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 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 that 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 [3]). 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, data-specific performance metrics like bounding-box overlap offer some insight into the causes of observed performance, but they require the designer

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*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.

to understand their correlation with the true 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 about that correlation 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. The analysis process is integrated with a collection of modules in an interactive, easy-to-use graphical environment. Those modules are: a visualization component for the tree structure and its contents (the latter user-extensible, so it can be adapted to a specific application domain); a facility for interactive execution of tree searches and updates as well as breakpoints and single-stepping through those commands, similar to functionality found in programming language debuggers; browsers for viewing performance numbers derived from the analysis framework. The salient features of `amdb` and its analysis framework are:

Universal Applicability The analysis framework and most of the `amdb` visualization 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 [4]. 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. `Amdb` supports this process 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 this extended abstract briefly introduces GiST, which lays the foundation for an understanding of the breakdown of the performance metrics, and presents an overview of `amdb` along with a description of the analysis framework and its intended usage. In addition, we present a hypothetical example to demonstrate how the performance metrics are calculated in a workload's analysis. Please see [6] for a full description of `amdb`.

2 Generalized Search Trees

A GiST is a balanced tree that provides "template" algorithms for navigating the tree structure and modifying the tree structure through node splits and deletes. Like all other (secondary) 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. A B^+-tree [2] 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. 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.

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 SEARCH and INSERT operations and how they use the extension methods.

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, an 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 an 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.

Although the GiST abstraction prescribes algorithms 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.

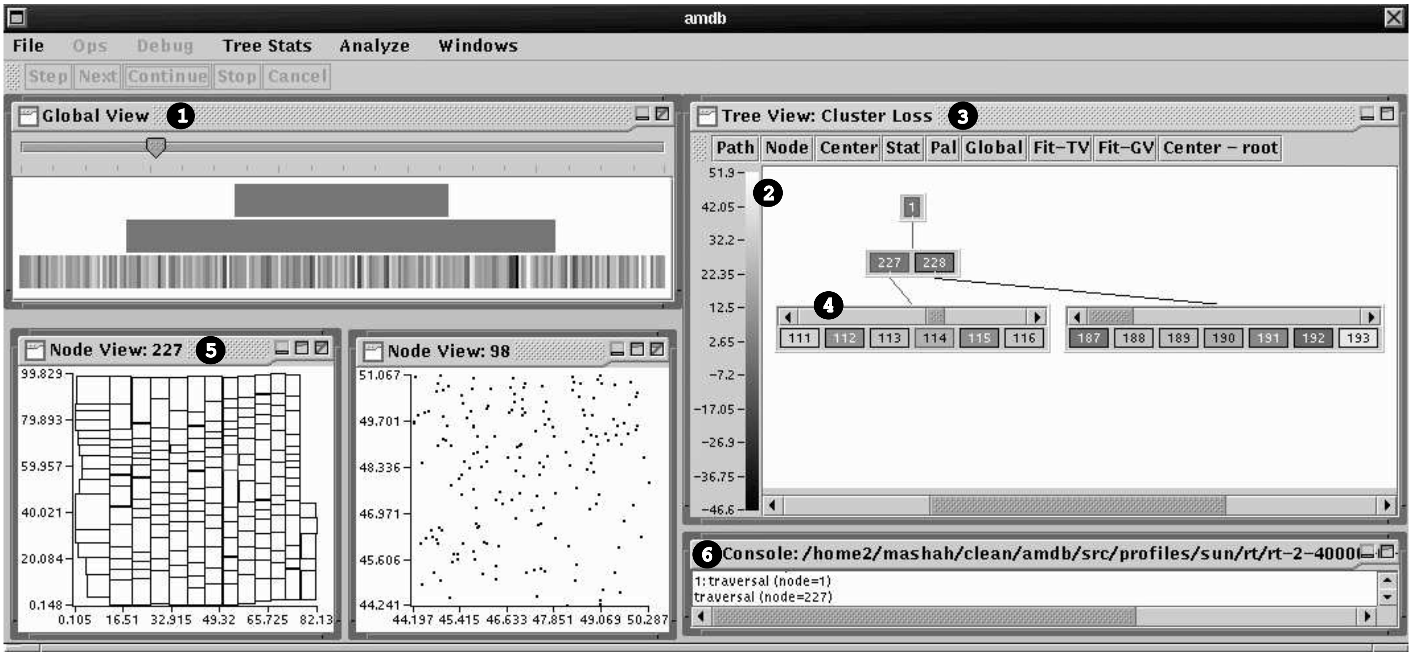

Figure 1: Amdb User Interface

Subtree Predicates While the size and shape of the indexed data is part of the input, 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.

