Access Methods for Next-Generation Database Systems

by

Marcel Kornacker

Diplom Informatik (Universität Hamburg) 1995
M.S. (University of California, Berkeley) 1997

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Joseph M. Hellerstein, Chair
Professor Michael R. Stonebraker
Professor Tito A. Serafini

Fall 2000

The dissertation of Marcel Kornacker is approved:

Chair Date

Date

Date

University of California at Berkeley

Fall 2000

Access Methods for Next-Generation Database Systems

by

Marcel Kornacker

Abstract

Access Methods for Next-Generation Database Systems

by

Marcel Kornacker

Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Joseph M. Hellerstein, Chair

Today's extensible object-relational database management systems (ORDBMSs) are being deployed to support non-traditional applications such as dynamic web servers and geographic information systems. ORDBMSs distinguish themselves from purely relational DBMSs by providing an extensible architecture, built around a richer and user-extensible type system combined with object-oriented concepts such as type hierarchies. They retain standard features of relational databases such as declarative access, multiuser operation, transactional isolation and recoverability.

One particular aspect of DBMS functionality that is critical to performance is their support for access methods (AMs). In traditional relational DBMSs, B^+-trees [Com79] serve as the AM of choice to provide a very high level of performance for applications dealing with the standard SQL datatypes (numeric data, character strings, dates, etc.). In

order to provide the same level of performance for non-traditional applications, B^+-trees are not sufficient; instead, novel kinds of datatype-specific AMs are required. The most promising approach to supporting those novel AMs is an extensible architecture in which the core services of the ORDBMS can be complemented with externally-supplied AMs.

In my dissertation, I investigate general issues that arise in the design and implementation of non-traditional AMs in an extensible ORDBMS. This research was undertaken in the context of the generalized search tree (GiST), a tree-structured template access method, which encapsulates standard AM search and update functions and is a suitable basis for AM extensibility in ORDBMSs. The dissertation contains three contributions. The first is an extension of the GiST API that makes it more flexible and at the same time improves performance when implemented in a typical commercial ORDBMS. The second comprises concurrency and recovery protocols that allow GiSTs to be useful in application scenarios where high concurrency and recoverability are required. With these protocols, GiSTs fully encapsulate physical concurrency, transactional isolation and recovery, and thereby relieve an external access method of the burden of dealing with these issues. The API extensions and the concurrency and recovery protocols together make GiSTs a high-performance alternative to custom AM development in commercial ORDBMS. The third contribution is an AM performance analysis framework, implemented in a corresponding tool, that gives the AM developer a detailed picture of an AM's performance deficiencies while still retaining the GiST framework's independence of the datatype and application

domain.

Professor Joseph M. Hellerstein
Dissertation Committee Chair

Contents

Acknowledgements

First, I wish to thank Joe Hellerstein for his support, encouragement, faith in my intellectual abilities, and patience with one of his more difficult students. I also wish to thank Mike Stonebraker for the considerable help and support I received from him throughout my undergraduate and graduate career; it is thanks to him that I was motivated and able to consider pursuing a PhD at Berkeley. Finally, I want to thank the co-inhabitants of 419 Soda Hall, in particular Paul Aoki, Mehul Shah and Noah Treuhaft, with whom I spent many hours discussing my research as well as other related and unrelated topics.

Chapter 1

Introduction

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, and so forth. ORDBMSs distinguish themselves from purely relational DBMSs by providing a richer and user-extensible type system, combined with object-oriented concepts such as type hierarchies; they distinguish themselves from purely object-oriented DBMSs by retaining standard features of relational databases such as declarative access, multiuser operation, transactional isolation and recoverability. The added flexibility and performance when dealing with non-traditional datatypes makes them a better match for the abovementioned application areas than standard relational DBMSs; as a consequence, object-relational extensions to the pure relational model are now part of most commercially significant DBMSs.

In the technically most accomplished commercial products such as Informix' Universal Server and IBM's DB2 UDB, a user-extensible type system is provided through an extensible architecture, supplemented by domain-specific database extensions supplied by external vendors. In this approach, the database server provides standard DBMS functionality, such as storage management and query optimization and execution, in a datatype-independent way. The datatype-specific functions and query predicates (e.g., clipping of geospatial objects or determining when a point lies within a rectangle) are provided by an external vendor in the form of a function library, which is linked in dynamically by the server. When processing queries over such user-defined datatypes, the DBMS calls the functions provided in the extension library. By internalizing part of the application functionality in this way, the DBMS server can avoid shipping the data out to the application, and can thereby realize a performance advantage.

The techniques used in these ORDBMSs are direct descendents of research prototypes pioneered in the 80s and 90s (such as Postgres [SR86], Starburst [SCF+86] and Shore [CDF+94]). Despite this long time period and the high degree of commercial acceptance, object-relational technology is far from mature: while type hierarchies and extension methods have been uniformly accepted and even standardized [ISO96], these new features have often not been complemented by corresponding storage-level techniques to assure good performance (see DeWitt's criticism of how object-relational concepts subvert parallel architectures [DeW96], which essentially argues that the translation of those concepts into physical storage structures still needs improvement). One particular area of

functionality in traditional relational DBMSs that is critical to performance is their support for access methods (AMs).

In traditional relational database management systems, B^+-trees [Com79] serve as the AM of choice to provide a very high level of performance for applications dealing with the standard SQL datatypes (numeric data, character strings, dates, etc.). In order to provide the same level of performance for the abovementioned applications, B^+-trees are not sufficient; instead, novel kinds of datatype-specific AMs are required. The most promising approach to supporting those novel AMs is an extensible architecture in which the core services of the ORDBMS can be complemented with externally-supplied AMs. The current state of the art in AM extensibility is not very well developed. The support found in commercial ORDBMSs for AM extensibility is crude, if at all existent. Likewise, the research community has largely neglected the issue, focussing instead on general extensibility questions, which systems like Exodus [CDF+86], Postgres [SK91] and Starburst [HCL+90] addressed, or on developing novel search trees to support each new application area. 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, very few of the structures developed since the B^+-tree have found their way into commercial products, because the cost of development and maintenance generally cannot be justified in light of the limited applicability of a domain-specific AM. This state of affairs is a consequence of the lack of an effective form of AM extensibility in ORDBMSs.

In this dissertation, I investigate general issues related to the design and implementation of non-traditional AMs. This research was undertaken in the context of the generalized search tree (subsequently referred to as GiST), which was originally introduced in [HNP95]. The GiST is a balanced, tree-structured template access method, which encapsulates standard AM search and update functions. By supplying a small set of custom functions, a GiST can be made to emulate a wide range of tree-structured AMs relatively effortlessly.

This dissertation contains three contributions. The first is an extension of the original GiST design that makes it more flexible and at the same time improves its performance when implemented in a typical commercial ORDBMS. The second contribution consists of concurrency and recovery protocols that allow GiSTs to be useful in application scenarios where high concurrency and recoverability are required. With these protocols, GiSTs fully encapsulate physical concurrency, transactional isolation and recovery, and thereby relieve the access method developer of the burden of dealing with these issues. The API extensions and the concurrency and recovery protocols together extend the GiST abstraction into an effective mechanism for AM extensibility in ORDBMSs. The third contribution is an AM performance analysis framework, implemented in a corresponding tool, that gives the AM developer a detailed picture of an AM's performance deficiencies while still retaining the GiST's independence of the datatype and application domain.

Chapter 2 explains the original GiST design and informally discusses its usefulness and the limits of its generality. Chapter 3 discusses existing approaches to AM extensi-

bility and the relevance of the GiST abstraction for an AM extensibility architecture. It goes on to describe an enhanced GiST model that was implemented in a commercial OR-DBMS, Informix Dynamic Server with Universal Data Option (subsequently referred to as IDS/UDO). The IDS/UDO implementation employs a newly designed GiST interface that reduces the number of user-defined function calls, which are typically expensive to execute, and at the same time makes the GiST a more flexible data structure. Experiments show that this enhanced form of GiST-based AM extensibility does not suffer from performance degradation in comparison to built-in AMs when indexing user-defined data types.

Chapter 4 presents general algorithms for concurrency control in tree-based access methods, as well as a recovery protocol and a mechanism for ensuring repeatable read. Although developed in a GiST context, the algorithms are generally applicable to many tree-based access methods. The concurrency control protocol is based on an extension of the link technique originally developed for B-trees, and completely avoids holding node locks during I/Os. Repeatable read isolation is achieved with a novel combination of predicate locks and two-phase locking of data records. A discussion of the fundamental structural differences between B-trees and more general tree structures like GiSTs explains why the algorithms developed here deviate from their B-tree counterparts.

Chapter 5 describes an analysis framework for tree-structured access methods. Designing and tuning access methods has always been more of a black art than a rigorous discipline, with performance assessments being mostly reduced to presenting aggregate runtime or I/O numbers. This chapter describes an analysis framework for AMs that defines perfor-

mance metrics that are more useful than traditional summary numbers and thereby allow the AM designer to detect and isolate deficiencies in an AM design. This framework was implemented in amdb, a comprehensive graphical design tool for AMs that are constructed on top of the GiST abstraction. Amdb complements the analysis framework with visualization and debugging functionality, allowing the AM designer to investigate the source of those deficiencies that were discovered with the help of the performance metrics. Several AM design projects undertaken at U. C. Berkeley have confirmed the usefulness of the analysis framework and its integration with visualization facilities in amdb. The analysis process that produces the performance metrics is fully automated and takes a workload—a tree and a set of queries—as input. The metrics characterize the performance of each query as well as that of the tree structure. Central to the framework is the use of the optimal behavior, which can be approximated relatively efficiently, as a point of reference against which the actual observed performance is compared. The framework applies to all GiST-compliant AMs and is not restricted to particular types of of data or queries.

Chapter 2

Generalized Search Trees: Motivation and Overview

Although the design of novel AMs has been a popular research topic in the past decade or so, 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 detailed 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.

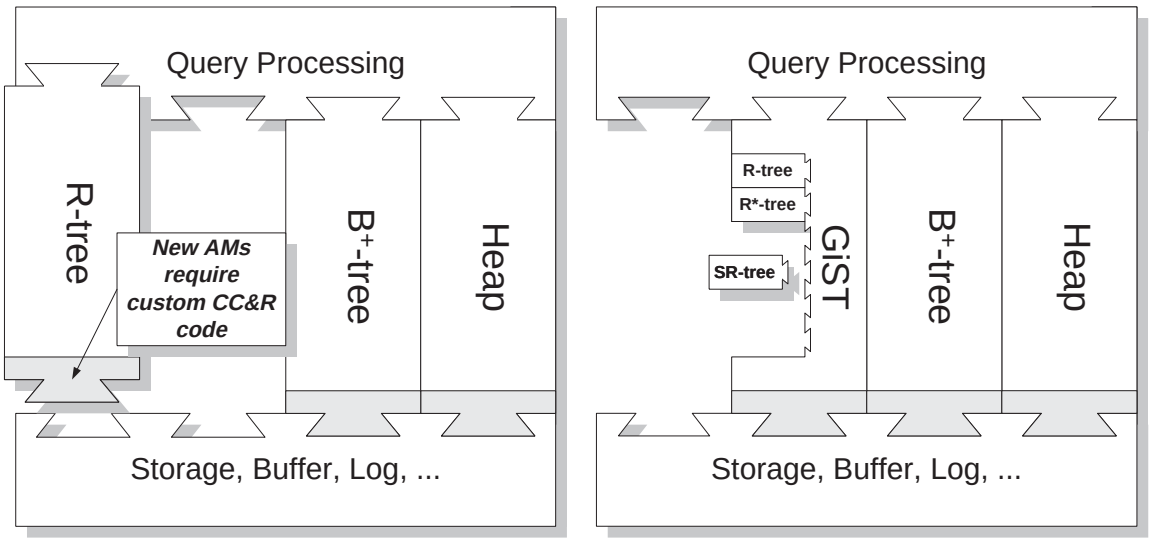
The commercial state of the art in access method extensibility, exemplified by IDS/UDO's Virtual Index Interface [Inf98b], Oracle's Extensible Indexing Interface [Ora98] and DB2

UDB's table functions [DM97],[1] does not reduce this complexity: essentially, these interfaces represent the access method as an iterator data structure, similar to a built-in AM, which the query executor calls directly to retrieve tuples (illustrated in principle in Figure 2.1 (a)). 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. As a result, few if any database extension vendors have undertaken the daunting task of implementing a custom-designed, high-quality access method from scratch for any of the popular ORDBMSs.[2] An iterator interface is most suitable for interfacing existing external retrieval engines to the database system (see [DM97] for an example). However, this approach is mostly limited to read-only data, because the concurrency and recovery regimes of the ORDBMS and external AM are not integrated.

The GiST abstraction was designed to address the problems inherent in AM extensibility. Compared to an iterator-based extensibility interface, the GiST interface raises the level of abstraction, only requiring the AM developer to implement the semantics of the data type 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. As we will see in Chapter 3, the level of abstraction offered by the interface

[1] As with most other object-relational mechanisms, the AM extensibility mechanisms present in those commercial systems are direct descendents of those pioneered in research systems such as Postgres [Aok91].

[2] See [BSSJ99] for an account of an implementation of an AM using IDS/UDO's VII.



(a) Standard ORDBMS (b) ORDBMS with GiST

Figure 2.1: Access method interfaces – the database extender's perspective.

can relieve 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 2.1 (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.

The original GiST publication [HNP95] introduced the basic abstraction of the data structure, but ignored those aspects of an AM extensibility architecture that are required for an industrial-strength solution, namely efficiency, concurrency control, and recovery.

This dissertation will address these issues in turn: Chapter 3 presents a redesigned GiST interface that avoids the performance problems inherent in the original design, and Chapter 4 presents concurrency and recovery mechanisms suited to the GiST framework. In order to give the reader a perspective on the extent of this work, the remainder of this chapter contains a description of the original GiST abstraction. The chapter closes with an informal discussion of its strengths and limitations.

2.1 GiST Overview

A GiST is a balanced tree that provides "template" algorithms for navigating the tree structure and modifying the tree structure through node splits and deletes. Like all other (secondary[3]) 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 nodes contain *(predicate, child page pointer)* pairs; the predicate evaluates to true for any of the keys contained in or reachable from the associated child page (see Figure 2.2). This 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 nodes 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 nodes. The

[3]The GiST abstraction is equally applicable to primary indices, but for simplicity we limit out discussion here to secondary indices.

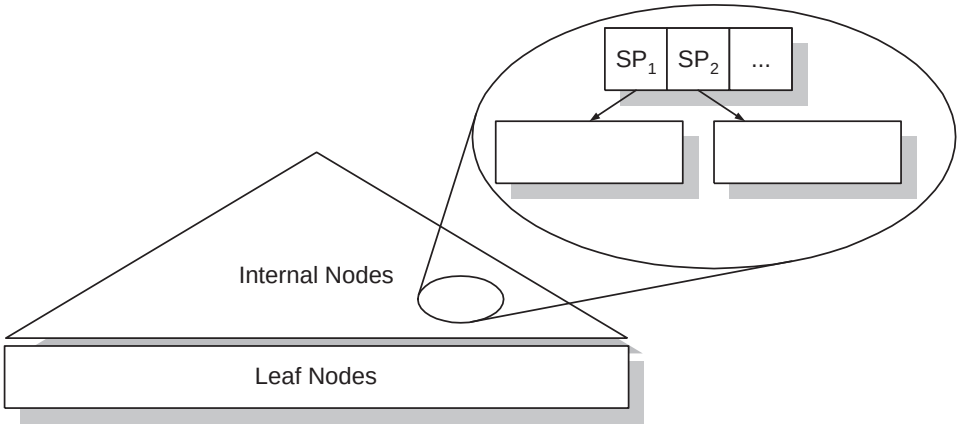


Figure 2.2: Basic GiST structure.

predicates in the internal nodes 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 the organization of data within and across nodes. In particular, the key space need not be ordered, thereby allowing multidimensional data. Moreover, the nodes 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.[4]

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

[4]This structural requirement excludes R^+-trees [SRF87] from conforming to the GiST structure, and generally precludes AMs that store data redundantly [HKP97].

```
search(search-pred)
  push(stack, root);
  while (stack is not empty)
    child = pop(stack);
    for each page entry e on child:
    if ( consistent(search-pred, e.pred))
      if (child is leaf)
        add e to search result set;
      else
        push(stack, e.child-ptr);
      end
    end
  end
```

Figure 2.3: Original GiST search algorithm.

with the help of a set of extension methods supplied by the access method developer. The GiST can be specialized to one of a number of particular access methods by providing a set of extension methods specific to that access method. These extension methods encapsulate the exact behavior of the search operation as well as the organization of keys within the tree.

2.2 GiST Template Algorithms

This section provides a sketch of the implementation of the operations and how they use the extension methods. For a more detailed description, together with examples of B-tree and R-tree extension methods, see the original publication [HNP95]. The user-supplied extension functions are summarized in Table 2.1.

SEARCH In order to find all leaf entries satisfying the search predicate, we recursively

Function	Input Parameters	Output Parameters	Purpose
consistent()	query qualifier, predicate	boolean	determine whether predicate satisfies the query qualifier
pick_split()	set of predicates	set of predicates destined for right node	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
penalty()	new key, SP	penalty value	compute penalty for inserting new key into subtree with given SP
union()	set of predicates	predicate	compute the union of a set of predicates
compress()	predicate	compressed predicate	computes a compressed representation of a predicate
decompress()	compressed predicate	predicate	computes a decompressed representation of a compressed predicate

Table 2.1: Summary of the original GiST interface.

descend *all* subtrees for which the parent entry's SP is consistent with the search predicate and return all leaf entries consistent with the search predicate (in both cases employing the extension method *consistent()*). Figure 2.3 shows an implementation of the search algorithm. Calls to the GiST interface functions are shown in italics.

INSERT Given a new *(key, RID)* pair, we must find a leaf on which to insert it. Note that because GiSTs allow overlapping SPs, there may be more than one leaf where the key could be inserted. The extension method *penalty()* compares a key and a subtree predicate and computes a domain-specific penalty for inserting the key within the

subtree summarized by the SP. Using this extension method, we traverse a single path from root to leaf, following the branches with the lowest insertion penalty.

If the leaf overflows and must be split, a extension method, *pick_split()*, is invoked to determine how to distribute the keys between two nodes. If, as a result, the parent also overflows, the splitting is carried out bottom-up.

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 expansion is done with a extension method *union()*, which computes the union of a set of predicates. Like node splitting, expansion of SPs in parent entries is carried out bottom-up until we find an ancestor node whose SP does not require expansion. Figure 2.4 shows the implementation of the insertion algorithm.

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, which is achieved by recomputing it with the *union()* extension method.

The extension methods *compress()* and *decompress()* are used to compress predicates prior to inserting them on a page and to decompress stored predicates prior to passing them to *consistent()*. The original GiST design also contained provisions for emulating B-trees efficiently (essentially by requiring an extension method that implements predicate comparisons), but this is does not affect the following exposition and will be ignored.

The original GiST search algorithm as described above is limited in the types of query operators it can support: the *consistent()* extension function is only allowed to examine the query qualification and page predicate in order to determine whether to traverse a subtree or return a leaf item. This limits the query qualification to specify logical containment, i.e.,

```
globals:
   stack; // records nodes on path

insert(new-key, RID)
   leaf = locateLeaf(new-key);
   if (not enough space on leaf)
      splitNode(leaf, new-key);
   else
            if ( union(entries on leaf, new-key) != SP(leaf))
         updateParent(leaf,
            union(entries on leaf, new-key), 0, 0);
      end
   end
   insert entry (new-key, RID) on leaf;

locateLeaf(new-key)
   p = root;
   loop
      if (p is not leaf)
         push(stack, p);
         minpenalty = penalty(new-key,
            pred of first entry on p);
         for each page entry e on p:
         if ( penalty(new-key, e.pred) ¡ minpenalty)
            minpenalty = penalty(new-key, e.pred);
               p = e.child-ptr;
            end
      else
         return p;
      end
   end

splitNode(p, new-key)
   create new node p';
   right-entries = pickSplit(entries on p, new-key);
      p-SP = union(entries on p without right-entries);
   p'-SP = union(right-entries);
   move right-entries from p to p';
   updateParent(p, p-SP, p', p'-SP);

updateParent(left, left-SP, right, right-SP)
   parent = parent(left, stack);
   if (not enough space on parent)
      splitNode(parent, union(left-SP, right-SP));
   else
      updateParent(parent,
            union(entries on parent, left-SP, right-SP), 0, 0);
   end
   update predicate of left entry on parent with left-SP;
   if (right != 0)
      insert entry (right-SP, RID(right)) on parent;
   end
```

Figure 2.4: Original GiST insert algorithm.

range queries, which are typically executed by depth-first tree traversal. In order to support more advanced query operators and traversal orders, such as nearest-neighbor searches or index-assisted sampling or statistical computations, Aoki proposes generalizations of the GiST search algorithm [Aok98]. The generalizations allow the search algorithm to traverse the tree in an application-specific way and let it compute user-defined aggregates; this added functionality requires additional extension methods. Aoki's proposed generalizations can be applied independently of the techniques that will be discussed in this dissertation.

2.3 Generality of GiSTs and Related Work

In order for a template data structure to be useful as a basis for AM extensibility, it needs meet two seemingly contradictory requirements: it needs to be able to model a wide variety of (data domain-specific) AM designs, yet at the same time standardize the structure so that template algorithms can operate without knowledge of the data domain. The original GiST design of [HNP95] showed that GiSTs are capable of the latter of the two requirements; I will argue that GiST incorporates the essential mechanisms of an AM and therefore also meets the first one. A comparison with prior work in this area shows that GiST was the first and so far remains the only data structure to balance functionality and generality in that way.

The task of an index AM is to speed up associative access to a (typically small) subset

of a (typically large) collection of data, compared to a sequential scan of the full data collection. This is achieved by a) grouping the data into clusters based on a fixed page size, and b) providing a directory structure that guides access to those clusters. The efficacy of an AM in comparison to a sequential scan hinges on the clustering to reflect the "typical query shape"[5] and on the directory structure to represent the clusters accurately. If both aspects are done well, an AM can limit the number of I/Os for a search to the number of pages required to store the result set (or a small multiple thereof), usually orders of magnitude below the number of I/Os needed for a full scan of the data.

The GiST framework provides those crucial clustering and directory abstractions, both of which are user-controllable: a) the clustering of the indexed data is determined by the *pick_split()* and *penalty()* extension methods, and b) the directory structure is implemented as a tree, but SPs, which determine accuracy, are user-defined. Therefore, it should be possible to map all or most of the design elements of an AM onto a GiST and still capture the order-of-magnitude performance improvement over the full scan.

