An Analysis Framework for Access Methods

Marcel Kornacker
U. C. Berkeley
Phone: (510) 642 8072
Fax: (510) 642 5615
marcel@cs.berkeley.edu

Mehul Shah
U. C. Berkeley
mashah@cs.berkeley.edu

Joseph M. Hellerstein
U. C. Berkeley
jmh@cs.berkeley.edu

Abstract

Designing and tuning access methods (AMs) has always been more of a black art than a rigorous discipline, with performance assessments being mostly reduced to presenting bottom-line runtime or I/O numbers. This paper presents an analysis framework for AMs that defines performance metrics which are more meaningful than bottom-line numbers and thereby allow the AM designer to detect and isolate deficiencies in an AM design. The analysis process takes a workload—a tree and a set of queries—as input and provides metrics that characterize the performance of each query as well as that of the tree structure and the structure-shaping aspects of the AM implementation. 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 measured. The performance metrics themselves reflect the fundamental performance-relevant properties of the input tree. The framework applies to most balanced tree-structured AMs and is not restricted to particular types of of data or queries. It is implemented in amdb, a comprehensive graphical design tool for AMs that are constructed on top of the Generalized Search Tree abstraction. Amdb complements the analysis framework with visualization and debugging functionality, allowing the AM designer to investigate the source of those deficiencies that were brought to light with the help of the analysis framework.

1 Introduction

Despite the large and growing number of access methods (AMs) that have been produced by the research community—and also despite their increasing importance, considering the explosion of data users find worth querying—the design and tuning of AMs has always been more of a black art than a rigorous discipline. Traditionally, performance analyses are presented in terms of aggregate runtime or page access numbers. The drawback is that these numbers do not allow the contributions of individual design ideas to be quantified. As a result, it is hard to explain performance differences between competing AM designs, if those deviate in more than one design aspect. Also, aggregate runtime or access numbers do not allow AMs to be assessed on their own, because competing AM designs are needed to put the numbers into perspective.

In this paper we present an analysis framework for tree-structured, height-balanced AMs that provides more meaningful performance metrics than just aggregate numbers and can be applied to any workload, regardless of the type of data or nature of the queries involved. Its salient features are:

- 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. This allows the framework to be used to tune an AM for a specific workload and to compare workloads by running them against the same AM.
- 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 thus reflect performance objectively. On the other hand, metrics that express performance in terms of semantic properties require the designer to understand their correlation with page accesses. Since such an understanding is a *goal* of the analysis process, any apriori assumptions are often incorrect and misleading.
- Central to the analysis is the comparison of observed performance with optimal performance, i. e., performance that would have been obtained with a tree that is optimal for the input workload. The performance metrics are derived from this comparison and express performance loss. With such a point of reference, the observed performance can be put into perspective without having to compare with a competing AM design. Moreover, this particular point of reference shows the potential for performance improvement, which cannot necessarily be discovered by comparing two alternative AM designs.
- The framework defines performance loss metrics for each query of the workload, for the nodes of the input tree and for the structure-shaping aspects of the AM implementation. Furthermore, those metrics are broken down to reflect the fundamental performance-relevant properties of tree-structured AMs. Such a breakdown is more useful than aggregate numbers, because it facilitates assessing the performance effects of AM design aspects individually.

The analysis framework is implemented in amdb, a comprehensive visual design tool for AMs built on top of the Generalized Search Tree (GiST) abstraction ([HNP95]). Its features include: interactive execution of search, insert and delete operations; support for breakpoints and single-stepping through operations; visualization of the tree structure and node contents (the latter being user-extensible); execution of query workloads, gathering of tracing information and visual presentation of performance metrics and tracing information. In order to compute performance loss metrics, amdb approximates part of the workload-optimal tree, namely the optimally clustered leaf level. This is achieved by modelling the input workload as a hypergraph and approximating the optimal clustering via a heuristic hypergraph partitioning algorithm.

