Amdb: A Design Tool for Access Methods

Marcel Kornacker*
marcel@cs.berkeley.edu

Mehul Shah
mashah@cs.berkeley.edu

Joseph M. Hellerstein
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 aggregate runtime or I/O numbers. This paper presents amdb, a comprehensive graphical design tool for AMs that are constructed on top of the Generalized Search Tree abstraction. At the core of amdb lies an an analysis framework for AMs that defines performance metrics that are more useful than traditional summary numbers and thereby allow the AM designer to detect and isolate deficiencies in an AM design. 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 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 most balanced tree-structured AMs and is not restricted to particular types of of data or queries.

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 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 [9]). 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 contribution 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,

*This work was supported by NASA grant 1996-MTPE-00099, NSF grant IRI-9703972, and a Sloan Foundation Fellowship. Computing and network resources for this research were provided through NSF RI grant CDA-9401156.

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 apriori 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 paper we present `amdb`, a comprehensive support tool for the AM analysis process. At the core of `amdb` is an analysis framework that defines performance metrics that are superior both to aggregate numbers and data-specific performance metrics. 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, 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:

Universally Applicable The analysis framework and most of the `amdb` visualization facilities 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 Generalized Search Tree (GiST) abstraction [10]. 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 on its own. 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 paper is structured as follows. Section 2 briefly introduces GiST, which lays the foundation for an understanding of the breakdown of the performance metrics. Section 3 gives an overview of amdb and describes the analysis framework and its intended usage, which is illustrated in Section 4 with two examples of AM design projects that made use of amdb. Section 5 discusses the analysis framework in detail, along with illustrative examples, among them a test for unindexability. Section 6 discusses related work and Section 7 contains the conclusion and an outline of future work.

2 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 [5] 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 [9], 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.[1]

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.

[1] This structural requirement excludes R^+-trees [22] from conforming to the GiST structure.

We now provide 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 paper [10].

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,[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

[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.

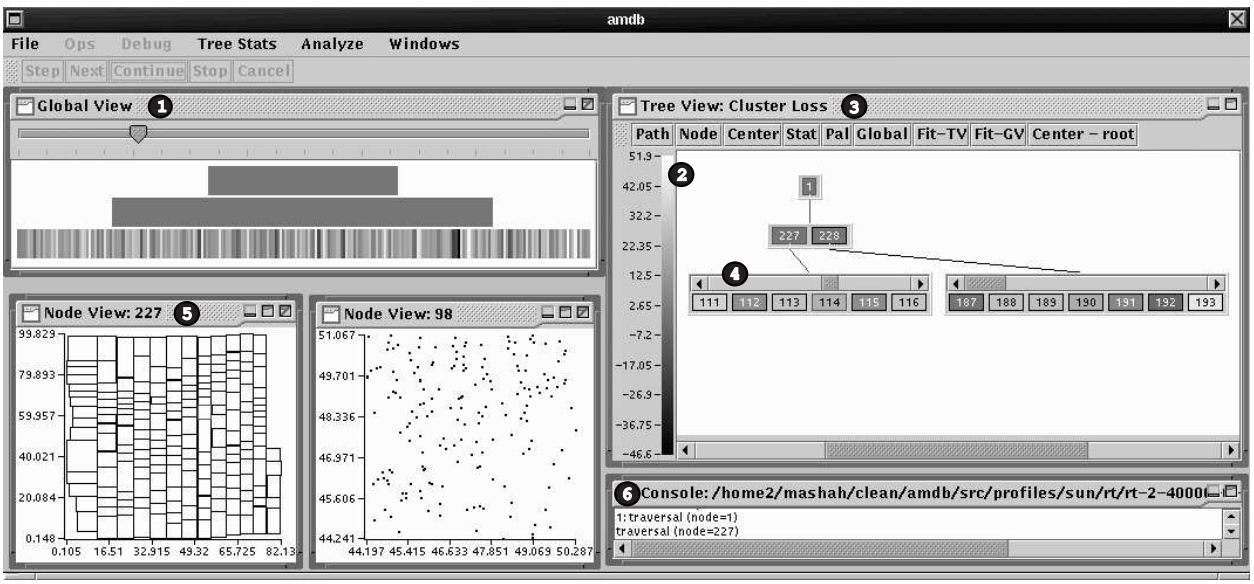

Figure 1: Amdb User Interface

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.

3 A Tour of `Amdb`

This section describes `amdb`'s visualization and debugging features (which are presented in greater detail in [21]) and gives an overview of the analysis framework and its intended usage.

`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/`.

3.1 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 3.4). 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 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 predicates, the access method designer must provide a module that displays the node given its contents. Currently, `amdb` contains a suite of modules which 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.

3.2 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.

3.3 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 5.4 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 semantics 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 difference between the number of page accesses in the actual tree and the optimal tree; we call this difference the *performance loss*. 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.[3] 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. Due to space constraints, we are unable to describe the implementation metrics in this paper and will ignore them in further discussions. Please refer to the technical report [14] for details.