Currently, the GiST abstraction appears to be the only generic index structure that was designed specifically with extensibility in mind; alternative data structures reported in the literature either lack extensibility or have not been specified in enough detail to be practicable. Lomet [Lom91] explores the concept of "grow and post" (GP) trees, which designates

[5] This consideration does not apply to point queries, i.e., queries that verify the existence of an individual data point. For such queries, a hash-based index is ideal and can be "extended" to any data domain by providing a function that maps the domain-specific key to a hashable value. In other words, the generalized indexing problem for point queries has already been solved and need not be dealt with in the context of a template AM like GiST.

a category of AMs with specific structural properties (which are similar to those of GiSTs): the category includes all AMs that are tree-structured, height-balanced trees that perform bottom-up splits and are capable of dynamic insertions and deletions. The author proposes using a multi-dimensional point index to index non-traditional datatypes and does not develop the ideas behind GP trees into an extensible data structure. Another generic tree structure is the π-tree [LS92], which was developed as a vehicle for concurrency control and recovery algorithms (and is therefore discussed in more detail in Section 4.6). The π-tree has template algorithms for search, insertion and deletion, but due to the focus on concurrency and recovery, was not developed into an extensible data structure. Furthermore, it requires a partitioning of the data space (i.e., no overlap among SPs at the same tree level), and is therefore less general than GiSTs. More recently, van der Bercken et al [vdBDS00a, vdBDS00b] described a Java library for storage structures and query processing algorithms, which also includes a framework of tree-based index structures. The authors report that this framework provides a template AM which is specialized through the use of functions, complete with generic query processing and bulk-loading algorithms. The paper contains no details, which makes it hard to judge whether this framework provides a more customizable or more usable abstraction than GiSTs.

An alternative to providing generic and extensible indexing functionality that is more prevalent in the research literature is the toolkit approach: a DBMS or storage system exports the primitives needed to asssemble an index structure from scratch (examples are [CDF+86, BBG+88, BRS96]). While this gives the AM developer almost unlimited flexi-

bility, it does not hide any of the challenging implementation issues and therefore does not address the fundamental problems that prevent widespread acceptance of non-traditional AMs.

Chapter 3

Performance-Oriented Extensions of GiSTs

This chapter demonstrates the advantages of a GiST-based approach to access method extensibility in industrial-strength systems. It describes the GiST-based extension architecture implemented in IDS/UDO and the extended GiST interface, which takes into account the performance considerations of an industrial-strength ORDBMS. A comparison of a GiST-based R-tree and its built-in counterpart in IDS/UDO shows that the flexibility afforded by GiST need not carry a performance penalty.

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

Datatype Extensibility 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 have some particular characteristics (such as a defined total order in the case of a B-tree). In order for an existing 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 Safety Overhead 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,[1] 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 must be executed in a separate address space, which adds even more 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 and make the case for extensible indexing

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

less compelling. The original GiST interface as described in Chapter 2 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.

Page Layout Customization The original GiST design assumes that index pages are organized as 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 facilitate customized page layouts that allow predicate compression and optimized intra-page search. Note that the original GiST design's *compress()* and *decompress()* extension functions are only applied to an individual page entry, which precludes such techniques as prefix compression, that require access to more than a single data item.

The remainder of this chapter is structured as follows: Section 3.1 describes the implementation of the GiST concept in IDS/UDO and how the GiST interface was extended to take the abovementioned considerations into account. The extended GiST interface is illustrated with two examples of specific node layouts, a B-tree and a k-d-tree node layout.

Section 3.2 compares the performance of GiST-based R-trees with their built-in counterparts in IDS/UDO.

3.1 GiST-Based Index Extension Architecture

3.1.1 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 3.1 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*. The GiST core is part of the database server and interacts with the AM extension via the extended *GiST interface*, which consists of 11 functions that each AM extension needs to implement. In addition to the original GiST interface, which encapsulates data semantics and the split and insertion strategies, the extended 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

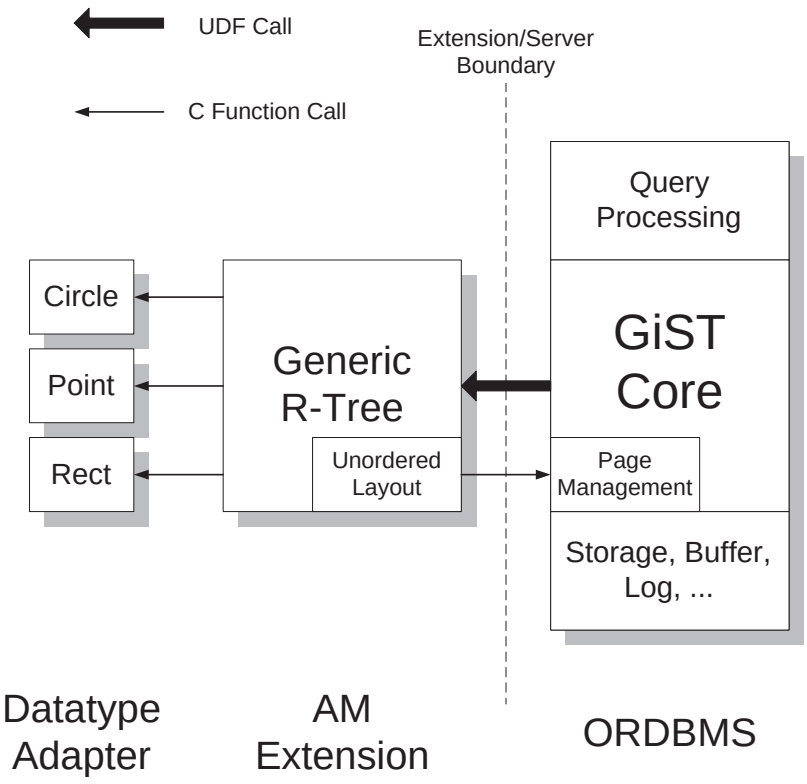


Figure 3.1: Example of an R-tree in the GiST AM extension architecture.

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], the collection of server functions that are callable from an external module). The GiST-specific page management 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 con-

trast, the externally-visible GiST-specific interface is greatly simplified, being stripped of 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 buffer pool regions to achieve the same effect.

AM Extension The AM extension implements the extended GiST interface and resides in an extension module outside the database server. The AM extension specifies its own custom interface, an *extension interface,* that encapsulates the behavior of the data it can index. This interface contains all the functions needed for the supported query operators, as well as additional functionality needed to implement the split and insertion strategies. For example, the B-tree extension's interface specifies a comparison function, which is needed to support queries and perform insertions. The R-tree extension's interface specifies a minimum of seven functions: four of those (*overlap(), contains(), equal()* and *within()*)

implement 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 3.1 for the R-tree extension). A standard page layout that 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:

the 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.

3.1.2 Extended GiST Interface

The functions of the extended GiST interface are summarized in Table 3.1. To provide context, I will go through each index operation, explaining each interface function as it is called by the generic algorithm.

A separation of the external functionality of an AM into the AM extension itself plus the datatype adapter increases flexibility, but also complexity, because the function entry points of the adapter need to be communicated to the AM extension. In a built-in, datatype-extensible AM such as the server-provided R-tree, this is achieved through an *operator class*, or *opclass*, a named collection of UDFs that essentially contains the type adapter's functions (see [Aok91] for details). In this architecture, the external, datatype-specific functions that the built-in AM needs to call are resolved using the server-supplied mechanisms for registered UDFs and called as UDFs. In contrast, the GiST-based architecture avoids these UDF calls and uses standard C function calls when executing external AM extension code. The pointers to the C functions of the datatype adapter module are bundled together into a C structure that is passed as a parameter to each of the GiST interface functions. The GiST core obtains this structure once at query initialization by calling a UDF that is registered with the database for the specific AM extension/datatype combination.

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.[2] 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 query operator and the qualification constants. These SAPI calls can involve catalog lookup overhead, which the AM extension may want to avoid incurring during each *search()* call. 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()*. Figure 3.2 shows the implementation of this search operation.

INSERT The insertion operation begins by traversing the tree from the root to the insertion target leaf. At each page on the path, it calls the *find_min_pen()* interface functions to determine the mininum-penalty entry, which provides the child pointer for the next page to traverse. 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,

[2] As mentioned in the preceding section, only the currently traversed page is accessible through the GiST-specific page management interface. For that reason, it is not necessary to pass this page explicitly to the GiST interface functions.

```
search(search-pred)
   state-ptr =  begin_scan(search-pred);
   push(stack, root);
   while (stack is not empty)
      child = pop(stack);
      match-entries =  search(child, search-pred,
         state-ptr);
      if (child is leaf)
         for each slot s in match-slots:
          add ( get_key(s), s.ptr) to search result set;
      else
         nsn = global NSN;
         for each slot s in match-slots:
            push(stack, [s.ptr, nsn]);
      end
   end
    end_scan(state-ptr);
```

Figure 3.2: GiST search algorithm for the extended interface.

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 SP for the original page (output parameter *left_SP* of the *pick_split()* function) 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 is 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

```
globals:
   stack; // records nodes and NSNs on path

insert(new-key, RID)
   leaf = locateLeaf(new-key);
   if (not enough space on leaf)
      splitNode(leaf, new-key, 0);
   else
      if ( union(leaf, SP(leaf), new-key)
            != SP(leaf))
         updateParent(leaf,
             union(leaf, SP(leaf), new-key), 0, 0);
      end
   end
    insert(leaf, new key, RID);

locateLeaf(new-key)
   p = root;
   loop
      if (p is not leaf)
         push(stack, p);
         p = find_min_pen(p, new key);
      else
         return p;
      end
   end

splitNode(p, key1, key2)
   create new node p';
   (split-info, p-SP, p'-SP) = pickSplit(p, SP(p),
      key1, key2);
   move data from p to p' according to split-info;
   updateParent(p, p-SP, p', p'-SP);

updateParent(left, left-SP, right, right-SP)
   parent = parent(left, stack);
   if (not enough space on parent)
      splitNode(parent, left-SP, right-SP);
   else
      updateParent(parent,
          union(parent,
             union(parent, SP(parent), left-SP),
            right-SP),
         0, 0);
   end
    updatePred(parent, slot of left entry, left-SP);
   if (right != 0)
      insert(parent, right-SP, RID(right));
   end
```

Figure 3.3: GiST insert algorithm for the extended interface.

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. Figure 3.3 shows the implementation of the insertion operation.

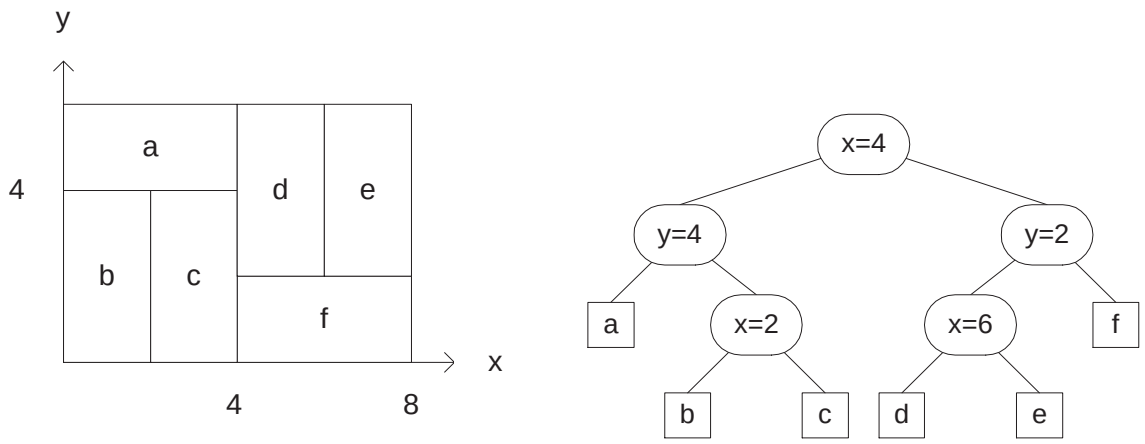
DELETE There are two scenarios for a delete operation. If it is preceded by a search operation in the same index, the leaf that holds the item to be deleted has already been located, and the deletion of the item can be performed immediately via the *remove()* interface function. If an initial lookup 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 extended GiST interface.

3.1.3 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 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



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

Figure 3.4: K-d-tree example.

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 an internal 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.

3.1.4 Example: K-d-tree Page Layout for GiST-Based Spatial Indices

The k-d-tree [Ben75] is a main-memory multidimensional binary search tree that is very efficient for storing iso-oriented rectangles that partition a given space. K-d-trees are

used as the layout of non-leaf pages of hB-trees [LS90], a multidimensional point access method that partitions the data space. Figure 3.4 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 GiST page entry. GiST page entries corresponding to internal nodes of the k-d-tree have four components: the coordinate value, two pointers to k-d-tree child nodes (with pointers being stored as slot indices) and one pointer back to the k-d-tree 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 derived from the ancestor nodes. Figure 3.5 illustrates this for the left branch (representing data rectangles a, b and c) of the k-d-tree shown in Figure 3.4.

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

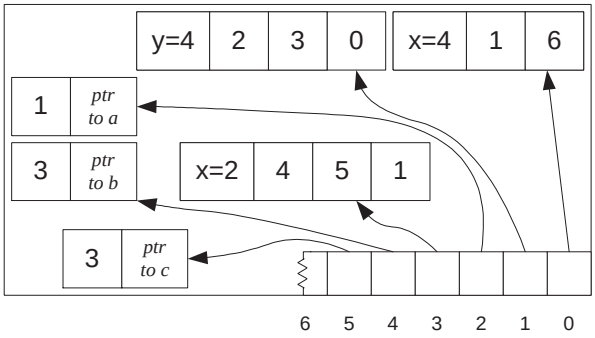


Figure 3.5: Example of a k-d-tree page layout (root: slot 0, internal nodes: slots 1 and 3, leaf nodes: slots 2, 4 and 5).

leaf entry, reconstructs 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.

A k-d-tree partitions the data space, so that SPs do not expand on insertion, which the *union()* function signals 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 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.

3.2 Performance Measurements

A comparison of GiST-based R-trees with the built-in R-trees available in IDS/UDO 9.2 shows not only that GiST-based AMs enjoy software engineering benefits, but that these benefits do not come at the expense of performance. In fact, in this particular scenario, GiST-based R-trees even offered *higher* performance than their built-in counterpart. As mentioned in Section 3.1.1, the IDS/UDO 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 (*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 I present here 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* [Rat] 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

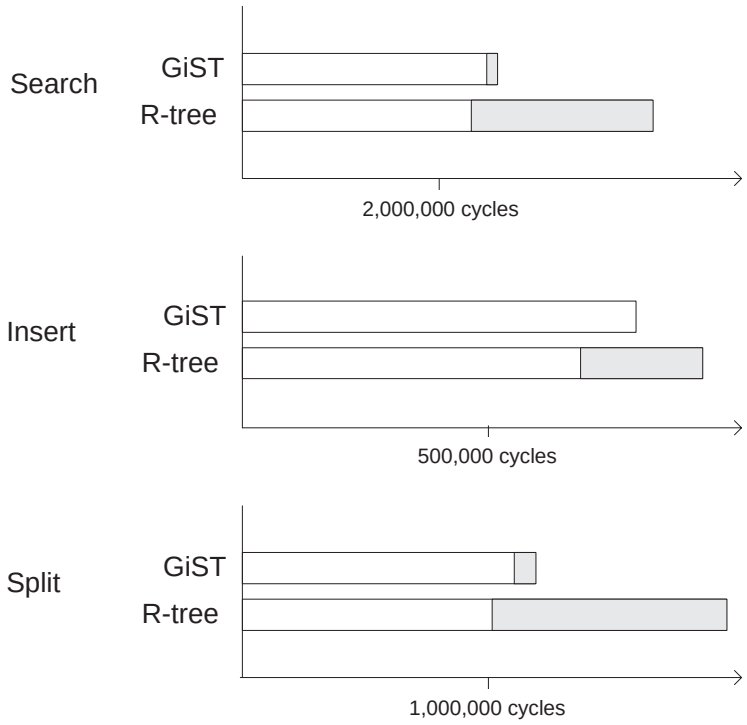


Figure 3.6: Comparison of GiST-based and built-in R-trees (shaded portion indicates UDF call overhead).

scenario described in this section. In other ORDBMSs, a UDF call might involve a context switch and interprocess communication, which would make it far more expensive.

The total execution times, excluding time spent in operating system calls, for three different operations are shown in Figure 3.6. The operations are: a query with a rectangle containment qualification that retrieves only a single rectangle, but traverses 78 pages in the tree; an insertion operation; an insertion 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. This performance loss is an artifact of the implementation of the built-in R-tree; if the external interface of the built-in R-tree (exposed through the opclass) were redesigned to mimic the extended GiST interface, the number of UDF calls in both R-tree implementations would be roughly the same. 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 in the test scenario, this cannot make up for the large number of UDF calls.

Overall, the overhead introduced by GiST-based AMs is limited to setup. The magni-

tude of this overhead depends on the particular AM (a datatype-extensible B-tree would only need to initialize a single descriptor for the comparison function, resulting in even less setup time than the built-in R-tree requires), but in comparison to the execution cost of a single index operation, this setup cost is small and could probably be reduced even further by more aggressive descriptor caching.

3.3 Summary and Conclusion

This chapter outlines the performance and functionality requirements of an AM extensibility architecture in a commercial ORDBMS and shows how GiSTs can be adapted to meet those requirements. The extended GiST interface addresses all three issues identified at the beginning of this chapter:

Datatype Extensibility By separating the server-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 Safety 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 extended interface is that it hides the details of the page layout from the GiST core, allowing the AM extension to

customize the page layout via a simple and externally visible page management interface. This additional flexibility does not necessarily come at the price of increased implementation complexity for the AM developer, because page layout functionality can be separated into libraries and re-used across AM extensions.

In summary, a GiST-based AM extension architecture has a number of advantages over the current state-of-the-art iterator-style AM extension interfaces:

- AM development is greatly simplified. The abovementioned B-tree and R-tree extensions were implemented 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 details. Such reuse is normally not possible when custom locking and logging protocols are intermixed with page layout functionality.

- 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.

- Despite part of an AM residing outside the server, the AM is still fully integrated into the DBMS and offers the same set of features as a built-in AM: integrated storage management, backup and recovery, support for SQL isolation semantics and a high degree of concurrency. Moreover, the performance of GiST-based AMs is as good or better than that of built-in AMs.

A crucial aspect of the engineering advantages of a GiST-based AM lie in the integration of concurrency and recovery protocols into the GiST framework, which is the topic of the next chapter.

Function	**Input Parameters**	**Output Parameters**	**Purpose**
insert()	predicate, heap pointer, adapter		insert (*predicate*, *heap pointer*) entry on page
remove()	set of slots, adapter		remove slots from page
update_pred()	slot, predicate, adapter		update predicate part of entry at given slot
begin_scan()	query qualifier, adapter	state pointer	transform query qualifier into AM-specific format
search()	query qualifier, state pointer, adapter	set of matching slots	return slots of page items matching query qualifier
end_scan()	state pointer, adapter		deallocate data allocated in *begin_scan()*
get_key()	slot, adapter	key	extract predicate of entry at given slot
pick_split()	original SP, new key 1, new key 2, adapter	set of slots destined for right node, left SP, right SP	determine which entries of a page are to be moved to the new right sibling page, taking into account that the given new keys get added to the two pages, and compute the SPs for the resulting left and right page
find_min_pen()	new predicate, adapter	slot	determine slot of internal page entry with smallest insertion penalty
union()	SP, new predicate, adapter	SP	compute union of an SP and a predicate
eq_op()	adapter	operator number	returns AM-specific equality operator number

Table 3.1: Summary of the extended GiST interface.

Chapter 4

Concurrency and Recovery for GiSTs

Multiuser access to AMs is a crucial feature of traditional RDBMSs; without it, OLTP applications would be practically impossible, and the staggering throughput numbers of modern systems on standard industry benchmarks such as TPC-C would be unthinkable. For next-generation database applications, such as GIS or multimedia content management, it is sometimes argued that datasets are more static and hence the traditional requirements of concurrency and recoverability do not apply. This view is debatable for two reasons: a) most application scenarios involve at least a small degree of update activity; b) online information systems have potentially very large user communities, and quiescing query activity in order to perform batch updates can be impractical.

More specifically, we can distinguish three related requirements: 1) support for concurrent search and insert and delete operations; 2) support for the degrees of transactional isolation offered by the query language of the DBMS; 3) integration with the recovery

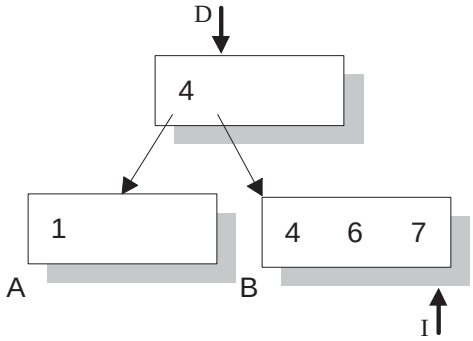
mechanism that guarantees the integrity of the DBMS's data. Most research on novel access methods ignores these issues, and algorithms for their support are scarce. In addition, experience with B-tree implementations and IDS/UDO's built-in R-tree has shown that these features cause much of the complexity of actual implementations and account for a major fraction of the code.

In this chapter I describe algorithms that address each of the three requirements in the context of the GiST data structure. These algorithms can be implemented fully within the template index operations, only using the lock and log manager of the DBMS server and the external GiST interface described in the preceding section. As a result, the externally-implemented AM is fully shielded from the details of concurrency and recovery and the corresponding components of the DBMS server. Although the algorithms are presented in the context of the GiST, they can also be applied individually to any particular access method that complies with the GiST structure. This is not a very restrictive requirement, because it only excludes access methods that are either not proper trees (e.g., the hB-tree, as described in [LS90]) or that have other structural peculiarities (for instance, R^+-trees, described in [SRF87], replicate leaf entries). Although the GiST structure is similar to that of a B-tree, it generalizes the B-tree structure in a way which makes most of the extensively researched concurrency control techniques for B-trees inapplicable in the less restricted context of the GiST structure.

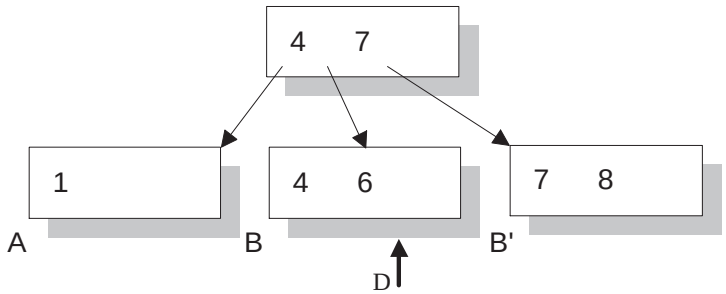
The literature distinguishes two types of concurrency control in the context of AMs: *physical* concurrency control allows multiple search and update operations to be active in

the index simultaneously, even during structure modifications like node splits and deletes; *logical* concurrency control, on the other hand, ensures transactional isolation. The solution for physical concurrency control in GiST that I present here is based on an extension of the link technique initially developed for B-trees and completely avoids holding page latches during I/Os. I achieve transactional isolation with a hybrid locking mechanism, which combines traditional two-phase locking of data records with predicate locking. More generally, I show how the structure of an access method affects the available choices of concurrency control techniques, and I explain why existing B-tree techniques cannot be directly applied to more general tree structures. I also address practical issues such as support for unique indices, and I describe implementation details of the protocols in the IDS/UDO GiST implementation.

The rest of this chapter is organized as follows. Section 4.1 extends the GiST structure for concurrent access. After these preliminaries, Section 4.2 explains the algorithms for index lookup, key insertion into non-unique and unique indices, and key deletion. Section 4.3 outlines the design of the hybrid locking mechanism and compares it with other locking approaches. Logging and recovery are described in Section 4.4. Section 4.5 discusses a variety of implementation issues and describes specific details of the IDS/UDO GiST implementation. Section 4.6 discusses related work and some of the implications of the structure of an access method for concurrency control techniques; it also explains why most of the prior work on B-trees cannot be directly applied in the GiST context. Section 4.7 concludes this chapter with a summary.