The rest of the paper is structured as follows. Section 2 gives an overview of amdb and describes the integration of the analysis framework into a graphical development environment. Section 3 briefly introduces GiST. Section 4 contains a discussion of the analysis framework, along with illustrative examples, among them a test for unindexability. Section 5 discusses related work and Section 6 contains the conclusion and an outline of future work.

2 Amdb: A Design Tool for Access Methods

The goal of the development of amdb[1] was to provide the AM designer with a comprehensive tool that would cover the entire design process, ranging from debugging the initial implementation to fine-tuning of an AM for a specific workload. At the core of amdb is the analysis framework that is the topic of this paper; it 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); 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.

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 packages can be downloaded from http://gist.cs.berkeley.edu/.

This section describes the visualization and debugging features and gives an overview of how the analysis framework is presented to the user.

2.1 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 1). 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 2.3). 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 1: 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 1: 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 1: 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 1: 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 1: 5). Since tree nodes contain arbitrary user-defined keys,[2] the access method designer must provide a module which displays the node given its contents. Currently, amdb contains a suite of modules which visualize two-dimensional projections of spatial data. One convenient feature of the node view is that it highlights the current path in the tree view. The node view also allows the user to simulate a split[3] 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.

2.2 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.

2.3 Overview of the Analysis Process

The analysis framework described in Section 4 defines performance metrics for each query of the owrkload, node of the input tree and for the structure-relevant split and insertion strategies of the AM design. These metrics point out deficiencies in the AM and tell the designer which parts of the input tree or which of the queries to focus on. The visualization and debugging features complement the

[1] An initial implementation is briefly described in [KSH98].

[2] The GiST abstraction, described in more detail in Section 3, factors out structural and algorithmic aspects that are common to most balanced tree-structured AM.

[3] This is achieved by calling the *pickSplit()* extension function, which will be introduced in Section 3.