3 A Tour of `Amdb`

This section describes `amdb`'s visualization and debugging features (which are presented in greater detail in [7]) 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 software 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. 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 that visualize two-dimensional projections of spatial data. The node view also allows the user to simulate a split (by calling the *pickSplit()* extension function) and visualize the results by separating the items with contrasting colors. In addition to user-defined data visualization, `amdb` provides a textual description of the keys, their sizes, and associated pointers.

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 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, `amdb` 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. For brevity, we concentrate on explaining observed page accesses; please see [6] for a discussion of the remaining components of the performance equation.

In Section 1 we 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. In general terms, it is a tree where (a) the data is clustered into leaf nodes to maximize the co-location of data that is co-retrieved, (b) the nodes in the tree are packed to the desired degree of utilization, and (c) the subtree predicates only guide the search algorithm to subtrees with query answers. While this hypothetical tree cannot be automatically synthesized for use, 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. More specifically, property (a) can be efficiently approximated via hypergraph clustering [5], and properties (b) and (c) can be simulated while gathering idealized performance results. The details are presented in [6].

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. Similarly, we can compute a *node performance loss*, which expresses a node's contribution to query or aggregate workload performance loss. The analysis framework also defines a number of additional *implementation metrics* that characterize aspects of the AM implementation; we refer the reader to [6] for more details.

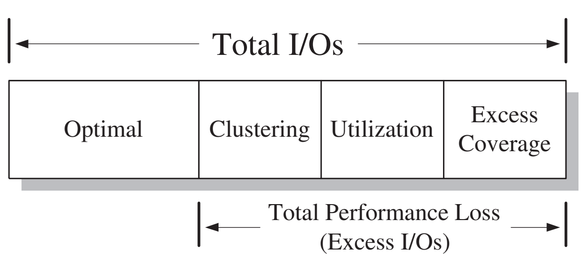

Figure 2: Decomposition of observed I/Os on a per-query and per-node basis

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[1]) 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.

[1] The reason this is restricted to leaf-level clustering is explained in [6].

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 [6].

The performance loss metrics express I/Os, not particular application-specific properties of the tree at hand or the AM design; the metrics can therefore only serve as an *indication* of, not an explanation for performance deficiencies. The explanation of performance deficiencies and a subsequent improvement of the AM design need to be done by the AM designer, based on an understanding of the semantics of the application domain. Gaining such an understanding is a creative process, which is helped by the amdb visualization facilities and their integration with the analysis framework: the performance metrics "flag" those parts of the tree and those queries that perform badly; the visualization facilities then let the designer navigate those index nodes and queries and investigate the reasons for their above-average performance loss. Aside from the user-extensible data visualizations, amdb also gives the designer access to a very comprehensive set of workload statistics, including per-query aggregate page access numbers, full traversal paths, the amount and specific location of data retrieved, *etc.* The performance metrics themselves are quite 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 [6].

4 Analysis Framework Illustration

In this section, we illustrate amdb's analysis framework. As a point of comparison, we define the optimal tree with respect to a search tree's structural characteristics. Then we show an example calculation of performance loss at the leaf-leaf level for a single query. The full details of our analysis framework can be found in [6].

4.1 Optimal Tree

The optimal tree is defined by the following characteristics:
No excess coverage, which eliminates page accesses due to overly general SPs – i.e., accesses that retrieve no items in the query result set.
Target page utilization, which would ideally be 100%, but this is unattainable in practice. Instead, the AM designer specifies a desired target page utilization, which can be estimated through external considerations,

