Amdb: A Visual Access Method Development Tool

Mehul A. Shah Marcel Kornacker Joseph M. Hellerstein*
University of California. Berkeley
{mashah, marcel, jmh}@cs.berkeley.edu

Abstract

The development process for access methods (AMs) in database systems is complex and tedious. `Amdb` *is a graphical tool that facilitates the design and tuning process for height-balanced tree-structured AMs. Central to* `amdb`*'s user interface is a suite of graphical views that visualize the entire search tree, paths and subtrees within the tree, and data contained in the tree. These views animate search tree operations in order to visualize the behavior of an access method.* `Amdb` *provides metrics that characterize the performance of queries, the tree structure, and the structure-shaping aspects of an AM implementation. The visualizations can be used to browse the performance metrics in the context of the tree structure. The combination of these features allows a designer to locate the sources of performance loss reported by the metrics and investigate causes for those deficiencies.*

1. Introduction

The recent explosion in the volume and diversity of electronically available information has prompted the need for efficient techniques for searching this data. To address this demand, object-relational database systems provide interfaces for managing data of various types and implement secondary-storage data structures (also referred to as access methods or indexes) for efficiently accessing this data. A commonly implemented access method (AM) for ordered data types is the B^+-tree [5].

Countless access methods for non-traditional data types have been proposed by the database research community, yet relatively few are supported in production scale databases. Part of the impediment is the complexity of the design, implementation, and refinement process for AMs. Often, it is difficult for a designer to assess which aspects of an AM design are responsible for observed performance. Even the research community is undecided on the best designs for non-traditional data types. For example, Gaede and Gunther [6] survey numerous techniques for AMs that manage spatial data, but their results are inconclusive. Furthermore, correctness of an AM implementation is often difficult to verify.

Traditional tools such as programming language debuggers and profilers are cumbersome and uninformative for developing AMs. Pinpointing implementation flaws and understanding the mechanics of an AM require the ability to visually inspect the AM's state and observe the AM's behavior during index search, insert, and delete operations. Programming language debuggers are tedious because they offer interactive execution at the level of single lines of source code and introspection features for low-level data structures. Moreover, profiling tools are designed for analyzing code execution paths rather than AM-specific properties, such as clustering of data items, that characterize AM performance. A tool that encapsulates knowledge about AM structures and operations and exposes AM-specific, higher-level interfaces is needed. Thus, we present `amdb`, a graphical tool that simplifies the design, verification, and tuning process for height-balanced tree-structured AMs.

`Amdb` is a comprehensive, data-type independent development tool for AMs built in the Generalized Search Tree (GiST) framework. GiST encapsulates core AM functionality such as page management and exposes a domain-independent extensible interface for implementing height-balanced search trees. `Amdb` leverages GiST because it reduces implementation effort while still encompassing a broad class of AMs [8, 11]. `Amdb` provides debugger-like functionality at the level of the basic actions that comprise search tree operations. These actions are node-oriented actions such as node traversal, node split, node update, etc. Thus, the designer can observe and reason about the larger-scale mechanics of the tree rather than individual lines of code to gain a better understanding of the AM's behavior. Furthermore, given a workload — a tree and a set of queries — `amdb` reports metrics that characterize the input tree's performance in terms of the fundamental performance-relevant properties of the tree such as cluster-

*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.

ing of data items. These metrics are further broken down on a per-node and per-query basis. Such a breakdown enumerates the sources of performance loss. Determining how to improve an AM's performance involves scrutinizing this large collection of metrics, and understanding a search tree's behavior also requires the ability to navigate it.

A key challenge in building `amdb` was to enable navigation of the search tree and facilitate browsing of the metrics in terms of tree structure. Several techniques have been proposed for representing and navigating large hierarchies [15, 13, 9, 4]. However, none are well suited to the requirements of search tree access methods, which are typically short and height-balanced with high fanout. `Amdb` provides a hierarchy of visualization tools tightly integrated with the debugging and analysis facilities that fulfill these unique requirements.

These visualizations are the *global view*, *tree view*, and *subtree view* shown in Figure 1. These linked views represent and help a user navigate the structure, contents, and properties of the search tree at various levels ranging from a global perspective to the individual entries in each node. In conjunction with the performance metrics, these views bring out sources of inefficiencies in the tree structure, and allow a designer to investigate causes for those inefficiencies. In this paper, we identify several modes of interaction during this investigative phase and show how these views address each. Finally, combined with the debugging features, they provide animations of AM operations to understand the behavior of those operations and elucidate flaws in an implementation.

In this paper, we present the details of `amdb`'s user interface. We begin with an overview of the GiST framework and some sample balanced trees in Section 2. In Section 3 we enumerate the design criteria which guided `amdb` development. A description of the `amdb` analysis, visualization, and animation facilities is given in Section 4. Section 5 describes some previous work in both generic tree visualizations as well as search tree and database visualizations. Finally, Section 6 concludes with future work.