An insert operation of key 8 and a delete operation of key 7 are executing concurrently in the tree. The insert operation reaches leaf B, which is full and must be split. The delete operation examines the parent and discovers that it must also examine leaf B (arrows indicate scan positions).



The delete operation reaches leaf B after it was split and incorrectly concludes that key 7 does not exist.

Figure 4.1: Illustration of incorrect interleaving of key search and node split.

4.1 GiST Extension for Physical Concurrency

When multiple operations are carried out on a GiST in parallel, their interactions may be interleaved in a way that leads to incorrect results. An example for a B-tree is shown in Figure 4.1, where a node split changes the location of some keys, which causes a concurrently executing search operation to miss them.

In order to avoid such situations, we apply the B-link tree strategy [LY81] of adding a link between a node and its split-off right sibling. All the nodes at each level are chained

together via links to their right siblings; the addition of this *rightlink* allows operations traversing this node to compensate for missed splits by following the rightlink. Of course, rightlinks cannot be followed blindly every time a node is traversed, or parts of the tree would be scanned multiple times. For the link strategy to work, a traversing operation must be able to (1) detect a node split and (2) determine when to stop following rightlinks (a node can have split multiple times, in which case the traversing operation must follow as many rightlinks as there were node splits).

For B-trees, both of these questions can be answered by examining the keys in the node, since the key domain has two restrictions: it is ordered and the keys are partitioned across the leaves. In particular, a comparison of the search key and the highest key on the node will tell a traversing operation if a node has split and if a right sibling might contain entries intersecting the search range. GiSTs do not impose these restrictions on the key domain, which means that the B-link strategy by itself is insufficient. We therefore need to augment the tree structure so that we can reconstruct the split history of a node, starting from the time we obtained a pointer to it. This can be achieved by assigning a sequence number to each node in addition to the rightlink. This node sequence number (NSN) is taken from a tree-global, monotonically increasing counter variable. During a node split, this counter is incremented and its new value assigned to the original node; the new sibling node receives the original node's prior NSN and rightlink. Figure 4.2 illustrates the extended tree structure and shows how a traversing operation can take advantage of the NSNs and the counter variable to detect splits and determine how many times a node was split. In

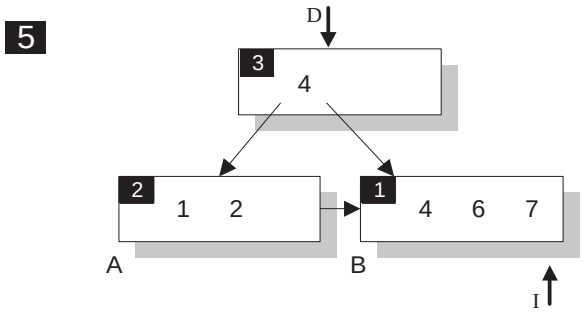
general, a traversing operation can now detect a split by memorizing the global counter value when reading the parent entry and comparing it with the NSN of the current node. If the latter is higher, the node must have been split and the operation follows rightlinks until it sees a node with an NSN less than or equal to the one originally memorized. A node with an NSN less than or equal to the memorized one cannot have been split after the parent has been visited and therefore its rightlink need not be traversed; since nodes are always split to the right, this node must demarcate the end of the rightlink sequence that the traversing operation has to follow.

4.2 Algorithms for Search and Update Operations

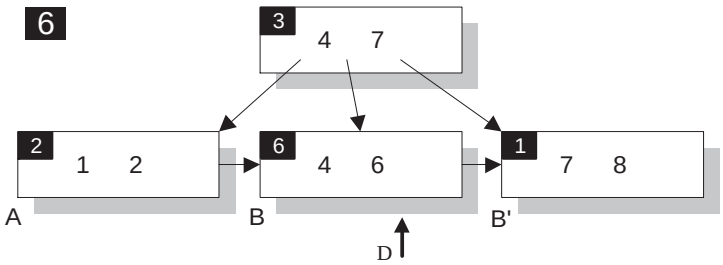
The algorithms presented in the following four subsections implement concurrent search, insertion, deletion and unique-index insertion operations in GiSTs extended with sequence counters. For now, the presentation ignores issues of transactional isolation, which are dealt with in detail in the following section.

4.2.1 Search

The search operation returns the set of leaf entries satisfying the search predicate. The function `search()`, shown in Figure 4.3, implements the search operation. Calls to the GiST interface functions of Chapter 3 are shown in italics. For clarity, the page to which the call applies is shown explicitly, although in the actual implementation this would be



The example B-tree GiST of Figure 1 has been extended with NSNs and rightlinks. Again, an insertion of key 8 and a deletion of key 7 are executing concurrently. The insertion is about to split leaf B. The delete operation records the value of the global counter (5) before it goes to B.



The insertion splits B, increments the global counter value from 5 to 6 and assigns the new value to B. The new sibling receives the old NSN and rightlink of B. When the delete operation reaches leaf B, it determines that B must have split, because the recorded global values (then 5) is less than the current NSN (6) of B. Likewise, when it follows the rightlink to B', it can determine that this is the end of the "split chain" it has to traverse, because the NSN of B' (1) is less than or equal to the recorded global counter value (5).

Figure 4.2: Example of the extended tree structure and how it avoids incorrect interleaving.

implicit.

The search operation starts by pushing the root pointer on the stack. It then repeatedly pops an entry off the stack, visits the corresponding node and pushes all entries with matching predicates on the stack. This results in a depth-first traversal of the tree and is repeated until the stack is empty. While examining a node for matching entries, we hold it latched[1]

[1]Latches differ from locks in two aspects: (1) latches, like mutexes, are addressed physically and can

to prevent concurrent modifications. The latch is released as soon as we are done with the node and before going to the next node, avoiding latch-coupling across I/Os.

To recognize node splits, we timestamp every page pointer stored on the stack with the value of the global counter as of the time the page pointer was read.[2] When the stack is used to visit a node, the recorded counter value is compared with the node's NSN. If the latter is higher, the node has been split, and we add the rightlink pointer of the node together with the originally recorded counter value to the stack. This guarantees that the right siblings split off the original node will be examined later on.

Although the `search()` function shown here is specific to range queries, different types of traversals, corresponding to different types of search states (as discussed in [Aok98]), can be accomodated easily. The reason is that the correctness of the latching protocol does not depend on the traversal order, because nodes are latched one at a time and split detection is not affected by traversal order.

4.2.2 Key Insertion

Key insertion is carried out in several phases:

---

therefore be set and checked much more efficiently than locks, which are usually organized into a hash table; (2) existing latches are not checked for deadlock by the DBMS, which requires the system-internal latch logic to make sure its usage pattern is deadlock-free. Latches are commonly used to synchronize access to physical "objects" of the DBMS such as buffer pool frames. Also, latches do not interact with locks, so that it is possible to latch the buffer pool frame holding a particular node while some other transaction holds a lock on the node. See [MHL+92] for more details.

[2] When reading the value of the global counter while examining a page (line "nsn = global NSN" in Figure 4.3), we need to synchronize access to the global counter in order to avoid conflicts with node splits, which update the counter. The implementation of this synchronization depends on the implementation of the global counter itself, which is discussed in more detail in Section 4.5.

```
search(search-pred)
   nsn = global NSN;
   push(stack, [root, nsn]);
   while (stack is not empty)
      [child, expected-NSN] = pop(stack);
      latch(child, S-mode);
      if (expected-NSN < NSN(child))
            push(stack, [rightlink(child), expected-
NSN]);
      end
      match-slots = search(child, search-pred);
      if (child is leaf)
         for each slot s in match-slots:
          add ( get_key(s), s.ptr) to search result set;
      else
         nsn = global NSN;
         for each slot s in match-slots:
            push(stack, [s.ptr, nsn]);
      end
      unlatch(child);
   end
```

Figure 4.3: Concurrent GiST search algorithm.

1. The insertion operation begins by traversing the tree along a single path from the root to a leaf, following branches with the lowest insert penalty.

2. If the chosen leaf would overflow as a result of the key insertion, the leaf must be split beforehand, which in turn might cause recursive splitting of ancestor nodes.

3. If no split is required, and if the insertion of the new leaf entry will change the leaf's SP, the updated SP is propagated to the parent entries by backing up the tree, until an ancestor node is encountered whose SP does not need to expand.

4. The new *(key, RID)* pair is inserted on the leaf.

Note that in contrast to B-trees, an insertion operation may back up the tree for two reasons: splitting a node requires the installation of a new parent entry, and expanding a leaf's SP requires the adjustment of parent entries. The latter step is missing in B-trees. Figure 4.4 shows the insertion algorithm.

The function *insert()* first locates the target leaf, then recursively splits or updates its SP, if necessary, and finally inserts the new *(key, RID)* pair. The function *locateLeaf()* traverses a single path, starting from the root, to locate the target leaf of the insertion. It detects node splits in the same manner as the search operation, but compensates for missed splits differently. Instead of following rightlinks, it returns to the lowest unsplit ancestor in order to re-establish the minimum-penalty path from that ancestor. The reason for this deviation is that the *findMinPen()* extension function can only be applied to a parent node to find the minimum-penalty child slot. To compensate for missed splits by following rightlinks, *findMinPen()* would need to be applied to each of the child nodes (original and split-off) directly.[3]

The eventual path to the leaf is recorded on a stack, which is used subsequently to ascend the tree for necessary structural modifications. The function *splitNode()* carries out a recursive split operation, including updating of affected SPs. When backing up the tree to update parent entries it can also become necessary to recognize splits of ancestor nodes that have taken place since the nodes were initially traversed on the way down to the leaf. In this instance, it is not even necessary to compare NSNs. If a parent node does not contain

[3]The reason why a search operation cannot adopt this repositioning strategy is explained in Section 4.2.3.

the child's pointer anymore, it must have been split and the search for the child's pointer is continued in the right sibling.

When arriving at the root node while backing up the tree, the insertion operation can be confronted with a missed root split that took place after we initially obtained the root pointer and began tree traversal. In order to update the additional level(s) created by the missed root split(s), we need to augment the stack that records the path with the corresponding nodes of those levels. Those nodes are determined by doing an exhaustive breadth-first traversal, starting with the now current root, until the old "root" is encountered. This has no detrimental effects on average performance, because it is an extremely rare event and there will usually only have been a single level added, which keeps the number of nodes to traverse fairly small.

Once the split is done, we can release the latch on the parent and the new right sibling (assuming for the sake of simplicity that the subsequent insertion will go on the original node). The function *updateSP()* updates a node's parent entry, if the SP has changed. This is carried out recursively, and may even result in node splits of ancestors, because an SP update may expand the parent entry and overflow the parent node.

4.2.3 Key Deletion

In order to delete a key from a tree, it is first necessary to locate the key on a leaf, which is equivalent to a search operation with an equality predicate. The item on the leaf will not be physically deleted but only marked deleted, i.e., a logical deletion will be performed.

The physical presence of this deleted key is useful for isolation and recovery purposes (as explained further in Sections 4.3 and 4.4).

Garbage Collection of Deleted Leaf Entries

A leaf entry may only be physically removed and the associated SPs—that of the leaf and those of the leaf's ancestors—shrunk after the deleting transaction has been committed. For that reason, those physical update operations must be performed as garbage collection by other operations which happen to pass through the affected nodes. Garbage collection for a single leaf node is performed as a *node reorganization*, which removes all those entries from a leaf which have been marked deleted and for which the initiating transactions have committed.[4] As a result of a node reorganization, the SP of that node may have changed, which can then be propagated to the parent nodes. Note that when removing entries from a node, the SP of that node should actually shrink, i.e., cover less of the data space. Since the *union()* extension function is implemented externally, a more robust assumption is that *union()* will change the SP in some way in response to an entry removal, not that it necessarily shrinks it.

Node Deletion

A node reorganization may also leave a node completely empty, in which case it should be removed from the tree so that it can be reused for later node splits. Unfortunately, it is

[4] [Moh90b] shows how this can be done cheaply in a WAL environment. Essentially, if the page's LSN is less than the first LSN of the oldest active transaction, then all entries must belong to committed transactions and no additional locks have to be tested.

impossible simply to remove the node's parent entry and retire the node, because ongoing tree operations might still have pointers to the node on their stacks.

Such an ongoing operation could not be completed correctly if it would visit the node after it has been deleted *and* reused for two reasons: (1) if the node had been split before being deleted, a search operation may still need to traverse its rightlink; (2) the node could be reused for the split of another node that one of the search operations has already visited, in which case this operation would traverse the same subtree multiple times and return duplicate results.

To avoid this situation, the reuse of a deleted node must be delayed until there are no more search operations targeting that node. For B-trees, the *latch-coupling* protocol is a popular way to achieve this (see for example ARIES/IM [ML92]): during tree descent, the latch on the parent node is not released until the latch on the child node has been obtained. To delay node reuse, ARIES/IM's delete operation first marks the empty node as a delete target, then releases the latch on that node and subsequently obtains an X-latch on the parent node. At that point, there can be no more search operations trying to access the marked node. Search operations that end up visiting the marked node suspend themselves until the node delete operation is over and then recover by repositioning within the tree.

While latch coupling is very effective for B-trees, it is practially impossible in the GiST context. In its simplest form, a search operation would hold a parent node S-latched while traversing all of its qualifying subtrees, which would seriously impact the degree of concurrency with regard to update operations, especially if SP updates after insertions are

frequent.

Alternatively, the search operation could release the latch on the parent after visiting the first child and later on reposition itself within the parent node, carefully avoiding those regions of the key space that have already been traversed. Repositioning is unfortunately not possible in the GiST context, because the key space is not necessarily partitioned and concurrent changes in the subtree make it impossible to keep track of which regions of the key space have been traversed (see Figure 4.5 for an illustration). Hence, latch-coupling and repositioning cannot be used to achieve high concurrency in any non-partitioning tree structure.

We therefore must look for a different way to delay the reuse of a delete node while active tree operations still hold *direct* or *indirect* pointers to that node. A direct pointer is a node pointer recorded in some operation's stack. A node pointer indirectly references some of that node's right siblings, if the stack entry also contains an NSN which would lead the operation to cross rightlinks to them. To make node deletion operations aware of existing direct pointers, an operation places a *signaling* lock on a node when it pushes a pointer to that node onto the stack. The signaling lock can be implemented as a short-duration S-mode lock on the node (released at the end of the index operation), so that other insert and delete operations are not prevented from physically accessing and modifying that node. A node deletion checks for signaling locks by trying to acquire an X-mode lock on the respective node. To extend the deletion protection to those nodes that have been split off and are therefore referenced indirectly, a node-splitting transaction copies the list of signaling locks

placed on the original node to the new right sibling (alternatively, splits could be blocked, lowering both implementation complexity and the degree of concurrency). A signaling lock is released as soon as the operation that sets it visits that node. The only exception to that rule is the signaling lock set on the target leaf of an insert or delete operation. The lock has to be retained until the end of the inserting or deleting transaction for recovery purposes, otherwise recovery-relevant parts of the link chain would be interrupted (details are given in Section 4.4).

4.2.4 Key Insertion into Unique Indices

When inserting a duplicate into a unique index, database semantics require that the insertion operation return with an error message. To insert into a unique index, we therefore first perform a search operation, followeed by an insertion. The search operation preceding the insertion verifies that no duplicates will be introduced. If it finds the new value in the tree, it returns an error condition. If the inserted value is not found in the tree, the phases of the regular insert operation are carried out: the leaf is located and after key insertion the parents' entries are updated, if necessary. In order to avoid a race between two insertion operations of the same value, the search phase must be executed in repeatable-read mode, which is the topic of the next section.

4.3 Transactional Isolation

The highest degree of transactional isolation is defined as Degree 3 consistency or repeatable read isolation [Gra78]. It implies that if a search operation is run twice within the same transaction, it must return the exact same result, even if the result set is empty or the operation results in an error. This requires regulation of the overlapping data access of read and update operations of competing transactions, either by restricting access to data or by replicating the data. This section presents an overview of mechanisms that guarantee repeatable read isolation by restricting data access; for approaches that employ replication, see for example [MPL92, Stü95].

The simplest solution would be to lock all involved tables for the duration of the entire transaction. Unfortunately, this leads to an unacceptably low degree of concurrency. DBMSs try to avoid this by accessing the tables through index structures and explicitly locking only as much as needed to guarantee repeatable read isolation. One part of what needs to be locked is the set of data records returned by the search; this will prevent modification or deletion of the data records in the result set. This is not sufficient, because it is also necessary to lock those regions of the search range for which no data records were returned, in order to prevent subsequent insertions into the search range. These insertions are known as phantom insertions [EGLT76] and result either from newly-inserted data records or rolling-back deletions of data records.

In the following discussion we assume the availability of a standard lock manager that supports locks of varying modes and durations. Furthermore, we assume to be dealing with

secondary indices, which store the RIDs of data records that are stored separately in tables.[5] We require that all insert and delete operations X-lock the data records they update and all search operations S-lock the data records they retrieve (except for those running under degree 0 consistency, which allows reading of uncommitted data). Lower isolation levels (degree 1 and 2) can be implemented entirely with data locks acquired in this manner, with the duration of S-locks varying according to the chosen isolation level (instant duration for degree 1 and transaction duration for degree 2). The data locks are also necessary for degree 3 search operations to protect the result set from modification. It is not sufficient to protect the result set of a search operation by only protecting one particular access path to it, because conflicting data record updates may not need to use this access path. For that reason, data record locks (or locks of larger granularity on the base table) are mandatory. In the rest of this section we will only be concerned with phantom protection.

Note that the practice of performing *reinsertions* to delay node splits, which was first introduced for the R*-tree [BKSS90] and was also adopted in the original GiST design of [HNP95], is problematic when insertions and searches are performed concurrently. When performing a reinsertion, a node split is delayed by removing some of the entries and re-inserting them into the tree, which can improve clustering. By relocating entries to different nodes of the tree, a search operation might run across a matching entry more than

[5] For the sake of simplicity and to avoid confusion, we limit the discussion here to secondary indices. Note that the techniques presented here are equally applicable to primary indices, which store the data records themselves in the leaf level of the index. The difference between primary and secondary indices from the perspective of concurrency control is that for secondary indices, the RID of the data record residing in the base table is an immutable ID; in primary indices, the RID of the index-resident data record can change due to node splits, and an immutable surrogate needs to be formed (e.g., the hash value of the key or a synthetic ID).

once; this would result in duplicates in the result set, which is not allowed at any of the isolation levels. For that reason, reinsertion is not part of the insertion algorithm described in Section 4.2.2.

We can identify four evaluation criteria for isolation mechanisms:

Computational Efficiency We are generally concerned with execution efficiency, and would like to keep the overhead required by the locking protocol as low as possible. The overhead consists of computing lock names, setting and checking locks in addition to the data locks, etc.

High Degree of Concurrency We would like the obtained degree of concurrency to be as close as possible to the maximum degree of concurrency. The maximum is determined by the overlap between qualification predicates of search operations and the set of target objects of conflicting update operations. The actual degree can be lower due to limitations of the isolation mechanism.

Gradual Expansion of the Locked Region Instead of taking place instantaneously, search operations usually accumulate their result set over a period of time, in particular if they are executed within a SQL cursor from an external application, which may have significant delays for user interaction, etc., and which may terminate before the full search range is exhausted. It is desirable to expand the locked region only gradually, in lockstep with the data space “covered” by the search operation, so as to maximize concurrency during the lifetime of the search operation. Note that this form of

concurrency is independent of the previous one.

Implementation Complexity Implementation complexity translates directly into engineering cost, which should be kept as low as possible. Although it seems irrelevant from an academic perspective, coding and debugging concurrency control protocols is notoriously difficult and time-consuming. Very complex designs are prone to bug-ridden implementations or may require a prohibitive engineering effort.

4.3.1 A B-Tree Solution: Key-Range Locking

One solution to the phantom problem in an ordered key domain is a technique called key-range locking,[6] which works as follows. Each data item with key k_i is treated as a surrogate lock name for the key range $(k_{i-1}; k_i]$. For a scan with a given search range, we retrieve and two-phase lock all the data records[7] within the range and we also lock the first data record *past the right end of the range*; this is typically done with the help of a B-tree index. As a result, all the key ranges $(k_{i-1}; k_i]$ intersecting the search range are locked. Before a leaf entry insertion, we check the data record to the right of the insertion point for existing locks, thereby making sure that the "gap" into which we are inserting is not locked by any scan.

Essentially, key-range locking requires the *ordering property of the key domain* and the *correspondence between logical key order and physical order of (key, RID) pairs* within and

[6] Other terms are key-value locking [Moh90a], or next-key locking [GR93].
[7] As in the data-only locking approach of ARIES/IM [ML92].

across nodes. This allows conflicting search and insert operations to agree on a sequence of physical data records as a surrogate for a logical search range.

With respect to our four evaluation criteria, we observe the following:

Efficiency Surrogate lock names are easy to compute (they are either the returned data records or the data record right next to the last one in the sequence); additionally, a logical lock range has been transfered into a sequence of purely physical locks, which can be set and checked very efficiently by the standard lock manager. Furthermore, only the final lock in the search range represents isolation overhead, because all of the preceding ones would have been acquired anyway.

Concurrency A drawback is that the degree of concurrency is not maximal, because the final lock might cover a substantially wider range than is required by the search predicate, depending on the key value. The reason is that the "density" of the lock predicates depends on the data stored in the index: range predicates depend on adjacent data items and cannot be chosen irrespective of the leaf-level data. This becomes a problem when a search operation's range qualification ends in between two leaf keys. The corresponding lock predicate will extend past the search range, which reduces the degree of concurrency. It is also problematic when a delete operation locks a range predicate (i.e., the next key) in order to protect the interval covering the key. In principle, only a single value needs to be locked, and the larger the interval, the more concurrency is restricted. Note that logical deletion does not suffer this drawback, since the deleted value is physically retained on the page and can serve as a surrogate

key. (See [Lom93, Moh90b, Moh95] for the details.)

Gradual Expansion The lock range is expanded gradually, because additional intervals of the data space are only locked when the next key in the key range is read.

Complexity Key-range locking is relatively straightforward and can be implemented with a standard lock manager, resulting in moderate implementation complexity. Some care must be taken when obtaining locks while holding page latches. This can lead to deadlock, which the lock manager cannot detect and which therefore must be avoided. The typical approach of releasing page latches, requesting the lock, then re-acquiring the page latches does not result in overly complex code, because repositioning a scan (or any traversal operation) is comparatively simple in B-trees.

Overall, the key-range locking protocol, even though it is not optimal, is very effective, and we will see that it is difficult to produce something of equal quality in the more general GiST case.

4.3.2 Predicate Locking

Key-range locking is not directly applicable to GiSTs, because we cannot assume an ordered key domain and therefore cannot count on finding a physically contiguous sequence of leaf entries as a surrogate for the possibly multi-dimensional search range. Two alternative techniques, predicate locking [EGLT76] and its more efficient relative precision locking [JBB81], circumvent this problem. The search operations register their search

predicates in a tree-global table, so that insert and delete operations can check for conflicting concurrent search operations. Symmetrically, insert and delete operations register their keys in a tree-global table, which is checked by search operations for conflicts with the search predicate. An operation can only start traversing the tree after it verifies that there are no conflicting predicates. A predicate lock is retained until the end of the transaction.