[3] 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.

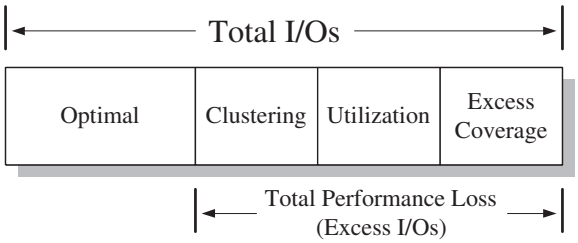

Figure 2: Decomposition of observed I/Os on a per-query and per-node basis.

Given a particular performance loss, we can further subdivide it to reflect the fundamental performance-relevant properties of GiST-based AMs, namely clustering, page utilization and excess coverage loss. *Clustering loss* specifies the part of performance loss that can be attributed to the difference between workload-optimal and achieved (leaf-level[4]) 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 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 2. 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 4 will illustrate this point.

3.4 Using the `Amdb` Analysis Framework

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.

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

[4]Why this is restricted to leaf-level clustering is explained in Section 5.

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 actually 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 Appendix A.

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.

4 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, designers were able to use amdb to achieve significant improvements.

4.1 Content-Based Image Retrieval

An AM design project was undertaken in the context of a content-based image searching, Blobworld [1]. 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 [16]. The details of this AM design project are described in [23].

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 nodes with particularly high loss. Visualizing the data in those nodes helped the designers

come up with ideas for a more accurate encoding of the space covered by the feature vectors. More specifically, the data visualizations showed nodes with a large fraction of empty "corner" space; the remedy for this is to encode the SPs 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;[5] 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 the shape of the SPs.

Similarly, 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 focused 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.

4.2 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 perviuosly 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

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

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.[6] 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 Details of the Analysis Framework

The following subsection discusses the optimal tree and how to construct it. Section 5.2 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 derives node metrics based on the query metrics. Various examples throughout this section illustrate the performance metrics.

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. Variables with subscript q are query-specific and variables with subscript p are page-specific. Use these as references when reading through the equations. 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.

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

[6] The possible reason for this is that it separates clusters 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.

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_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 paths, $I_q^l = \{p \lvert p \in I_q \setminus I'_q \wedge \neg(child(p) \in I_q \cup L_q)\}$
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, $r_q = \lvert L_q^o \rvert * C * u_t / R_q$
$R_{p,q}$ [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 1: Input Variables (Profiling Data, Tree Statistics and Derived Variables)

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.

CL_q	clustering loss	$CL_q = u_q/u_t \lvert L'_q \rvert - \lvert L_q^o \rvert$
EL_q^l	leaf-level excess coverage loss	$EL_q^l = \lvert L_q \rvert - \lvert L'_q \rvert$
UL_q^l	leaf-level utilization loss	$UL_q^l = \lvert L_q \rvert (1 - u_q/u_t)$
$EL_{p,q}^i$	internal-level excess coverage loss on page p	$EL_{p,q} = \begin{cases} 0 & \text{if } p \in I'_q \\ 1 & \text{if } p \in I_q^l \\ u_p/u_t & \text{otherwise} \end{cases}$
EL_q^i	internal-level excess coverage loss	$EL_q^i = \sum_{p \in I_q \setminus I'_q} EL_{p,q}^i$
$UL_{p,q}^i$	internal-level utilization loss on page p	$UL_{p,q} = \begin{cases} 1 - EL_{p,q}^i & \text{if } p \in I_q \setminus I'_q \\ 1 - u_p/u_t & \text{otherwise} \end{cases}$
UL_q^i	internal-level utilization loss	$UL_q^i = \sum_{p \in I_q} UL_{p,q}^i$
I_q^r	remainder of internal-level accesses	$I_q^r = \sum_{p \in I'_q} u_p/u_t$
CL_p	clustering loss	$CL_p = \sum_{q \in Q'_p} (u_p - Q_{p,q}^o/C)/u_t$
EL_p^l	leaf-level excess coverage loss	$EL_p^l = \lvert Q_p \setminus Q'_p \rvert$
UL_p^l	leaf-level utilization loss	$UL_p^l = \sum_{q \in Q_p} 1 - u_p/u_t$
EL_p^i	internal-level excess coverage loss	$EL_p^i = \lvert \{q \lvert p \in I_q^l\} \rvert$
UL_p^i	internal-level utilization loss	$UL_p^i = \sum_{\{q \in Q_p \lvert p \notin I_q^l\}} 1 - u_p/u_t$
Q_p^r	remainder of internal-level accesses	$Q_p^r = \sum_{\{q \in Q_p \lvert p \notin I_q^l\}} u_p/u_t$

