2004 IEEE International Conference on Systems, Man and Cybernetics

Search for Approximate Matches in Large Databases*

Eugene Fink
Language Technologies
Carnegie Mellon University
Pittsburgh, PA 15213
e.fink@cs.cmu.edu

Aaron Goldstein
DYNAMiX Technologies
12330 Perry Highway
Wexford, PA 15090
agoldstein@dynamixtechnologies.com

Philip Hayes
DYNAMiX Technologies
12330 Perry Highway
Wexford, PA 15090
phayes@dynamixtechnologies.com

Jaime G. Carbonell
Language Technologies
Carnegie Mellon University
Pittsburgh, PA 15213
jgc@cs.cmu.edu

Abstract – *We present a system for indexing large sets of records, and retrieving exact and approximate matches for a given query. We define records, queries, and matches between them, describe an indexing structure for fast identification of exact and approximate matches, and give results of testing the system on a database of hospital patients.*

Keywords: Massive data, indexing and retrieval, approximate matches.

1 Introduction

We have developed a data structure and search algorithms for identifying exact and approximate matches in a large set of records, and tested them on a database of patients admitted to the Massachusetts hospitals in the 2001 and 2002 fiscal years. We explain the related representation of records and queries (Section 2), describe the developed technique for retrieval of records that match a given query (Sections 3 and 4), and show how its performance depends on the number of records, query type, and size of the available memory (Section 5).

2 Records and queries

We first define records and queries, and illustrate them with an example of patient data. We specify a table of records by a list of attributes; as a simplified example, we can describe patients by their sex, age, and diagnosis. A *record* includes a specific value for each attribute; for example, it may indicate that a patient is female, her age is 30, and her diagnosis is asthma. On the other hand, a *query* may include multiple values for each attribute; that is, it may include some set I_1 of values for the first attribute, some set I_2 for the second attribute, and so on. For example, if we need to retrieve all patients between 20 and 40 years old, who have either asthma or flu, we can use the query ({male, female}, [20..40], {asthma, flu}). We may specify a query attribute by a single value, a numeric range, or a set of several values or ranges. An n-attribute record $(i_1, i_2,..., i_n)$ is an *exact match* for a query $(I_1, I_2,..., I_n)$ if $i_1 \in I_1$, $i_2 \in I_2$, and so on; that is, every value in the record belongs to the respective set in the query. If a query specifies one value for each attribute, it is called a *point* query; if it includes ranges or sets, it is a *region* query.

The notion of *approximate matches* is based on distances between records. We specify a distance function for each attribute, along with an n-argument function that combines attribute distances into an overall distance. An attribute distance is a two-argument function, with nonnegative values, which must give zero for identical arguments. The n-argument combination function must give zero when all arguments are zeros, and it must be monotonically increasing on each argument. The distance from a record to a region query is defined as the smallest distance between the record and the region specified by the query. In particular, if the record exactly matches the query, the distance is zero. When looking for approximate matches, we specify not only query attributes, but also a distance function, number of matches, and distance limit; note that we may use different distance functions with different queries. If the database includes more than the given number of matches within the specified distance limit, the system retrieves the given number of the closest matches; else, it returns all matches within the distance limit.

* 0-7803-8566-7/04/$20.00 © 2004 IEEE.

3 Indexing structure

The system arranges all records into a tree, with height equal to the number of attributes, as shown in Figure 1. The root node encodes the first attribute, and its children represent different values of this attribute. The nodes at the second level divide the orders by the second attribute, and each node at the third level corresponds to specific values of the first two attributes. In general, a node at level i divides records by the values of the ith attribute, and each node at level $(i + 1)$ corresponds to all records with specific values of the first i attributes.

Every node includes a red-black tree, which indexes its children by the corresponding attribute; for example, each "age" node in Figure 1 includes a red-black tree that arranges its children by age values. This tree supports fast addition and deletion of children, as well as fast retrieval of all children in a given range. The nodes also include summary data that help to identify approximate matches; specifically, every node stores the minimal and maximal value of each numeric attribute in the corresponding subtree.

When the system adds a new record, it creates the appropriate new branch in the indexing tree, and updates the summary data of the ancestors of this branch. When removing a record, it deletes the corresponding leaf node, and updates the summary values of the ancestor nodes. If the deleted leaf is the only leaf in some subtree, the system removes this subtree; for example, the deletion of the leftmost leaf in Figure 1 leads to the removal of its parent.