When applying our evaluation criteria, we see that predicate locking has two disadvantages—its efficiency and degree of gradual expansion—in comparison to key-range locking:

Efficiency As a result of not using surrogate keys that correspond to physical data items, predicate evaluation is potentially expensive—in fact the cost cannot be pre-determined—and requires user code. Moreover, every lock acquisition goes through the entire list of potentially conflicting locks, making this locking protocol more costly as the number of concurrent operations increases.

Concurrency The accuracy of the lock predicates is maximal, because this technique does not rely on surrogate lock names, but instead uses the search predicate directly.

Gradual Expansion A search operation must set its predicate lock before the index is accessed and any data records are retrieved. Unlike key-range locking, the locked key range is not expanded gradually, which can be very detrimental to concurrency if the search is done as part of an interactive cursor. An even worse scenario is a search qualification whose “coverage” of the data space also depends on the search state, e.g., a nearest-neighbor search (the qualification only specifies a center point, while

the search state—the number of points retrieved so far—determines the area that has been examined by the search operation). In this case, an update operation cannot determine whether it conflicts with the ongoing search and would conservatively need to suspend itself while the search is ongoing. In other words, any search operation of this nature would lock the entire table.

Complexity The implementation complexity of this approach is moderate. All lock requests are issued before tree traversal begins, which eliminates complexity due to lock/latch problems. In addition to the standard lock manager, a predicate lock manager is needed, which can be patterned after the standard lock manager.

4.3.3 A Hybrid Locking Protocol

Instead of resorting to pure predicate locking, we can use a hybrid mechanism, which synthesizes two-phase locking of data records with predicate locking. The underlying idea is to rely on two-phase locking for existing data records and augment that technique with a restricted, more efficient version of predicate locking for phantom avoidance. In the hybrid mechanism, *existing* data records that are scanned, inserted or deleted are still protected by the two-phase locking protocol. In addition to data locks, search operations acquire predicate locks to prevent phantom insertions. That way, only insertion operations, which *add* data, check for conflicting predicates; search and delete operations simply check for conflicting data locks. Furthermore, the predicate locks are not registered in a tree-global list;

instead, they are *directly attached to the index nodes they apply to.* Predicate attachments[8] are performed so that the following invariant is true at all times: if a search operation's qualification predicate matches a node's SP, the qualification predicate must be attached to the node. An insert operation can therefore limit itself to checking *only* the predicates attached to its target leaf. A delete operation must be carried out as a logical delete, so that the respective leaf entry is not physically deleted but is only marked as deleted (see [Moh90b] for a discussion of this technique in the context of B-trees). The physical presence of the leaf entry and the lock on the corresponding data record ensure that search operations will block on the deleted entry until the delete is committed.

The advantage of this hybrid approach over pure predicate locking is that two-phase locking of data records is still used for every *existing* data record, eliminating the need for search and delete operations to check for conflicting predicates. Only insert operations, which *add* to the set of existing records, check for search predicates; furthermore, they only check the set of predicates attached to their target leaves, which is typically substantially smaller than the tree-global set of predicates.

This isolation mechanism requires augmentation of the basic algorithms with testing and setting of data and predicate locks. For example, search operations attach their predicates to the nodes as they are visited and the predicates and their node attachments are only removed when the owner transaction terminates. Since the tree structure changes dynamically as nodes split and SPs are expanded during key insertions, the predicate attachments

[8]Note that these predicate attachments can be implemented through an in-memory data structure, similar to a lock table; a physical modification of the index page in question is not necessary.

must adapt to the structural changes. We distinguish two cases that require existing predicate attachments to be replicated at other nodes. The first case is a node split, which creates a new node whose SP might be consistent with some of the predicates attached to the original node. The invariant is maintained by attaching those predicates to the new node. The second case involves the expansion of a node's SP, causing it to become consistent with additional search predicates. Again, those predicates need to be attached to the node; they are found in the ancestor nodes accessed during the SP update phase of the insertion operation, and they are "percolated" to all the child nodes whose expanded SP is consistent with the predicate. The following paragraphs describe the details.

Search While traversing the tree in a top-down fashion, the search operation S-locks each data item in the result set and attaches its predicate to every node that it visits. The attachment takes places immediately before visiting a node (it must not take place when the node is held latched and it should not take place much earlier because the lock limits concurrency).

Similar to B-trees, there is an opportunity for undetectable deadlock if the latches are not released when blocking on a data lock. To avoid this situation, the latches must be released before blocking on the data lock and then reacquired when the search operation is unblocked. If the latched leaf has been split in the meantime, we also visit the split-off leaf nodes guided by the node's original NSN. When revisiting a leaf, we have to make sure that leaf entries are not included in the result set multiple times, even if the leaf has been mod-

ified or split in the meantime. We can keep track of which entries of a particular leaf have already been processed by including their *data* RIDs (not the entries' RIDs themselves) in a list. Note that this is only possible with leaf entries, not with entries of internal nodes; the discussion in Section 4.2.3 makes this clear.

Insertion The phases of the insertion operation as described in Section 4.2.2 are augmented as follows:

1. As part of the two-phase locking protocol, the new data record is X-locked *before* the tree insertion is initiated.

2. If the target leaf has search predicates attached that conflict with the new key, the operation blocks on those predicates until their owner transactions commit.

3. If a node split is necessary, we attach to every new node all of the original node's predicates that are consistent with the new node's SP.

4. The SP updates are performed atomically, i.e., all involved ancestor nodes are held latched simultaneously,[9] and the updated SPs are only installed in the ancestor nodes if the new key does not conflict with any of the search predicates attached to the ancestor nodes. During the SP update phase, we "percolate" search predicates from ancestor to child nodes, if the ancestors' predicates are consistent with the child's updated SP.

[9] Note that this form of latch-coupling does not have the same detrimental effect on the degree of concurrency as latch-coupling during tree *descent*; at the point where the SP updates are performed, the involved ancestor nodes are most likely still cached in the buffer pool and therefore no lengthy I/Os are necessary.

5. If any conflicting search predicates are detected, we block on those until their owner transactions commit. If no conflicting predicates are detected, we proceed to insert the new *(key, RID)* pair on the target leaf.

If the insertion detects any conflicting search predicates along its update path, it must be suspended until the conflicting search transactions are terminated. When resuming the insertion, the leaf originally located might have been split. This again is recognized by comparing the memorized NSN with the leaf's current NSN, followed by repositioning in the parent and retraversal, as is done in *locateLeaf()* in Figure 4.4. The structural modifications (node splits and SP updates prior to inserting item on leaf) then need to be reattempted. This repetitive process may result in starvation, which can be dealt with in a relative simple and crude way (after a number of attempts, the inserter acquires an X-lock on the index for the duration of the insertion operation).

Deletion Delete operations do not set or check predicate locks, and therefore the algorithm described in Section 4.2.3 need not be augmented. We perform logical deletion to ensure that repeatable-read search operations have an opportunity to be suspended when they encounter such a key.[10] For the same reason, the delete operation must not shrink parent entries before the deletion is committed, because this would remove the path to the key and make the key inaccessible for concurrent search operations. A subsequent rollback

[10] In B-trees, the delete operation could physically remove the key, but would have to leave a lock on the next key. This is not applicable to GiSTs for the reasons already mentioned in Section 4.3. Even in B-trees, logical deletion is preferable if increased concurrency is important (see [Moh90b] for further discussions of logical deletions). In fact, the index manager of DB2/MVS V4 has adopted it for that reason.

of the deletion, including expansion of the ancestor SPs, would then cause a phantom to appear.

Insertion into Unique Index The search operation for the new key value preceding the actual insertion must be carried out like a standard search operation with a qualification of the form "$= key$", executed in repeatable-read isolation mode. If two insert operations happen to be interleaved in a way that they both miss each other's new key during their search phases, the predicates will ensure that each operation blocks on the other's predicate. This will result in a deadlock, which can be resolved in a standard manner by the lock manager. Once the insert operation is finished, the predicates left behind from the search phase can be released. If the new key value is already present in the index, the search operation requests an S-mode lock on the corresponding data record. Note that in this case no predicate locks need to be left behind by the search operation; the lock on the data record alone is sufficient to ensure repeatability of the error condition and availability of the original data item for subsequent searches.

Garbage Collection Shrinking an SP as a result of removing logically deleted entries from a page cannot cause the SP to become consistent with additional search predicates. From that perspective, recursive SP updates performed during garbage collection need not replicate search predicates from ancestors to children. As pointed out in Section 4.2.3, the GiST interface function *union()*, which implements the SP re-computation, may actually *expand* the SP (it may do this as a result of a design or coding error or on purpose). For

robustness, we therefore percolate search predicates, just like in the SP update phase of an insertion.

Applying our four evaluation criteria, the hybrid technique can be characterized as follows:

Efficiency The hybrid protocol is more efficient than pure predicate locking, because only insert operations need to check predicates. Furthermore, the number of predicates to check should on average be less, since only the predicates attached to the leaf need to be considered, but the actual number relative to the total number of active search operations is impossible to predict.

Aside from that, structural modifications now incur an overhead, because they need to replicate predicate locks in order to maintain the invariant. Such overhead is not present in the key-range locking technique or in pure predicate locking.

Concurrency Just like regular predicate locking, the accuracy of the lock predicates is perfect and concurrency is not diminished. Note that the use of data record locks does not impede concurrency, since these locks are not used as surrogate lock names for intervals of the data space.

Gradual Expansion Unlike pure predicate locking, the lock range is expanded gradually, although not to the same extent as in key-range locking. The reason is that predicates are attached to the visited nodes top-down, starting with the root.[11] This can block

[11] We cannot attach predicates bottom-up, i.e., only attach a predicate to a parent node when all child nodes

an insertion into the search range, even if the leaf where the insertion takes place has not been visited by the search operation. For this to happen, the insertion must cause SP updates in ancestor nodes where the search predicate is already attached. Gradual expansion is retained (although diminished) even for nearest-neighbor queries: although the search predicate cannot be used to differentiate conflicting from non-conflicting updates, the absence of a global predicate lock list will still allow some updates to proceed (unless they perform structural updates that touch nodes on the traversal paths of the search operations).

Complexity The implementation complexity of the hybrid technique is fairly high. It requires a predicate lock manager that can handle per-node predicate lists and will allow transactions to update other transactions' predicate attachments.

4.3.4 A Node Locking Protocol

The hybrid mechanism presented in the preceding subsection can be simplified in a variety of ways. This reduces implementation complexity at the expense of concurrency.

One simplification is to ignore the predicate-specific information of a predicate node attachment and treat each predicate as if it conflicted with any insertion.[12] This way, predicate node attachments can be implemented by simple node locks, with search operations

of interest have been visited, because the search operation does not revisit the parent. It would therefore be unable to pick up additional traversal paths that have been added by concurrently executing insertions—these traversal paths would then result in phantoms if the search operation is re-executed. Section 4.2.3 discussses additional problems involved in repositioning a search operation.

[12] This simplification was inspired by [CM00]. Section 4.6 will describe their approach in more detail and contrast it with the node locking approach.

setting locks by requesting transaction-duration S-locks on traversed nodes and insertions checking them by requesting instant-duration X-locks on updated nodes. The results are more efficient lock checking and a simpler implementation. This obviously reduces the degree of concurrency, but it does not prevent concurrent searches and insertions entirely, because insertions still only check for conflicting locks at their target leaf.

Replacing predicate locks with node locks also eliminates the need for node split and SP update structure modifications to perform lock replication, which is typically not supported by a standard lock manager. The reason is that the existence of any node locks will now prevent the structure modification from taking place (as described in the preceding section), whereas in the hybrid protocol the structure modification is able to proceed if the predicate and the new key to not conflict.

These simplifications turn the hybrid locking protocol into a node locking protocol, in which search operations protect their traversal trees by locking visited nodes and insertions are blocked if the nodes on their update paths collide with a traversal tree. This protocol has the following properties:

Efficiency Setting and checking page locks is relatively cheap, which reduces the computational complexity of the node locking protocol. Furthermore, the computation resources involved in these operations are known, whereas those for predicate operations less predictably depend upon the datatype and the specific search predicate. The handling of structure modifications conflicting with searches also has different computational requirements. The node locking protocol blocks those structure mod-

ifications, which requires them to be reversed.[13] The hybrid protocol, on the other hand, replicates existing predicate locks, which is potentially more costly.

Concurrency Compared to the hybrid mechanism, the obtainable degree of concurrency is reduced, because the discriminating power of predicates is lost. The actual reduction of the degree of concurrency depends on the quality of the SPs, which in turn depends on the particular AM, the data and the specific workload. Although it is impossible to generalize, published evidence [TCH00] suggests that overly "wide" SPs can be a problem in spatial workloads with even moderate dimensionality. For these workloads, the loss in concurrency of the node locking protocol might be unacceptable.

Gradual Expansion Since search operations traverse the tree in the same way in both protocols, the degree of gradual expansion is the same.

Complexity The node locking protocol does not need a specialized predicate lock manager, which lowers the implementation complexity. On the other hand, it needs to block structure modifications that conflict with search operations. This will increase the implementation complexity of those structure modifications considerably. Overall, the node locking protocol most likely is less complex to implement than the hybrid protocol.

[13] A split begun at the leaf level and carried up the tree recursively will need to be reversed if it needs to split a node with a lock. Reversing a node split or SP update is at least as expensive as the node split or SP update itself.

In summary, the hybrid and the node locking approaches have very different characteristics, and a comprehensive experimental comparison has not been attempted yet. It is unlikely that a clear winner will emerge, i.e., one technique that dominates the other in every conceivable aspect and under every workload. A comparison of a technique similar to the node locking approach with the hybrid approach in [CM00] reports that the node locking approach is much more efficient and offers similar degrees of concurrency, but the observation was made for workloads with 2- and 3-dimensional data and where all search operations required repeatable-read isolation. In different scenarios, e.g., higher dimensions or very few repeatable-read searches (as one would expect from highly dynamic databases), the outcome may well be reversed, because the computational disadvantage of predicate checking may be insignificant, but its concurrency advantage may matter a lot. The book on this subject cannot be closed yet, and it would be interesting to see a comprehensive experimental evaluation.

4.4 Logging and Recovery

The goal of recovery is twofold. First, it ensures that the tree's leaf data only reflects insertions and deletions of committed transactions and none of those of uncommitted transactions. Second, the physical tree structure must be brought back into a consistent state after a system crash. As an example of an inconsistent state, consider a node split that was interrupted by a system crash before a parent entry could be installed for the new child.

Since the global counter value has been incremented, a subsequent traversal operation will not recognize the "missed split" and the tree structure will remain corrupted. The following GiST recovery protocol is targeted at a write-ahead logging (WAL) environment with page-oriented redo and logical undo (for a description of WAL environments, see [MHL+92] and [GR93]).

4.4.1 Logging for High Concurrency

In order to obtain high concurrency, a logging protocol must separate update operations into their content-changing (key insertion and logical deletion at the leaf level) and structure-modifying parts (node splits, SP updates, node deletions). This allows us to ascribe the content-changing operations to the initiating transactions, whereas structure modification operations (SMOs) are carried out separately from any transactions as individually committed atomic units of work.[14] The advantage of this approach is that an SMO can be "committed" as soon as it finishes and the latches on the involved nodes can be released immediately (within an atomic unit of work, we employ two-phase latching: once acquired, a latch is only released when the atomic unit of work finishes). This is in contrast to content-changing operations, whose updates are protected by data record locks until the end of their enclosing transactions (page latches are of course also released once the operation is fin-

[14] These atomic units of work have also been called "atomic actions" [LS92] or "nested top actions" [MHL+92] in the research literature. A technique for executing a sequence of page updates as individually committed atomic units of work is described in [MHL+92]. Essentially, the log records written for these page updates are separated from those of the surrounding transaction by appropriate setting of the backchain pointers in the log record headers.

ished). As an example of what would happen without atomic units of work, consider how an SP update would be carried out if it were to execute as part of the same transaction that initiated the preceding key insertion. In this case, the updated parent entry containing the SP would have to be locked until the end of the insert transaction, otherwise another update to the same parent entry by a different transaction would be incorrectly rolled back if the insert transaction were aborted. Locking the parent entry, however, would serialize all key insertions that need to update the same parent entry.

By exclusively latching all nodes involved in an SMO for the duration of the SMO, the SMO prevents all intervening access to its nodes. This allows SMOs to be undone *physically*, i.e., by physically reversing the original modifications. Content-changing operations, on the other hand, cannot latch their target leaves for the duration of their enclosing transaction, because this would prevent structural modifications of that leaf. As a consequence, they require *logical* undo, during which part of the tree structure may need to be re-traversed to relocate the inserted or marked leaf item. Another differentiating characteristic is that content-changing operations are undone when their enclosing transaction is rolled back, but an SMO is only undone if a crash prevents it from finishing, i.e., if the log at the time of the crash does not contain all of the SMO's log records. This implies that if an SMO only writes a single log record, it need never be undone and can therefore be logged as a redo-only operation, which reduces log record size. The functionality of the log manager determines whether this optimization can be employed: if only a single page is updated during the SMO, the log manager must be able to tag this as a redo-only record;

for a multi-page SMO, the log manager must also be able to relate a single log record to multiple pages.

Sections 4.2.2 and 4.2.3 used the following atomically executed SMOs as the building blocks of insert and delete operations: (1) split of a single node, including installation of the parent entry for the new node and update of the parent entry for the split node; (2) parent entry update on a single ancestor node; (3) deletion of a node, including the deletion of the parent entry; (4) garbage collection of a node. An SMO writes a single log record per modified page, with each log record containing enough information to redo and undo the corresponding page operation. The log record data must be sufficient to execute the redo and undo actions *independently* of any other pages in the index. The reason is that the WAL protocol does not guarantee that pages are written to disk in the order they were modified, and therefore at restart time the database may not be in a state that corresponds to a chronological order of page modifications. As an example, consider the split SMO. The fact that it moves some page entries from the left page to a new right page cannot be captured by simply logging the slot indices of the moved entries, because the redo action would need to refer to the original node to re-create the new right sibling. The latter would fail if the on-disk image of the original node *post*dates the split and therefore does not contain the entries in question. For that reason, the split SMO log record for the new right sibling must include the page entries destined for this node. The SMO's redo and undo actions execute the same GiST interface functions that are executed during normal operation, which requires that the corresponding catalog information that enables access

to those UDFs be recovered *before* any indices are recovered. Note that no additional user-supplied extension code is required to write the log records or execute any special logging-related actions, so that logging can be handled independently by the core DBMS component and need not be taken into account by the external AM designer.

4.4.2 Logical Undo Recovery

Key insertion and logical deletion at the leaf level are undone logically, by revisiting the leaf holding the key, which may have changed its location since the original operation was performed. The latter happens if between the time the index operation was performed and the time the transaction is aborted, a split takes place that moves the relevant entries rightward onto other leaves (for the same reason, logical undo is also employed in ARIES/IM [ML92] and ARIES/KVL [Moh90a]). In order to relocate the leaf holding the entry of interest, we traverse rightlinks, starting from the original leaf. Note that a full retraversal of the tree starting at the root is not always possible during restart recovery, because the tree may still contain unfinished splits. For the same reason, the undo recovery phase must not execute any structure modifications as part of a logical undo of a leaf entry insertion or deletion (if they intersect with the yet-unfinished SMOs, they will be lost). The GiST logging protocol fulfills this requirement, because it can avoid or defer all SMOs during the undo recovery phase. More specifically (a) undoing a leaf entry deletion only involves unmarking the entry, not updating the parent entries; (b) undoing a leaf entry insertion only involves physical deletion of the entry; even if the node becomes empty, no node deletion

is performed.

4.5 Implementation Issues

The previous sections describe the algorithms for concurrent operations in a GiST, but omit the implementation of some important details. This section discusses two issues that arise during the implementation of a high-concurrency GiST: sequence number generation and predicate management.

4.5.1 Node Sequence Numbers

A key component of the concurrency protocol for GiSTs is the tree-global counter used to generate node sequence numbers. This counter is incremented during splits and must be made recoverable in order for split detection to work after a crash. In a write-ahead logging environment, this can be achieved easily without having to write additional log records, using the existing infrastructure.

First, instead of maintaining a separate counter for each index in a database, it is possible to use a single database-wide counter. WAL-based recovery systems often have log sequence numbers (LSNs) associated with their log records, which are also reflected in the page to which the log records refer (for a description of logging and restart in WAL environments, see [MHL+92] and [GR93]). These LSNs are guaranteed to be monotonically increasing, which makes the LSN of the last log record written an ideal candidate for the

global counter value. During a split SMO, a log record is written for the original node, which implicitly increments the global counter. Furthermore, this counter is automatically recoverable without having to write any log records.

Using end-of-log LSNs to generate NSNs gives us an opportunity for a second optimization. When a search operation examines an internal node, it needs to memorize the end-of-log LSN along with all the qualifying subtree pointers. Reading the end-of-log LSN requires synchronization within the log manager, which could cause it to become a bottleneck. To alleviate the traffic on the end-of-log LSN, descending operations can memorize the node's LSN instead. This is possible, because the LSN and the NSN of a page come from the same source. If the parent's log record is written after those of the children, the parent's LSN will be greater than any of the child NSNs, except for those whose recent splits are not reflected in the parent.

If this infrastructure is not internally available within the DBMS, as was the case in the IDS/UDO implementation of GiST, the global counter must be stored and maintained explicitly in the database. By storing the counter on a separate page in the index, the buffer manager's latches can be used to synchronize access to it. Updates of the counter now require explicit logging, but the logging frequency can be reduced with the following trick: instead of logging every single counter increment, only write a log record every nth increment, with the actual counter value *plus n*. An example will clarify this. Suppose only every 100th increment is logged, and we are incrementing the counter to 200, logging a value of 300. The next 99 increments need not be logged, because after a crash, the value

will be restored to 300. The search optimization—looking at the parent's LSN instead of the global counter—can also be applied, even if no node LSNs are available. It is sufficient to store the highest child NSN in the parent, which can be updated easily during a node split. The additional storage cost in the parent node is low, and the value can be used in the same way as the node LSN.

4.5.2 Predicate Management

Predicate locking is one of the two centerpieces of the hybrid locking protocol. A predicate manager component can be used in conjunction with the regular lock manager to offer the required functionality.

The predicate management functions required by the search and insert operations are: (1) attaching search predicates to nodes; (2) removing search predicates from nodes; (3) checking all of the predicates attached to a node for conflicts with a key of an insert operation; (4) replicating predicate attachments between two nodes; (5) replicating predicate attachments at sibling nodes when doing a node split. These functions are best realized by a predicate manager component, which can be implemented along the lines of a lock manager within a DBMS (see [GR93] for an example). The major data structures within a predicate manager would be: (a) a list of predicates per transaction, (b) a list of node attachments for each predicate, and (c) a list of the predicates attached to each node. With the help of the standard lock manager, an operation can block "on a predicate" by blocking on the owner transaction of the predicate. This is easily achieved by constructing a lock name

from the owner transaction ID and requesting an S-mode lock on that name, assuming that every transaction acquires an X-mode lock on its own ID when it starts up.

