people at Informix who facilitated this work, I especially want to thank David Ashkenas, Paul Brown, Toni Guttman, Scott Lashley, Kumar Ramayer and Robert Uleman. My thanks also to the anonymous reviewers, who pointed out items that needed clarification.

References

[Ben75] L. Bentley, J.˙ Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[Com79] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(4):121–137, 1979.

[GG98] Volker Gaede and Oliver Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD Conf.*, pages 47–57, June 1984.

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st Int'l Conference on Very Large Databases (VLDB), Zürich, Switzerland*, pages 562–573, September 1995.

[Inf98a] Informix Corp. *Universal Server DataBlade API Programmer's Manual, Version 9.12*, 1998.

[Inf98b] Informix Corp. *Virtual Index Interface Guide*, 1998.

[KMH97] M. Kornacker, C. Mohan, and J. Hellerstein. Concurrency and Recovery in Generalized Search Trees. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Tucson, Arizona*, pages 62–72, May 1997.

[LS90] D. Lomet and B. Salzberg. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM TODS*, 15(4):625–685, December 1990.

[LS92] D. Lomet and B. Salzberg. Access Method Concurrency with Recovery. In *Proc. ACM SIGMOD Conf.*, pages 351–360, 1992.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1), March 1992.

[Ora98] Oracle Corp. *All Your Data: The Oracle Extensibility Architecture*, November 1998.