2. Balanced Search Trees

Databases store and retrieve data from disk in units of pages. Retrieving specific data items from disk is a relatively expensive operation. It involves locating the page containing the item, which may require sifting through many pages. Access methods are algorithms and data structures that attempt to minimize the number of I/Os necessary for this operation. Balanced search trees encompass a broad category of hierarchical access methods whose tree nodes correspond to pages on disk.

We give an overview of the GiST framework to describe the class of balanced search trees we are attempting to model with the visualizations. In addition, we provide a brief overview of the R-tree [7] in terms of the GiST framework. The R-tree serves as the example GiST AM that is used to highlight the utility of the user interface.

2.1 Generalized Search Trees

A generalized search tree (GiST) is a balanced tree which provides "template" algorithms for navigating the tree structure and modifying the tree structure through node splits and deletes. A 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. 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 many data types including 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. The GiST can be specialized to one of a number of particular access methods by providing a set of extension methods specific to that access method. These extension methods encapsulate the exact behavior of the search operation as well as the organization of keys within the tree.

We now provide a sketch of the implementation of the operations and how they use the extension methods. For a more detailed description see the original paper [8].

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

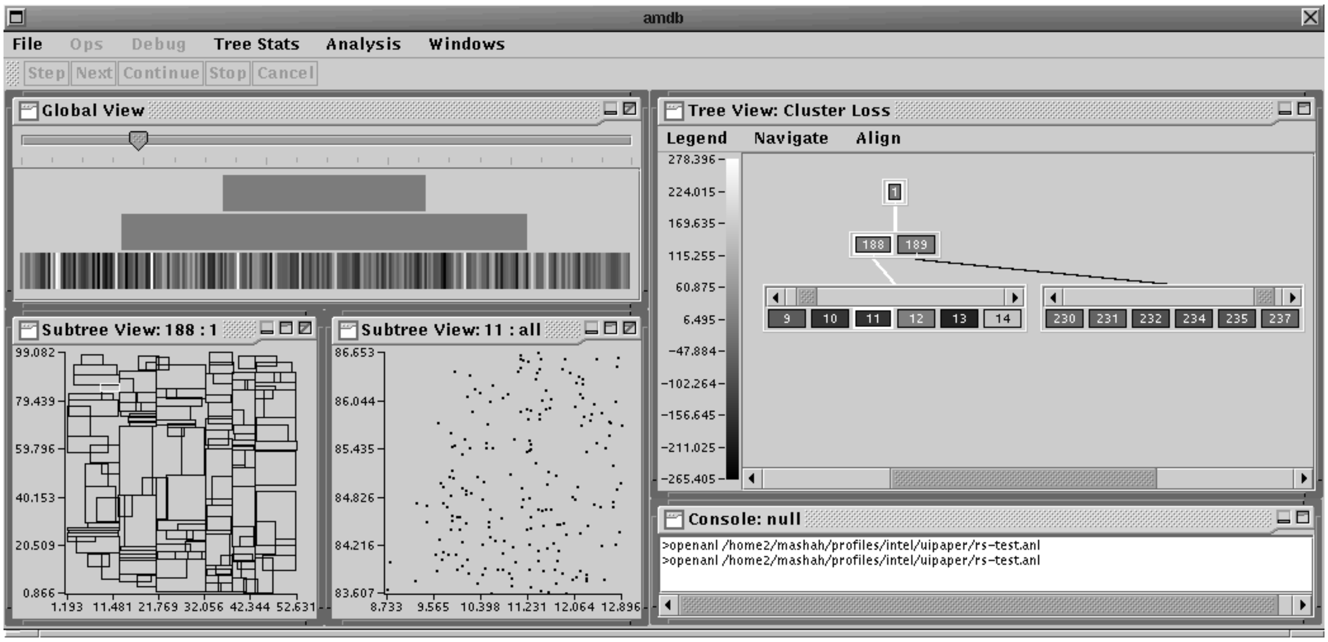

Figure 1. The `amdb` **user interface.**

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 recursively 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.

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 an algorithm for searching and inserting, the AM designer still has full control over the performance-relevant structural characteristics of the AM. We briefly describe these characteristics below.

Clustering is the organization of the indexed data in leaf pages, and SPs in the internal pages. The clustering determines the amount of extra data that a query needs to access in order to retrieve its result set. A poor clustering causes a query to fetch many pages each of which may contain only a few relevant items. An AM design controls the clustering through the *pickSplit()* and *penalty()* extension methods.

Page Utilization is the fraction of the page occupied with data. Typically, the utilization varies across nodes and ranges from some fixed tunable lower bound to 100%. 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 (SPs) describe, or cover, that part of the data space which is present at the *leaf* level of each SP's associated subtree. 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 pages that contain no relevant data.