If the tree size is smaller than the size of the available memory, the system keeps the entire tree in memory; otherwise, it divides the tree into fixed-size blocks and stores them on disk. We usually use 1-Kbyte blocks, which means that the total size of nodes in a block must be at most 1 Kbyte. In Figure 1, we show blocks by dashed boxes; a block may include a single node, several adjacent nodes, or part of a large node that does not fit into a block. If a block includes multiple nodes, they form a tree, which is part of the indexing tree. The overall block structure is also a tree, where the edges between blocks correspond to the cross-block edges of the indexing tree. This structure supports addition and deletion of records, which involves dynamic splitting of overfull blocks and dynamic merging of adjacent underfull blocks.

4 Search for matches

We use two algorithms that retrieve matches for a given query; the first is depth-first search, which identifies exact matches, and the second is best-first search, which finds approximate matches.

The depth-first algorithm begins by identifying all children of the root that match the first attribute of the query, and then recursively processes the respective subtrees. For example, suppose that we are looking for 30-year old patients of both sexes with asthma or ulcer, and the indexing tree is as shown in Figure 1. The algorithm determines that both children of the root match the first attribute of the query, and then processes the respective subtrees. It identifies two matching nodes for the second attribute, and three matching leaves for the third attribute; we show these nodes by thick boxes.

The best-first algorithm uses a node's summary data to estimate the distance to the closest possible match in the node's subtree. It arranges nodes into a priority queue by their distance estimates; at each step, it processes the smallest-estimate node. If this node is a leaf, it returns the respective record; else, it adds the node's children that match the next attribute of the query to the priority queue.

5 Experiments

We have experimented with a medical database, obtained from the Massachusetts Health Data Consortium, which contains a list of all patients admitted to the Massachusetts hospitals in the 2001 and 2002 fiscal years, that is, from October 2000 to September 2002. These data are anonymous, which means that the database does not contain enough personal information to identify individual patients. The database includes 1.6 million patient records, described by seventy attributes; we have used twenty-one of these attributes to build an indexing tree. The size of the tree that includes all 1.6 million records is 890 MBytes. We have used a 2.4-GHz Xeon computer with 400-MHz bus, and we have run the system with five different sizes of the main memory: 64 MBytes, 128 MBytes, 256 MBytes, 512 MBytes, and 1,024 MBytes.

In Figure 2, we show the dependency of the retrieval time on the number of records, using logarithmic scale. We have measured the speed of retrieving exact matches for point queries (dotted lines), exact matches for region queries (dashed lines), and approximate matches (solid lines). If the memory size is 1,024 Mbytes, the system stores all 1.6 million records in memory; if the memory is smaller, the system stores part of the indexing tree on disk. The graphs show that, when the tree grows beyond the memory size, the retrieval time increases sharply. In Figure 3, we give a different view of the same results; specifically, we plot the dependency of the retrieval time on the memory size for three fixed sizes of the indexing tree.

The results show that the disk indexing is 10 to 50 times slower than the memory indexing. When the system keeps the tree in memory, the retrieval of matches for exact point queries is 2 to 5 times faster than that for region and approximate queries. When the system uses the disk indexing, exact point queries are 50 to 200 times faster than region and approximate queries.

The retrieval time for exact point queries is proportional to the logarithm of the number of records; that is, its time complexity is $O(\lg N)$, where N is the number of

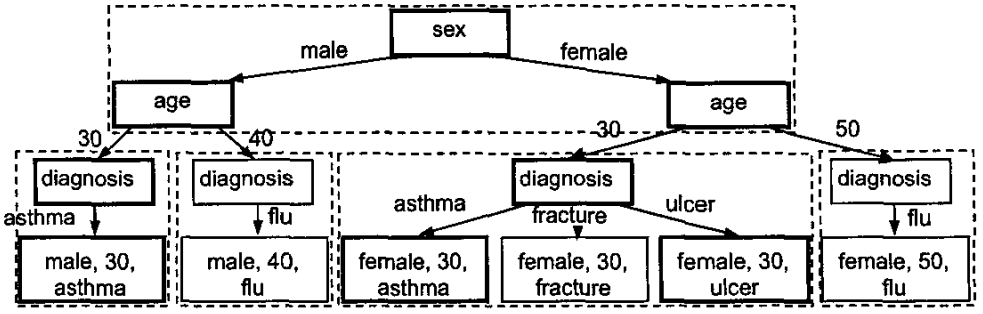

Figure 1: Indexing tree with patient records. The solid boxes are nodes of the tree, and the large dashed boxes are blocks used in the disk indexing. We use thick boxes to show the retrieval of 30-year old asthma and ulcer patients of both sexes.