Analogous to the replication of predicate attachments as a consequence of node splits, it is also necessary to replicate the signaling locks set on a node. This requires an extension of the standard lock manager functionality.

4.6 Related Work

In this section, I compare my approach to concurrency and recovery wih the prior work in this area, which has mostly been restricted to B-trees [BS77, LY81, Sag86, SG88, Moh90a, ML92]. The comparison shows that structural differences between B-trees and the class of trees represented by GiSTs make most of the concurrency techniques developed so far for B-trees inapplicable in the GiST context and hence in the context of structures such as R-trees [Gut84], TV-trees [LJF94] and so forth.

The link technique, on which GiST concurrency is based, was first introduced in [LY81] as the B-link tree, and has been the basis of many subsequent papers on B-tree concurrency (for example [Sag86]). Its superiority over subtree-locking concurrency protocols, as described in [BS77], has been confirmed by two performance studies [SC91, JS93].

A different approach than node linking was taken in ARIES/IM [ML92], which employs a conventional non-link tree structure and allows latch-coupling during tree descent, but is still able to propagate splits bottom-up without having to lock subtrees. Instead of

following rightlinks, the traversing operations recognize an ongoing split and compensate for it by repositioning themselves in an ancestor node and retraversing the tree from there. As explained in Section 4.2.3, repositioning a search operation within an ancestor node requires partitioning of the key space, so this technique is, like the original B-link tree technique, also not applicable in a GiST context.

Another general tree structure for which concurrency algorithms were developed is the π-tree [LS92], which was designed for multidimensional point data, and partitions its key space at each level of the tree. It deviates from GiSTs in that it is not a proper tree but a DAG: two index entries on different parent nodes can point to the same child node. The π-tree also employs the link technique to allow traversing operations to recover from missed splits. Because each tree level partitions the key space, it can rely on SPs to detect splits. Node deletion is not discussed in [LS92], but is possible by latch-coupling during descent and repositioning in an ancestor node for each traversal of a subtree. No algorithms for transactional isolation were developed for the π-tree, but a slight modification of the the method described in Section 4.3 is also applicable to them.

Chakrabarti and Mehrotra [CM00] describe an approach to repeatable-read isolation for GiSTs that they characterize as an application of "dynamic granular locking," similar in spirit to key-range locking in B-trees and derived from a protocol developed specifically for R-trees [CM98]. It is largely identical to the node locking protocol of Section 4.3.4 (in fact, it inspired the simplifications of the hybrid protocol): a search operation's traversal tree is protected by locks on the traversed nodes and an insert operation avoids conflicts by

blocking on those locks set along its update path; splits of locked nodes are also avoided, otherwise the locks would need to be replicated to the new sibling node. Chakrabarti and Mehrotra's technique deviates from the node locking protocol as follows:

- (Non-singleton) search operations rely purely on node locks and do not acquire any data record locks. While this is appealing from an efficiency point of view (fewer locks set), the DBMS server may require acquisition of data locks in order for those locks to be visible to conflicting operations that do not traverse the same index. Aside from that, node locks reduce concurrency for insertions and do not work for isolation levels below repeatable read.

- Conflicting locks on the update path of an insertion are checked during descent, not ascent. This may lead to a simpler implementation.

Note that the description of this technique as "granular locking" is misleading, because the SPs are not used as granules for the purpose of locking (otherwise the lock on the root node would lock the entire data space). Instead, these locks protect parts of the physical tree structure, namely the *traversal paths* of search operations and the *update paths* of insert and delete operations (and their undo operations).[15]

The fundamental ideas for access method recovery—essentially, the requirement for

[15] As a result of this misunderstanding, the authors state incorrectly that the shrinking of SPs after a commited deletion must be prevented, if it would affect the SP of a node that is locked by a search operation. The reasoning is that such shrinking would decrease the "coverage" of the node lock and thereby allow future insertions into the now "unprotected" data space. This is incorrect, because the existing lock protects the *node*, not the associated data space, from updates; it will prevent future insertions that would alter the search operation's traversal tree even after shrinking the SP.

separate atomic units of work to carry out structure modifications—have been recognized early on and have been published in a number of articles [ML92, Moh90a, GR93, LS92]. The GiST logging and recovery protocol as presented in Section 4.4 directly builds on that prior work.

The basis of the GiST concurrency protocol was developed in [KB95] in the context of R-trees, which have the same structural properties as GiSTs (non-partitioning, non-linear keys). The R-tree concurrency control protocol replicates the NSN of a node in its parent entry, which adds extra space overhead. The paper also does not sufficiently address the problems of transactional isolation, recovery and node deletion. The data-only locking approach and logical deletion have been adopted from [ML92, Moh95, Moh90b].

4.7 Conclusion

This chapter presents search tree algorithms for physical and logical concurrency as well as recovery. Although presented in the context of GiSTs, these algorithms are applicable to a broad class of search trees. In conjunction with the algorithms developed in this paper, the GiST structure can serve as the basis of truly extensible indexing in commercial-strength database systems. The core DBMS plus GiST can be extended with a new access method simply by supplying it with a set of pre-specified methods, which specialize the abstract GiST structure into the desired access method. Details such as concurrency and recovery—which usually account for a major fraction of the complexity of the code and

are error-prone and hard to debug—can be ignored by the extension code.

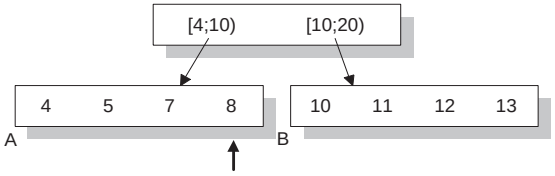
The key features of the GiST concurrency protocol are that it does not hold any latches during I/Os and is deadlock free, resulting in a degree of concurrency that should match that of the best B-tree concurrency protocols. The basic idea is derived from the link technique pioneered for B-trees, which allows compensating for unexpected splits by moving across rightlinks. To make this work for a broader class of tree structures, it is necessary to add sequence numbers to the nodes to make node splits visible without reference to the stored keys.

In order to ensure repeatable read transactional isolation, which guarantees the absence of "phantoms" across search operations, GiST can use a hybrid locking mechanism, which combines two-phase locking of data records with a novel form of predicate locking. Predicate locking is responsible for avoiding phantoms, whereas more efficient data record locking is used for retrieved index entries. This division of responsibilities is necessary, because data record locking alone cannot reasonably solve the phantom problem in a key space without linear order. A simplification of this protocol leads to a less complex node locking protocol.

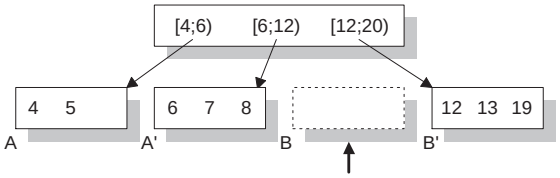
These algorithms are general enough to work for all tree-based access methods with a "traditional" B-tree-like structure, because structural elements, not semantic knowledge about the data, are exploited.

```
globals:
   stack; // records nodes and NSNs on path

insert(new-key, RID)
   leaf = locateLeaf(new-key);
   // at this point, only the leaf is X-latched
   if (not enough space on leaf)
      splitNode(leaf, new-key, 0);
      release all latches at ancestor levels,
      target leaf stays X-latched;
   else
     if ( union(leaf, SP(leaf), new-key) != SP(leaf))
         updateParent(leaf,
            union(leaf, SP(leaf), new-key), 0, 0);
      end
   end
    insert(leaf, new key, RID);
   unlatch(leaf);

locateLeaf(new-key)
   p = root;
   p-NSN = global NSN;
   loop
      if (p is leaf)
         latchmode = X-mode;
      else
         latchmode = S-mode;
      end
      latch(p, latchmode);
      if (p-NSN < NSN(p))
         // encountered a split
         unlatch(p);
         [p, p-NSN] = pop(stack);
         go to beginning of loop;
      end
      if (p is not leaf)
         push(stack, [p, NSN(p)]);
         child-ptr = find_min_pen(p, new key);
         p-NSN = global NSN;
         unlatch(p);
         p = child-ptr;
      else
         // leave leaf p X-latched
         return p;
      end
   end

splitNode(p, key1, key2)
   create new node p';
   latch(p', X-mode);
   (split-info, p-SP, p'-SP) = pickSplit(p, SP(p),
      key1, key2);
   move data from p to p' according to split-info;
   NSN(p') = NSN(p);
   rightlink(p') = rightlink(global);
         // the following increment is exe-
cuted atomically
   <global NSN = global NSN + 1;>
   NSN(p) = global NSN;
   rightlink(p) = p';
   updateParent(p, p-SP, p', p'-SP);
   unlatch(p');

updateParent(left, left-SP, right, right-SP)
   latch(parent(left, stack), X-mode);
                  if (NSN(parent) != par-
ent NSN recorded on stack)
      parent = node in rightlink chain starting with
            currently recorded parent, holding en-
try for left;
      latch correct parent;
   end
   if (not enough space on parent)
      splitNode(parent, left-SP, right-SP);
   else
      updateParent(parent,
         union(parent,
            union(parent, SP(parent), left-SP),
            right-SP),
         0, 0);
   end
    updatePred(parent, slot of left entry, left-SP);
   if (right != 0)
      insert(parent, right-SP, RID(right));
   end
   unlatch(parent);
```

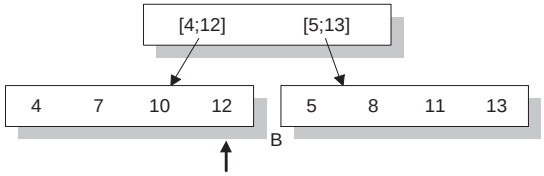
Figure 4.4: Concurrent GiST insert algorithm.



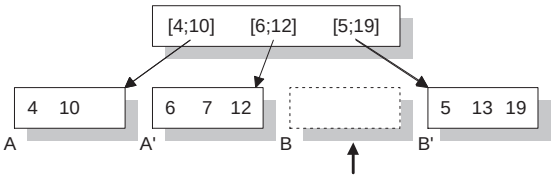
A regular B-tree, where entries in internal nodes are shown as intervals for clarity (normally only stored with their left or right boundary). A scan of the key range $[5; 20]$ starts in leaf A.



After the scan is done with leaf A, leaf A and B are split as a result of inserting keys 6 and 19. Additionally, keys 10 and 11 are deleted and as a result, leaf B is also deleted. The scan notices this and repositions itself in the parent node, remembering that it already covered range $[5; 10)$ (by having scanned former leaf A). The next leaves to visit are A' and B'. Leaf A' does not contain any items contained in the remaining scan range $[10; 12]$, and the scan moves on to leaf B'.



A fictitious non-partitioning B-tree, where the key space is not partitioned across the leaves and as a result internal entries have overlapping intervals. A scan of the key range $[5; 20]$ again starts in leaf A.



After the scan is done with leaf A, leaves A and B are split as a result of inserting keys 6 and 19. Keys 8 and 11 are deleted, resulting in the removal of leaf B. The scan notices this, and, in order to continue correctly, would have to go to leaf B' and ignore A'. Notice that it is impossible to determine this by repositioning in the parent node: (1) although the scan has already covered former leaf A with the interval $[4; 12]$, it would be incorrect to go to leaf B' and ignore entry 5; (2) for that same reason, if the scan would go to leaf A', it would incorrectly repeat scanning of entries 7 and 12.

Figure 4.5: Illustration of why repositioning requires partitioning.

Chapter 5

An Analysis Framework for Access Methods

The preceding two chapters have dealt with the technical problems involved in making non-traditional AMs work inside a DBMS. In this chapter, I will address the more fundamental problem of AM performance analysis.

Despite the large and growing number of AMs that have been produced by the research community, the design and tuning of AMs has always been more of a black art than a rigorous discipline. Traditionally, performance analyses focus on summaries of observed performance, such as aggregate runtime or page access numbers, or on performance metrics that express data-specific properties of index pages (e.g., spatial overlap between the pages of an R-tree [Gut84]). The drawback of *aggregate numbers* is that they do not provide any insight into the causes of observed performance. As a result, it is hard to quantify the con-

tribution of individual design ideas or explain performance differences between competing AM designs, if those deviate in more than one design aspect. Also, aggregate numbers do not allow AMs to be assessed on their own, because competing AM designs are needed to put the numbers into perspective. In contrast, *data-specific performance metrics* offer some insight into the causes of observed performance, but they require the designer to understand their correlation with the optimization objective, i.e., the minimization of aggregate runtime or page access numbers. Since such an understanding is a *goal* of the analysis process, any *a priori* assumptions are often incorrect and misleading. If the correlation of the data-specific performance metric with the optimization objective is not perfectly clear, using such a performance metric to guide AM design is problematic.

In this chapter I present an analysis framework that defines performance metrics that are superior both to aggregate numbers and data-specific performance metrics. This framework is implemented in amdb, a comprehensive support tool for the AM analysis process. Within amdb, the analysis framework is integrated with a collection of modules in an interactive, easy-to-use graphical environment. Those modules are: a visualization component for the tree structure and its contents (the latter user-extensible, so it can be adapted to a specific application domain); a facility for interactive execution of tree searches and updates as well as breakpoints and single-stepping through those commands, similar to functionality found in programming language debuggers; browsers for viewing performance numbers derived from the analysis framework. The salient features of amdb and its analysis framework are:

Universal Applicability The analysis framework and most of the amdb visualization fa-

cilities are independent of the semantics of the data and queries of the application domain, which makes them universally applicable to any AM design that is based on the GiST abstraction. The analysis framework treats the workload—a tree and a set of queries—as an input parameter, allowing the designer to tune an AM for that particular workload.

Better Performance Metrics The analysis framework defines performance metrics that reflect *performance loss*, measured in I/Os and derived from a comparison of observed performance with the performance of a workload-optimal tree. This tree minimizes the total number of I/Os for the input workload and can be approximated relatively efficiently. The advantage of these performance metrics in comparison to aggregate I/O measurements is that they reflect the potential for performance improvement, allowing an AM design to be assessed individually. The loss metrics are further broken down to reflect the performance-relevant characteristics of the tree, which gives the designer a clearer understanding of the effects of individual design ideas or the differences between two competing AM designs.

Fully Automated Analysis The fully automated analysis process executes the user-supplied set of queries, gathers tracing data, uses that to approximate an optimal tree and computes the performance metrics.

Visualization Integration The analysis framework is integrated into amdb to the extent that the metrics as well as tracing information gathered during workload execution

are visualized using the data-independent tree structure visualization facilities. This integration is particularly helpful, because it lets the designer investigate poorly performing parts of the tree and queries. The analysis framework and the visualization tools are complementary: the performance metrics highlight the sources of poor performance, thereby focusing the designer's attention. The visualization tools are then used to investigate those parts of the tree or those queries which have been flagged by the performance metrics.

Designing AMs is a creative process that amdb supports with an analysis framework that points out specific sources of performance degradation and visualization tools for investigating them. The experience we have gathered so far with amdb justifies our claims about its usefulness: in two AM design projects undertaken at U.C. Berkeley, amdb was instrumental in quickly locating performance problems in existing AM designs and verifying that the remedies to those problems worked as intended.

The rest of the chapter is structured as follows. Section 5.1 describes the analysis framework and its intended usage and gives an overview of amdb . Section 5.2 demonstrates amdb's usefulness with two examples of amdb-based AM design projects. Section 5.3 discusses the analysis framework in detail, along with illustrative examples, among them a test for unindexability. Section 5.4 discusses related work and Section 5.5 contains the conclusion and an outline of future work.

5.1 A Tour of Amdb

This section gives an overview of the analysis framework and its intended usage and describes amdb's visualization and debugging features (which are presented in greater detail in [SKH99]).

Amdb supports access methods developed using the public domain libgist package which implements the GiST abstraction. Amdb and libgist are written in Java and C++ and are portable across many versions of UNIX as well as Microsoft Windows NT. The software can be downloaded from http://gist.cs.berkeley.edu/.

5.1.1 Overview of the Analysis Framework

The goal of the analysis framework is to explain the observed performance of an AM running a user-supplied workload. The single ultimate performance number is the total execution time of the entire workload. This total depends on the number and nature of page accesses, the buffering policy and the CPU time spent examining pages. We will for now concentrate on explaining observed page accesses and ignore the other components of the performance equation. Section 5.3.4 addresses these issues.

The introduction mentioned the deficiencies of the current practice of reporting performance with aggregate I/O numbers or data-specific metrics. To be effective and universally applicable, an analysis framework should have three properties:

1. The performance metrics should be data-independent and not be tailored to the se-

mantics of a particular application domain, so that the analysis framework is applicable in the full generality of the GiST AM design framework.

2. The performance metrics must give an indication of the quality of measured AM performance in terms of the optimization objective, i.e., minimization of I/Os.

3. The metrics should give the designer an understanding of the causes of observed performance.

In order to ensure data-independence of the framework, the workload—a tree and a set of queries—is an input parameter of the analysis, and the metrics characterize the performance of an AM specifically in the context of that workload. Also, the performance metrics directly characterize the observed performance of the workload execution, namely the page accesses. They are not stated in terms of data or query semantics, and are therefore data-independent.

Instead of simply reporting the number of observed page accesses, a more meaningful performance metric is the *performance loss*, i.e., the difference between the number of page accesses in the actual tree and the optimal tree. The optimal tree is defined as minimizing the total number of page accesses over the entire workload. Knowing the magnitude of performance loss is a clear indication of the quality of an AM, expressed in the units of the optimization objective, I/Os. Moreover, the performance loss shows the potential for performance improvement, which cannot necessarily be discovered even when comparing two competing AM designs using traditional performance metrics. We can compute a *query*

performance loss, which expresses the difference in the number of I/Os of a query executed against the actual tree and the workload-optimal tree.[1] Similarly, we can compute a *node performance loss*, which expresses a node's contribution to query or aggregate workload performance loss. Furthermore, we can also compute *implementation metrics* in order to characterize aspects of the AM implementation. The extension methods *pickSplit()* and *penalty()* directly control the tree structure and performance loss metrics for these functions should express to what extent they are responsible for the structural deterioration that causes performance loss.

Given a particular performance loss, we can further subdivide it to reflect the fundamental performance-relevant properties of GiST-based AMs, which are:

Clustering The clustering of the indexed data at the leaf level and of the SPs at the internal levels determines the amount of extra data that a query needs to access in order to retrieve its result set. An AM design controls the clustering through the *pickSplit()* and *penalty()* extension methods.

Page Utilization The page utilization determines the number of pages that the indexed data and the SPs occupy and therefore also influences the number of pages that a query needs to visit. Similar to the clustering, the page utilization is controlled by the *pickSplit()* and *penalty()* extension methods.

[1] Having knowledge of the execution profile of the workload, in particular the result sets of the queries, allows us to approximate the optimal tree relatively accurately. The details of how the metrics are computed are presented in Section 5.3.

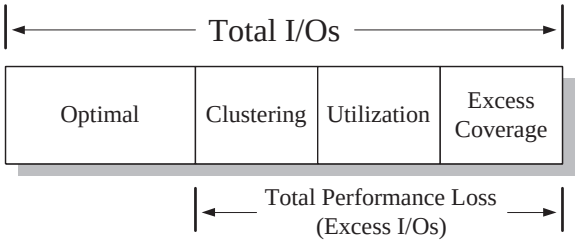


Figure 5.1: Decomposition of observed I/Os on a per-query and per-node basis.

Subtree Predicates While the size and shape of the indexed data is part of the input,[2] the size and shape of the SPs are parameters of the design and considerably influence performance. A SP's task is to describe, or cover, that part of the data space which is present at the *leaf* level of its associated subtree (i.e., the perfect SP would simply enumerate all the data items contained in the leaves of its subtree; of course, this is problematic with regard to the size of the SPs). We speak of SP *excess coverage* if the SP covers more of the data space than is needed in order to represent the data contained in the subtree. If a SP exhibits excess coverage, it may cause queries to visit more than the minimum number of pages determined by the clustering and page utilization.

Clustering loss specifies the part of performance loss that can be attributed to the difference between workload-optimal and achieved (leaf-level[3]) clustering in the index tree; *utilization loss* specifies the part that is attributable to node utilization deviating from a target utilization; *excess coverage loss* specifies the part that is due to accesses to leaf nodes

[2] One could argue that the size of the indexed data can be changed by applying compression in the index. We will ignore this possibility by assuming that a similar form of compression can be applied to the data as a pre-processing stage.

[3] Why this is restricted to leaf-level clustering is explained in Section 5.3.

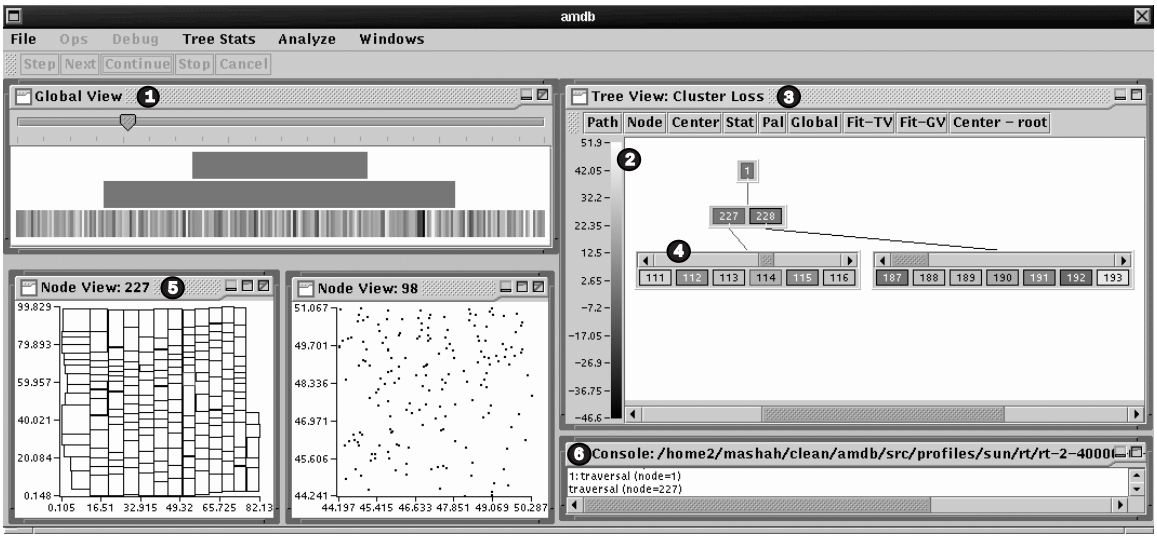


Figure 5.2: Amdb user interface.

that contain no relevant data to a query. All of these subdivisions of performance loss are also specified in I/Os—possibly fractions of I/Os; They are summarized in Figure 5.1. Such a breakdown of performance loss is more useful than aggregate numbers, because it helps the designer understand the nature of the loss and thereby provides more insight into the causes of observed performance. The breakdown of the node metrics in particular helps the designer identify anomalies in the tree structure. The examples in Section 5.2 will illustrate this point.

5.1.2 Amdb's Visualization Functionality

Understanding flaws in an AM design requires inspecting the corresponding tree; thus, amdb provides interactive graphical views of the entire tree, paths and subtrees within the tree, and contents of nodes within the tree. These are the global view, tree view, and node

view, respectively (Fig 5.2). These views not only help visualize the tree structure and its contents, but also help visualize profiling data and performance metrics by associating them with nodes in the tree (discussed in detail in Section 5.1.4). Finally, they provide navigation features, which enables designers to drill down to the source of a deficiency.