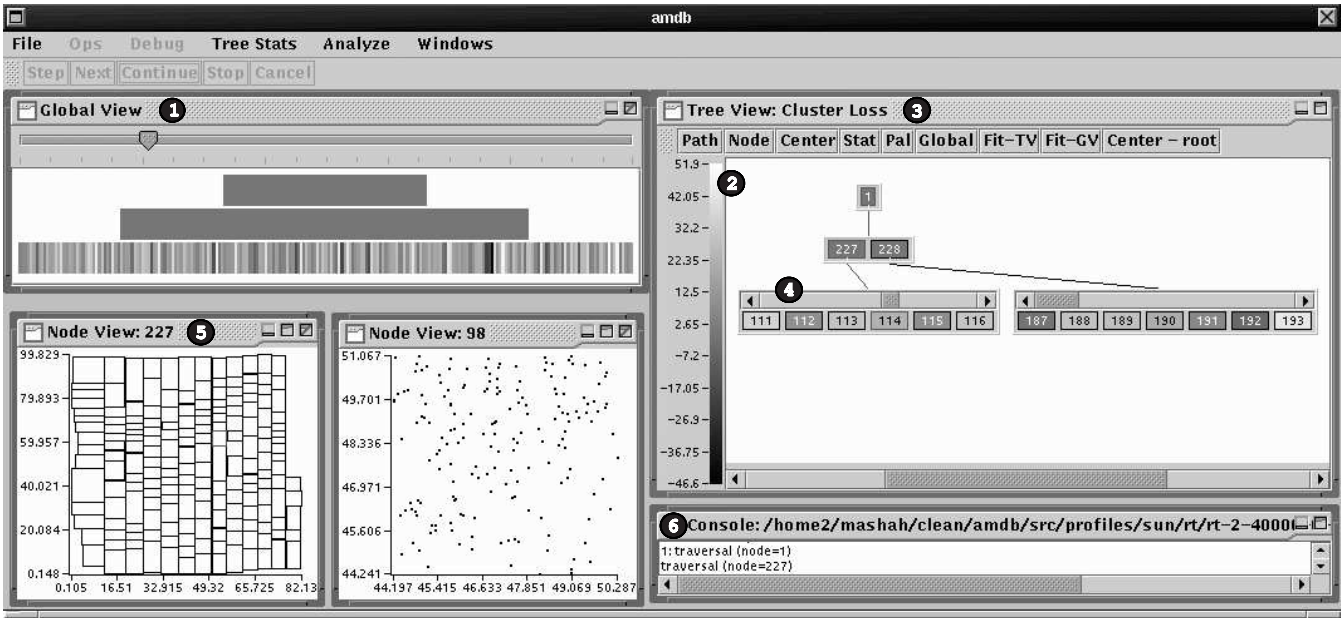

Figure 1: Amdb User Interface

performance metrics by giving the designer the means to investigate and understand the source of the deficiencies.

The per-query metrics show the performance loss for each query and pinpoint badly performing queries. These metrics are complemented with tracing data gathered during query execution, including traversal paths, CPU execution time, the amount and specific location of data retrieved, *etc*. This tracing data gives the developer a very detailed view of the behavior of each query and is instrumental in understanding poorly performing queries.

Per-node metrics show which nodes in the tree contribute to performance loss; they are computed for each query and for the aggregate workload. The performance numbers are visualized via coloring of nodes in the global and tree view, so that ill-behaved parts of the tree can be identified easily. The navigation and data visualization features of these views let the developer examine those parts of the tree structure and the data contained therein. Aside from performance numbers, these views also visualize per-query tracing data; for example, traversal paths and per-node CPU execution times can be visualized very effectively through node coloring.

AM implementation metrics show how workload performance is affected by splits and insertions. This gives the developer direct feedback about the quality of the AM design and points out cases where the design fails. The actual splits and insertion paths that deteriorate workload performance can be visualized with the node and tree view, respectively.

Performing an analysis of an existing tree requires very little user interaction. Essentially, the developer only needs to prepare a script containing the queries of the workload (and a file with keys for the insertion strategy analysis). `Amdb` executes this script against the input tree, collects the required tracing data, and computes the performance numbers, which are then shown in dialog boxes for easy browsing. The tracing data and performance numbers are stored in a file to avoid recomputation for subsequent `amdb` sessions.

3 Generalized Search Trees

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

We now provide a sketch of the implementation of the operations and how they use the extension methods. For a more detailed

[4] This structural requirement excludes R^+-trees ([SRF87]) from conforming to the GiST structure.

description, together with examples of B-tree and R-tree extension methods, see the original paper ([HNP95]).

SEARCH In order to find all leaf entries satisfying the search predicate, we recursively descend *all* subtrees for which the parent entry's predicate is consistent with the search predicate (employing the user-supplied extension method *consistent()*).

INSERT Given a new *(key, RID)* pair, we must find a leaf to insert it on. Note that because GiSTs allow overlapping SPs, there may be more than one leaf where the key could be inserted. A user-supplied extension method *penalty()* compares a key and predicate and computes a domain-specific penalty for inserting the key within the subtree whose bounds are given by the predicate. Using this extension method, we traverse a single path from root to leaf, following branches with the lowest insertion penalty.

If the leaf overflows and must be split, a extension method, *pickSplit()*, is invoked to determine how to distribute the keys between two leaves. 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 takes two predicates, one of which is the new key, and returns their union. Like node splitting, expansion of predicates in parent entries is carried out bottom-up until we find an ancestor node whose predicate does not require expansion.

DELETE In order to find the leaf containing the key we want to delete, we again traverse multiple subtrees as in SEARCH. Once the leaf is located and the key is found on it, we remove the *(key, RID)* pair and, if possible, shrink the ancestors' SPs.

Although the GiST abstraction prescribes algorithm for searching and inserting, the AM designer still has full control over the performance-relevant structural characteristics of the AM. These structural characteristics 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.

Subtree Predicates While the size and shape of the indexed data is part of the input (if the data can be compressed, this should be done in any case), 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.

4 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 4.5 addresses these issues.

