01-AE

World Multiconference on Systemics, Cybernetics and Informatics

July 22-25, 2001
Orlando, Florida, USA

PROCEEDINGS

Volume I

Information Systems Development

Organized by IIIS
International
Institute of
Informatics
and Systemics

Member of the International
Federation of Systems Research

IFSR

Co-organized by IEEE Computer Society
(Chapter: Venezuela)

EDITORS
Nagib Callaos
Ivan Nunes da Silva
Jorge Molero

Algorithms for Efficient Data Compression in Databases using the Semantic Binary Model*

Art SHAPOSHNIKOV, Naphtali RISHE, Daniel MENDEZ
High Performance Database Research Center, Florida International University
Miami, Florida 33199, U.S.A.

Ouri WOLFSON
Department of Electrical Engineering and Computer Science
University of Illinois at Chicago
Chicago, IL 60612, U.S.A.

ABSTRACT

Databases created using the Semantic Binary Model represent real world data more naturally than those created using the table representation of the relational model. This results in databases requiring less storage space as well as smaller programming efforts. This paper presents data structures used in the Semantic Binary Object-Oriented Database. Algorithms for B-tree index and suffix compression are described. Compression results in significantly reduced index data size and fast database index lookups.

Keywords: B-tree, database indexes, compression, semantic database.

1. INTRODUCTION

B-trees and B+Trees [1] are used as indexing structures for variable length records in most databases. A B-tree is a large collection of lexicographically ordered strings (keys) with attached data records. Both keys and the records attached to them can be of variable length. The lexicographic order allows index lookups to be performed in O(log(N)) time where N is the total number of records in the B-tree. While this runtime is close to optimal for variable length data, there are algorithms that work faster than B-trees for fixed length data, for example hash tables that can give O(1) runtime. Therefore, B-trees should be used primarily for variable length data. Many commercial databases use predominately fixed length data due to the above mentioned performance considerations. However, a large part of real life data is of variable length. Street addresses, names, and phone numbers are examples of variable length data. To deal with this, software engineers have to model the variable length data either by reserving more space than necessary to accommodate all possible variable length records or by writing additional software code that represents variable length data using fixed length records. Variable length data containers, such as B-trees, can be utilized to save both space and programming efforts. For example, the Semantic Binary Object-Oriented Database (Sem-ODB) [7], described in more detail in section 2, is based on B-trees. Sem-ODB efficiently accommodates variable length database records using B-trees. Performance comparisons with commercial databases [4] demonstrated significant improvements in space and runtime database performance. B-trees are also used by Sem-ODB to efficiently represent one-to-many relationships. One-to-many relationships cannot be modeled in relational databases without either duplicating the records in a table or creating additional artificial tables. Sem-ODB results in significantly shorter application programs partly due to this fact. [8] This paper describes the implementation of Sem-ODB using B-trees and shows how the storage and run-time performance of B-trees

* This research was supported in part by NASA (under grants NAG5-9478, NAGW-4080, NAG5-5095, NAS5-97222, and NAG5-6830) and NSF (CDA-9711582, IRI-9409661, HRD-9707076, and ANI-9876409).

can be improved by using compressed B-tree indexes.

2. SEMANTIC DATABASES

The semantic database models in general, and the Semantic Binary Object-Oriented Model (Sem-ODM) ([7] and others) in particular, represent information as a collection of elementary facts categorizing objects or establishing relationships of various kinds between pairs of objects. The central notion of semantic models is the concept of an abstract object. This is any real world entity about which we wish to store information in the database. The objects are categorized into classes according to their common properties. These classes, called categories, need not be disjoint, that is, one object may belong to several of them. Further, an arbitrary structure of subcategories and supercategories can be defined. The representation of the objects in the computer is invisible to the user, who perceives the objects as real-world entities, whether tangible, such as persons or cars, or intangible, such as observations, meetings, or desires.

The database is perceived by its user as a set of facts about objects. These facts are of three types: facts stating that an object belongs to a category; facts stating that there is a relationship between objects; and facts relating objects to data, such as numbers, texts, dates, images, tabulated or analytical functions, etc. The relationships can be of arbitrary kinds; for example, stating that there is a many-to-many relation address between the category of persons and texts means that one person may have an address, several addresses, or no address at all.

Logically, a semantic database is a set of facts of three types: categorization of an object denoted by **xIC**; relationship between two objects denoted by **xRy**; and relationship between an arbitrary object and a value denoted by **xRv**. Efficient storage structure for semantic models has been proposed in [6].

