amdb: An Access Method Debugging Tool

Marcel Kornacker, Mehul Shah, Joseph M. Hellerstein
U. C. Berkeley
{marcel,mashah,jmh}@cs.berkeley.edu

1 Introduction

The design and tuning of new access methods (AMs) for non-traditional data types and application areas has always been more of a black art than a rigorous discipline. The designer can only rely on intuition to come up with an effective design; its evaluation and profiling require tedious instrumentation of complex AM code and a host of hand-written scripts.

To address these issues, we developed amdb, a visual AM "debugging" tool to support the AM design and implementation process. It is based on the GiST (Generalized Search Tree, [HNP95]) framework for AM construction, which offers the designer an abstracted view of a tree-structured AM and factors out the mechanical aspects of an AM implementation, such as tree traversal, concurrency control and recovery. Amdb is a visual analysis, debugging and profiling tool for AMs that are written as extensions of libgist, a public-domain stand-alone C++ implementation of GiSTs.

2 System Features

Amdb was developed with the entire AM design and implementation process in mind and supports the designer in three areas:

1. Analysis of the dataset (i.e., the search keys of the data) and the index tree structure to evaluate the general indexability of the dataset ([HKP97]) and the effectiveness of the design.
2. Debugging of dynamic tree operations to pinpoint implementation flaws.
3. Profiling of a defined query workload to measure the level of end-user performance.

Central to the user interface of amdb is a graphical display of the tree structure, which greatly contributes to the ease-of-use of the debugging and analysis functions. To enhance the clarity of the display, individual subtrees can be collapsed and expanded via a point-and-click interface.

As an initial step in AM design, the dataset should be evaluated with regard to the target query workload to determine its degree of indexability, i.e., whether an AM can succeed in clustering the data to outperform a sequential scan of the dataset for the particular queries in the workload. The starting point of an indexability analysis is a query workload and the materialized result sets (or rather: enumerations of the set elements) of the workload queries. The latter reflect the "clustering affinity" of groups of data items and serve as the actual input to the analysis algorithm. Given this input, amdb uses a hypergraph partitioning algorithm to cluster the data. Based on this clustering, statistical indicators are computed that characterize the indexability of the dataset/workload combination.

The debugging facility of amdb was designed to make the individual steps of the insertion, deletion and search operations visible, so that bugs and shortcomings in the AM implementation can be easily located and corrected. The debugging operations available to the user are similar to those found in programming language debuggers, except that the smallest unit of execution is not a single line of source code, but a single node-oriented action such as a node insertion or node traversal. Specifically, the debugging functions include:

- single-stepping through insertion, deletion and search operations; as an aid to the user, the progression of each operation through the tree is visualized by highlighting the traversed nodes and updating the display in response to node splits and deletions
- breakpoints on specific tree nodes and/or node events such as split, deletion and node updates/traversal
- display of the current traversal path from the root; the nodes on the path from the root are highlighted in the display of the tree
- display of relevant node-related data (space utilization, fanout, more advanced node statistics, node contents—the latter with a user-supplied function)
- a tracing window, where global statistics and statistics on selected nodes are shown and continuously updated
- batch execution of commands via scripts, in order to restore the state of an index tree without having to re-execute all the modifications manually

The analysis functions found in amdb give the designer feedback on the structural aspects of the index tree that are likely to affect retrieval performance. Simple statistics such as node utilization, fanout or average entry size can be computed either for individual nodes or for user-selected groups of subtrees; in the latter case, the statistics are displayed visually by coloring the nodes in the tree display. A visualization of the node contents can also be helpful; this is displayed in a separate window and requires a user-supplied display function. More advanced (built-in) analysis functions display indicators for the quality of the clustering achieved at each level in the tree and for the quality of the bounding predicates. Examples for indicators of clustering quality are node cohesiveness and overlap, both of which can be derived from profiling statistics and are displayed by coloring the corresponding nodes in the tree display. Indicators for the quality of bounding predicates are their storage size and accuracy—how concisely they represent the data contained in the subtree. They are visualized in a manner similar to the other indicators.

A central part of the evaluation of a newly designed AM is profiling and comparison with existing AMs. Amdb facilitates this by allowing the user to specify batch operations via scripts and by collecting performance-relevant data during the execution of tree operations (examples: number of page accesses/reads/writes, number of aborted tree descents, BP accuracy metrics, etc.). The accumulated internal counters can be written to files for further processing or displayed graphically.

3 Implementation

The GUI frontend and tree visualization are implemented in Java and the GiST algorithms for tree updates and searching are provided by libgist, which is implemented in C++. The remaining analysis functions are also written in C++.

4 Demonstration Description

We will demonstrate amdb by going through a comparison of three tree-structured spatial AMs, namely R*-trees, SS-trees and SR-trees. Our group initially undertook this comparison as a research project ([WHL]), but without the help of tools. The goal of the comparison is to characterize the performance differences between the AMs and to determine the source of these differences. The AMs are structurally very similar and we will investigate three aspects on which they differ: split algorithms, insertion heuristic and representation of the bounding predicates. The analysis functions of amdb show which of these aspects contribute to the performance differences.

The second part of the demonstration will show how the debugger can be used to track down deficiences in an AM implementation. Starting with an R-tree implementation that employs a naive split algorithm, we will single-step through individual insertions and node splits to show the effect the split algorithm has on the tree structure. The debugging facility lets the developer examine the effects of design decisions at a smaller scale than the global statistics do, and we will use this more focused perspective in our demonstration to refine the flawed split algorithm.

References

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

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st Int'l Conference on Very Large Databases (VLDB)*, pages 562–573, September 1995.

[WHL] S. Wang, J. Hellerstein, and I. Lipkind. Near-Neighbor Query Performance in Search Trees. Submitted for publication.