The highest-level, *global view* provides a manageable aggregate view of the entire index (Fig 5.2: 1). This representation factors out much of the tree structure by mapping it onto a triangle with an adjustable baseline and height. The purpose of this view is to project a user-selected tree statistic or performance metric onto this abstract display and depict the variation of the statistics across the total tree. The user can choose both a color map (or palette, Fig 5.2: 2) and a statistic; the global view assigns colors to the statistical values and renders the nodes accordingly. Nodes are visually concatenated and merged if necessary with other nodes on the same level. Thus, the pixel density of nodes increases geometrically with the level. The user can also perform an approximate drill-down into an area of interest by clicking on it. Subsequently, a path from the root node to a node in the neighborhood of the specified point will be shown in the tree view, a lower-level view which shows more detail.

The *tree view* shows the structure of the search tree (Fig 5.2: 3). It offers an intuitive point-and-click interface for browsing the tree while improving on conventional tree navigation interfaces which become cumbersome for high fanout trees. In this view, the tree's nodes are represented by boxes and labeled with a unique number for reference. Each node is enclosed in a scrollable and stretchable container which displays its direct siblings. This

container (Fig 5.2: 4) allows users to focus on nodes of interest while bounding the amount of detail displayed. Any node can be expanded or contracted by clicking on it. When a node is expanded, the container holding its children is displayed below it with a line linking the two; when contracted, the entire subtree below the node is removed. Like the global view, the tree view represents a user-selected tree statistic or performance metric by coloring the nodes. With these features, a user can simultaneously focus on several paths and subtrees of interest without being overwhelmed by the width of the search tree.

After drilling down from the global view and tree view, the user can investigate the contents of specific nodes using amdb's node view (Fig 5.2: 5). Since tree nodes contain arbitrary user-defined predicates, the access method designer must provide a module that displays the node given its contents. Currently, amdb contains a suite of modules that visualize two-dimensional projections of spatial data. The node view also allows the user to simulate a split (by calling the *pickSplit()* extension function) and visualize the results by separating the items with contrasting colors. In addition to user-defined data visualization, amdb provides a textual description of the keys, their sizes, and associated pointers.

5.1.3 Amdb's Debugging Functionality

The behavior of an AM can be difficult to understand without being able to observe its mechanics. Previously, only standard programming language debugging tools were available for examining libgist AMs. Because these tools are designed for analyzing low level actions, such as a single line of source code, they are too cumbersome for gaining an

understanding of how search and update operations behave and interact with the tree.

Amdb allows a designer to single-step through tree search and update commands. Those commands generate events for various node-oriented actions, such as node split, node traversal, *etc.*, which permits users to step from event to event. Since manual stepping can become tedious, it also supports breakpoints. Breakpoints can be defined on generic events, e. g., node update, or can be tied to a specific tree node, e.g., update of node 227. When a breakpoint event is encountered, execution is suspended, and the user has an option to single-step through events or continue until the next breakpoint. Additionally, amdb allows batch execution of commands via scripts so users can conveniently restore state.

5.1.4 Using Amdb to Analyze Access Method Performance

To use amdb in order to analyze an AM design, the designer constructs an index tree and decides on a set of queries to run against that tree. Together, these two items constitute the *target workload*. Taking this workload as input, amdb then runs the analysis that produces the performance metrics described in the previous section. The analysis process consists of running the queries against the index tree, gathering tracing data such as traversal paths, and approximating an optimal tree based on the tracing data. Given this optimal tree approximation, amdb computes the performance metrics for each query and the aggregate workload. These are broken down further into per-node loss metrics, which are also computed for each query and the aggregate workload. A detailed description of the tracing data, the nature of the optimal tree and the computation of the performance metrics are

given in Section 5.3.

The performance loss metrics express I/Os, not particular application-specific properties of the tree at hand or the AM design; the metrics can therefore only serve as an *indication* of, not an explanation for performance deficiencies. The explanation of performance deficiencies and a subsequent improvement of the AM design need to be done by the AM designer, based on an understanding of the semantics of the application domain. Gaining such an understanding is a creative process, which is helped by the amdb visualization facilities and their integration with the analysis framework: the performance metrics "flag" those parts of the tree and those queries that perform badly; the visualization facilities then let the designer navigate those index nodes and queries and investigate the reasons for their above-average performance loss. Aside from the user-extensible data visualizations, amdb also gives the designer access to a very comprehensive set of workload statistics, including per-query aggregate page access numbers, full traversal paths, the amount and specific location of data retrieved, *etc.* The performance metrics themselves are very voluminous—there are three loss metrics for each query and each node of the tree–which makes it necessary to find good visualizations for them.

The node metrics are visualized by coloring nodes in the global and tree view, so that ill-behaved parts of the tree can be identified easily without having to browse through each node's metrics individually. The navigation and data visualization features of these views let the developer navigate those parts of the tree structure and examine the data contained therein. The global and tree views are also used to visualize the per-query loss metrics

and trace data on a per-node basis (for example, traversal paths can be visualized very effectively through node coloring). This tracing data in combination with the visualizations give the developer a very detailed view of the behavior of each query and are instrumental in understanding poorly performing queries.

Before designing an AM for a particular workload, it is instructive to determine whether that workload is possibly unindexable, i.e., whether no index structure will be able to outperform a sequential scan on that workload. The amdb analysis process produces all the data necessary to perform such a test; the details are given in Section 5.3.3.

The next section describes two amdb-assisted AM design projects in which the amdb performance metrics were used to assess the merits or demerits of an AM design. In these examples, total I/Os or execution times were inconclusive or, at worst, misleading.

5.2 Sample Applications of Amdb

Since the time amdb was implemented and made available to the public, two AM design projects undertaken at U.C. Berkeley made use of this tool. We will describe each one in turn in order to illustrate how amdb was used to help the design process. In both of these projects, the designers were able to use amdb to achieve significant improvements.

5.2.1 Content-Based Image Retrieval

An AM design project was undertaken in the context of a content-based image searching, Blobworld [CTB+99]. The Blobworld system addresses content-based querying by breaking the images into "blobs" of homogeneous characteristics, and searching for images by specifying the characteristics of the blobs in the desired images. A full Blobworld query must perform computationally complex comparisons of the high-dimensional feature vectors of the blobs in the images. For the purpose of indexing this data set, the dimensionality of the feature vectors was reduced from 218 to five dimensions by doing a singular value decomposition. The data set was then bulk-loaded into an R-tree using the STR partitioning algorithm [LLE97]. The details of this AM design project are described in [TCH00].

Using amdb, the designers found that while clustering and utilization were good (i.e., the corresponding losses were 3 and 1 percent of the total number of about 200,000 I/Os for the entire workload), excess coverage contributed a very large percentage to the total I/O count (about 31 percent). The tree visualization of the excess coverage loss statistics actually showed that this loss was not distributed evenly across the entire tree, but that some nodes attracted significantly higher loss than others. The data visualizations of those high-loss nodes showed that they contained a large fraction of empty "corner" space. The designers drew the conclusion that SPs should be encoded as polygons instead of simple hyperrectangles in order to "cut away" empty corners.

One particular design idea for SPs was to combine two hyperrectangles instead of just

a single one, as in the standard R-tree. Running the benchmark workload in amdb quickly showed that, as implemented (rectangles were chosen from a set of randomly constructed bounding rectangles), this design resulted in a small total performance degradation in comparison to the original R-tree. Looking at amdb's metrics made it clear that this design decreased excess coverage loss at the leaf level, but increased I/Os at the internal levels. The reason is that at internal levels, having two hyperrectangles was not an effective way of excluding "empty" corners;[4] the combination of two hyperrectangles therefore ended up being no more discriminating than just a single one, but used up more space. This particular example illustrates the value of the performance breakdown: had only aggregate I/O numbers been available, the varying effects on the leaf and internal levels would not have been visible, making it harder to draw the same conclusion. In this example, the integration of the available metrics with the visualization tools was also very important, because it facilitated examining those nodes with high excess coverage loss and drawing conclusions about a more accurate encoding of the space covered by the feature vectors.

The amdb analysis also established that another design alternative—convex minimum-bounding polygons—causes almost no excess coverage loss and is therefore close to optimal for the given workload. Taking this into account, the designers then focussed on finding an approximation to this fairly CPU-intensive design, rather than searching for a yet more accurate SP design. In this case, the amdb performance metrics clarified that no substantial improvements could be gained from investigating more accurate SPs.

[4]This might be an effect of the particular algorithm used to construct the SPs, but that is not the point here.

5.2.2 Multidimensional Point AM for Window Queries

As part of the graduate database class at U.C. Berkeley (CS286, Spring 1999), the students were required to design an improved AM for a particular synthetic multidimensional point dataset (containing 8-dimensional data arranged into 200 clusters of 100 points each). The workload consisted of 10,000 range queries centered on randomly chosen data points. The starting point was the performance achieved with an R*-tree, which the students needed to improve.

A confirmation of the efficacy of amdb and the analysis framework in particular was that many of the design groups managed to improved performance to a great extent (some by a full 50 percent), although none of the students had previously worked on spatial point AMs (in fact, any AMs at all) and each group only spent about a week on the assignment. We believe that without amdb, such results would not have been possible.

All groups started their design process by looking at the breakdown instead of just the total numbers of aggregate I/Os and proceeded to address one or more of the performance factors which proved to be problematic. At the leaf level, the initial total number of 26,600 I/Os broke down into roughly 5,400 I/Os due to clustering loss, 1,800 I/Os due to utilization loss, 9,050 I/Os due to excess coverage loss and 10,350 optimal I/Os.

One of the design ideas that the students came up with was to relax the utilization restrictions in the R*-tree split algorithm (which allows at most a 40/60 imbalance). The purpose was to to allow a node split to separate two clusters cleanly instead of forcing it to divide up individual clusters between two nodes to satisfy utilization restrictions. This

resulted in a substantial performance improvement, reducing the total number of leaf I/Os to 19800. Aside from clustering and excess coverage loss, it also reduced the utilization loss component, which was unexpected, because splits were allowed to be less balanced.[5] The breakdown of the aggregate I/O number therefore clarified the effects of this design idea and in this particular case allowed the design group to conclude that further work on rectifying an assumed utilization problem was not necessary. Another design group had a contrary experience: their SP design resulted in a total reduction of 4,000 leaf I/Os. The breakdown showed that the cause for this was a reduction of excess coverage loss by about 5,000 I/Os, mitigated by an increase of utilization loss by about 1,000 I/Os. Again, the breakdown conveyed more useful information that just the aggregate number and gave a more insightful assessment of the effects of this particular design idea.

Generally speaking, all groups stated in their reports that the performance metrics were essential in finding which aspects of the AM needed improvement. In addition, some groups complained that the multidimensional data visualization supplied with amdb (which consists of a simple projection on the first two dimensions) was not sufficiently powerful. This illustrates our earlier point about the complementary nature of amdb's data-independent performance metrics and data-specific visualizations, namely that the latter is necessary for gaining an intuition of the nature of the problem, whereas the former tells the designer which particular subtrees or queries to investigate.

[5] The possible reason for this is that it allowed clusters to separate into their own nodes; in this particular data set, if a node contains more than one cluster, it will be forced to split at some point.

5.3 Details of the Analysis Framework

The following subsection discusses the optimal tree and how to construct it. Section 5.3.2 derives the query performance metrics, first for the leaf level, then for internal levels, and presents examples of analyses conducted with these metrics. Section 5.3.3 derives node metrics based on the query metrics. Various examples throughout this section illustrate the performance metrics.

The discussion of the metrics in this section is purposely informal and relies mainly on examples; I felt this would improve readability. The input variables and metrics are formally defined and summarized in Table 5.1 and Table 5.2, respectively. Variables with subscript q are query-specific and variables with subscript p are page-specific. Also note that the performance metrics are a complete partitioning of the I/Os observed for the workload; an I/O or fraction thereof is not attributed to more than one loss category.

5.3.1 Construction of the Optimal Tree

The optimal tree is defined by the following characteristics:

no excess coverage, which eliminates page accesses due to overly general SPs;

target page utilization, which would ideally be 100%, but this is unattainable in practice. Instead, the AM designer specifies a desired target page utilization, which will also be used as a parameter for the optimal tree. For some AM design projects, this value will be determined by external considerations, e.g., the existence of a competing AM

Q	set of queries q in workload
L	set of leaf nodes in tree
I	set of internal nodes in tree
C [bytes]	page capacity
R_q [bytes]	size of result set
L_q^o	set of accessed pages in optimal clustering
L_q	set of accessed leaves in actual tree
L'_q	set of relevant leaves in actual tree (leaves that contain items of q's result set)
u_t [%]	target utilization
u_p [%]	utilization
u_q [%]	average utilization seen by query, $u_q = \sum_{p \in L'_q} u_p / \lvert L'_q \rvert$
I_q	set of accessed internal nodes in tree
I'_q	set of accessed internal nodes on paths to L'_q
I_q^l	internal "leaves" of traversal tree, $I_q^l = \left\{ p \lvert (p \in I_q \setminus I'_q) \wedge \neg(child(p) \in I_q \cup L_q) \right\}$
Q_p	set of queries that access p
Q'_p	set of queries for which p is relevant leaf
r_q	optimal ratio of accessed to retrieved data, $r_q = \lvert L_q^o \rvert * C * u_t / R_q$
$R_{p,q}$ [bytes]	size of fraction of q's result set found on p
$Q_{p,q}^o$ [bytes]	optimal amount of accessed data, $Q_{p,q}^0 = r_q * R_{p,q}$
Q_p^o [bytes]	optimal amount of accessed data aggregated over workload, $Q_p^o = \sum_{q \in Q'_p} r_q * R_{p,q}$

Table 5.1: Input variables of the analysis process (profiling data, tree statistics and derived variables).

with a well-known average utilization.[6] If no such point of reference is available, one or more reasonable utilizations (in the 50–80% range) should be tried. The value we often used in practice was the average workload page utilization. We will see that the absolute level of the target page utilization does not affect the significance of the performance metrics for the comparison of nodes within the tree structure.

optimal clustering, which minimizes the total number of "relevant" page accesses (at the

[6] In this case, the target utilization should be in the vicinity of the known average utilization. Also, for purely static trees, a value of 100% is attainable and should be used.

CL_q	clustering loss	$CL_q = (u_q/u_t)\|L'_q\| - \|L^o_q\|$
EL^l_q	leaf-level excess coverage loss	$EL^l_q = \|L_q\| - \|L'_q\|$
UL^l_q	leaf-level utilization loss	$UL^l_q = \|L'_q\|(1 - u_q/u_t)$
$EL^i_{p,q}$	internal-level excess coverage loss on page p	$EL_{p,q} = \begin{cases} 0 & \text{if } p \in I'_q \\ 1 & \text{if } p \in I^l_q \\ u_p/u_t & \text{otherwise} \end{cases}$
EL^i_q	internal-level excess coverage loss	$EL^i_q = \sum_{p \in I_q \setminus I'_q} EL^i_{p,q}$
$UL^i_{p,q}$	internal-level utilization loss on page p	$UL_{p,q} = \begin{cases} 1 - EL^i_{p,q} & \text{if } p \in I_q \setminus I'_q \\ 1 - u_p/u_t & \text{otherwise} \end{cases}$
UL^i_q	internal-level utilization loss	$UL^i_q = \sum_{p \in I_q} UL^i_{p,q}$
I^r_q	remainder of internal-level accesses	$I^r_q = \sum_{p \in I'_q} u_p/u_t$
CL_p	clustering loss	$CL_p = \sum_{q \in Q'_p}(u_p - Q^o_{p,q}/C)/u_t$
EL^l_p	leaf-level excess coverage loss	$EL^l_p = \|Q_p \setminus Q'_p\|$
UL^l_p	leaf-level utilization loss	$UL^l_p = \sum_{q \in Q_p} 1 - u_p/u_t$
EL^i_p	internal-level excess coverage loss	$EL^i_p = \|\left\{q \| p \in I^l_q\right\}\|$
UL^i_p	internal-level utilization loss	$UL^i_p = \sum_{\left\{q \in Q_p \| p \notin I^l_q\right\}} 1 - u_p/u_t$
Q^r_p	remainder of internal-level accesses	$Q^r_p = \sum_{\left\{q \in Q_p \| p \notin I^l_q\right\}} u_p/u_t$

Table 5.2: Performance metrics produced by the analysis process.

leaf level, those are accesses to pages containing items of the result set of a query, see Table 5.1) for the entire workload.

A tree with these properties will execute the investigated workload with the minimal number of page accesses. This tree is only a theoretical construct, since it is generally impossible to create reasonably-sized SPs with no excess coverage. Nevertheless, it is possible to approximate this tree well enough to be able to infer the page access pattern of the workload queries.

To construct the optimal leaf level, we partition the indexed data items so that the total number of leaf accesses is minimized over the workload[7] and the partition size is equal to the target page capacity. This task can be converted into a hypergraph partitioning problem by modelling the workload as a hypergraph (each indexed data item is a node with a weight that is equal to its size in bytes; each query, identified by its result set, is a hyperedge). Hypergraph partitioning is provably NP-hard [GJ79], but existing approximation algorithms work reasonably well in practice. Section 5.3.5 discusses the implementation in more detail, in particular the hypergraph partitioning.

To construct the optimal internal levels, we need to create reasonably-sized SPs with no excess coverage, which is generally not possible. Nevertheless, it is still possible to report utilization and excess coverage loss metrics for those.

Figure 5.3 serves as a running example throughout the rest of this section. It shows the traversal tree of a query (its traversal paths in the index, which form a subtree of the index) that retrieves five data items, for which it needs to access four leaves in the actual tree and two leaves in the optimal tree. The page capacity is four items (to keep the example simple, data items and SPs are assumed to have the same size) and the target utilization is 75 percent. Occupied slots are shaded, and the pages in the actual tree are enumerated for reference.

[7] Note that clustering to minimize the number of leaf accesses over the *entire* workload will generally not minimize the number of leaf accesses for each query *individually*. The minimum number of leaf accesses for a single query is the size of its result set divided by the page size. This usually cannot be achieved for the entire workload, because the queries' result sets overlap and their clustering requirements are contradictory.

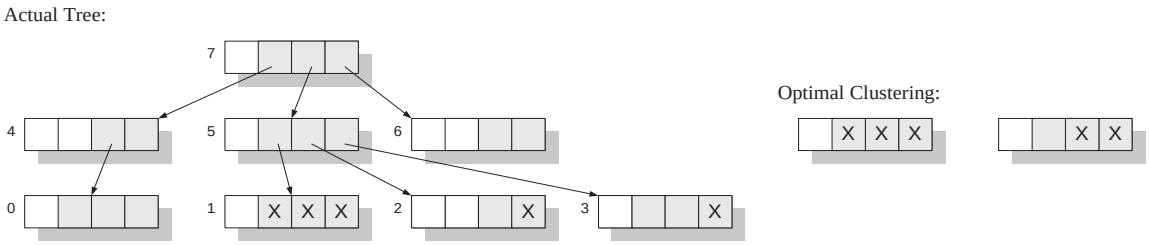


Figure 5.3: Traversal paths and optimal clustering for example query.

5.3.2 Query Performance Metrics

The per-query performance metrics express performance loss due to suboptimal clustering, page utilization and SPs in the index. At the leaf level, these numbers are derived by comparing the page access pattern in the actual tree with the corresponding pattern in the optimal tree. At the internal level, the corresponding optimal structure is not available for comparison, but we can still derive a reduced set of the metrics, namely excess coverage and utilization loss. The next two subsections in turn describe how the loss metrics are derived for the leaf level and the internal levels.

Leaf-Level Performance Metrics

For each query, the performance loss at the leaf level—actual minus optimal number of leaf accesses—is divided up into utilization, excess coverage and clustering loss. More formally:

$$|L_q| = |L_q^o| + EL_q^l + UL_q^l + CL_q.$$

In the example, the query experiences a performance loss of two leaf accesses when compared against the optimal tree. The following paragraphs how to compute the losses

for this example.

Excess coverage loss When accessing a leaf during query execution that does not contain any items of the result set, the leaf access is due to excess coverage in the leaf's SP. Those pages do not count toward utilization loss, even if they are underutilized, because packing them more densely would not lower the total number of leaf accesses (unless retrieved data were added, but then the accesss would not count as excess coverage to begin with). For the same reason, the access cannot count as clustering loss, because the feature of that node relevant to the query is its SP, not its page utilization or clustering. In the example in Figure 5.3, leaf 0 is accessed but contains no matching items, and therefore the access counts as excess coverage loss.

Utilization loss Deviation from the target utilization in the remaining leaves is summed up as utilization loss. In the example, leaf 2 has a utilization of 50%, which is $2/3$ of the target utilization of 75%, resulting in a loss of $1 - 0.5/0.75 = 1/3$. The idea behind this accounting is that if the pages had been packed more densely, part of the accesses could have been avoided. Note that a page utilization in excess of the target utilization counts as a negative performance loss, i.e., a performance gain.

Clustering loss Clustering loss is the difference between the conceptually "tightly packed" leaves in the index and the corresponding leaves in the optimal tree. The accessed leaves in the index become "tightly packed" by subtracting the utilization loss. In the example, the

result set is spread over three leaves, or $8/3$ tightly packed leaves. The difference between that and the two leaf accesses in the optimal tree is $2/3$, the clustering loss.

To summarize the leaf-level metrics established for the example query: excess coverage loss is 1 I/O, utilization loss is $1/3$ I/Os and clustering loss $2/3$ I/Os. The sum is 2 I/Os, which is the total performance loss that the example query experiences at the leaf level.

Internal-Level Performance

Although it is not possible to construct the optimal internal levels for the workload in a manner similar to the leaf level, the characteristics of the accessed internal nodes in the actual tree still allow us to derive two of the three metrics, namely excess coverage loss and utilization loss. The remaining internal-node accesses cannot be subdivided any further. More formally:

$$|I_q| = I_q^r + EL_q^i + UL_q^i.$$

Excess coverage loss Similar to the leaf-level metric, accesses to internal nodes without any matching entries are counted as excess coverage loss. In addition, we also count internal pages that do not lead to any leaves containing retrieved data; these internal pages are accessed due to excess coverage of SPs in the subtree. In the example, page 6 does not carry any matching SPs and its access is fully counted as excess coverage loss. Page 4 has a matching SP, but it only matches because of excess coverage in page 0's SP, so we count its utilization, $2/3$ of the target utilization, as excess coverage. The remaining $1/3$ are counted

as utilization loss, because, unlike the leaves of the traversal tree, the property of relevance of these nodes is not their SP but the SPs of their children, *i. e.*, the data contained in this node.

Utilization loss Similar to the corresponding leaf-level metric, the sum of the deviations from the target utilization is the utilization loss, excluding from consideration leaf nodes of the traversal path of the query. In the example, only page 4 causes the query to experience utilization loss at the internal levels in the amount of $1/3$ I/Os.

To summarize the preceding observations: of the 4 page accesses to internal nodes, $5/3$ are caused by excess coverage and $1/3$ by underutilization. The remaining 2 accesses to nodes 5 and 7 cannot be subdivided any further.

