STORAGE TYPES IN THE SEMANTIC BINARY DATABASE ENGINE

Naphtali Rishe, Malek Adjouadi, Maxim Chekmasov,
Dmitry Vasilevsky, Scott Graham, Dayanara Hernandez
Florida International University, Miami, FL

Ouri Wolfson
University of Illinois at Chicago, Chicago, IL,

Keywords: Semantic binary data model, storage type, database management system.

Abstract: Modern database engines support a wide variety of data types. Native support for all of the types is desirable and convenient for the database application developer, as it allows application data to be stored in the database without further conversion. However, support for each data type adds complexity to the database engine code. To achieve a compromise between convenience and complexity, the semantic binary database engine is designed to support only the binary data type in its kernel. Other data types are supported in the user-level environment by add-on modules. This solution allows us to keep the database kernel small and ensures the stability and robustness of the database engine as a whole. By providing extra database tools, it also allows application designers to get database-wide support for additional data types.

1 INTRODUCTION

Conceptually, a semantic binary database is a set of facts about objects, (Rishe, 1992). Objects belong to categories. Relations are defined between categories, and objects are connected by relations. Objects can also have attributes, which are considered to be relations from objects to values. The original semantic database engine stores information about schema and abstract objects as a set of facts at the logical level. The following facts are stored:

- `xC`. This is a fact that an abstract object `x` belongs to a category `C`. If an object belongs to several categories, one fact is stored for each of these categories. While the object may belong to any number of different categories that are not disjoint, it may most commonly belong to several subcategories of one category. If the object belongs to the subcategory, it also belongs to the corresponding category.
- `xRy`. The object `x` is connected to the object `y` by the relation `R`. Two objects can not be connected twice by the same relation. They are either connected or not.
- `xRv`. The attribute `R` of object `x` has value `v`. Object attributes are like relations, therefore they may be multi-valued. However, the object can not have one specific value of the attribute two times.
- `Cx`. This is an inverse (redundant) fact for `xC`. It is used to query the objects that belong to a certain category.
- $yR^{-1}x$. This is an inverse (redundant) fact for `xRy`. It is used to traverse relations in a reverse order.
- $R^{-1}vx$. This is an inverse (redundant) fact for `xRv`. Inverse facts for attributes have a different structure than the inverse facts for the relations with abstract objects. They are used in queries that search for all the objects with a certain attribute value.

The facts are encoded as binary strings using reversible encoding. Every `y`, `x`, `C`, `R`, or R^{-1} in the above facts are encoded with object IDs. The object ID is an integer; it is encoded with a variable length encoding that maps the natural ordering of integers into the lexicographical ordering of strings. This encoding is compressed in the sense that small values of object IDs result in short strings. The encoding also allows the database to find the end of

Rishe N., Adjouadi M., Chekmasov M., Vasilevsky D., Graham S., Hernandez D. and Wolfson O. (2005).
STORAGE TYPES IN THE SEMANTIC BINARY DATABASE ENGINE.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 437-440
DOI: 10.5220/0002535804370440
Copyright © SciTePress

the encoded integer when it is used as a prefix of a binary string. Every encoded string starts with an object ID; we insert different separators to distinguish between pairs and triples.

The only facts that have values other than object IDs are xRv and R^{-1}vx. The values are encoded in such a way that natural ordering on the values is mapped into the lexicographical order of encoded strings. Encoding of xRv is a concatenation of three encoded values and a separator. However, encoding of R^{-1}vx presents a problem. Simple concatenation is no longer sufficient, since we need to be able to find the end of the encoded v; this imposes a strict restriction on encoding. Another separator is added between v and x at the end of the fact.

The semantic binary database distinguishes between two groups of data types – abstract data types and concrete data types. Abstract data types are categories of abstract objects, for example Person, Professor, Student. Concrete data types are Integer, String, Datetime among others.

The hierarchy of data types shown in Figure 1 has a single data type Binary at the root; only Binary data type is supported by the engine kernel. Other data types are supported in the user-level environment by the add-on modules. This solution allows us to keep the database kernel small and to get database-wide support for additional data types.

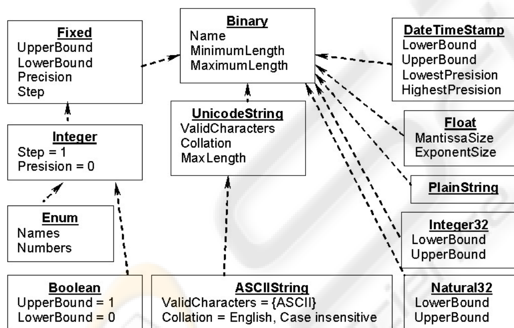

Figure 1: Sub-schema of concrete types

2 FACT STORAGE

As we have mentioned, the original semantic binary database engine stores the schema and content of the database using facts. Every object is given a unique object ID; the information about the object is represented as a set of direct facts xC, xRy and xRv. For every direct fact there is one redundant reverse fact stored in the database. All facts are encoded into binary strings which are stored in a B-Tree structure in lexicographical order. We will call this storage method fact storage.