Instead of simply measuring the number of page accesses, a more meaningful performance metric is the difference between the number of page accesses in the actual tree and the optimal tree; we call this difference the the *performance loss*. The optimal tree is defined as minimizing the total number of page accesses over the entire workload. 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 analysis framework defines performance metrics that are based on the performance loss and fall into three groups:

Query Metrics A query will experience a performance loss if the actual tree has inferior clustering, page utilization, or SPs relative to the optimal tree. In order to understand the nature of the loss, we break down the total loss to reflect each of these shortcomings. The breakdown reveals how much of a query's performance loss is due to suboptimal clustering, page utilization and SPs.

Node Metrics Similar to the query metrics, the framework defines node metrics that express an individual node's contribution to aggregate workload performance loss, broken down to reflect the losses cause by the node's clustering, utilization and SP. Such metrics are valuable because they help the AM designer identify anomalies in the tree structure.

Implementation Metrics The extension methods *pickSplit()* and *penalty()* directly control the tree structure and their performance metrics should express to what extent they are responsible for the structural deterioration that causes performance loss. Unlike query and node metrics, the implementation metrics cannot be derived from the tracing information gathered during workload execution. Instead, we execute additional splits and insertions and observe how workload performance changes. Like query and node metrics, the implementation metrics reflect a comparison to an optimum, in this case the optimal split and insertion.

The following subsection discusses the optimal tree and how to construct it. Section 4.2 derives the query performance metrics, first for the leaf level, then for internal levels, and presents examples of analyses conducted with these metrics. Section 4.3 derives node metrics based on the query metrics. Section 4.4 discusses the optimal split and insertion and derives metrics for the *pickSplit()* and *penalty()* extension methods; an example illustrates these metrics and completes one of the analyses begun in Section 4.2.

The presentation of the metrics in this section is purposely informal and relies mainly on examples; we felt this would improve readability. The input variables and metrics are defined and summarized in Table 1 and Table 2, respectively.[5] Variables with subscript q are query-specific and variables with subscript p are page-specific.

4.1 Construction of the Optimal Tree

The optimal tree is defined by the following characteristics:

[5]We leave out the definition of the split and penalty metrics, because these are cumbersome and can be derived from the descriptions in Section 4.4.

CL_q	clustering loss	$CL_q = u_q/u_t \lvert L'_q \rvert - \lvert L^o_q \rvert$
EL^l_q	leaf-level excess coverage loss	$EL^l_q = \lvert L_q \rvert - \lvert L'_q \rvert$
UL^l_q	leaf-level utilization loss	$UL^l_q = \lvert L_q \rvert (1 - u_q/u_t)$
$EL^i_{p,q}$	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	$EL^l_p = \lvert Q_p \setminus Q'_p \rvert$
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	$EL^i_p = \lvert \{ q \mid p \in I^l_q \} \rvert$
UL^i_p	internal-level utilization loss	$UL^i_p = \sum_{\{ q \in Q_p \mid p \notin I^l_q \}} 1 - u_p/u_t$
Q^r_p	remainder of internal-level accesses	$Q^r_p = \sum_{\{ q \in Q_p \mid p \notin I^l_q \}} u_p/u_t$

Table 2: Performance Metrics

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^o_q	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_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^l_q	internal "leaves" of traversal paths, $I^l_q = \{ p \mid p \in I_q \setminus I'_q \wedge \neg(child(p) \in I_q \cup L_q) \}$
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, $r_q = \lvert L^o_q \rvert * C * u_t / R_q$
$R_{p,q}$ [bytes]	size of fraction of q's result set found on p
$Q^o_{p,q}$ [bytes]	optimal amount of accessed data, $Q^0_{p,q} = r_q * R_{p,q}$
Q^o_p [bytes]	optimal amount of accessed data aggregated over workload, $Q^o_p = \sum_{q \in Q'_p} r_q * R_{p,q}$

Table 1: Input Variables (Profiling Data, Tree Statistics and Derived Variables)

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. For that reason, the AM designer can specify a desired target page utilization, which serves as a point of comparison for nodes within the tree structure. 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.

optimal clustering, which minimizes the total number of "relevant" page accesses (at the leaf level, those are accesses to pages containing items of the result set of a query, see Table 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[6] 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 4.6 discusses the implementation, in particular the hypergraph partitioning, in more detail).