5.3.3 Node Performance Metrics

The per-node loss numbers are derived from the per-query loss numbers and show which parts of the tree contribute to performance deterioration. More specifically, these metrics show how a node's utilization and clustering properties as well as its SP affect workload performance. Generally, we sum up the per-query loss metrics across the nodes to arrive at per-node metrics. Similar to per-query metrics, we subdivide the accumulated performance loss of a leaf page into excess coverage, utilization and clustering loss. More formally:

$$|Q_p| = Q_p^o + EL_p^l + UL_p^l + CL_p, p \in L.$$

At the internal levels, we can only identify excess coverage and utilization loss; the remaining accesses cannot be subdivided any further. More formally:

$$|Q_p| = Q_p^r + EL_p^i + UL_p^i, p \in I.$$

Figure 5.3 will again be used as our running example.

Excess coverage loss A node's excess coverage loss is simply the number of times the node was accessed but no matching data was found. This does not take into account accesses to internal nodes that are caused solely by excess coverage in the children's SP, which are also classified as excess coverage loss. In this particular case it is the shared responsibility of the children, and it needs to be apportioned to them in some way. It is not clear how that should be done, so this type of excess coverage loss is presently not accounted for in the node performance metrics.[8]

In the example, we have pages 0 and 6 with excess coverage loss of 1 I/O each. The excess coverage loss of page 0 should also include the data accessed in page 4, but apportitioning this excess coverage loss to the children is not generally possible, as explained in the preceding paragraph.

[8] In the experiments conducted so far, those accesses played an insignficant role in comparison to the workload total. Note that the term Q_p^r also includes excess coverage loss created by child nodes that cannot be apportioned to the child nodes themselves.

Utilization loss A node's utilization loss is the product of its traversal count (minus those accesses caused by excess coverage) and its deviation from target utilization. In the example, pages 2 and 4 both have a utilization of 50%, a deviation of $1/3$ from the 75% target utilization.[9] If each of these pages were traversed 100 times across the entire workload, each one would contribute $33\frac{1}{3}$ I/Os to the entire workload utilization loss.

Clustering loss Each query's clustering loss needs to be distributed according to how much each accessed, non-empty leaf contributes to total clustering loss. We use as the guiding principle the quality of the clustering in a node *for the particular query in question.* The quality of clustering can be expressed as the ratio of accessed to retrieved data, and the optimal clustering establishes a benchmark ratio against which the accessed leaves in the actual tree will be measured.[10] In the example, the query accesses 2 leaves in the optimal tree to retrieve 5 data items, which fill up $5/3$ pages, resulting in a benchmark ratio of $6/5$. At leaf 3, the example query accesses 1 page worth of data in order to retrieve $1/3$rd of the page, although according to the benchmark ratio it should only have accessed $1/3 * 6/5 = 2/5$ of a page. The difference of $3/5$ is the clustering loss that the node contributes to this query. The corresponding numbers for pages 1 and 2 are $-2/10$ and $4/15$. The sum across these leaves is $2/3$, which is the total clustering loss for the query

[9] Conversely, if the target utilization were 45%, those pages would have recorded a utilization gain. Since utilization metrics record *deviation* from a constant, changing this constant does not affect performance difference between any two nodes.

[10] More formally: the pages in L'_q cause a loss of CL_q that needs to be distributed according to how much each page in L'_q contributes. Given L^o_q, we define a benchmark overhead ratio $r_q = |L^o_q| * C * u_t / R_q$. Given that ratio, we expect to access $r_q * R_{q,p}$ on each page p if clustering in the actual tree were as efficient as in the optimal tree. The difference $u_p * C - r_q * R_{q,p}$ is p's contribution to query q's clustering loss.

established in Section 5.3.2. The total per-node clustering loss is simply the sum of the per-node losses over the queries.

Example 1: Comparison of R- and R*-Trees

This example illustrates how to make an initial performance assessment with the help of the per-query and per-node metrics. We compare R- and R*-trees for range queries over 8-dimensional point data; we purposely chose to compare two well-known data structures, because knowing how they work will make the results of the analysis easier to follow.

The data set used in the experiment consists of 40,000 8-dimensional points, with each dimension limited to the interval $[0, 100)$, arranged into clusters of 100 points each. The clusters are box-shaped and have a diameter of 10; the center points of the clusters are distributed randomly. The trees were produced by bulk-loading 20,000 randomly selected data items and individually inserting the remaining 20,000.[11] This ensures that the split and insertion strategies are reflected in the resulting trees. Bulk-loading was done using the STR technique, which partitions the data points into iso-oriented tiles. We ran 20,000 square range queries over the trees, each with a side length of 12. The center points of the queries were randomly selected items from the data set, so that every query intersected with a cluster. On average, each query retrieved 20.6 items.

The aggregate results of this analysis are summarized in Table 5.3. We only report leaf-level performance numbers, since for this type of workload, R- and R*-trees are relatively

[11] The R*-tree variant used for this comparison does not attempt reinsertion, which was intentionally left out of the revised GiST abstraction of Chapters 3 and 4.

short and the upper levels can be buffered. Section 5.3.4 talks more about how to account for buffering.

	R*-tree [I/Os]	R-tree [I/Os]
actual tree, total	72,044	97,414
optimal clustering	23,262	23,224
utilization loss	4,650	3,906
excess coverage loss	16,895	30,171
clustering loss	27,237	40,113
sum	72,044	97,414

Table 5.3: Comparison of leaf-level performance in R- and R*-trees.

The performance numbers indicate that, as expected, R*-trees outperform R-trees, but that there is is still room for improvement.

Low utilization losses indicate that underutilization is not a problem. The target utilization was set to 80% and the average workload utilizations are close to that number (74.28% for the R*-tree and 75.75% for the R-tree).

Comparing clustering losses with those in the initial bulk-loaded tree confirms that the initial clustering is deteriorated by splits and insertions, although only to a moderate extent in the case of R*-trees. This can be deduced from the clustering *overhead*, which is the ratio of optimal accesses plus clustering loss to optimal accesses. For the R*-tree, this ratio is $(23262 + 27237)/23262 = 2.17$ and for the initial bulk-loaded tree it is $(10412 + 8903)/10412 = 1.86$. A possible reason for the relatively high clustering loss in the bulk-loaded tree is that by creating equi-distant partitions along each dimension, the STR algorithm cuts through clusters that exist in the data; since the queries are centered on the data points, breaking up clusters will also cause more page accesses.

Using amdb, we can see that in both cases the clustering loss is not spread evenly across the entire leaf level, but mostly confined to a few hot spots (this is shown in the global view, which is described in Section 5.1; we omit a screen shot of this particular scenario here because it is not fundamentally different from the one in Figure 5.2). The difference is that for the R-tree, these hot spots are more frequent and more stretched out.

Looking at per-node excess coverage loss in both trees, we can see that this is roughly co-located with clustering loss. This seems to suggest that the SP design works well for the clustering requirements of the workload, because we do not experience excess coverage loss where clustering loss is low. Intuitively, this is what we expect for minimum-bounding rectangles, because good clusters for this workload are rectangular, which results in tightly-fitting MBRs.

Example 2: Comparison of SPs for Nearest-Neighbor Searches on Multidimensional Points

This example illustrates how to evaluate and compare different SP designs independently of the remaining AM design aspects. We compare three different SP designs for a popular type of workload, nearest-neighbor queries on multidimensional point data. The three types of SPs are: minimum bounding rectangles, as employed in R*-trees [BKSS90]; minimum bounding spheres, as employed in SS-trees [WR96]; a combination of the two, which is used in SR-trees [KS97]. The latter two AMs were specifically designed for the type of workload that underlies our comparison.

The data set used in the experiment consists of 40,000 8-dim points, with each dimension limited to the interval $[0, 100)$, arranged into (uniformly distributed) clusters of 100 points each. The clusters are box-shaped and have a diameter of 10. The query set consists of 20,000 nearest-neighbor queries, each centered on a randomly selected (without replacement) data point and retrieving 20 items. In order to eliminate the effects of page utilization and clustering, the R*-, SS- and SR-trees were built by bulk-loading the leaf level, so that only their internal levels differ.

	Leaves	Internal	Total
R*	15061	51486	66547
SR	15003	61699	76702
SS	134094	173350	307444

Table 5.4: Excess coverage loss of SPs of R*-, SS- and SR-trees.

The measured excess coverage losses for the entire workload are shown in Table 5.4. Essentially, R*- and SR-tree SPs cause about the same amount of excess coverage loss, whereas the spheres of the SS-tree have about 10 times as much excess coverage loss. The reason is that the point sets in the leaves form clusters for which the MBRs have an aspect ratio that significantly deviates from 1. The corresponding spheres, which have a similar diameter as the MBRs, suffer from a much higher volume. The higher excess coverage loss of the SR-tree in comparison to the R*-tree is due to the increased storage requirements of their SPs, which decreases the fanout of internal nodes. Reducing the fanout leads to an increase in the number of nodes, which also increases the number of traversals caused by excess coverage.

The bad performance of spherical SPs in this example may well be an artifact of bulk-loading, which produces clusters that are often skinny along one or more dimensions. If the clusters would have a spherical shape, the result of the comparison might even favor spherical SPs. Intuitively, though, spherical SPs are less robust regarding the shape of the clusters, because, unlike rectangles, they have the same extent in all dimensions.

Another performance study that compares sphere and rectangle SPs [KS97] comes to a contrary conclusion, namely that spheres result in smaller-diameter SPs, because three separate elements of AM designs were evaluated together: by comparing insertion-loaded SR- and R*-trees, the insertion and split strategies also come into play and mask the performance effects of the SP design. This example illustrates the value of the excess coverage metric and the importance of separating individual aspects of an AM design.

Example 3: Unindexability Test

As part of constructing the optimal leaf level, we can perform a simple test that will tell us if a workload is not indexable,[12] even if it were possible to construct an optimal tree for it. This test is not limited to GiST-compliant AMs, but applies to all index structures that store indexed data on fixed-size pages.

The test can be stated as follows: *If in the optimal tree the aggregate number of leaf access for the entire workload takes longer than sequentially scanning the leaf level for each query, the workload should be considered unindexable.* The aggregate number of

[12]This test assumes that total execution time of the workload under consideration is dominated by page access cost.

leaf accesses in the optimal tree is a lower bound on the total number of page accesses for the entire workload, because minimally each query needs to access its result set. If this lower bound takes longer to execute than a sequential scan of the leaf level for each query, no actually constructed tree can be expected to outperform sequential scans. Since index accesses usually result in random accesses, a relatively small number of leaf accesses will take as long as a sequential scan of the entire level. The exact ratio of sequential to random accesses depends on the disk drives and the OS overhead, and we will assume a ratio of 14:1 as a conversion ratio representative of current technology.[13] Note that this test cannot be reversed: failing this criterion does not necessarily mean that a workload is indexable, because it might not be possible in practice to come close enough to the optimal clustering and SPs to achieve performance that will on average be better than a sequential scan. Also note that this test does not constitute a proof of unindexability, since in practice we can only approximate the optimal leaf-level clustering. Rather, the test should be seen as a strong hint, which becomes particularly compelling if one is unable to improve on the generated clustering by hand.

To illustrate the usefulness of the test, we look at two different kinds of workloads: nearest-neighbor queries on both uniform and clustered synthetic point data of moderate dimensionality (16 and 32). Such datasets are very popular for performance studies of ac-

[13]Using Seagate Barracuda ultra-wide SCSI-2 drives, [Rie98] measures a throughput of ca. 9MB/s under Windows NT. The average seek time and rotational delay for this drive are 7.1ms and 4.17ms, respectively. For 8KB transfers, this results in a ratio of 14 sequential I/Os for each random I/O. In the past years, raw drive throughput has increased faster than seek times and rotational delay have decreased, so the conversion ratio is likely to increase in the future.

cess methods for high-dimensional data such as feature vectors ([BBK98] is one example). The datasets we use for the analysis contain 10000 points each (experiments with 20000 and 40000 points give identical results for appropriately scaled result set sizes). When applying the unindexability test, the average result set size of the workload queries is important: if the average result set contains fewer items than the number of leaf pages divided by the conversion ratio, unindexability cannot be established. For the 16-dimensional data set, with with a target page capacity of around 40 points and 250 leaves, the threshold result set size is 18 points, or 0.18% of the data set. There is also a corresponding upper bound for the result set size, beyond which unindexability is ensured: a result set size in excess of the size of the data set divided by the conversion ratio. For the preceding example, this upper threshold is at around 7% of the data set.

Figure 5.4 plots the leaf accesses as a function of the result set size for the example data sets. To establish unindexability, it is sufficient for a workload to access more than 7% of the leaves. For the uniform 16-dimensional workload, this threshold is reached when result set sizes exceed about 0.3% of the data set size, a surprisingly small number. For the uniform 32-dimensional workload, the situation is a little better, because doubling the number of dimensions also doubles the storage size. Note, though, that the threshold result set size does not double as well. In contrast to uniformly distributed data sets, unindexability cannot be established for corresponding workloads involving clustered data sets, even for much larger result set sizes.

Unindexability of uniformly-distributed high-dimensional point data is confirmed by

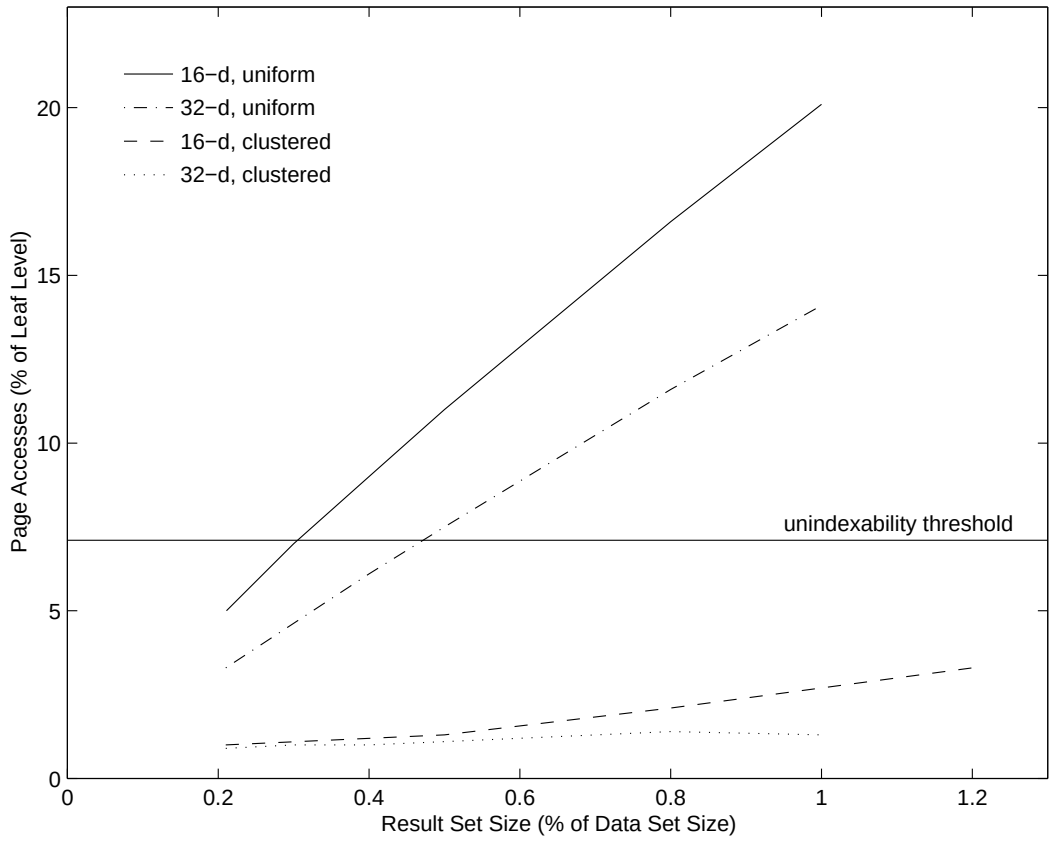


Figure 5.4: Results of the unindexability test for 16- and 32-dimensional uniformly distributed and clustered data.

a recently published theoretical analysis of nearest-neighbor queries([SBGR99]), which notes that for this type of data, increasing the dimension decreases the distance between the nearset and the farthest points. This implies that a given point is more likely to be a "nearest neighbor" for any query point in higher dimensions than in lower dimensions. As a result, a given point can be co-retrieved with a larger variety of points, making it more difficult to co-locate with all co-retrieved points. Note that our unindexability test is able to reach the same conclusion without knowledge of the data domain or the particular indexing problem. It can therefore be used as an automated first step in the AM design process.

Even if unindexability cannot be established, it is still instructive to look at the ratio of the number of workload leaf accesses in the optimal clustering to the number of pages needed to store the result sets. This ratio, which we will call the workload-optimal access overhead, is a measure of the inter-query "tension" in the workload: the higher this overhead, the more extra data must be accessed, even if the index achieves optimal clustering and is able to construct SPs without excess coverage. For example, the optimal access overhead of B-tree workloads is never worse than 2, and that of 2-dimensional uniform point data is 1.5 on average for 20-item result sets. On the other hand, that of 16-dimensional uniform point data is 12.2 and for 32 dimensions the corresponding ratio is 16.3. A correspondingly defined query-optimal access overhead can be used to find "atypical" queries in a workload, for which the overhead deviates noticeably from the average.

5.3.4 Other Performance Factors

The analysis framework presented so far completely ignored a number of components of the performance equation (CPU time, buffering, and comparison with approximations). We will now address these components individually and also comment on the usefulness of approximation numbers as the basis for our comparisons.

CPU Time Although CPU time can play an important role in the overall performance of an AM, it is excluded from the analysis framework. Since CPU time is not amenable to the same type of analysis as page accesses, it is unclear how to construct a model of optimal

CPU time behavior. Another drawback of CPU time is that it depends on the quality of the implementation and the particular hardware platform on which the analysis is run. This implies that these metrics are less general than page access-related metrics. Since CPU time can play an important role in overall execution cost, we suggest that an AM designer weigh it judiciously against the page access metrics of the analysis framework when deciding which aspects of the AM implementation need to be improved.

Buffering Buffering has been shown to reduce the number of I/Os for AM queries [LL98] and its presence—a standard feature in all commericial DBMS—will therefore change observed workload performance. We will outline several ways of taking buffering into account in the context of our analysis framework. A popular buffering technique for tree-structured AMs is to pin the first few levels of the tree ([LL98] reports that this technique never performed worse than LRU replacement in their experiments). Modifying the analysis metrics to take this into account is straightforward: the observed page accesses to those upper levels can simply be subtracted. For other buffering techniques, we can estimate an average hit rate and reduce the performance metrics uniformly by that rate. Either way, buffering can be dealt with separately and need not be integrated into our framework. Note that in order to integrate a realistic view of buffering into the framework, it is not sufficient to simulate a buffer pool/replacement strategy against a serial execution of the queries. In production DBMSs, queries are typically executed concurrently and index access is most likely interleaved.

Comparison with Approximation Numbers The performance metrics use the optimal tree as a point of reference. Unfortunately, in practice we can only approximate the optimal tree, which questions the utility of reported performance numbers. First, note that in the optimal tree, only clustering is approximated. Page utilization and SPs are stipulated to be perfect, and therefore the corresponding numbers accurately reflect the true performance loss. However, since no bounds on clustering quality are known for the heuristic algorithm we use for optimal clustering, the reported clustering loss numbers are only with regard to a "good" clustering rather than the optimum. Nevertheless, those numbers are still useful information for the AM designer: if the reported clustering loss is positive, clustering in the actual tree cannot be optimal and should therefore be a target for performance improvement. The number of cases in which negative clustering loss will be reported depends on the effective quality of the clustering algorithms. With the algorithm currently in use, we have not seen a single workload for which substantial negative clustering loss was reported.

5.3.5 Implementation

During the execution of the workload, amdb collects profiling data for each query individually, consisting of query result sets (references to retrieved items), visited pages, the number of bytes retrieved per page, etc. This puts a burden on the workload execution that is proportional to the cost of the execution itself, i.e., profiling a single page access or item retrieval incurs a small, constant cost, and is negligible. For example, 2,500 nearest-neighbor queries on 5,000 2-dimensional points took 12.3 seconds without profiling and

13.06 seconds with profiling on a Dell Dimension Workstation 333MHz Intel Pentium II processor. The size of the stored profiling data and performance metrics depends on a number of factors, such as the size of the result sets, tree size and excess coverage present in the tree, so it cannot be stated as a simple percentage of the tree size. Informally speaking, the sizes are fairly moderate. For example, the profile sizes for the workloads used in the unindexability tests in Section 5.3.3 range from 1.4MB (for 5,000 queries retrieving 21 of 10,000 16-dimensional points) to 40MB (for 20,000 queries retrieving 120 of 40,000 16-dimensional points).

Hypergraph partitioning is used to construct the optimal leaf level used for the query and node analysis, the optimal tree used for the implementation analysis and the optimal split used for the *pickSplit()* analysis. This task is performed by the public domain package `hMetis` from the University of Minnesota [KAKS97]. `HMetis` employs heuristics to approximate the optimal partitioning (which itself is NP-hard). Although designed primarily with VLSI applications in mind, we nevertheless found it to produce high-quality partitionings. As an example, we compared an R-tree bulk-loaded with 2-dimensional, Hilbert-value-sorted points with the equivalent `hMetis`-partitioned leaf level. The latter even slightly improved the clustering of the Hilbert-sorted leaf level (one has to keep in mind that even a perfectly square grid partitioning might be suboptimal for a given set of queries, because the queries might prefer a different grid origin or a different aspect ratio). We also found cases where the `hMetis`-produced clustering was inferior to space-partitioned [LLE97], bulk-loaded leaf levels, but the performance difference was minuscule

and the two clusterings were practically of the same quality. Using hypergraph partitioning to arrive at a clustering of the data items requires that each data item be covered by a sufficiently large number of queries, and furthermore that the queries themselves are sufficiently diverse (where establishing "sufficiently" is an area of future work). For the experimental results presented earlier, we tried to be conservative and executed half as many queries as there were data items. The queries themselves were centered on uniformly selected data items so that even coverage was ensured.

5.4 Related Work

5.4.1 Index Performance

Pagel, et al. [PSW95] study index clustering in a manner very similar to that of our analysis framework, also using an idealized goal of an optimal clustering to establish lower bounds on page accesses. They focus on window queries over multidimensional datasets, and apply simulated annealing to find an approximation to the optimal clustering. In their complexity analysis, they use a graph model for clustering that is not unlike the use of hypergraph partitioning.

The literature is rife with performance studies of various index structures, especially for multidimensional querying. Gaede and Günther survey over 50 different multidimensional index structures [GG98], most of which were introduced with a performance study to demonstrate their efficacy. [GG98] also surveys a number of comparative studies of mul-

tidimensional indexes, and attempts to unify the results into a partial ordering of quality; this is complicated by the variance in the workloads that the studies examine.

Most of the studies in the literature do not analyze performance results beyond comparing the number of page accesses on a given workload. Some studies provide analyses or intuitions of varying complexity to justify the page access measurements, often with domain- and workload-specific arguments. As an example, [BKSS90] explains (and visually illustrates) the efficacy of their node split technique with arguments about the virtues of square bounding boxes, which are not clearly translatable to other data domains, or to workloads of queries with high aspect ratio.