Table 2: Performance Metrics

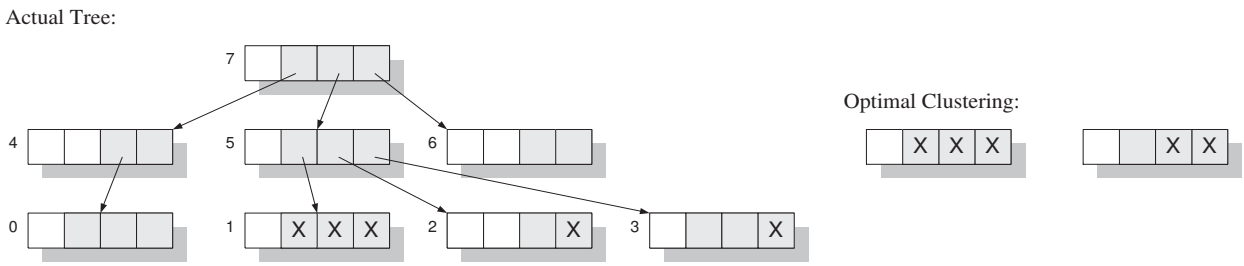

Figure 3: Traversal Paths and Optimal Clustering for Example Query

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 [8], but existing approximation algorithms work reasonably well in practice (Section 5.5 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 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%. Occupied slots are shaded, and the pages in the actual tree are enumerated for reference.

5.2 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.

[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 individual queries' clustering requirements are contradictory.

5.2.1 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 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.

5.2.2 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 $1/3$ 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 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 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]

[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.

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.

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 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.[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 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 5.2.1. The total per-node clustering loss is simply the sum of the per-node losses over the queries.

5.3.1 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 [16], 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.

[9] 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.

[10] More formally: the pages in L'_q cause a loss of $C\,L_q$ 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.

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 5.4 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 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 3; 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.

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.

5.3.2 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 [3]; minimum bounding spheres, as employed in SS-trees [24]; a combination of the two, which is used in SR-trees [12]. 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 [12] 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.

5.4 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. 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 [15] 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 ([15] 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 substantial negative clustering loss was reported.

5.5 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. 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 A 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 [11]. `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 [16], 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.

6 Related Work

6.1 Index Performance

Pagel, et al. [18] 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 [7], most of which were introduced with a performance study to demonstrate their efficacy. [7] 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, [3] 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 ([6, 18] 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.

6.2 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. [11] 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 [4]. 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 [17] 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.

7 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 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

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

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 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 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 we want to investigate in more detail in the future. Section 5 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.

References

[1] S. Belongie, C. Carson, H. Greenspan, and J. Malik. Color- and Texture-Based Image Segmentation Using EM and its Application to Content-Based Image Retrieval. In *Proc. of the 6th ICCV*, 1998.

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

[3] 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.

[4] 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.

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

[6] 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.

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

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

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

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

[11] 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.

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

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

[14] M. Kornacker, M. Shah, and J. Hellerstein. An Analysis Framework for Access Methods. Technical Report UCB//CSD-99-1051, University of California at Berkeley, 1999.

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] M. Shah, M. Kornacker, and J. Hellerstein. Amdb: A Visual Access Method Development Tool In *User Interfaces to Data Intensive Systems, Edinburgh, UK*, 1999.

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

[23] M. Thomas, C. Carson, and J. Hellerstein. Creating a Customized Access Method for Blobworld. In *Proc. ICDE*, 2000.

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

A 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 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

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

[13] Using Seagate Barracuda ultra-wide SCSI-2 drives, [19] measures a throughput of ca. 9MB/s under Windows NT. The average seek time and

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 ([2] 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 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 a recently published theoretical analysis of nearest-neighbor queries [20], 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

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.

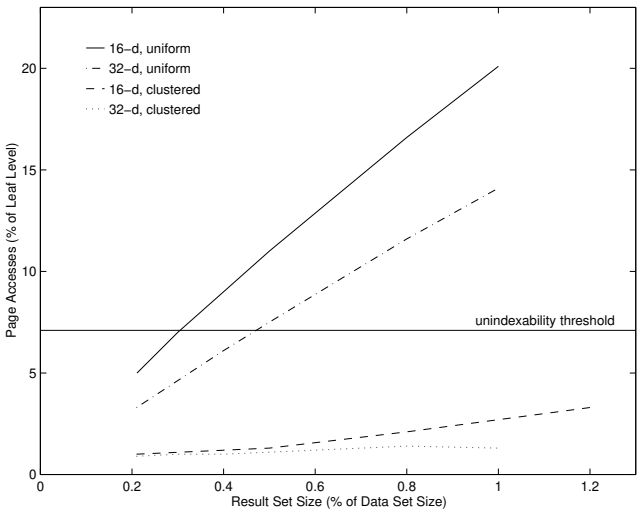

Figure 4: Unindexability Test: 16- and 32-dimensional uniformly distributed and clustered data

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.