With the fact storage method, every attribute of an object is stored as one direct fact. The opposite is also true, with every direct fact storing only one attribute. This approach is inefficient in two ways: space overhead and performance overhead.

With respect to space overhead, the direct fact stores an object ID, a relation ID, and some data related to the B-Tree structure in addition to its attribute data. For example, an integer attribute of 4 bytes is stored in a fact with the object ID and the relation ID. With the object ID requiring 4 bytes of storage, the relation ID requiring 4 bytes, and the storage infrastructure requiring 4 bytes, a total of 16 bytes is needed to store a single 4-byte integer.

The issue of space overhead was addressed in the original semantic database design by introducing two space saving techniques: special encoding of integers and prefix compression of B-Tree blocks. Unfortunately, the implementation of these algorithms does not have acceptable performance. Encoding of integers requires complex operations at the bit level. Prefix compression of B-Tree blocks makes binary searches within the block impossible. Also, on average, only 4 bytes are saved for every fact with the prefix compression, which is less than the overhead introduced. Therefore this technique does not eliminate space overhead, and significantly decreases performance.

With respect to performance overhead, retrieval of information from the fact storage requires several time-consuming operations. The B-Tree has to be searched for the correct range of facts. A linear search of B-Tree blocks takes more time than a binary search or direct record access. Complex bitwise operations are needed to unpack relevant information from the facts. If all the attributes for one object are needed, unpacking of several facts is required.

Another problem with fact storage was uncovered by the standard TPC-D benchmark (TPC-D, 1998). For some queries in TPC-D, it is necessary to scan all the objects in a category and retrieve most of their attributes. While only one elementary query is required to find all objects in a category, retrieving all the attributes of every object requires one elementary query per object. This may result in as many disk accesses as there are objects in the category. However, if the objects belonging to one category are stored closely together in one linear file, many of them would fit into one disk block. Scanning the set of objects would only require as many disk accesses as there are blocks. These inefficiencies prompted us to search for a better representation of the data.

3 RECORD STORAGE

Comparing the semantic binary database model and the relational database model, we can state that a category roughly corresponds to a table, an object corresponds to a table row and an attribute corresponds to a column. Relational databases do not have a large overhead for infrastructure when storing data. Typically, all attributes of an object are stored in a fixed-size record in a positional form. A fixed amount of space is reserved for the attribute, requiring no overhead for object IDs or infrastructure at this level. Several records are grouped in pages and pages are stored in a linear file.

The relational databases offer a different way of storing information, which is called a clustered index (England, 2001). In a clustered index, a combination of attributes is designated as a key of a clustered index. All other attributes are organized in a record. The record is stored in a B-Tree along with the key. Since a fixed amount of space is allocated for every column (attribute) in a row (object), `m:m` attributes do not fit into the picture, therefore they are not directly supported by the relational database model. Also, if the attribute has a value of `NULL`, the record would still have space reserved for it. If data is sparse and many `NULL`s are present, this form of storage becomes space-inefficient.

We propose to adapt clustered index storage for semantic binary database storage in certain cases to improve space allocation and performance. Instead of storing every object attribute in a separate fact, one fact can be created to store some of the object's attributes. Not all the attributes can be stored in a record. The following conditions have to be met for the attribute to be qualified for record storage:

1. The attribute should be total. In other words it can not be `NULL`, so no special handling of `NULL`s is necessary.
2. Cardinality of the attribute should be `1:1` or `m:1`. There is at most one attribute for every object in the category, so no special handling is required for multi-value attributes. In combination with condition (1), this implies that only one value of the attribute is present for each object in the category.
3. The value of the attribute should have fixed length and the length is the same for all the objects; this allows us to allocate appropriate space for the attribute in a record.

The attributes concatenated together result in a record that is stored for all the objects in the category and that has a fixed length equal to the sum of lengths of all concatenated attributes. The record `v` can be placed in one fact `xR'v` where `R'` is a special system relation introduced for the purpose of storing the record. We will call this storage method record storage. Figure 2 illustrates fact and record storage.

A semantic binary database engine based on the record storage was implemented and analyzed. It achieved substantial performance improvements over the original semantic engine on certain types of workloads. The typical workload that benefits from this representation is the TPC-D benchmark.

For some queries in TPC-D, it is necessary to scan all the objects in a category and retrieve most of their attributes. Records for the objects belonging to the same category were placed in a separate file. To answer the query, a full scan on the entire file was performed and every record was retrieved. This required only as many disk accesses as needed for the size of the result set. In addition, no complex operations were required to extract information from the records. Once the record is in memory, several CPU instructions are enough to retrieve an attribute.