2.2 R-trees

An R-tree [7] is a common AM that is easily modeled in the GiST framework. It is a height-balanced search tree which indexes multidimensional spatial objects. Each SP in an R-tree is the minimum bounding rectangle (MBR) that encloses the spatial objects contained in the subtree below. Search is performed starting from the root. To find all objects contained within a rectangle, all subtrees whose MBRs overlap the query rectangle are explored further. Once a leaf is reached during a search, the data items contained within it are filtered by the query rectangle. Thus, for a GiST implementation of an R-tree that supports containment queries, the *consistent()* function returns all MBRs or data items which overlap the query rectangle. Insertion is also carried out top-down. At each level, the subtree in which to insert is determined by the picking the MBR which requires the minimum volume enlargement. For a GiST implementation, the *penalty()* function is defined to return the enlargement in volume of the MBR. Finally, upon an overflow during insertion, a node is split by separating its contents into two disjoint subsets whose MBRs overlap the least. Various other heuristics for separating the contents have been proposed. Split heuristics are encapsulated in the *pickSplit()* function. For a more detailed description, of the R-tree extension methods, see the original GiST paper [8].

Variants of the R-tree modify one or more of the following three aspects: SP design, *penalty()*, and *pickSplit()*. For example, the R*-tree [2], modifies the *penalty()* and *pickSplit()* routines. SS trees and SR trees [10, 16] modify the SP in addition. Each of these designs attempt to create a better AM by adhering to two goals. The first is to minimize SP overlap. The second is to minimize the volume spanned by the SPs.

3. Design Criteria

There are a number of principles which guided the design for `amdb`. These criteria were inspired by our experience in developing GiST-based AMs and initial development efforts to visualize their structure. We summarize these goals before we describe the features of `amdb`.

High-level and interactive. Debugging AMs requires the ability to step through index search, insert, and delete commands, but programming language tools are too tedious for this purpose. They provide interactive execution at the source code level rather than at the level of the salient actions that comprise the index operations. For search trees, these basic actions are node-oriented. Examples include node traversal, node split, item insertion, item deletion, etc. Index operations often take convoluted paths through the code. Hence, with a source code debugger, it is difficult for the user to determine why and when these node-oriented actions are invoked. Raising the level of abstraction allows the designer to follow the essential aspects of a search tree's behavior while hiding unnecessary details. Thus, `amdb` allows interactive execution of index operations at the level of node-oriented actions.

Performance feedback for a workload. An AM's performance characteristics can not be deduced from single executions or from aggregate numbers alone. Characterizing the performance of a particular AM in general is a difficult problem as evidenced by the efforts in the research literature [12, 6]. One reason is because an AM's performance is dependent on many factors: the queries run against it, the data it contains, and its structure. A more tractable approach for evaluating an AM is to characterize its performance for a given workload – a set of queries run against a fixed tree. This is the approach taken by `amdb`.

Visualization and animation. Interactive execution facilities and performance metrics are not enough for facilitating the AM development process. Recognizing patterns in large collections of data is difficult for humans. Both the search tree and performance data reported by `amdb` are typically large enough to make manual browsing tedious. Visualizations and animations are necessary because they stimulate pattern recognition. Software visualization tools are too low-level to be generally useful in this context [1]. Like programming language debuggers and profilers, they do not encapsulate any knowledge about AM-specific data structures or operations. Thus, `amdb` must provide a visualization of the search tree structure and its contents. This in turn can be leveraged to browse reported statistics as well as animate debugging operations in the context of the AM structure.

Many tree visualization schemes have been proposed; however, none of them are directly appropriate for the task at hand. The following requirements are crucial for our visualizations.

Focus + context visualizations. A common form of interaction for inspecting search trees involves traversing several paths, or subtrees. For example, a window query often traverses several paths in an R-tree. During an interactive execution of a query, a user may want to compare the nodes along the traversed paths to better understand the query's behavior. While a user is focusing on this subset of the tree, it is also useful to see its relationship to the whole tree. A similar argument applies when a user is looking at the contents of a single node. Context helps place the "focused" items. Any visualizations that model the search tree must provide focus with context at all levels.

AM specific visualizations. `Amdb` models height-balanced trees that usually have bounded and high fanout. Database search trees typically have a fanout between

100 and 200 and are 3 to 4 levels deep. A completely generic tree visualization is unnecessary, especially if it sacrifices one of the other criteria.

Preserve data-type independence. GiST is a generic, data-type independent framework for implementing search trees. `Amdb` leverages GiST to encompass as many flavors of AMs as possible. Accordingly, the features and analysis framework that `amdb` implements should preserve data-type independence whenever possible.

4. Analysis, Visualization, and Animation

In this section, we give an in-depth description of the functionality provided by the `amdb` user interface and illustrate how it can be used to refine an AM design. First, we give an overview of the performance metrics that `amdb` reports. These metrics can be browsed naturally using the graphical views that `amdb` provides. We describe these visualizations next. Finally, we describe the debugging functionality and how it is integrated with the views to provide animations of index operations.