The collection of facts forming the database is represented by a file structure which ensures approximately one disk access to retrieve any of the following:

1. For a given abstract object **x**, verify/find to which categories the object belongs.
2. For a given category, find its objects.
3. For a given abstract object **x** and relation **R**, retrieve all **y** such that **xRy**.
4. For a given abstract object **y** and relation **R**, retrieve all abstract objects **x** such that **xRy**.
5. For a given abstract object **x**, retrieve (in one access) all (or several) of its categories and direct and/or inverse relationships, i.e. relations **R** and objects **y** such that **xRy** or **yRx**. The relation **R** in **xRy** may be an attribute, i.e. a relation between abstract objects and values.
6. For a given relation (attribute) **R** and a given value **v**, find all abstract objects such that **xRv**.
7. For a given relation (attribute) **R** and a given range of values $[\mathbf{v}_1, \mathbf{v}_2]$, find all objects **x** and **v** such that **xRv** and $\mathbf{v}_1 \leq \mathbf{v} \leq \mathbf{v}_2$.

The operations 1 through 7 are called *elementary queries*. The entire database can be stored in a single B-tree. This B-tree contains all of the facts of the database (**xIC, xRv, xRy**) and additional information called inverted facts: **CIx, Rvx,** and $\mathbf{yR}_{inv}\mathbf{x}$. (Here, **I** is the pseudo-relation IS-IN denoting membership in a category.) The inverted facts allow answers to the queries 2, 4, 6, 7 to be kept in a contiguous segment of data in the B-tree and answer them with one disk access (when the query result is much smaller than one disk block). The direct facts **xIC** and **xRy** allow answers to the queries 1, 3, and 5 with one disk access. This allows both sequential access according to the lexicographic order of the items comprising the facts and the inverted facts, as well as random access by arbitrary prefixes of such facts and inverted facts. The facts which are close to each other in the lexicographic order reside close in the B-tree. (Notice, that although technically the B-tree-key is the entire fact, it is of varying length and typically is only several bytes long, which is a typical size of the encoded fact **xRy**.)

Numeric values in the facts are encoded as substrings using the order-preserving variable-length number encoding of [5].

Table 1: Implementation of elementary queries summarizes how the elementary semantic queries are implemented using the B-tree interval operators. We use notation **S + 1** to denote a string derived from the original string **S** by adding 1 to the last byte of **S.** (For strings encoding abstract objects, this operation never results in overflow.)

Query	B-tree Implementation
1. **x?**	**[xI, xI+ 1]**
2. **C?**	**[CI, CI+ 1]**
3. **xR?**	**[xR, xR + 1]**
4. **?Rx**	**[xR_{inv}, xR_{inv}+ 1]**
5. **x??**	**[x, x + 1]**
6. **?Rv**	**[Rv, Rv + 1]**
7. **R[v_1..v_2]?**	**[Rv_1, Rv_2 + 1]**

Table 1: Implementation of elementary queries

For most elementary queries (queries 1, 3, 4, 5, and 6) the number of binary facts is usually small. Some queries (queries 2 and 7), however, may result in a very large number of facts, and it may be inefficient to retrieve the whole query at once.

3. B-TREE COMPRESSION

Compression in databases not only reduces the storage requirements but also improves the runtime performance of queries by reducing the number of disk accesses needed to traverse the B-tree index. [2] [9] As explained in the previous section, the Semantic Binary database can be represented as a lexicographically ordered set of strings where each string represents a binary relationship between two objects. Physically this set is organized as a B+ tree:

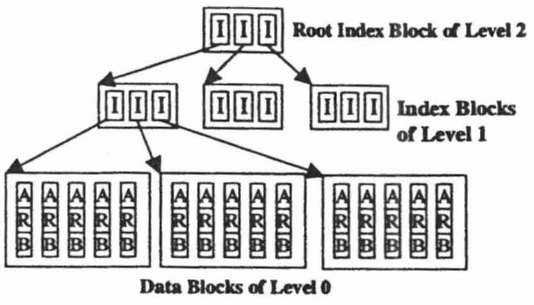

Figure 1: Semantic Database as a B+ tree where:

I is an index string,

A
R
B

is a binary string representing a binary fact ARB that an abstract object A is related to object B by relation R.

It is reasonable to expect that in a large B+ tree we will have many data strings with common prefixes. In the Semantic Binary database we also need to store inverted facts - binary facts where the relation identifier and the concrete object (such as character string) precede the abstract object identifier: RVA.

There will be many facts that start with the same relation identifier R. Moreover, the concrete objects V are also likely to have large common prefixes. For example, in case of a relation Last Name in a category PERSON we will have large average common prefixes that one can observe by looking into a residential telephone book.