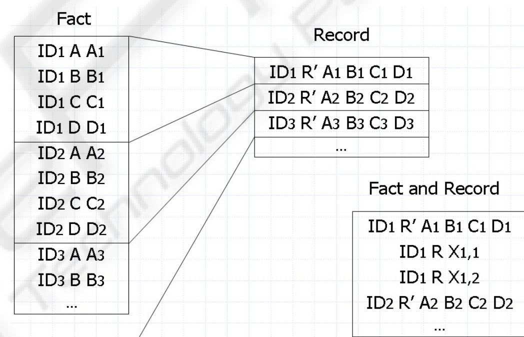

Figure 2: Fact and record storage in semantic binary engine

4 BITMAP STORAGE

Modern database management systems use bitmaps as another way of storing information. Bitmap indexes are available in many commercial systems, such as IBM DB2 (Winter, 1999) and Oracle (Jakobsson, 1997). Bitmaps offer the advantage of saving storage space and improving disk retrieval speed by requiring only one bit to store a Boolean value. In the semantic database engine, bitmaps can be used for storing reverse-fact information on Boolean, enumeration, and small integer attributes efficiently. An example of such a case is representing category membership for objects.

A bitmap is an array of bits, where one bit corresponds to one database object. Object ID is used as an index value in a bitmap array. Eight consecutive bits in a bitmap are stored as one byte.

Consecutive bytes are stored in blocks. Typical block sizes vary from 4 kilobytes to 64 kilobytes. Blocks can be stored sequentially or organized and referenced by a control structure. Logically, a bitmap represents a set of objects as a set membership vector. For every object in the database, the corresponding Boolean element of the vector is `True` if the object belongs to the set. For objects that do not belong to the set, the corresponding elements are `False`. Boolean values of the vector are represented by bits.

Consider the following method for storing category membership. Membership information for each category `C` is represented by one bitmap, where each bit corresponds to an object ID. Bit `N` in the array is `1` if an object with the object ID `N` belongs to category `C`, and `0` otherwise. Thus, the length of the array in bits will be equal to the number of objects in the database and access to this information will be fast since it is equivalent to direct access to an element of an array.

The size of the bitmap is proportional to the domain size. Thus, one bit is allocated in a bitmap for every abstract object regardless of whether it belongs to a represented set or not. For example, consider a set of objects that have certain Boolean attribute values set to `True`. The bitmap requires just one bit to represent each fact; it would otherwise be represented in fact storage as a string consisting of object ID, relation ID, and a Boolean value, thus using at least 10 bytes of storage. Being stored in a B-tree, this string would also require some space to maintain the B-Tree block structure. Thus, the bitmap is about 80 times more efficient in terms of space for total Boolean attributes. Access to facts in a B-Tree requires a B-Tree search and unpacking of facts, which are both non-trivial and time-consuming operations compared to retrieval of values from a bitmap, which, in most cases, is a simple direct access operation.

Operations to access individual bits are simple. Suppose `m` is the first object ID in a block and we need to access a bit that corresponds to the object with object ID `n`. The byte number within the block that the bit belongs to is `i=(m-n)/8`. Bit number within the byte is `j=(m-n)MOD8`, where `MOD` is the modulo operation (remainder of a division).

To read the bit, the following formula is used: `B[i]&(1<<j)`, where `B` represents block as a byte array. Result is `0` if bit is `0` and non-zero if bit is `1`. Use `B[i]|=(1<<j)` to set the bit to `1` and `B[i]&=~(1<<j)` to reset the bit to `0`. To set the bit to value `x`, use `B[i]=(B[i]&~(1<<j))|(x <<j)`.

Bitmaps have several attractive properties:

1. Bitmaps represent a set of objects in a compact way since only one bit is used per object. Under favourable conditions, this can be further improved.
2. Operations on bitmaps are fast since only one CPU instruction is needed to act on several objects simultaneously.
3. A bitmap is a simple structure and the overhead for accessing and maintaining it is small.

5 CONCLUSION

While designing a semantic database, the general approach is to be flexible in selecting storage types. Fact, record, and bitmap storage might be utilized simultaneously for different purposes. Record storage might be used for those attributes it suits best, whereas fact storage might be used for other attributes. Category membership might be stored as bitmaps. With fact storage, it should be up to the user to select the best storage type for various types of data.

ACKNOWLEDGEMENTS

This material is based on work supported by the National Science Foundation under Grants No. HRD-0317692, EIA-0320956, EIA-0220562, CNS-0426125, IIS-0326284, CCF-0330342, IIS-0086144, and IIS-0209190.

REFERENCES

Rishe, N., 1992. *Database Design: the Semantic Modeling Approach,* McGraw-Hill. 528 pp.

TPC-D, 1998. Transaction Processing Performance Council. *TPC Benchmark D*, Standard Specification Revision 2.1.

England, K., 2001. *Microsoft SQL Server 2000 Performance Optimization and Tuning Handbook*, Digital Press; 1st edition, 320 pp.

Winter, R., 1999. *Indexing Goes a New Direction*, Intelligent Enterprise, 2(2), pp. 70-73.

Jakobsson, H., 1997. *Bitmap Indexing in Oracle Data Warehousing.* Database seminar at Stanford University. http://www-db.stanford.edu/dbseminar/Archive/FallY97/slides/oracle.