4.1. Analysis Framework

We give a brief overview of the `amdb` analysis framework; for a more complete description see [12]. The analysis framework defines performance metrics that characterize the page access behavior of a specific workload—an input tree and a set of queries. These metrics are more meaningful than aggregate page access or runtime numbers and thereby allow the AM designer to detect and isolate deficiencies in an AM design. The analysis framework applies to all GiST AMs.

Central to the analysis is the comparison of observed page accesses with optimal page accesses, i.e., the number of page accesses in a tree that is optimal for the input workload (a model of which can be approximated relatively efficiently). The performance metrics are derived from this comparison and express *performance loss*, which is the difference between actual and optimal page accesses. The framework defines metrics for each query of the workload, for each node of the input tree, and for the structure-shaping aspects of the AM implementation, namely the *pickSplit()* and *penalty()* GiST extension functions. In order to provide additional insight into the sources of performance deterioration, the loss metrics are further broken down to reflect clustering loss, page utilization loss, and excess coverage loss in the input tree.

`Amdb` implements this framework and offers text-oriented browsers to step through individual metrics. Since these interfaces can be cumbersome, it also offers a combination of visualization tools to browse the structural metrics more naturally.

4.2. Visualization

In this section, we present the graphical interfaces that `amdb` provides to support the process for determining the causes of AM performance degradation. The main contributions of our design are as follows. First, we provide new mechanisms that help a designer navigate database search trees without being overwhelmed by their width. Second, we identify three modes of interaction with the search tree during this investigative process and provide a visualization that addresses each. The three views that support this process are *global view*, *tree view*, and *subtree view*. They are tightly integrated via various methods of linked views.

To illustrate the utility of these interfaces, we refer to a running example of an R-tree which indexes 2-D point data (see Figure 2). It contains 40,000 points which are clustered around 200 randomly distributed centers in the range [0..100] in both dimensions. The clusters are square with an average side-length of 5. We ran 20,000 queries against this R-tree to compute the performance metrics. These were square queries of side-length 2.5, and centered on randomly chosen points from the indexed data.

4.2.1. Global View. The purpose of the global view is to provide a manageable aggregate view of the entire index with respect to a particular node property (see Figure 2 and Figure 3). This property can be a workload-performance metric, a generic attribute like page utilization, or a boolean attribute like traversal during a query. The global view is meant to help designers recognize patterns in a property among all nodes in the tree and correlations between different properties. Thus, a designer can use it to quickly spot regions in the search tree which are responsible for performance loss. In order to provide a manageable overview, the global view approximates the search tree. It factors out much of the exact tree structure while trying to preserve the relative positions of nodes within the tree.

The global view is constructed by mapping a conventional 2-D layout of the entire tree onto a triangle (see top of Figure 2). A conventional 2-D layout of a tree is one in which parents are centered above their children. In the global view, each node is represented by a vertical colored bar which reflects a user-chosen node property for the entire tree. The links between parents and children are not shown, and all nodes on the same level are concatenated. In this layout, the distance between nodes on a particular level is roughly related to the distance to a common ancestor — a property users are accustomed to in conventional 2-D tree layouts. The height of the tree is changed by adjusting the size of the window, and the baseline of the triangle is adjusted with the scale at the top of the view. Since search trees usually have high and roughly constant fanout, the number of nodes on each level increases geometrically while the allocated screen space only increases

Figure 2. In this `amdb` session, we begin with the global view and narrow down upon the sources of excess coverage loss through the tree view to the subtree views. We discover the cause of high excess coverage loss for node 192 is high overlap with node 63. Thus, the SP or split algorithm are design aspects for further refinement.

linearly. Thus, the pixel density of nodes increases roughly geometrically. This implies that at the bottom levels, it is possible that a vertical bar that is one pixel wide represents a collection of nodes. In this case, the default behavior is to use the average of the property across the nodes to represent the collection. The mapping between colors and the node-related property is displayed in the legend on the left of the *tree view* (see center of Figure 2), which is described in the next section.

The global view elucidates patterns in a chosen metric and correlations between two metrics. For example, the global view in Figure 2 displays excess coverage loss, the extra I/Os incurred because of overly general MBRs, in our example R-tree. Notice that there are localized spots of high excess coverage loss in the leaf and internal levels, indicated by white regions. Correlations of excess coverage loss with another metric may provide insight into the cause of the empty traversals. A designer can search for correlations by projecting the new metric onto the same view. To show how the global view can point out correlations between metrics, we digress from our current example to another tree, an R*-tree which indexes 8-D point data. Figure 3 shows the correlation between clustering loss and excess coverage loss for the leaf level of this tree. In both views, black represents

Figure 3. The top view shows clustering loss and the bottom view displays excess coverage loss for an R*-tree indexing 8-D clustered point data. These views have been extracted from the user interface and placed side by side to highlight the correlation.

low loss and white represents high loss. Notice the correlation for the leaf nodes slightly right of center. The correlation between these metrics is not exact, but strong enough to merit further investigation. With the approximation of the search tree provided by the global view, the designer can identify areas of interest which can be explored further using a more detailed view of the search tree structure, the *tree view*.