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 2 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%. Occupied slots are shaded, and the pages in the actual tree are enumerated for reference.

[6] 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 individual queries' clustering requirements are contradictory.

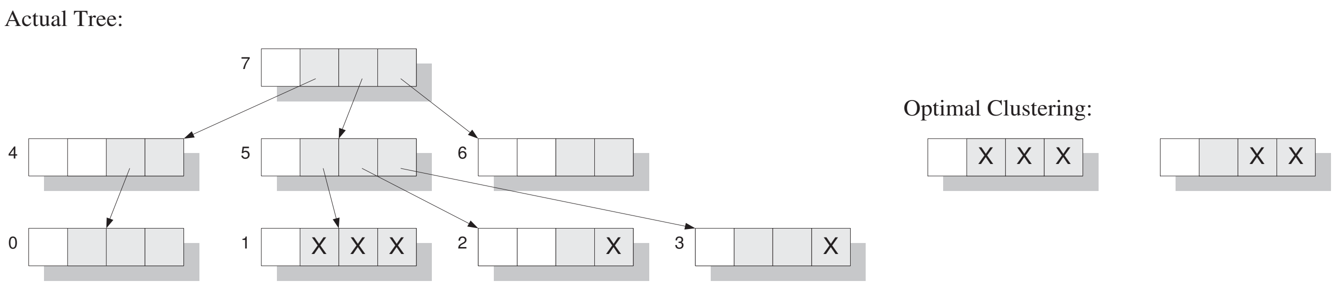

Figure 2: Traversal Paths and Optimal Clustering for Example Query

4.2 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.

4.2.1 Leaf-Level Performance Metrics

For each query, the performance loss at the leaf level—actual minus optimal 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. We show 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. Even if those pages are underutilized do they not count toward utilization loss, 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 2, 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, utilization loss is $1/3$ and clustering loss $2/3$. The sum is 2 accesses, which is the total performance loss that the example query experiences at the leaf level.

4.2.2 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 $1/3$.

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.

4.3 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 2 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 metrics.[7]

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

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.[8] If each of these were traversed 100 times across the entire workload, each one would contribute $33\frac{1}{3}$ accesses to the entire workload performance.

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.[9] 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 1.2. 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 $1/3 * 1.2 = 40\%$ of a page. The difference of 60% is the clustering loss that the node contributes to this query. The corresponding numbers for pages 1 and 2 are -0.2 and $4/15$. The sum across these leaves is $2/3$, which is the total clustering loss for the query established in Section 4.2.1. The total per-node clustering loss is simply the sum of the per-node losses over the queries.

4.3.1 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 40000 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 20000 randomly selected data items and individually inserting the remaining 20000. This ensures that the split and insertion strategies are reflected in the resulting trees. Bulk-loading was done using the STR technique ([LLE97]), which partitions the data points into iso-oriented tiles. We ran 20000 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 3. We only report leaf-level performance numbers, since for this type of workload, R- and R*-trees are relatively short and the upper levels can be buffered. Section 4.5 talks more about how to account for buffering.

	R*-tree		R-tree	
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 3: Comparison of leaf-level performance in R- and R*-trees

The performance numbers indicate that R*-trees outperform R-trees, which is what is expected, but that there's 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 2; we omit a screen shot of this particular scenario here for brevity). The difference is that for the R-tree, these hot spots are more frequent and more stretched out.

[7] 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.

[8] Conversely, if the target utilization is 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.

[9] More formally: the pages in L_q^l cause a loss of CL_q that needs to be distributed according to how much each page in L_q^l contributes. Given L_q^o, we define a benchmark overhead ratio $r_q = |L_q^o| * C * u_t / R_q$. 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.