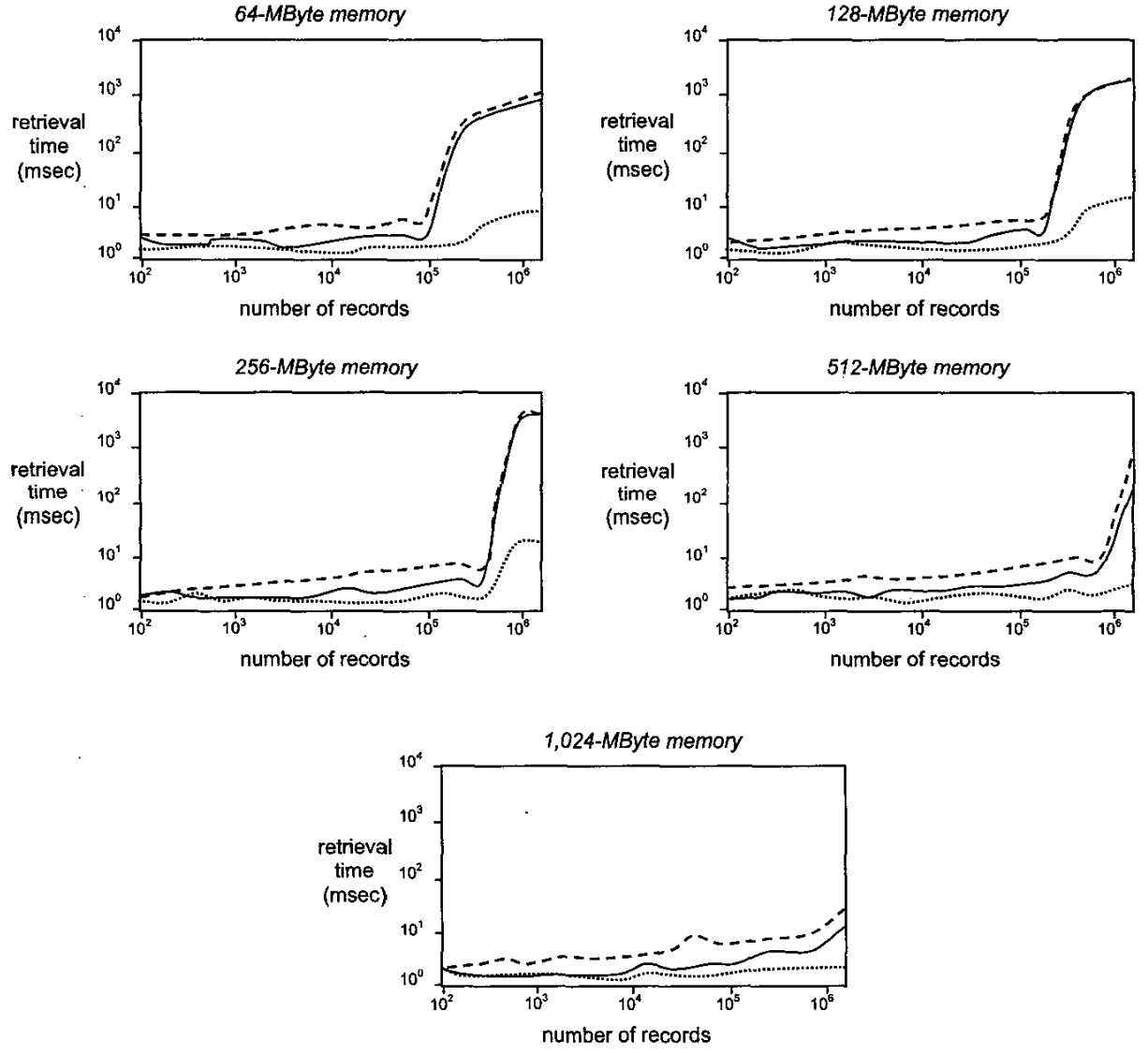

Figure 2: Dependency of the retrieval time on the number of records; the scale of both horizontal and vertical axes is logarithmic. We give the results of experiments with five sizes of the main memory: 64 MBytes, 128 MBytes, 256 MBytes, 512 MBytes, and 1,024 MBytes. For every memory size, we show the time of retrieving exact matches for point queries (dotted lines), exact matches for region queries (dashed lines), and approximate matches (solid lines).