The user can navigate to an area of interest in the tree by clicking on it in the global view. Subsequently, a path to a node in the vicinity of the click is shown in the tree view. For example, in our example R-tree we notice several areas of high excess coverage loss. It is worth investigating those areas further to determine the causes for such a performance loss. Figure 2 shows the path that is highlighted in the tree view if we click on the first white bar in the lowest level of the global view.

4.2.2. Tree View. The purpose of the tree view is to allow the user to focus on several paths or subtrees at once within the search tree. This form of interaction is useful during animations of index operations. In addition, it is necessary when investigating sources of performance loss. Looking at several paths or subtrees at once helps the user recognize correlations in statistics between parents and children and among different paths. These correlations are obscured by the global view. Furthermore, the relationship between the global view's representation of the tree and the paths shown in the tree view is obscured or lost when there are numerous paths displayed. Since a focus+context visualization is an important criterion for our design, one requirement of the tree view is to provide as much context as possible in relation to the search tree to compensate for this drawback.

The tree view displays the true structure of the subtrees and paths that are visited by exactly depicting ancestral relationships among the visited nodes (see center of Figure 2). It also offers an intuitive point-and-click interface for browsing the search tree while improving on conventional tree navigation interfaces that become cumbersome for high-fanout trees. Conventional 2-D tree interfaces display all the children of a node at once and provide a single scrollbar for navigation in the tree if it becomes too large to fit on screen. This can be unwieldy for browsing search trees because even a single level often cannot fit on screen. Below, we describe how the tree view provides context without overwhelming the user with the tree's width.

In the tree view, the tree's nodes are represented by boxes and labeled with a unique number for reference. The nodes are colored according to the same node property shown in the global view. The mapping between colors and the node property value is displayed in a color legend in left of the tree view. Each node is enclosed in a scrollable and stretchable container, a *sibling container*, which displays its direct siblings. This container allows users to focus on nodes of interest while providing as much context as possible and bounding the fanout of each node. Figure 2 shows an example for the children of nodes 188 and 189; they cannot all fit on the screen, but since they are enclosed in sibling containers the visualization works naturally. A scroll bar is provided to find nodes of interest within a container.

Any node can be expanded or contracted by clicking on it. When a node is initially 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. If it is subsequently expanded, the previously contracted subtree is re-displayed.

The conventional 2-D tree layout algorithm is used to lay out the sibling containers. The only subtlety is in determining what width the containers should be. The goal is to provide as much context as possible without cluttering the display or going beyond its boundary. Initially, the width of the window is divided among containers at the same level, and this allocation is dynamically maintained until the user resizes the container. After resizing, the container's width remains fixed until it is resized again.

The tree view also provides the notion of a "current path", a path from the root to a given node in the tree. This notion is useful while browsing and debugging. During debugging, the "current path" represents the progress of the index operation which moves from node to node. While browsing, it serves as a visual cue for linking this view to the data-type specific visualization of the tree contents described in the next section. If a "current path" is shown, then it is highlighted, and the sibling containers that hold nodes along the path are aligned vertically.

Continuing with our example, the tree view in Figure 2 shows a highlighted path to node 192, which we narrowed down upon from the global view. Notice that the children of node 189, which has high excess coverage loss, have almost no excess coverage loss. But, the children of node 188, which has low excess coverage loss, have a relatively higher excess coverage loss. Such a non-intuitive and suggestive observation is difficult to discern with only the global view. Once we have located sources of performance loss, we need to determine the causes of these losses to refine our AM design. To do so, we need to investigate the data contained in the candidate nodes or subtrees. `Amdb` offers the *subtree view* which visualizes these data.

4.2.3. Subtree View. Unlike the previous two views which provided data-type independent visualizations, the subtree view is a data-type specific visualization of data in the search tree. It provides a graphical display of some user-specified subset of the items and SPs contained in the search tree. The goal of the subtree view is to help the designer translate the sources of performance loss into domain-specific intuitions that reflect the causes of these losses. These intuitions can then be used to refine the AM design for reducing the losses. For example, the subtree view should help us explain the excess coverage loss in terms of the points and MBRs contained in our example R-tree, so that we may reduce or eliminate the loss. Since tree nodes contain arbitrary user-defined SPs, the access method designer must provide a module which displays the data items and SPs. Currently, `amdb` is released with a built-in suite of modules which visualize two-dimensional projections of multi-dimensional data. A designer may install custom modules if needed. As a default, if no visualization modules are applicable, `amdb` provides a textual description of the SPs, their sizes, and associated pointers contained in any chosen node. We describe the features that the 2-D subtree view modules support.