Looking at per-node excess coverage loss, 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 are rectangular, which results in tightly-fitting MBRs.

4.3.2 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 40000 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 20000 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, we built the R*-, SS- and SR-trees 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 4: Comparison of SPs of R*-, SS- and SR-trees

The measured excess coverage losses for the entire workload are shown in Table 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.

This example illustrates the value of the excess coverage metric and the importance of separating individual aspects of an AM design. Another performance study that compares sphere and rectangle SPs ([KS97]) comes to a conclusion contrary to ours, 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.

4.3.3 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,[10] 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 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.[11] 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 access 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 3 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

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

[11] 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.

data sets, unindexability cannot be established for corresponding workloads involving clustered data sets, even for much larger result set sizes.

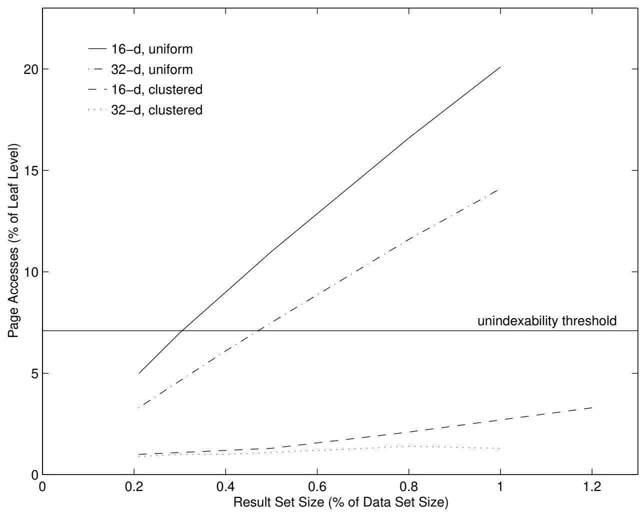

Figure 3: Unindexability Test: 16- and 32-dimensional uniformly distributed and clustered data

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

4.4 Implementation Performance Metrics

In addition to analysing existing tree structures, we also want to assess the performance of the structure-shaping extension methods, *pickSplit()* and *penalty()*. Our goal is to measure how these functions deteriorate the tree structure, expressed by the derioration of the workload performance caused by splits and insertions. This cannot be derived from the tracing information, because the workload only contains queries, and the effects of structure changes cannot be inferred indirectly. Instead, we simulate splits and insertions and observe the changes in workload performance; the splits and insertions are not carried to avoid actually deteriorating the tree during the evaluation process. Similar to the query and structural metrics, the implementation metrics should reflect the performance loss in comparison to the optimum, which we obtain by comparing the effects of a split of a particular node or insertion of a particular data item with the effects of an optimal split or insertion. The following two subsections in turn derive the split and penalty metrics.

4.4.1 Split Performance Metrics

We evaluate a split of a particular leaf node by comparing the actual split as produced by the *pickSplit()* extension method to the optimal split. The optimal split minimizes the total number of page accesses to the two post-split nodes by (a) producing perfect SPs with no excess coverage and (b) optimally partitioning the items on the leaf node so that non-empty accesses to the successor nodes are also minimized. Like the optimal tree, the optimal split is a theoretical construct, because partitioning the leaf items optimally will generally not result in SPs that completely eliminate excess coverage loss.

This definition of an optimal split actually ignores the effects of page utilization or the balance of the page utilizations produced by the split. The balance of a split clearly has an effect on the performance of a dynamic tree structure, since a perfectly balanced split is usually better at maintaining overall higher page utilization (in an unbalanced split, the fuller node is more likely to be the next node to be split again—assuming subsequent insertions are not biased toward the less utilized node—which will result in an overall lowered page utilization). On the negative side, a perfectly balanced split might have less desirable clustering properties. Unfortunately, the effects of the degree of balance of a split cannot be quantified, at least not in the workload context we consider. For that reason, we leave page utilization our of our split analysis and simply stipulate that the optimal split should be at least as balanced as the actual split. This way, both the utilization properties and the clustering of the optimal split are at least as good as that of the actual split.