There is also a body of work on describing or predicting multidimensional index performance using formal models ([FK94, PSW95] are two examples). These papers provide insight into the performance of different indexing techniques on various synthetic workloads of queries and data. They often make rather strict assumptions about the workloads they model (e.g., many study only square queries). These models shed light on the challenges of multidimensional indexing in general, but are not necessarily helpful to a user studying a particular workload of queries and data. Mapping from a user's workload to one of these models is not generally possible.

5.4.2 Index Visualization and Animation

To our knowledge, amdb is the first tool of its kind to allow index developers to debug and analyze their implementations. Naturally, its various visualization and debugging

components have precedents in the literature. Amdb significantly extends many of these approaches, and unifies them into a single framework for index development.

There are a number of tools for visualizing and animating search tree data structures and algorithms; a compendium of references is maintained on the World-Wide Web [CCA]. Most of these tools focus on displaying tree structures, typically in a "nodes and arrows" visualization. This is useful only for pedagogical purposes, since such diagrams do not scale to the size of database indexes.

Brabec and Samet provide a suite of Java applets for a variety of 2-dimensional spatial database search trees, including R-trees and a host of quad-tree variants [BS98]. The visualizations focus on a geographic, 2-dimensional view of the *data domain*, akin to amdb's "node view" but spanning all nodes of one or more levels. Users may observe SPs and data items during insertion, deletion and splitting, with a large but fixed set of split algorithms. Some simple domain-specific statistics are displayed per level. Again, the focus of these tools seems to be pedagogic; the authors note that the visualizations do not scale to the fanouts typical in most trees. DEVise [LRB+97] is a general-purpose data exploration and visualization system, which has been demonstrated to be effective in helping R-tree development and debugging. As in the work of Brabec and Samet, DEVise was used in this scenario to visualize a 2-dimensional space containing data points and bounding rectangles. DEVise itself provides no facility for animating index algorithms or characterizing performance.

5.5 Conclusion

This chapter presents an analysis framework for tree-structured balanced AMs that can be used to evaluate the page access performance of user-defined query workloads. The framework is independent of the particular type of data to index or the nature of the queries. It only requires as input the data and tracing information gathered during query execution and the tree. The performance metrics it produces reflect actual performance loss, obtained by comparing the observed performance against that of an assumed optimal tree structure. The loss numbers are further decomposed to reflect the three fundamental structural performance factors: clustering, page utilization and the subtree predicates.

The AM design tool `amdb` incorporates the analysis framework as well as other features that support the design of GiST-compliant AMs. `Amdb` lets the user single-step through individual index operations and set breakpoints on events of interest. The visualization features allow navigation and inspection of the tree structure and the data contained in tree nodes. The latter is user-extensible, so that the visualization is not tied to a fixed set of data types. To facilitate the analysis process, `amdb` gathers the required tracing information during workload execution and displays the computed performance metrics both visually and textually.

In amdb, the analysis framework is combined with tree and data visualization and animation functionality to create a powerful design tool for access methods. The analysis process begins with the inspection of performance metrics to locate sources of deficiencies. Unlike data-dependent measures, these metrics objectively reflect access method perfor-

mance. The visualization and animation functionality then enable users to investigate those sources of performance loss and gain an understanding of how domain-specific properties affect performance. Based on this understanding, the designer incorporates improvements into the design and repeats the analysis process to evaluate their efficacy. This methodology was employed in several projects at U.C. Berkeley, in which amdb was an indisposable tool that allowed effective fine-tuning of AMs, showing significant improvements in a short amount of time.

There are several questions that could be investigated in more detail in the future. Section 5.3 mentions that for the hypergraph partitioning to produce "good" clusters—those that reflect semantic proximity of the data items—the queries in the workload must not only be representative, but also cover the entire data set to a sufficient degree. What the required number and shape of queries in a workload should be needs to be established more clearly. We also plan on extending the analysis framework to other, more exotic tree-structured access methods (such as non-balanced trees or key-transforming trees, such as R^+-trees) and hash-based access methods. The main challenge will be the construction of optimal structures for these AMs. Furthermore, we want to add functionality to amdb that allows it to compute user-defined metrics for queries, nodes and the split and insertion strategies. The metrics would express properties of the data and their organization within the tree that the designer believes to affect performance (for example, "small minimum-bounding rectangle overlap in R-trees results in good performance"). Comparing the user-defined metrics with those produced by our framework lets the designer verify the accuracy of his intuition and

forces him to revise it, if necessary.

Chapter 6

Conclusion

This dissertation explores a number of issues related to the design and implementation of non-traditional access methods. This chapter summarizes the contributions and gives an outline of unfinished areas of work that would benefit from future exploration.

6.1 Summary of Contributions

Chapter 3 extends the original GiST interface to address its functionality and performance deficiencies and reports on experiences with a GiST-based AM extensibility architecture implemented in IDS/UDO. The original GiST interface provides an abstraction of the indexed data domain and the specific operational properties of the AM, such as the split and insertion strategies. The extended GiST interface also provides an abstraction of the index pages themselves, which allows the AM developer to take full control over the internal layout of index pages. Additionally, it consolidates the call-per-entry extension

functions of the original interface into call-per-page functions, thereby reducing the UDF calling overhead substantially. The experiences with the IDS/UDO implementation of a GiST-based AM extension architecture validate the superiority of this approach to the traditional iterator-style mechanisms: B-trees and R-trees were implemented and debugged in a matter of days, rather than weeks or months, and each of these AMs requires only a fraction of the code necessary for a full implementation. Moreover, the performance tests showed that the flexibility and functionality of a GiST-based AM extension architecture does not compromise performance when compared to built-in, datatype-extensible AMs; in fact, the GiST-based R-tree exhibited better performance than its built-in counterpart.

Chapter 4 presents algorithms for physical concurrency control, transaction isolation and recovery in the GiST framework. Although concurrency control techniques have been studied extensively for B-trees, their application in the context of the more general GiST framework is often not possible. The reasons are the lack of a defined ordering and semantic information about the data in the GiST abstraction, as well as the lack of a partitioning of the data space. The proposed latching protocol that controls concurrent operations in GiSTs was derived from the B-link tree protocol and enhanced to avoid any references to the data stored in the index; its salient features are its deadlock freedom and that it does not hold any latches during I/Os, resulting in a degree of concurrency that should match that of the best B-tree concurrency control techniques. The proposed transaction isolation mechanism is a combination of 2-phase locking of existing data items and a node-local form of predicate locking, which is responsible for phantom protection. This hybrid

locking mechanism retains the accuracy and concurrency of pure predicate locking, but at the same time allows search operations of different isolation levels to be mixed freely and reduces runtime overhead by only requiring insertion operations to participate in the protocol. A simplification of this approach leads to a node-locking protocol, with lower implementation and runtime cost but also a lower degree of concurrency. Recoverability in the GiST framework is ensured by adapting a popular and effective B-tree logging protocol, which separates structure modifications from key insertions and deletions and which can be implemented in a standard write-ahead logging environment.

The extensions and techniques presented in Chapters 3 and 4 complete the original GiST design and turn it into a viable alternative to custom AM development in commercial environments, where performance and robustness are crucial. The experience gained with the GiST-based AM extensibility architecture implemented in IDS/UDO confirms that this approach is not merely of academic interest, but resulted in substantial time savings during AM implementation and generally a very high level of performance.

Chapter 5 presents an analysis framework for AMs that conform to the GiST model. The framework characterizes the I/O performance of an AM in the context of a given query workload. The central idea of this framework is the comparison of actual observed performance with performance in a workload-optimal tree, which leads to metrics that express performance loss. These loss metrics have a number of advantages over traditional performance metrics, such as aggregate runtime or I/O numbers or data-specific properties of the tree: they indicate the potential for future performance improvement, they present a fixed

point of reference and thereby allow an AM design to be assessed on its own, and they are not application or data specific. The workload-optimal tree can be partially determined by modeling the workload as a hypergraph and mapping the optimal clustering problem into a minimum-cut hypergraph partitioning problem. The latter is NP-complete, but can be approximated with sufficient quality. The availability of an (approximated) workload-optimal index leaf level also makes it possible to formulate a test for unindexability of a dataset under a given query workload. Using this test, it is possible to show that several data/workload combinations, that are popular for performance evaluations in the research literature, are in fact unindexable. The tool amdb combines the analysis framework with a visualization environment for the tree structure and contents, which allows the AM developer to recognize performance deficiencies in the tree structure and identify particularly badly performing queries. The effectiveness of this tool and the underlying framework has been confirmed in a number of AM design projects, where amdb was instrumental in focusing the developers' attention on the most substantial performance deficiencies and in measuring and clarifying the effect of design changes.

6.2 Future Work

The work presented in this dissertation has also led to a better understanding of additional research opportunities related to the design and implementation of non-traditional AMs. I will first discuss remaining problems concerning the GiST data structure and then

challenges regarding AM evaluation and automated design.

6.2.1 GiST Extensions

While it is clear that a template AM such as the GiST is necessary in order to integrate new AMs into DBMSs at reasonable cost and effort, it is less self-evident whether the abstraction offered by GiST will be sufficient to capture a wide enough range of AMs. The experience gained with the IDS/UDO implementation suggests that at least an extension to deal with multi-entry objects (such as documents, which are represented in a keyword index with multiple entries) would be beneficial.

6.2.2 AM Concurrency Control and Recovery

Out of the various subtopics that comprise the general area of AM concurrency control and recovery, transactional isolation appears to be the one that still lacks a satisfying solution. The latching and logging protocols guarantee a high degree of concurrency, while introducing little runtime overhead or implementation complexity. This is not true of the proposed transactional isolation mechanisms, which present a clear tradeoff between complexity and runtime overhead on one hand and concurrency on the other. Fortunately, this problem is less pressing than that of physical concurrency control, and in practice seems to be handled by sacrificing isolation for concurrency; i.e., applications are willing to lower their isolation levels to obtain higher concurrency. Nonetheless, a solution that offered high degrees of concurrency *and* isolation would certainly be preferable. Specific questions for

future research are:

- Can the simple node locking protocol provide a high enough degree of concurrency, or does the more complex hybrid protocol offer significant advantages? This probably depends on the application scenario and the workload, but a characterization of which workloads are amenable to the simple node locking protocol would be useful.

- Assuming that the simple node locking protocol is not sufficient in all cases, is there a technique that does not require a concurrency versus implementation complexity/runtime overhead tradeoff? In practice, high concurrency is crucial in most high-throughput web-driven application, and even the degree of concurrency attainable with a predicate-based locking scheme may be too low. The popularity of products that employ transient versioning to improve concurrency (e.g., Oracle), which eliminates read-write conflicts and waits, might be an indication that future research on transaction isolation should investigate versioning instead of locking.

6.2.3 AM Design and Evaluation

While the implementation of AMs has been studied for some time, the same is not true for a rigorous approach to design and evaluation, and many questions remain in this area. The analysis framework of Chapter 5 gives the designer a usable procedure for the evaluation of static index trees, but leaves open questions regarding workload construction, evaluation of dynamic index trees and automated index repair. More specifically:

- What is the appropriate number of queries in a workload? The analysis framework introduced in Chapter 5 requires as part of the input a query workload, which is crucial to the significance of the analysis: if the workload does not "cover" the data set sufficiently, the effectiveness of the hypergraph partitioning is compromised (uncovered items are assigned randomly to clusters) and the performance metrics would present a distorted view. Currently, there are no guidelines to evaluate whether a workload's coverage is sufficient.

- How can a trace-gathered workload be reduced in size? Workloads derived from traces that are gathered from production databases tend to be very large, which makes the analysis process prohibitively time consuming. The challenge is to reduce the data set and workload to make them more manageable, while at the same time preserving the performance-relevant properties of the workload, such as the overlap between result sets.

- How can the framework be extended to *explain* performance rather than just assess it? The analysis framework presents a performance picture of a static version of an index tree, but it does not tell the developer how performance changes as the tree evolves dynamically. A straight-forward way to achieve this would be to re-evaluate the workload after each update operation, but this seems problematic for two reasons: first, the runtime overhead this introduces would be prohibitive for all but the smallest workloads; second, the content of the tree changes, while the workload stays

static, which means that the workload will not be able to cover the tree to the same extent during the entire evaluation. A more feasible alternative could focus on user-defined metrics, which reflect data semantics and could therefore be computed more cheaply from the contents of the tree nodes. A chronological record of the user-defined metrics in response to tree updates would let the designer trace back and explain the formation of performance deficiencies in the tree structure. Of course, metrics reflecting data semantics are only useful if they are correlated with observed performance, which could be established using amdb's standard performance metrics.

- Can the performance metrics be used to guide an automated index "repair" process? Given an insight into the causes of performance deficiencies, it would be interesting to investigate the extent to which these can be rectified automatically by an online index repair algorithm. This algorithm would (1) monitor the performance of search operations in the tree and (2) rebuild parts of the tree structure, triggered by observed performance deterioration. Rebuilding the tree structure should be performed without unduly restricting concurrency and would consist of two separate tasks: reorganization of subtrees to improve clustering and page utilization, and automatic SP refinement to reduce excess coverage loss. The former could be accomplished with the tree bulk-loading algorithm presented in [VdBSW97]; the latter could be achieved by employing the *pick_split()* extension function to compute a composite SP for a page, consisting of the left and right SP of a "virtual page split", thereby in-

creasing the amount of descriptive information recorded for a page. Monitoring the performance of search operations and detecting performance deterioration could be based on the analysis framework, whose benchmark numbers for clustering (ratio of accessed to retrieved data), page utilization and excess coverage could be used to calibrate trigger thresholds during runtime. The design of such a repair algorithm in the GiST context is particularly challenging if the introduction of additional extension functions is to be avoided: using the same extension function that led to the performance deficiencies in the first place to rectify those deficiencies may not always be successful.

Bibliography

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

[Aok98] P. Aoki. Generalizing “Search” in Generalized Search Trees. In *Proc. of the 14th Int. Conf. on Data Engineering, Orlando, USA*, pages 380–389, 1998.

[BBG+88] D. Batory, J. Barnett, J. Garza, K. Smith, K. Tsukuda, B. Twichell, and T. Wise. GENESIS: An Extensible Data Management System. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, 1988.

[BBK98] S. Berchtold, C. Böhm, and H.-P. Kriegel. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Seattle, Washington*, pages 142–153, 1998.

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

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. ACM SIGMOD Conf.*, pages 322–331, 1990.

[BRS96] S. Blott, L. Relly, and H.-J. Schek. An Open Abstract-Object Storage System. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Montreal, Quebec*, June 1996.

[BS77] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. *Acta Informatica*, 9:1–21, 1977.

[BS98] Frantisek Brabec and Hanan Samet. Visualizing and Animating R-Trees and Spatial Operations in Spatial Databases on the Worldwide Web. In *Proceedings of Visual Database Systems—VDB 4, L'Aquila, Italy*, May 1998.

[BSSJ99] R. Bliujute, S. Saltenis, G. Slivinskas, and C. Jensen. Developing a DataBlade for a New Index. In *Proc. of the 15th Int. Conf. on Data Engineering, Sydney, Australia*, pages 314–323, March 1999.

[CCA] The Complete Collection of Algorithm Animations. http://www.cs.hope.edu/ãlganim/ccaa/ccaa.html.

[CDF+86] M. Carey, D. DeWitt, D. Frank, G. Graefe, M. Muralikrishna, J. Richardson, and E. Shekita. The Architecture of the Exodus Extensible DBMS. In *Proc.*

of the 1st Int'l Workshop on Object-Oriented Database Systems, pages 52–65, 1986.

[CDF+94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Minneapolis, Minnesota*, pages 383–394, 1994.

[CM98] K. Chakrabarti and S. Mehrotra. Dynamic Granular Locking Approach to Phantom Protection in R-Trees. In *Proc. of the 14th Int. Conf. on Data Engineering, Orlando, USA*, pages 446–454, 1998.

[CM00] K. Chakrabarti and S. Mehrotra. Efficient Concurrency Control in Multidimensional Access Methods. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Philadelphia, PA*, pages 25–36, 2000.

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

[CTB+99] C. Carson, M. Thomas, S. Belongie, J. Hellerstein, and J. Malik. Blobworld: A System for Region-Based Image Indexing and Retrieval. In *VISUAL*, pages 509–516, 1999.

[DeW96] D. DeWitt. Combining Object-Relational and Parallel: Like Trying to Mix Oil and Water? http://www.cs.wisc.edu/ dewitt/vldbsum.ps, 1996.

[DM97] S. Dessloch and N. Mattos. Integrating SQL Databases with Content-Specific Search Engines. In *Proc. 23rd Int'l Conference on Very Large Databases (VLDB), Athens, Greece*, pages 528–537, 1997.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate locks in database systems. *Communications of the ACM*, 19(11):624–633, November 1976.

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

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

[GJ79] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1979.

[GR93] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[Gra78] J. Gray. Notes on Database Operating Systems. In *Operating Systems –*

An Advanced Course, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

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

[HCL+90] L. Haas, W. Chang, G. Lohman, P. McPherson, P. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, Carey. M., and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *TKDE*, 2(1):143–160, 1990.

[HKP97] J. Hellerstein, E. Koutsoupias, and C. Papadimitriou. On the Analysis of Indexing Schemes. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Tucson, Arizona*, pages 249–256, 1997.

[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.

[ISO96] ISO/IEC JTC1/SC21 N10491, ISO//IEC 9075, Part 8, Committee Draft (CD), Database Language SQL Part 8: SQL/Object, July 1996.

[JBB81] J. R. Jordan, J. Banerjee, and R. B. Batman. Precision Locks. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 143–147, 1981.

[JS93] T. Johnson and D. Shasha. The Performance of Current B-Tree Algorithms. *ACM TODS*, 18(1), March 1993.

[KAKS97] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. In *Proc. ACM/IEEE 34th Design Automation Conference*, 1997.

[KB95] M. Kornacker and D. Banks. High-Concurrency Locking in R-Trees. In *Proc. 21st Int'l Conference on Very Large Databases (VLDB), Zürich, Switzerland*, pages 134–145, September 1995.

[KS97] N. Katayama and S. Satoh. The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Tucson, Arizona*, pages 369–380, May 1997.

[LJF94] K. Lin, H. Jagadish, and C. Faloutsos. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB Journal*, 3, October 1994.

[LL98] S. T. Leutenegger and M. A. López. The Effect of Buffering on the Per-

formance of R-Trees. In *Proc. of the 14th Int. Conf. on Data Engineering, Orlando, USA*, pages 164–171, February 1998.

[LLE97] S. T. Leutenegger, M. A. López, and J. M. Edgington. STR: A Simple and Efficient Algorithm for R-tree Packing. In *Proc. of the 13th Int. Conf. on Data Engineering, Birmingham, England*, pages 497–506, April 1997.

[Lom91] D. Lomet. Grow and Post Trees: Role, Techniques and Future Potential. In *Proc. 2nd Symposium on Spatial Databases*, pages 183–206. Springer-Verlag, August 1991.

[Lom93] D. Lomet. Key Range Locking Strategies for Improved Concurrency. In *Proc. 19th Int'l Conf. on Very Large Databases (VLDB)*, August 1993.

[LRB+97] Miron Livny, Raghu Ramakrishnan, Kevin Beyer, Guangshun Chen, Donko Donjerkovic, Shilpa Lawande, Jussi Myllymaki, and Kent Wenger. DEVise: Integrated Querying and Visual Exploration of Large Datasets. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Tucson, Arizona*, 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.

[LY81] P.L. Lehmann and S.B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM TODS*, 6(4):650–670, December 1981.

[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.

[ML92] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *Proc. ACM SIGMOD Conf.*, June 1992.

[Moh90a] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *Proc. 16th Int'l Conference on Very Large Databases (VLDB)*, August 1990.

[Moh90b] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proc. 16th Int'l Conference on Very Large Databases (VLDB)*, August 1990.

[Moh95] C. Mohan. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, chapter Concurrency Control and Recovery Methods for

B+-Tree Indexes: ARIES/KVL and ARIES/IM. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[MPL92] C. Mohan, H. Pirahesh, and R. Lorie. Efficient and Flexible Methods for Transient Versioning of Records to Avoid Locking by Read-Only Transactions. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Diego, California*, June 1992.

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

[PSW95] B.-U. Pagel, H.-W. Six, and M. Winter. Window query-optimal clustering of spatial objects. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, SanJose, California*, pages 86–94, 1995.

[Rat] Rational Software Corp. Quantify for Sun. http://www.rational.com.

[Rie98] Erik Riedel. A Performance Study of Sequential I/O on Windows NT 4. In *Proc. 2nd USENIX Windows NT Symposium, Seattle, WA*, 1998.

[Sag86] Y. Sagiv. Concurrent Operations on B*-Trees with Overtaking. Journal of Computer and System Sciences, 33(2):275–296, 1986.

[SBGR99] U. Shaft, K. Beyer, J. Goldstein, and R. Ramakrishnan. When Is “Nearest

Neighbor" Meaningful? In *7th International Conference on Database Theory, Jerusalem, Israel*, January 1999.

[SC91] V. Srinivasan and M. Carey. Performance of B-Tree Concurrency Control Algorithms. In *Proc. ACM SIGMOD Conf.*, pages 416–425, 1991.

[SCF+86] P. Schwarz, W. Chang, J. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst Database System. In *Proc. of the 1st Int'l Workshop on Object-Oriented Database Systems*, pages 85–92, 1986.

[SG88] D. Shasha and N. Goodman. Concurrent Search Structure Algorithms. *ACM TODS*, 13(1), March 1988.

[SK91] M. Stonebraker and G. Kemnitz. The Postgres Next Generation Database Management System. *CACM*, 34(10):78–92, 1991.

[SKH99] M. Shah, M. Kornacker, and J. Hellerstein. Amdb: A Visual Access Method Development Tool. In *UIDIS, Edinburgh, UK*, pages 130–140, 1999.

[SR86] M. Stonebraker and L. Rowe. The Design of Postgres. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Washington, DC*, pages 340–355, 1986.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic

Index for Multidimensional Objects. In *Proc. 13th Int'l Conference on Very Large Databases (VLDB)*, pages 507–518, September 1987.

[Stü95] G. Stürner. *Oracle7: A User's and Developer's Guide*, chapter 4.4. International Thomson Computer Press, 1995.

[TCH00] M. Thomas, C. Carson, and J. Hellerstein. Creating a Customized Access Method for Blobworld. In *Proc. of the 16th Int. Conf. on Data Engineering, San Diego, USA*, page 82, 2000.

[vdBDS00a] J. van den Bercken, J.-P. Dittrich, and B. Seeger. javax.XXL: A Prototype for a Library of Query Processing Algorithms. http://www.mathematik.uni-marburg.de/DBS/xxl/XXL.pdf, 2000.

[vdBDS00b] J. van den Bercken, J.-P. Dittrich, and B. Seeger. javax.XXL: A Prototype for a Library of Query Processing Algorithms. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Dallas, TX*, 2000.

[VdBSW97] J Van den Bercken, B. Seeger, and P. Widmayer. A Generic Approach to Bulk Loading Multidimensional Index Structures. In *Proc. 23rd Int'l Conference on Very Large Databases (VLDB), Athens, Greece*, pages 406–415, 1997.

[WR96] D. A. White and Jain. R. Similarity Indexing with the SS-Tree. In *Proc. of*

the 12th Int. Conf. on Data Engineering, New Orleans, USA, pages 516–523, February 1996.