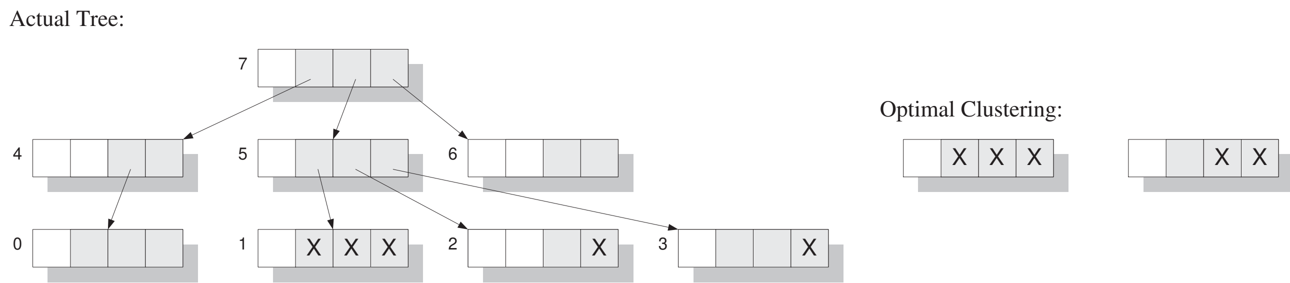

Figure 3: Traversal Paths and Optimal Clustering for Example Query

e.g., the existence of a competing AM with a well-known average utilization. This utilization also is used as a parameter to determine the optimal clustering.

Optimal clustering, which minimizes the total number of "relevant" page accesses. At the leaf-level, those are accesses to the pages that contain items in the query result set. 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. This clustering problem is equivalent to a hypergraph partitioning problem which is provably NP-hard. Thus, to solve this optimization, `amdb` relies on HMETIS [5] a tool that implements existing approximation algorithms that work reasonably well.

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 construct a tree that achieves a combination of the optimum in each category. Still, it is possible to approximate this tree along each axis well enough to infer page access patterns of a workload.

4.2 Performance Loss Breakdown

In this section, we illustrate the breakdown of performance loss at the leaf-level of a hypothetical search tree for a single query. The aggregate workload, internal-level, and per-node metrics are extensions of these calculations [6].

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) which retrieves five data items. This query accesses four leaves in the actual tree and only 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 traversal tree are numbered for reference.

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 excess coverage, utilization, and clustering loss. Ideally, leaf-level I/Os would visit the same number of nodes and return the same number of items as in the leaf-level of the optimal tree, and each visited leaf would be fully packed. Anything short of this is a performance loss. We begin by assigning leaf-level losses that are due entirely to bad SPs, and then consider the more complex interplay between utilization and clustering in partially useful leaf I/Os. 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 in each category for this example.

Excess coverage loss. During query execution, if a leaf node is visited but contains no items in the result set, we consider the entire leaf I/O to be excess coverage loss, i.e., due to an overly general SP for the leaf. 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. A leaf-level I/O that returns some useful items may contribute to performance loss in two ways. One way is through underfull leaf nodes. 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. Once we have factored away any utilization loss, the remaining I/Os reflect the performance of a "tightly packed" leaf level. Clustering loss is the difference between the conceptually tightly packed leaves in the index and the corresponding leaves in the optimal tree. 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 one I/O, utilization loss is $1/3$ I/Os and clustering loss $2/3$ I/Os. The sum is two I/Os, which is the total performance loss that the example query experiences at the leaf level.

5 Conclusion

`Amdb`'s analysis facilities, in concert with its visualization features, are an invaluable tool for understanding the performance characteristics of an AM and pinpointing the causes of deficiences. The analysis framework provides a breakdown of an AM workload's performance along three axes relevant to tree-based AMs: clustering, utilization, and the quality of subtree predicates. For each of these categories, `amdb` reports a performance loss in I/Os using an approximation to a workload- optimal tree as the basis for comparison. Such a breakdown provides a better characterization of AM performance than aggregate numbers and is universally applicable to any AM design based on the GiST abstraction. In [6] we detail how these metrics are computed for an aggregate workload as well as on a per-node and per-query basis, and we illustrate the use of these metrics on traditional AMs. `Amdb` has been instrumental in several experimental design projects for improving the performance of traditional AMs for specific applications [8, 9]. In [6] we highlight experiences in optimizing bulk-loaded R-trees for content-based image retrieval tasks, and summarize a user study in which a graduate database class was asked to improve the performance of AMs on a synthetic dataset.

References

[1] P. Aoki. Generalizing "Search" in Generalized Search Trees (Extended Abstract). In *Proc. 14th ICDE*, 1998.

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

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

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

[5] 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.

[6] M. Kornacker, M. Shah, and J. Hellerstein. Amdb: A Design Tool for Access Methods. Technical Report UCB//CSD-03-1243, University of California at Berkeley, 2003.

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

[8] M. Thomas, C. Carson, and J. Hellerstein. Creating Customized Access Methods for Blobworld In *Proc. 16th ICDE*, 2000.

[9] M. Thomas and J. Hellerstein. Boolean Bounding Predicates for Spatial Access Methods In *Proc. DEXA*, 2002.