Excess coverage loss Assuming that the optimal split eliminates excess coverage, the excess coverage loss of the actual split is the combined excess coverage in the left and right post-split nodes. A split is also an opportunity to improve SPs: describing data that previously resided on a single node with two SPs allows the description to be more specific. The success metric is the ratio of the decrease in excess coverage loss to the pre-split excess coverage loss, which constitutes the maximal improvement. Note that this ratio can drop below 0, if the split produces SPs with more excess coverage loss than the original SP.

Clustering loss The quality of clustering is expressed by the ratio of accessed to retrieved data: the higher the ratio, the more data a query needs to access in order to retrieve its result set and the poorer the clustering from that query's perspective. The amount of data that is accessed but not retrieved expresses clustering-related overhead, which the optimal split minimizes. The clustering loss of a split therefore is the difference in overhead data—limited to the left and right nodes of a split—between the actual and the optimal split. This is the same as the difference in the total amount of accessed data, because the volume of retrieved data remains unchanged by the splits. Note that the total amount of accessed data on a node cannot go up after a split: even if each query in the workload that visits the original node would have to visit both successor nodes. We call the amount by which data access decreases *clustering savings*. The ratio of actual clustering savings to optimal clustering savings serves as a "success' metric of the split that expresses to what extent the split realizes the potential for improvement of clustering.

4.4.2 Penalty Performance Metrics

We compare a penalty-guided insertion of a particular data item with the corresponding optimal insertion. The optimal insertion is defined as: (a) not adding to the excess coverage of the optimal target leaf and (b) choosing as the target the leaf which causes the smallest number of additional accesses in the workload. Note that the optimal target leaf does not correspond to the one that, if the data item were inserted and the SP actually updated, would result in the smallest number of total additional page accesses, including those due to excess coverage. Rather, it represents the true theoretical optimum, which optimizes each performance factor independently.

Performing a top-down, penalty-guided insertion has the disadvantage of accumulating the effects of multiple penalty computations. This could be avoided by scanning directly the level above the leaves for the minimum penalty leaf. However, a top-down traversal is more realistic and also reflects the quality of internal SPs.

In our analysis of the penalty function, we will again ignore the effects on page utilization. In the GiST framework, the shape of the SP cannot take the page utilization into account—the *union()* method is not informed of it—so that *penalty()* cannot direct an insertion based on the page utilization at the leaf level. For that reason, we assume change in the page utilization in response to insertions to be more or less random.

Excess coverage loss This is the number of additional excess coverage accesses to the actual target leaf after the insertion, assuming that optimally no additional excess coverage would be produced. When determining pre-insertion excess coverage, those queries that intersect with the new key need to be ignored, because they would falsely show up as a reduction in excess coverage.

Clustering loss The change in clustering quality in response to an insertion is reflected by the change in overhead data that the workload queries need to access. By definition, the optimal insertion minimizes additional overhead data access. The clustering loss is the difference in overhead data access between the actual and the optimal split.

4.4.3 Example 4: Comparison of R-tree and R*-tree Split and Insertion Strategies

This example continues the analysis begun in Section 4.3.1. We compare the split and insertion strategies of R- and R*-trees on a workload similar to that used in the previous example. For the implementation analysis, we use the intial bulk-loaded tree containing 20000 data items, and a correspondingly scaled back set of only 10000 queries. Using identical input trees for both the R-tree and R*-tree analysis simplifies the comparison, because the metrics reflect *changes* in workload performance due to splits and insertions.

Table 5 summarizes the split and insertion performance numbers. As expected, the R*-tree strategies are superior to those of the R-tree. The R*-tree split produces a better clustering and is also more effective at eliminating excess coverage than the R-tree split; the R*-tree insertion strategy also creates better clusters and marginally better SPs.