The 2-D subtree view modules are integrated with the tree view and provide the following features. First, the user can visualize, on a single canvas, the entire contents of any chosen node or the subtree rooted at that node. With the subtree option, the user must specify the number of levels (from the subtree root) that are displayed simultaneously. For example, the subtree view in the lower left of Figure 2 shows the data items in node 192. The axes represent the orthogonal axes of the data domain. The left view in Figure 4 shows the SPs contained in the first two levels of the tree; the SPs in node 1 are superimposed upon the SPs in node 188 and 189 from our example R-tree.

The subtree view provides other convenient features that link it to the tree view. These features help the user discover the context of the displayed items in terms of the tree structure. First, the subtree view highlights all SPs contained in the view that describe nodes along the "current path." For example, in the center subtree view of Figure 2, the MBR of node 192 is highlighted (in white) because node 192 is on the "current path." Another example is shown in Figure 5. The subtree view of node 1 shows the MBR of node 189 highlighted, and the subtree view of node 189 shows the MBR of node 60 highlighted. These MBRs are highlighted because they lie on the "current path" shown in the tree view. In addition to the "current path", subtree view provides a facility to highlight the contents of entire subtrees that are contained within a subtree view. For example, the left view in Figure 4 highlights the contents of the subtree rooted at node 188. Only the MBRs contained in node 188 are highlighted since it is the only node from the subtree whose contents are being displayed. Finally, in a single node visualization, one has the option to display the potential results of a node split in contrasting colors. The right side of Figure 4 shows an example in which the black points would be placed in the left node and the white points in the right if node 218 were split. This feature is indispensable for debugging the *pickSplit()* method.

Continuing with our example in Figure 2, we want to determine the cause of excess coverage loss in node 192. Its contents are shown in the left most subtree view of Figure 2. Notice, the node contains points from different clusters in an arrangement for which MBRs are not well suited. Thus, node 192's MBR has a lot of empty space. In order to find node 192's MBR in relation to its siblings, we need to take a look at the contents of node 188. We can see that node 192 has an MBR which has high overlap with another MBR contained within it, the one of node 63. [1] From these views, we see that two factors, the MBR's overlap and the MBR's inability to describe data from different clusters, are causing the excess coverage loss. Some alternatives for

[1] Currently, the subtree view does not allow the user to directly select an MBR to find out which node it describes. Thus, we manually searched the textual descriptions of node 188 to determine which node's MBR overlaps node 192's MBR.

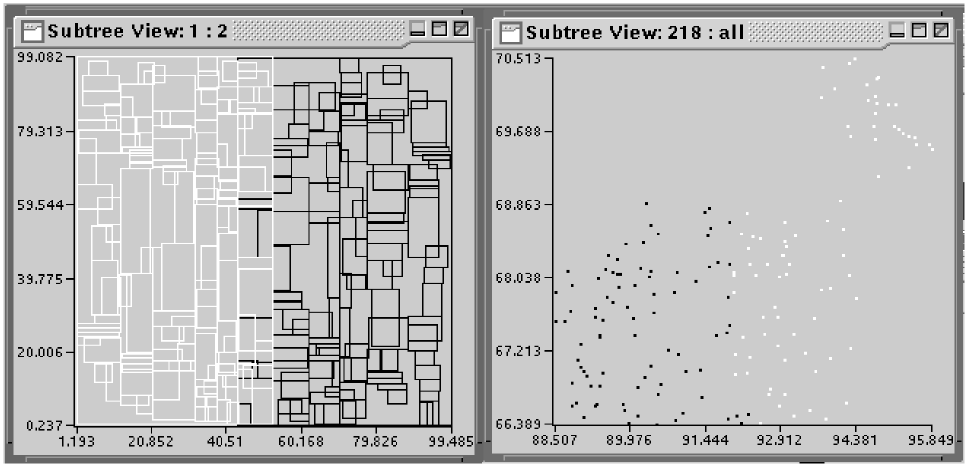

Figure 4. The two figures represent data contained in our example R-tree. The left view shows the SPs contained in the first two levels, nodes 1, 188, and 189. The SPs of internal node 188 are highlighted. The right shows a visualization of *pickSplit()* results for node 218.

improving our design include changing the SPs, changing the *pickSplit()* method to reduce overlap, or changing the *penalty()* method to seek clusters.

4.3. Animation Features

The behavior of an AM can be difficult to understand without an ability to observe its mechanics. Previously, only standard programming language debugging tools were available for examining GiST 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. Hence, `amdb` provides visual animations of these operations to help understand their behavior.

`Amdb` allows a designer to single-step through index search, insert, and delete commands. Those commands generate an event for each node-oriented action, such as node split, node traversal, etc. `Amdb` permits users to step from event to event. Since manual stepping can become tedious, `amdb` also supports breakpoints. Breakpoints can be defined on generic node-oriented actions, e.g., node traversal or node update. Breakpoints can also 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. Textual descriptions of an interactive execution are provided in a console window. However, these descriptions are often not very effective for grasping the overall behavior of the particular operation.