Common B-tree string prefixes can be compressed by a modified [3] encoding, where the first character of the string R represents the number of matching characters in the previous string. Since the length of the strings in our B-tree is variable, we also need to store the length of the string L. It is reasonable to limit the compression block length to the length of the data or index block in a B-tree, otherwise we will have to perform more disk accesses. When the prefix compression is limited to one block, the first string in each block is stored uncompressed. A typical B-tree block would looks as follows:

0LSX RLSX RLSX ... RLSX 00

where R is the common prefix length, L is the length of the string, S is the string suffix that cannot be compressed, and X is a *data suffix* string. Characters in the data suffix do not affect the lexicographical ordering. So, the data suffix can be used to store some additional information about the string. For example, a suffix of an index string can contain the address of the corresponding data block in the B+ tree. The last string indicates the end of data and has R = 0 and L = 0.

The search algorithm for a given string S can be implemented efficiently without decompressing a block. The search procedure is a sequential scan, but each step of this scan can be implemented so efficiently that it requires only one byte comparison operation to compare two strings. So, this sequential scan in compressed strings can be faster than a binary search in the same decompressed block since the binary search requires R comparisons for a pair of strings.

We also used an alternative compression technique that results in a smaller degree of compression but permits us to use a faster binary search within a block. At the beginning of each data or index block we store the largest common prefix and an addressing map of the string suffixes within the block. A binary search is performed using the addressing map.

4. INDEX COMPRESSION

Consider an index block *I* that contains the B-tree index strings. Each index string S in I has a data suffix three bytes in length. This suffix stores the disk address of a data block B that corresponds to the index string S. Each string in the block B is greater or equal to the index string S. The first string in a data block can serve as the index string S. However, any string which is shorter than the string S and is still greater than the last string in the previous block, can be used as the index string as well. So, the index string of a data block B can be chosen as the shortest prefix of the first string in the block B which is still greater than the last string in the previous block.

Note that we will not need to adjust the index string when several strings are inserted or deleted from the block B or its neighbors. This is because such insertions/deletions do not affect the main property of the index string: after an insertion or deletion each string in the block B will still be greater or equal than S and each string in the previous block will be less than S.

The prefix and suffix compression in index blocks results in very short index strings. Indeed, each index string consists of the repetition counter R, the length indicator L, the string body, and the 3 byte data address. It turns out that the average size of the index string body in a large B+ three is only slightly greater than one byte. So, the average index string length is less than 7 bytes. Such short indexes allow most of B+ tree index blocks to be stored in cache memory, which reduces the average number of disk accesses per user query.

Note that the index strings of an upper level index block that reference the lower level index blocks can not be compressed further because this will violate the property of an index string with respect to the data blocks: an index string is a lexicographical boundary between two data blocks.

5. CONCLUSION

Our compression algorithms were implemented and their performance was compared with relational databases using standard benchmarks as well as other real life applications benchmarks [4]. We compared the compression and CPU performance of Semantic Binary database with relational databases such as Oracle. The Semantic Database was fully indexed by its nature, while relational databases used only the indexes required to achieve the best performance for queries. Variable length data and B-tree compression allowed us to compress the databases by a factor of 2 or more compared to relational databases containing the same data, while still outperforming or having close performance to relational databases that used fixed length data.

6. REFERENCES

[1] D. Comer. "The Ubiquitous B-tree," *ACM Computing Surveys*, Vol. 11 No. 2., June 1979.

[2] Susan J. Eggers, Frank Olken, Arie Shoshani. "A Compression Technique for Large Statistical Data-Bases", *Proceedings of Very Large Data Bases, 7th International Conference*, September 9-11, 1981, Cannes, France, VLDB 1981: 424-434.

[3] J.Ziv and A.Lempel. "A Universal Algorithm for sequential Data compression", *IEEE Transactions on Information theory*, Vol IT-23, No.3, May 1977, PP337-343.

[4] Naphtali Rishe, Alexander Vaschillo, Dmitry Vasilevsky, Artyom Shaposhnikov, Shu-Ching Chen. "A benchmarking technique for semantic databases", *Proceedings of ACM SIGMOD ADBIS-DASFAA Symposium on Advances in Databases and Information Systems*, September 2000.

[5] N. Rishe. "Interval-based approach to lexicographic representation and compression of numeric data", *Data & Knowledge Engineering*, n 8, 1992, pp. 339-351.

[6] N. Rishe. "A File Structure for Semantic Databases", *Information Systems*, v 16 n. 4, 1991, pp. 375-385.

[7] N. Rishe. *Database Design: The Semantic Modeling Approach*, McGraw-Hill, 1992.

[8] N. Rishe, J. Yuan, R. Athauda, S. Chen, X. Lu, X. Ma, A. Vaschillo, A. Shaposhnikov, D. Vasilevsky. "SemanticAccess: Semantic Interface for Querying Databases", *Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt*, 2000 pp. 591-594.

[9] G. Ray and J.R. Haritsa, S. Seshadri. "Database Compression: A Performance Enhancement Tool", *Proceedings of International Conference on Management of Data*, 1995.

980-07-7541-2

57000>

9 789800 775417

ISBN: 980-07-7541-2