	R*-tree	R-tree
Splits		
pre-split accesses	75.44	
post-split accesses	40.04	44.62
pre-split exc. cov. loss	26.6	
post-split exc. cov. loss	20.8	33.0
Insertions		
clustering loss	1.28	1.88
excess coverage loss	8.74	8.8

Table 5: Performance numbers for R- and R*-tree split and insertion strategies

4.5 Other Performance Factors

In the analysis framework presented so far we 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, we excluded it 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. This is exacerbated by the fact that the underlying GiST framework has no knowledge of the internals of the stored data and the associated extension functions. 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 our 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] mentions that in their experiments, this technique never performed worse than LRU replacement). 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 real 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 usefulness 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 positiv, 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 negative clustering loss was reported.

4.6 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. The burden this puts on the workload execution 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, 2500 nearest-neighbor queries on 5000 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 4.3.3 range from 1.4MB (for 5000 queries retrieving 21 of 10000 16-dimensional points) to 40MB (for 20000 queries retrieving 120 of 40000 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 identical. 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 Related Work

5.1 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 our 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 multidimensional 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.

5.2 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 developers.

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. [12] 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.

[12] http://www.cs.hope.edu/ alganim/ccaa/ccaa.html

6 Conclusion

This paper 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. 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 refined to reflect the three fundamental structural performance factors: clustering, page utilization and the subtree predicates.

In `amdb`, the 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 performance. The visualization and animation functionality then enable users to investigate those sources of performance loss and gain an understanding of how semantic properties affect performance. Based on this understanding, the designer incorporates improvements into the design and repeats the analysis process to evaluate their efficacy.

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.

There are several questions we want to investigate in more detail in the future. Section 4 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.

Acknowledgement

Paul Aoki not only gave valuable comments on the paper and contributed greatly to the clarity of the presentation, he also implemented the `libgist` R-, SS- and SR-trees and provided us with an STR partitioning tool for spatial data.

References

[BBK98] S. Berchtold, C. Böhm, and H.-P. Kriegel. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proc. ACM-SIGMOD Conf.*, 1998.

[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.*, 1990.

[BS98] Frantisek Brabec and Hanan Samet. Visualizing and Animating R-Trees and Spatial Operations in Spatial Databases on the Worldwide Web. In *Proc. of Visual Database Systems*, 1998.

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

[FK94] Christos Faloutsos and Ibrahim Kamel. Beyond Uniformity and Independence: Analysis of R-trees Using the Concept of Fractal Dimension. In *Proc. 13th ACM SIGACT-SIGMOD-SIGART PODS* , 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.

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

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st VLDB*, 1995.

[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.

[KS97] N. Katayama and S. Satoh. The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proc. ACM-SIGMOD Conf.*, 1997.

[KSH98] M. Kornacker, M. Shah, and J. Hellerstein. amdb: An Access Method Debugging Tool. In *Proc. ACM-SIGMOD Conf.*, 1998.

[LL98] S. T. Leutenegger and M. A. López. The Effect of Buffering on the Performance of R-Trees. In *Proc. 14th ICDE*, 1998.

[LLE97] S. T. Leutenegger, M. A. López, and J. M. Edgington. STR: A Simple and Efficient Algorithm for R-tree Packing. In *Proc. 13th ICDE*, 1997.

[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 *Proc. ACM-SIGMOD Conf.*, 1997.

[PSW95] B.-U. Pagel, H.-W. Six, and M. Winter. Window query-optimal clustering of spatial objects. In *Proc. 14th ACM SIGACT-SIGMOD-SIGART PODS*, 1995.

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

[SBGR99] U. Shaft, K. Beyer, J. Goldstein, and R. Ramakrishnan. When Is "Nearest Neighbor" Meaningful? In *7th ICDT*, 1999.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-Tree: A Dynamic Index for Multidimensional Objects. In *Proc. 13th VLDB*, 1987.

[WR96] D. A. White and Jain. R. Similarity Indexing with the SS-Tree. In *Proc. 12th ICDE*, 1996.