`Amdb` integrates the debugging features with the visualizations to produce animations that provide more useful feedback than textual descriptions. The breakpoints encountered during the interactive execution mark the "frames" of the animation. For each frame, the path to the current node being considered is expanded and highlighted in the tree view. Likewise, the SPs of all nodes along this "current path" are also highlighted in the subtree views. Integrating these views with debugging features not only provides context for the progress of the execution, but also produces a cogent animation while single-stepping.

A snapshot of an interactive execution of a rectangular window query on an R-tree indexing 2-D point data is shown in Figure 5. The tree view highlights the path to the current node (60) and textual descriptions of the interaction are shown in the console. The subtree views display the MBRs contained in the root and node 189. The MBRs of node 189 and node 60 are highlighted in top and bottom subtree views respectively since they lie on the current path. The user can control the execution with the buttons at the top of the screen.

5. Related Work

5.1. Tree Visualization

Numerous techniques have been proposed for visualizing large hierarchies. *Cone Trees*, hyperbolic browsers, and treemaps are a representative subset. Each of these have their merits but were passed over for various reasons.

The Cone Tree [15] embeds the hierarchy in a three-dimensional space; the children of a node are wrapped around the base of a circular cone with the parent located at the apex. Nodes can be rotated to the front of the view to bring a path into focus while maintaining its context in

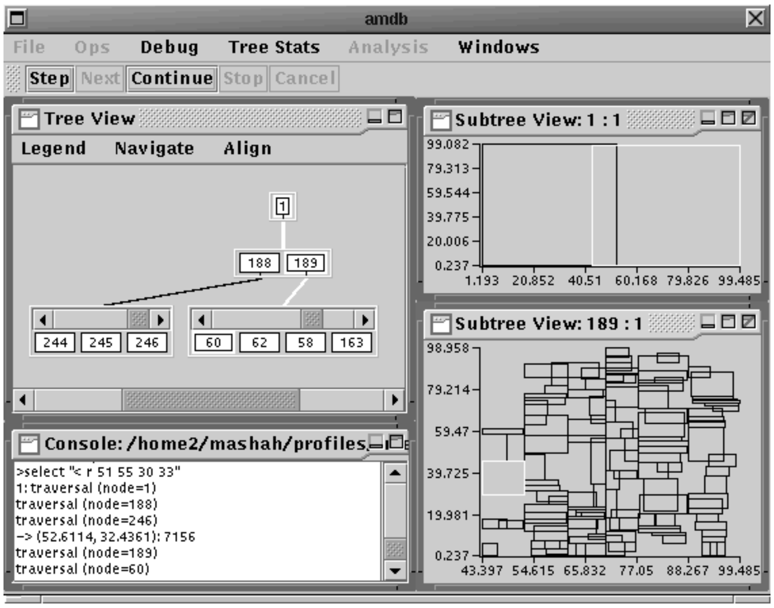

Figure 5. An interactive execution of a rectangular query on an R-tree.

the entire structure. In addition, subtrees can be pruned or expanded. One limitation is that only a single path can be brought into focus at any one time. Index operations often take several paths and hop around the tree. This property makes an execution animation hard to follow with the Cone Tree because nodes and paths are constantly rotated in and out of focus. The authors note the Cone Tree becomes unwieldy at about 1000 nodes or with a fanout higher than 30, and is more effective for unbalanced structures. Database search trees typically have more nodes, higher fanouts, and are balanced. User experience with our initial prototype indicated this representation was not effective at high fanouts. It obscured parts of the tree, hindering the ability to grasp the tree's global structure. Finally, the Cone Tree requires 3-D rendering and animation support. Our current implementation is 2-D and relies on simple Java toolkit primitives which makes `amdb` easy to port.

Hyperbolic browsers are a focus+context technique for visualizing hierarchies [13]. They lay out the hierarchy in a hyperbolic plane which is then mapped to a circular disk. This places the root at the center with the node density increasing exponentially towards the circumference. This is similar to our global view but in a circular orientation. Change of focus is performed by dragging a node of interest towards the center. The drawback to this approach is that only one node and its direct ancestors and children can be brought into focus at a time. Furthermore, a user study indicated that it provides a "weaker sense of directionality and location in the overall space" [13]. If applied to search trees, this would tend to obscure the difference between internal and leaf nodes, a critical distinction in our application.

Finally, treemaps represent hierarchies with screen real estate [9]. A node is assigned a section of the display which is then divided among its children. The area allocated is related to some property of the node. This layout helps find certain patterns easily. However, because the physical layout for one property can be quite different from another, finding rough correlations between two distinct properties is difficult.