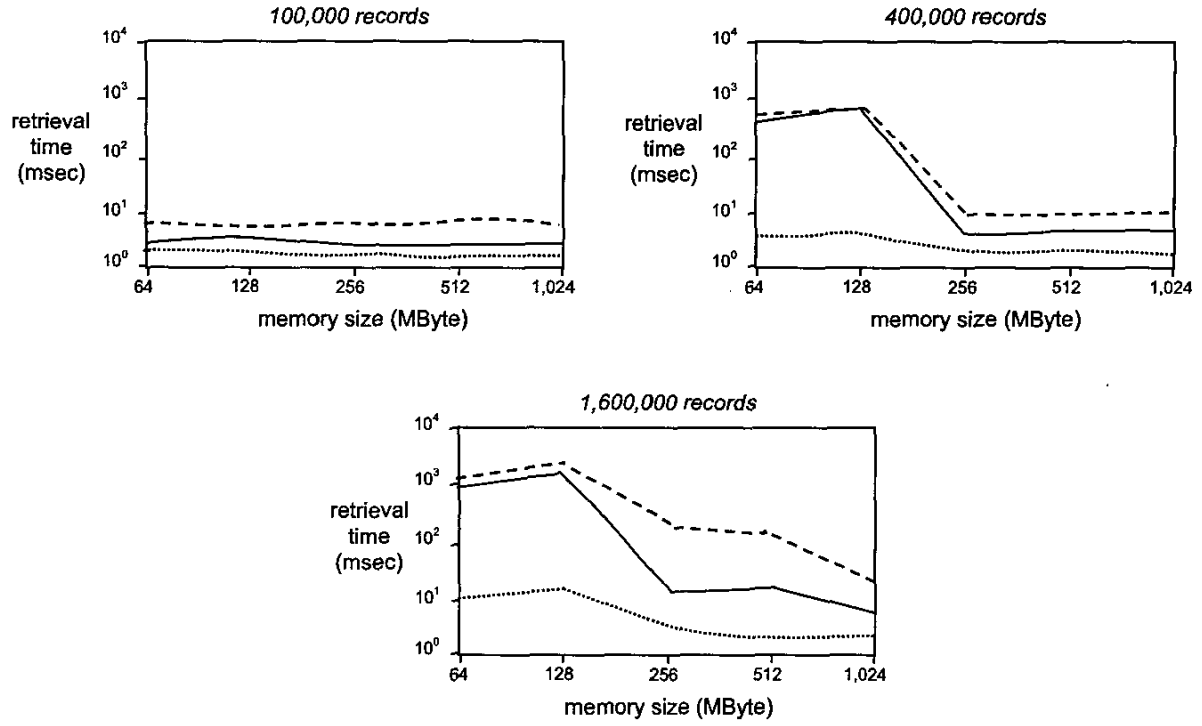

Figure 3: Dependency of the retrieval time on the size of the main memory; the scale of both horizontal and vertical axes is logarithmic. We show the dependency for three sizes of the indexing tree: 100,000 records, 400,000 records, and 1,600,000 records. For every size, we plot the retrieval time for exact point queries (dotted lines), exact region queries (dashed lines), and approximate queries (solid lines).

records. On the other hand, the time of region and approximate queries is proportional to a low power of the number of records; specifically, it is approximately $O(N^{0.2})$ for the memory indexing, and $O(N^{0.5})$ for the disk indexing.

We have also measured the disk utilization, that is, the ratio of the actual tree size to the total size of disk blocks; it varies from 54% to 58%, with mean at 55%.

6 Conclusions

The reported work is a step toward the development of tools for fast identification of patterns in large databases. It is part of a joint project involving Carnegie Mellon University and DYNAMiX Technologies, aimed at finding both known and surprising patterns in massive data streams [3]. The described system supports fast access to available data, and serves as a tool for the development of data-mining algorithms.

The experiments have confirmed that the system scales to large databases, which do not fit into the main memory; however, the observed time complexity of retrieval for such databases is $O(N^{0.5})$, which means that the retrieval time grows fairly quickly with the database size, hence restricting the system's scalability.

To improve the retrieval speed, we are working on the integration of the current indexing tree with *k-d* trees [1, 5] and PATRICIA trees [4], and on techniques for determining the most effective ordering of attributes in the tree. We also plan to enhance scalability by improving the disk utilization and developing a distributed version of the system. The long-term goal is to scale the system to databases with tens or hundreds of billions of records.

Acknowledgements

We are grateful to Johny Mathew for his extensive help with experiments. We thank Dwight Dietrich and Ganesh Mani for their valuable comments and suggestions. This work has been partially supported by the Advanced Research and Development Activity (ARDA), Novel Intelligence from Massive Data Program.

References

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), pages 509–517, 1975.

[2] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, Cambridge, United Kingdom, 1997.

[3] Chun Jin and Jaime G. Carbonell. ARGUS: Rete + DBMS = efficient continuous profile matching on large-volume data streams. Language Technologies Institute, Carnegie Mellon University, 2004. Technical Report CMU-CS-04-181.

[4] Donald R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4), pages 514–534, 1968.

[5] Jack A. Orenstein. Multidimensional tries used for associative searching. *Information Processing Letters*, 14(4), pages 150–157, 1982.

[6] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, second edition, 2003.

[7] Hanan Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.

[8] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.