5.2. Index Visualization and Animation

To our knowledge, `amdb` is the only tool that provides both debugging and analysis functionality for AM development. It is also the only one which integrates a scalable representation of the search tree structure with these features. There have been precedents in the literature of debugging tools geared toward domain-specific indexes. DEVise is a general purpose visualization tool that has been useful in debugging R-tree implementations [14]. It simply provides a 2-D view of the points and their bounding rectangles contained within the R-tree, akin to `amdb` 's default subtree view. It offers no facilities for animating index operations. Similarly, Brabec and Samet [3] provide a collection of Java applets that encapsulate a wide variety of 2-D R-tree and quad-tree variants. Again, like `amdb` 's subtree view, they focus on 2-D geographic visualizations of nodes spanning one or more levels. These views offer the ability to zoom in and out of the spatial representation, with domain-specific statistics reported per level. They do animate insertions, deletions, and splits in which users can observe changes in the points and bounding boxes indexed.

The authors indicate that their visualizations do not scale for high-fanout trees, as we have noticed with our subtree view (without the context provided by the other views). Neither of these two tools provide detailed feedback about an AM's workload performance.

6. Conclusion

Amdb is a tool that aids designers in the development process for GiST-based AMs. It animates interactive executions of search tree operations. It reports metrics which provide detailed feedback about the workload performance characteristics of an AM. It provides visualization tools with intuitive controls that are well suited for representing and navigating database search trees.

These tools allow the user to investigate sources of deficiencies and drill down to determine their causes. The performance metrics provide the input to the visualization tools for locating sources of performance loss. The global view provides an approximation of the tree structure which helps elucidate patterns in the reported metrics. The tree view allows the designer to focus on paths and subtrees of interest without cluttering the display. Finally, the subtree view is a visualization of the data that helps a desginer explain the performance loss. These visualization tools are implemented using primitives available in Java class libraries, which makes them easy to port.

There are a number of challenges that still need to be addressed by amdb. First, amdb currently does not provide visualization facilities for effectively browsing the per-query metrics. These metrics are just as significant as node metrics for pinpointing sources of performance loss. Second, the domain-independent visualization tools and analysis metrics only point out the locations of deficiencies without proposing deeper intuitions on causes or solutions. The subtree visualization partly assists the designer, but it is not enough for fully characterizing the domain-specific causes for the losses. Also, the state of a search tree is often the result of a sequence of operations and is highly dependent on their order. For example, for an R-tree the insertion order is critical to its performance. Visualizations of SPs and data give little feedback about such cumulative effects. One possible approach to remedy these drawbacks is to provide the AM designer with facilities to collect user-defined metrics over the workload execution. These metrics can then track cumulative effects and also incorporate domain-specific knowledge. Allowing the designer to visualize and compare these with the domain-independent metrics would verify intuitions about observed performance.

The current release of amdb can be downloaded from http://gist.cs.berkeley.edu/. Amdb is bundled with the libgist package, which implements the GiST abstraction. Amdb is written in Java and libgist is written in C++. The packages are available for several platforms.

Acknowledgments

We want to thank Megan Thomas and the students in the graduate database course at Berkeley for being guinea pigs. We would also like to thank Paul Aoki, Allison Woodruff, Vijayshankar Raman, and Bennet Vance for valuable comments.

References

[1] T. J. Ball and S. G. Eick. Software Visualization in the Large. *Computer*, Apr. 1996.

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

[3] F. Brabec and H. Samet. Visualizing and Animating R-Trees and Spatial Operations in Spatial Databases on the World-wide Web. In *Visual Database Systems*, May 1998.

[4] J. Carri re and R. Kazman. Interacting with huge hierarchies. In *Information Visualization Symposium*, pages 90–96, Oct. 1995.

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

[6] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

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

[8] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *VLDB*, pages 562–573, Sept. 1995.

[9] B. Johnson and B. Shneiderman. Treemaps: A space-filling approach to the visualization of hierarchical information. In *IEEE Visualization Conf.*, pages 284–291, 1991.

[10] N. Katayama and S. Satoh. The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *ACM SIGMOD*, pages 369–380, May 1997.

[11] M. Kornacker. High-Performance Generalized Search Trees. To appear in *VLDB*, Sept. 1999.

[12] M. Kornacker, M. Shah, and J. Hellerstein. An analysis framework for access methods. Submitted for publication to ICDE 2000.

[13] J. Lamping, R. Rao, and P. Pirolli. A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *ACM SIGCHI* , May 1995.

[14] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. DEVise: Integrated Querying and Visual Exploration of Large Datasets. In *ACM SIGMOD* , 1997.

[15] G. Robertson, J. Mackinlay, and S. Card. Cone trees: Animated 3D Visualizations of Hierarchical Information. In *ACM SIGCHI* , pages 189–194, 1991.

[16] D. A. White and J. R. Similarity Indexing with the SS-Tree. In *ICDE*, pages 516–523, Feb. 1996.